PowerJ Programmer's Guide

VERSION 98.03.25

Copyright © 1996-1998 Sybase, Inc. and its subsidiaries. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc. and its subsidiaries.

PowerBuilder, Powersoft, S-Designor, SQL Smart, and Sybase are registered trademarks of Sybase, Inc. and its subsidiaries. Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, AppModeler, InfoMaker, the Column Design, ComponentPack, DataArchitect, DataExpress, Data Pipeline, DataWindow, dbQueue, Dynamo, InfoMaker, Jaguar CTS, jConnect, MetaWorks, NetImpact, ObjectCycle, Optima++, Power++, PowerBuilder Foundation Class Library, PowerDesigner, PowerJ, PowerScript, PowerSite, PowerTips, Powersoft Portfolio, Powersoft Professional, ProcessAnalyst, SDP, SQL Remote, SQL Server, StarDesignor, Sybase SQL Anywhere, Watcom, Watcom SQL, and web.works are trademarks of Sybase, Inc. and its subsidiaries. Certified PowerBuilder Developer and CPD are service marks of Sybase, Inc. and its subsidiaries. DataWindow is a proprietary and patented technology of Sybase, Inc..

All other company and product names used herein may be the trademarks or registered trademarks of their respective companies.

Information in this manual may change without notice and does not represent a commitment on the part of Sybase, Inc. and its subsidiaries.

About this guide

This guide describes the PowerJ Java development environment. The guide assumes that you are familiar with the material in PowerJ *Getting Started*, supplied as part of your PowerJ package. The guide also assumes that you are familiar with the basic principles of using Windows, including how to:

- Start a Windows program.
- Reposition, resize, and close windows.
- Create, open, copy, and delete files and folders.
- Point, click, double-click and drag with a mouse or other pointing device.

If you are not familiar with such features, consult the Windows documentation (for example, *Introducing Microsoft Windows 95*).

Ways to use this guide

If you have used other rapid application development tools:

Two important differences between PowerJ and other RAD tools are drag-and-drop programming and using view windows. See the *Getting Started* guide for information on drag-and-drop programming using the Reference Card and Parameter Wizard, and for a description of view windows in PowerJ.

If you are a Java programmer and would like to plunge right in:

You should scan the *Getting Started* guide for information on drag-and-drop programming using the Reference Card and Parameter Wizard, and for a description of view windows in PowerJ. Then you should read Chapter 1, Basic concepts of PowerJ. Chapters 2 to 4 describe the basics of using PowerJ (how to save projects, how to edit source code, debugging, and so on). The specific details needed for writing PowerJ code begin with Standard types.

The chapters in Part I describe the mechanical aspects of using the PowerJ development environment. They describe the basic concepts of PowerJ, targets and projects, the different views and tools, debugging facilities, deployment issues, support for team programming and tips for coexisting with other tools.

The chapters in Part II discuss the most common aspects of writing PowerJ programs: standard types, fundamentals of using Java components, working with standard objects, creating your own menus, and using grids, forms, graphics and multiple threads.

The subsequent parts cover specific areas of Java programming with PowerJ. The chapters in Part III address topics related to JavaBeans and ActiveX components. The chapters in Part IV deal with database applications. Finally, the chapters in Part V deal with Internet (and intranet) applications, including distributed computing and creating applications for the Jaguar CTS transaction server.

Getting help while you work

In addition to online and printed manuals, PowerJ provides extensive online help facilities which explain how to use the software.

At any point during a PowerJ session, you can obtain help in a variety of ways:

- Point the cursor at anything on the screen and press F1. This provides a description of the purpose and use of the item indicated by the cursor.
- Click the question mark button, then click anything on the screen. PowerJ provides information about anything you click. When you have finished obtaining information, click off the question mark button.

Document conventions

The printed manuals for PowerJ use the following typographic conventions.

Typeface or symbol	Meaning
`Monospace type`	A monospaced font is used for code or for anything you must type. It is also used for the names of files and folders.
Bold face	Bold face is used for menus and other interface elements such as buttons and labels. It is also used for the names of components, classes, methods and events.
Italics	Italicized text is used to emphasize words such as new terms, the names of object properties and for text that is acting as a placeholder.
SMALL CAPITALS	Small capitals are used for the names of keys and combinations of keys, such as ENTER or SHIFT.
◆	This symbol denotes the beginning of a procedure for performing a task.

Important: A paragraph placed in a box often describes an exception to general rules given in the main body of the text.

You and the user

When the documentation contains a phrase like "you click the **OK** button", the word "you" refers to the person using PowerJ to develop a program. When the documentation contains a phrase like "the user clicks the **OK** button", "the user" refers to the person who will use the programs you develop using PowerJ.

Programs

This guide loosely uses the word "program" to refer to many sorts of applications. These may include applets, standalone executables, libraries, and so on.

Using the mouse

Unless specified otherwise, you use the *left* button in all actions with the mouse. For example, if the guide tells you to click or double-click an object on the screen, you use the left mouse button. Similarly, you use the left mouse button for all drag-and-drop operations, unless the documentation explicitly says to use a different button.

Part I. Development environment

This part describes the mechanical aspects of using the PowerJ development environment. It describes the basic concepts of PowerJ, targets and projects, the different views and tools, debugging facilities, deployment issues, team programming features and tips for coexisting with other tools.

Chapter 1. Basic concepts of PowerJ

This chapter presents basic information about PowerJ. Later chapters provide step-by-step instructions for performing the operations described here.

Java

Note: The online documentation for PowerJ provides an introduction to Java and Java programming titled *Thinking in Java*. If you are unfamiliar with the Java language, *Thinking in Java* is a good place to start learning how to write Java code. See Using Java tutorials for important information about using *Thinking in Java* with PowerJ.

Java is a programming language developed by Sun Microsystems. The syntax of Java source code has strong similarities with C++, with one important exception: Java does not support pointer types.

The original implementation of Java compiled Java source code into an executable code format designed to run on a *virtual machine*. Virtual machine code does not match any actual hardware. In order to run the virtual machine code, you need an *interpreter* program which reads the code and executes appropriate instructions to perform the operations described in the virtual machine code. Roughly speaking, the interpreter translates Java's abstract instructions into real instructions that can be executed by your computer hardware.

Note: At present, there are two popular implementations of the Java virtual machine: one from Microsoft and one from Sun's JavaSoft. PowerJ code can run on either version.

Standard file name extensions

A file containing Java source code typically has a name ending with `.java`. A file containing virtual machine code typically has a name ending with `.class`. For example, the Java source file `abc.java` might be compiled into the virtual machine code file `abc.class`.

JDK 1.02 vs. JDK 1.1

The specifications for Java are evolving quickly in the current computing industry. The most widely available versions are currently JDK 1.02 and JDK 1.1 (JDK is short for Java Development Kit). At the time of this writing, JDK 1.02 is more widely used than JDK 1.1, but that is likely to change as time goes on.

There are significant differences between the two versions of Java. Furthermore, the JDK 1.1 standard has labeled some JDK 1.02 features as deprecated or obsolete. Therefore, each time you begin creating an application, PowerJ asks whether you want to be compatible with JDK 1.02 or with JDK 1.1.

The code you write with one package is often not compatible with the other. As an example, the code for enabling a control with JDK 1.02 is typically:

```
enable( true )
```

The preferred form with JDK 1.1 is:

```
setEnabled( true )
```

These problems only occur when you are coding for objects defined in the "standard" Java libraries (such as AWT, the Abstract Windows Toolkit). When you are coding for objects defined by PowerJ itself, there is no difference whether you are using JDK 1.02 or 1.1. In other words, the methods defined by PowerJ are compatible with each other, whether you are using JDK 1.02 or 1.1, but the methods defined by Java and its associated libraries may not be compatible.

Note: PowerJ can help you to convert your PowerJ project from JDK 1.02 to JDK 1.1. You may also be able to write your Java code so that it will work with multiple JDK versions. For further information, see Migration from JDK 1.02 to JDK 1.1.

Event handling differences

An important difference between using JDK 1.02 and JDK 1.1 is that with JDK 1.1 PowerJ generates events using the delegation model while with JDK 1.02 PowerJ uses the old AWT event model.

JNI

JDK 1.1 introduces an interface for calling non-Java routines from a Java program. This feature is called JNI, and can be used in JDK 1.1 applications created with PowerJ. Note that the feature is not available with JDK 1.02 applications. Furthermore, the Microsoft 1.5 and 2.0 virtual machines for Java do not support JNI.

Using your own version of the JDK

The PowerJ package includes Sun's standard versions of JDK 1.02 and JDK 1.1; the package also includes Microsoft's Java SDK 1.5. However, it is possible that you will want to use different versions of these in your applications. For example, Sun is expected to release updated versions of JDK 1.1 from time to time. When Sun creates a new release of JDK 1.1, you may wish to use that release instead of the one that you received with PowerJ.

PowerJ makes it easy to use your own version of the JDK. For further information, see JDK configuration options.

Java and the Web

Since Java's virtual machine code is not specific to any hardware or operating system, you can run a Java program on any computing platform, provided that you have an appropriate interpreter. This makes Java ideal for use with the World Wide Web. If you make a compiled Java program available on the web, the program can be used by people on many different types of computing platforms.

Furthermore, Java was designed to be safe for users to execute. Ideally, you should be able to run Java programs obtained from any source and be confident that the programs cannot damage your system or access private information. In practice, some early implementations of Java had security holes that could be exploited by malicious programmers. However, these holes are quickly closed whenever they are found, because security is a primary requirement for all Java users.

The following sections describe fundamentals of the World Wide Web. For background information on other Internet basics, including descriptions of many acronyms that you are likely to encounter, see Internet fundamentals.

Web pages

People using the World Wide Web typically use a *web browser* program to display *web pages* obtained from other sites. Throughout this guide, we will use the following definitions:

- The *user system* is the computer where a user reads a web page. The web browser runs on the user system. The user system is also called the *client system*.
- The *server system* is the computer that actually supplies the web page. The server system has a program called a *web server* which finds web pages requested by users and delivers the contents of those pages to the user system. The server system is also called the *host system*.

URLs

Web browsers refer to web pages using *Universal Resource Locators* (URLs). Here is a typical URL:

```
http://www.sybase.com/products/powerj/index.html
```

This URL specifies the name of a computer system on the Internet (`sybase.com`) and the name of a file on that system (`products/powerj/index.html`). When you ask your web browser to open this URL, the browser contacts the given system and passes the URL to the web server running on that system. The web server then transmits the contents of the appropriate file back to your web browser, and your web browser displays the file.

URLs can do more than just specify the name of a file. For example, they may ask the web server to execute a program. In this case, the web server collects the output of that program, then sends that output to your web browser in response to the URL.

URLs may also contain various kinds of information specified by the user. For example, when you search for information using Yahoo, Alta Vista, or some other search engine, the strings that you're searching for are sent to the search engine as part of a URL. The search engine obtains these strings from the URL and performs the search you've requested.

HTML

Web pages are written in a text-based language called *HTML* (HyperText Mark-up Language). HTML has commands for simple text formatting (for example, breaking the text into paragraphs, creating section titles, putting strings of characters in special fonts, and so on). HTML also makes it easy to specify *links* to other web pages. When the user clicks on one of these links, the web browser automatically loads the web page associated with the link. This facility lets the user navigate through a sequence of web pages that may be spread over different machines on different continents.

HTML formatting directives are enclosed in angle brackets. For example, the directive for starting a new paragraph is `<P>`. Many directives come in pairs: an opening directive and a closing directive, as in

```
The following <I>word</I> is in italics.
```

The `<I>` directive tells the browser to start displaying text in italics, and the `</I>` tells the browser to return to the previous font.

Static vs. dynamic web pages

A *static* web page has fixed contents. You can picture this type of web page as a normal computer file. Whenever a user asks to see that web page, the web server sends the contents of the web page file to the user.

A *dynamic* web page is one whose contents can change. Dynamic web pages typically have a static "skeleton", plus various placeholders which are filled in at the time a user accesses the page. For example, consider a web page that offers a price list for various products. The static part of the web page might consist of a heading, product descriptions, and so on. However, the actual prices of the products are represented by placeholders which are filled in dynamically at the time the page is accessed. This ensures that the prices are always up to date.

The static "skeleton" of a dynamic web page is called a *template*. The placeholders in a template can come in various forms, but typically they specify instructions for obtaining the information needed to "fill in the blanks", plus layout instructions to indicate how to display the result.

The placeholder instructions in a template are executed by software on the server system: either the web server itself or software invoked by the web server. The placeholders are filled in with HTML code which is then transmitted to the user as part of the web page. This process is transparent to the user—by the time the user's web browser sees the page, all the placeholders have been filled in and the whole thing just looks like normal HTML.

Note: PowerJ can help you create dynamic web pages. In particular, the NetImpact Dynamo product that is included with PowerJ makes it easy to create web pages that obtain information from a database. For more information on how to do this, see A Dynamo WebApplication.

Java applets

HTML also makes it possible for a web page to execute a Java program. A Java program executed through a web browser is called a *Java applet*. Executing an applet works like this:

1. Your web browser program begins loading a web page written in HTML.
2. The HTML for the web page may contain an `<APPLET>` directive. This directive specifies the location of the Java applet you want to execute (usually a file containing virtual machine code on the same system that contains the web page).
3. The web browser loads the applet from the specified file and invokes an interpreter to execute the applet on your system.

Typically, the applet will create a form that may have push buttons, list boxes, and so on. The web browser displays this form as part of the web page that contained the `<APPLET>` directive. For example, the web page may begin with some paragraphs of text describing how to fill out the form. Then the web page contains an `<APPLET>` instruction which produces the actual form. When you look at this kind of web page with a browser, you will see the text instructions at the top of the page, followed by the form itself.

It is important to note that the user doesn't have to know anything about Java to use this kind of form. The user just clicks buttons, types text into text boxes, and so on. The user may not even realize that Java code is being executed—everything is handled transparently by the browser program.

Note: Before a Java applet can be executed on the user's system, the applet's virtual machine code must be transferred from the server system to the user system. The larger the program, the longer this transfer takes. Therefore, simple applets will load much faster on the user's system than complicated ones.

Sending information to the server

The usual purpose of a Java applet is to create a form that obtains information from the user. For example, a company that sells products over the Web may create a web page that makes it possible for customers to submit a purchase order. A Java applet invoked through this web page could create a form where customers can enter their names, addresses, what they want to buy, and so on.

When a customer has filled in the information required by the Java applet's form, the information must be sent back to the company so that it can process the purchase order. In other words, the applet program must deliver the information from the user system to the server system. This can be done in a variety of ways, all of which are described in Server extensions.

Once the server system has received information from the user, the information must be processed. The processing is done by an application on the server system. This application may be the server itself, a set of specialized routines attached to the server, or a completely separate program. Again, these different possibilities are discussed in Server extensions.

The software that processes user information often needs to send a response to the user. The response is sent as another web page. This web page may be simple (a confirmation that the information was received) or it may be complicated, invoking another applet, which produces another form to obtain more information.

Other uses of Java

Java can be used for more than just writing applets. For example, it is possible to write a Java *application*: a standalone program running on a single system, without using the Internet. This kind of application looks like an application written in Power++ or a standard programming language—it interacts with the user in the same way as other Windows programs.

Servlets

A servlet is a type of program intended to run on the server system. Servlet code is invoked by the web server in response to requests sent by the user system. For example, suppose that the user calls up a web page that invokes a Java applet. The applet displays a form and obtains information from the user, then transmits that information to the server system. The web server on the server system can process the information received by executing a Java servlet.

For more information about servlets and other types of Java applications, see Server extensions.

Note: Servlets are also known as web server extensions.

JavaScript

JavaScript is a language designed to complement Java. An application written in JavaScript is simply a sequence of text instructions that are executed one after another. If a Java program is similar to an executable file (with extension `.exe`), a JavaScript application serves a purpose similar to a DOS batch file (with extension `.bat`).

A web page may invoke JavaScript code in much the same way that it invokes a Java applet:

1. The HTML code specifies the location of the JavaScript code you want to execute.
2. The web browser loads the JavaScript code and executes it with a JavaScript interpreter.

Obtaining more information about Java

PowerJ includes an electronic copy of the book *Thinking in Java*, which you can read for an introduction to Java and object-oriented programming. The following web sites also provide general information about Java.

`http://java.sun.com:80/nav/read/Tutorial/`
The "official" Java tutorial from JavaSoft.

`http://sunsite.unc.edu/javafaq/javafaq.html`
A list of Frequently Asked Questions (FAQs) about Java.

`http://java.sun.com/faqIndex.html`
An index of many other FAQs about Java.

The Java sandbox

Sometimes you'll see people refer to the Java *sandbox* when talking about Java applets. The name comes from the idea of placing a child in a sandbox and letting them dig away without worrying that they're digging up your flower bed. In the same way, you want to restrict what an applet is allowed to do, thereby limiting the damage it can do to your system. So how does the sandbox work?

When a browser starts a Java virtual machine, the browser installs a *security manager* into the VM. A security manager is any class based on the `java.lang.SecurityManager` class. It has methods in it like **checkRead**, **checkWrite**, **checkDelete**, and so on that determine the set of low-level system operations that are acceptable to the browser. In effect, the security manager establishes a security policy.

Sprinkled throughout the standard Java class library is code that asks the security manager if an action is acceptable. For example, the code for `java.io.File.exists` looks like:

```
public boolean exists() {
    SecurityManager security = System.getSecurityManager();
    if( security != null ){
        security.checkRead( path );
    }
    // code to check for existence....
}
```

If a read operation is not allowed, **checkRead** throws a security exception, preventing the operation from taking place. All potential security risks in the core Java classes are protected in this way.

Java applications don't have these restrictions: they aren't running under a browser, so there is no security manager installed in the Java virtual machine. If the application installs security manager (by calling `System.setSecurityManager`), an exception is thrown if the application or applet tries to install a new security manager. This prevents users from getting around the security restrictions by installing a more permissive security manager.

JDK 1.1 introduces the idea of a *trusted* applet. This is an applet that is allowed to perform either some or all the "risky" operations that an application can perform. Trusted applets are digitally signed and users must explicitly tell browsers to trust a particular applet. By default, applets are still untrusted. For more about trusted applets, see Trusted applets.

Forms

A *form* is a window that can be displayed on the user's system. When you use PowerJ to build a Java project, you usually create one or more forms. To create a form, you:

- Design the form by laying out objects (buttons, boxes, and so on).
- Specify properties for each object.
- Write Java source code to handle the events that might happen to each object (for example, when the user clicks a button or types text into a text field).

Note: Do not confuse PowerJ forms with HTML forms. A PowerJ form is part of a graphic interface, used in creating various types of programs. An HTML form is typically built with textual HTML instructions, and is contained by a larger HTML file.

One PowerJ form can open another PowerJ form. For example, if the user clicks a button on one form, the associated source code may open a second form to obtain additional information from the user. There is no limit to the number of forms that may be associated with a project.

Forms in Java

Forms in Java

In PowerJ, each form is implemented as a Java *class*. This has several consequences:

1. A form is a data type, not a data object. In order to make a form appear on the user's screen, you must create an object of the form type.

   One of your form types is designated the *main form*. Your program automatically creates an object of this type when it starts up, to serve as the *initial window* that the user sees.

2. You can have multiple objects of the same form type. For example, suppose you design a form that displays the contents of a document. Your program can create multiple copies of this form, letting the user examine several documents at once.

3. Each form class has associated *properties*. Some properties affect the appearance of the form (for example, its color and its size). Other properties affect the behavior of the form (for example, whether the size of the form can be changed).

   When you design a form, you may specify initial values for the form's properties. These initial values are used whenever your program creates an object of the form class; however, you can change many of these properties later in program execution.

4. Each form class has a set of associated *methods*. A method is a function that lets you perform an action using the form. For example, a form has methods to examine or change the form's properties.

5. Using the Classes window, you can add your own methods to a form class. This is useful when you want to define a routine that can be used by other functions within the class, or when you want to provide controlled access to the class for objects outside the class.

The name given to a form is the name of the associated Java class. For example, if you name a form `Form1`, the associated Java class is also called `Form1` and its source file is named `Form1.java`.

Note: PowerJ lets you create many different types of forms. For example, a *dialog box* is a type of form which typically is opened to obtain information from the user and is closed once the user has filled in that information.

Objects

The first stage of creating a program with PowerJ is placing objects on a form. To do this, you select a component type from the component palette, then click on the form design window to specify the position of the object. In this way, you place buttons, text boxes, etc. on the form to design what the user will see when the form appears.

Object names

PowerJ associates a name with each object on the form. For example, PowerJ chooses names like `cb_1`, `lb_2`, and `Menu_1`. It is often a good idea to change these into more descriptive names: `cb_OK`, `fileList`, `menuSave`, and so on.

The name associated with an object is used as a variable name for referring to the object in Java code. For example, when you place a command button on a form, PowerJ might create the following declaration in the form class's source code:

```
Button cb_1 = new Button();
```

This indicates that `cb_1` refers to a Button object. PowerJ also generates code in the form class to initialize the Button object. For example, this code sets various properties controlling the button's size and position. The variable `cb_1` is declared as a private member of the class for the form.

Since objects on the form are normally referenced by private variables within the form class, they cannot be referenced directly by entities outside the form. If `Form1` needs to affect an object on `Form2`, `Form1` must communicate with `Form2` and have `Form2` do the actual work.

Notation: Other parts of this guide use the notation *name_N* to describe the format of default object names. The *N* stands for an integer value, with values beginning at `1`. For example, label objects are given names of the form *label_N*. This means that the first label on the form is named `label_1`, the next is `label_2`, and so on.

Object methods

Each type of object has a set of associated methods. For example, a list object has functions for adding new items to the list, deleting existing items, and so on.

```
lb_1.delItem( 3 );
```

uses the **delItem** method of a list object to delete item 3 from `lb_1`.

Some methods return a status value indicating whether they succeeded or failed. In many cases, this is a boolean value: `true` means the method succeeded and `false` means it failed. When the purpose of a method is to return a value, the function may return a special value to indicate failure. For example, if the purpose of a particular method is to return a string, the function will return a null string if the proper value cannot be determined.

Object properties

The properties of an object control the object's appearance and behavior. PowerJ lets you set the properties of an object while you are designing the form. Some properties can also be changed as your program executes. For example, you might disable the use of a command button at times when clicking that button is not appropriate.

Some properties can be changed at run time using appropriate function calls. For example,

```
cb_1.setLabel( "New text" );
```

changes the label of `cb_1` to the given string. Methods that change the value of a property have names beginning with **set**; methods that return the current value of a property have names beginning with **get**. Every property has a corresponding **get** method and every property that can be changed at run time has a corresponding **set** method.

Events

An *event* is a message received by a form or an object on a form. Events may be generated directly by the user; for example, when the user clicks on a button, an **Action** event is triggered for that button. Events may also be triggered by the code of your program; for example, if `Form1` creates a `Form2` window, the action generates an **objectCreated** event for the `Form2` object.

There may be many ways to cause the same event. For example, the **ListSelect** event means that an item has been selected in a list. This may happen because the user clicks an item, or because the user is running through the list of items by pressing the arrow keys.

Writing PowerJ programs is mainly a matter of writing Java routines to handle events. You do *not* have to write a mainline for the program; that is handled automatically by PowerJ. Instead you write a routine for what happens when the user clicks on one button, what happens when the user double-clicks an item in a list, and so on. These routines are called *event handlers*.

An event handler is a method belonging to the form class that contains the object receiving the event. This means that the handler has access to all the private members of the form object. In particular, it can work with all the objects defined on the form. For example, the **Action** event handler for `cb_1` can change the contents of a list elsewhere on the form.

A typical event handler has a name like

```
cb_1_Action
```

This incorporates the name of the object that receives the event (`cb_1`) and the name of the event itself (**Action**).

Event handler calling sequence

Event handler calling sequence

Every event handler is called with one argument: a block of information describing the event. Often, your event handlers will ignore this argument; the information is not necessary. In some cases, your event handler must place data in the information block to return a response to the caller.

Event handlers return `true` to indicate they handled the event completely or `false` if the default event handling of PowerJ should finish dealing with the event.

See Standard events for details about the way information is passed to an event handler, and how event handlers use their return values.

PowerJ and AWT

The *Abstract Windows Toolkit* (AWT) is a standard library of Java class definitions which define the common elements of graphical user interfaces: command buttons, lists, scroll bars, and so on. AWT was designed by Sun Microsystems to provide the basic building blocks for creating applets and other Java applications. AWT specifies operations that can be performed on elements (for example, adding or deleting items in a list) and events that can happen (for example, the user clicking a command button or entering text in a text field). AWT also defines other useful data classes (such as a TextField class representing a box where users can enter a line of text and an Image class for "pictures").

The *PowerJ component library* is built on top of AWT. For example, a PowerJ DBList object is based on the AWT List object, but has additional methods and properties which make it possible to use the DBList as a bound control in database applications.

In general, the PowerJ component library is a Java *wrapper* around the basic data structures and operations of AWT. The components of the library parallel the facilities of AWT itself.

However, the library provides more than a simple wrapping: it gives you a complete structure for creating programs quickly and easily. Complex operations can be performed in a single instruction, avoiding tiresome set-up and tear-down operations. At the same time, the library gives you full access to low-level AWT features, for those rare occasions when you need to program directly with AWT classes.

Learning to create programs with PowerJ is primarily a matter of learning to use the design environment and the PowerJ library.

The rest of this guide outlines the major features of the library and presents simple examples using the library functions. The goal is to introduce the most common classes and methods within the library, to help you get results fast. For full details, however, you should consult the *PowerJ Component Library Reference*.

Important: With PowerJ, you do not have to use sophisticated features of Java. The PowerJ library helps you create efficient and effective programs, even if you do not have extensive knowledge of Java or AWT. Most of the user interface code that you write will consist of simple calls to library functions.

Chapter 2. Using targets and projects

This chapter explains the basic principles of working with PowerJ targets and projects.

Targets

A *target* is a Java applet, a standalone Java application, a WebApplication, a ZIP, JAR or CAB archive, a set of Java class files, or a servlet.

PowerJ builds targets from *source files*. The following are all considered source files:

- Files containing Java source code (`.java` extension)
- Files containing HTML source code (`.html` or `.htm` extension)
- Compiled Java class files (`.class` extension)
- Java libraries
- Files that contain images that are used by an application (for example, GIF or JPEG files)
- Files that contain form definitions (`.wxf` extension).
- Managed class files (`.wxc` extension); for an explanation of managed classes, see Adding classes to a target.

Notice that the source files of a target do not necessarily contain Java source code. Files like libraries and image files may be considered source files for the target because they are used in building the target.

Target folders

Target folders

Each PowerJ target is kept in its own separate folder. This is necessary because different targets often have source files that have the same name. To keep the source files from overwriting each other, by default all the source files associated with a particular target are kept in the target folder.

By default, PowerJ creates target folders in the `Projects` folder under the main PowerJ folder. However, you can create target folders elsewhere if you wish.

The name for a target folder is the name of the target, without the filename extension. PowerJ gives you the chance to specify a different name for the target (and its folder) when you first create the target or when you use **Save Project As** to save the target.

Projects

A *project* is a collection of one or more targets. A project may consist of several programs which perform related tasks.

Some PowerJ operations apply to single targets, while others apply to the project as a whole. For example, the **Run** command runs a single target. On the other hand, the **Save Project** command saves all the source files for all the targets in the current project.

The project file

PowerJ creates a *project file* for each project. This is a text file listing all the targets that belong to the project. The project file can be regarded as a summary of the entire project.

PowerJ project files have the file extension `.WXJ`. By default, PowerJ places the project file in the same folder as the first target created for the project, but you can save the project file in a different folder if you wish.

You will never have to edit a project file directly; PowerJ maintains the file for you.

Tip: If you double-click on a project file, the system will open the project in PowerJ.

New projects

When you begin using PowerJ, PowerJ automatically sets up a new project. This is called an *untitled project* since you haven't assigned a name to it yet.

You can work with the untitled project in the same way as any other project: you can design forms, set properties, write code, and even run the resulting program. In order to do some of this work, however, PowerJ has to create a number of files and store these files on your system. Therefore, PowerJ stores the untitled project under the *temporary folder* that you specified when you installed the PowerJ package.

At any time, you can save the untitled project with a name of your choosing. Many people find it convenient to do this before doing any work on the project. When you save an untitled project (or move an existing project to a new folder), PowerJ must rebuild the files associated with the project. Therefore, if you save an untitled project before you start doing any work on it, you can avoid building some files twice.

Once you have named and saved the project, PowerJ deletes any files that were created for the project in the temporary folder.

Starting a new project

Starting a new project

As the previous section described, PowerJ automatically starts a new project for you when you start your PowerJ session. You can also start a new project in the middle of your PowerJ session.

- **To start a new project in the middle of your PowerJ session:**

1. From the **File** menu of the main PowerJ menu bar, click **New Project**.

The new project will automatically be an untitled project.

Opening an existing project

When you start your PowerJ session, you will often want to start work on a project that you saved in an earlier session. To do this, you must *open* the project.

- **To open an existing project:**

1. From the **File** menu of the main PowerJ menu bar, click **Open Project**. By default, PowerJ shows the last folder where you opened a project.
2. Locate the folder with the project file. By default, this is the folder of the first target in the project.
3. Click the name of the project file in this folder (with the extension `.wxj`), then click **Open**.

Note: You can only work on one project at a time in a single PowerJ session. If you are working on one project and then open a different one, PowerJ closes the first project before opening the next one. If you want to work on more than one project at a time, you need to start a separate PowerJ session for each project.

Loading projects saved with a previous version of PowerJ

Loading projects saved with a previous version of PowerJ

This version of PowerJ can load projects created with earlier versions of the product (for example, beta releases). You don't have to do anything special for the conversion to take place; just load your project with PowerJ, then save it.

If you are loading an old project that you had checked into a source control system, you should check out all of the project files before loading the project in PowerJ. This makes it possible for PowerJ to convert all files from old save formats (if necessary).

Closing a project

When you terminate your PowerJ session, PowerJ automatically closes the project before quitting. Similarly, if you start a new project or open an existing project, PowerJ automatically closes the project that you previously had open.

- **To close a project explicitly:**

1. From the **File** menu of the main PowerJ menu bar, click **Close Project**.

When PowerJ closes a project, it first checks to see if the project has unsaved changes. If so, PowerJ asks whether you want to save the changes before closing. If you do not save the changes, they are discarded.

Running a target

You can run a target program at any time during your PowerJ session.

- **To run your target:**

1. From the **Run** menu, click **Run**.
2. You can also click the **Run** button on the PowerJ toolbar:

Before your program starts executing

When you run a target, PowerJ first *builds* the target. For a Java applet or application, PowerJ compiles your Java source code into Java class files. Compilation can be a lengthy process if you have a lot of code; however, PowerJ tries to compile most of your code in the background, as you work on other parts of the project. This reduces the delay when you finally ask to run the target. Even so, it may take a little while before your program actually starts execution.

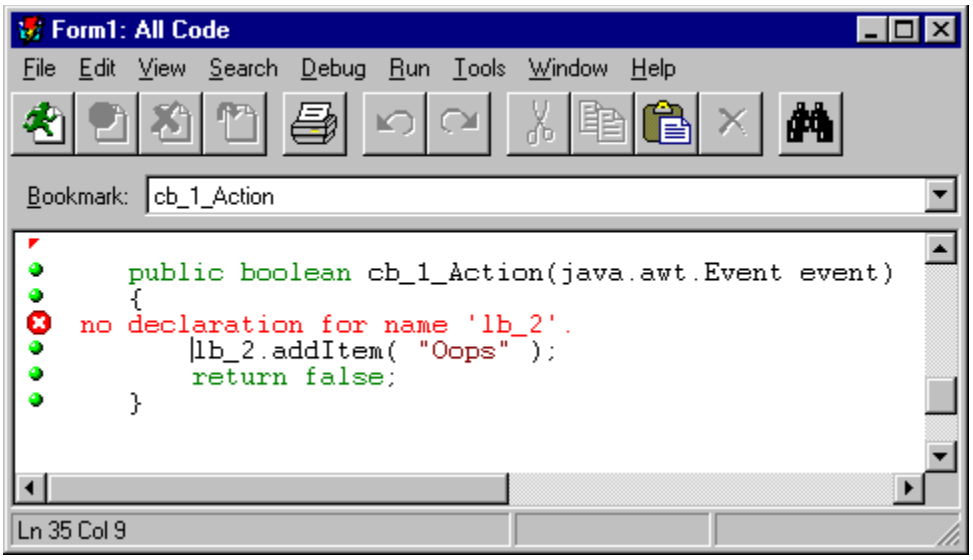
For Java targets, the amount of time required to build the target is affected by the build strategy that you have chosen. For more information, see Build strategy.

While building your target, PowerJ displays a dialog box with a progress bar showing how far PowerJ has got in preparing your program. If you click the minimize button on this dialog box, it minimizes your entire PowerJ session. This can be useful if you want to do something else while PowerJ is building your target.

Program execution starts when you see the target's initial form displayed on your screen. You can then interact with your program as a user would.

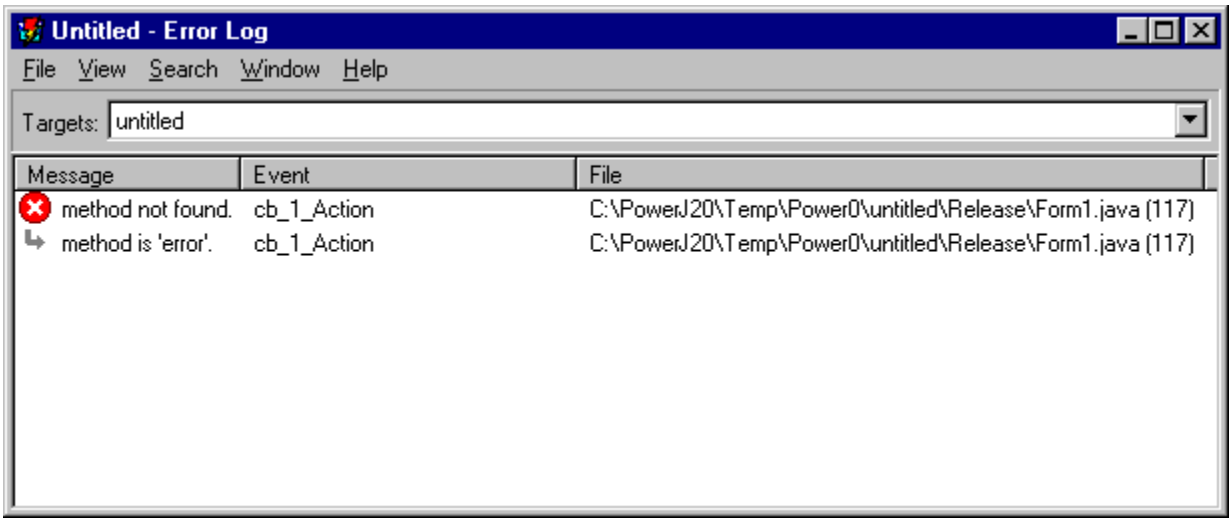
Compilation errors

If PowerJ encounters errors compiling your target, it opens a code editor window showing the code where the first error was found. One or more error messages will be displayed immediately before the line where the error occurred. The error messages are marked with a white X in a red circle on the left margin.



You can jump from one error to the next using the **Search** menu of the code editor. Clicking **Next Error** on this menu moves to the next error found by the compiler. Clicking **Previous Error** on this menu moves back to the previous error.

PowerJ can also display the *Error Log* window showing the errors. This window describes each error, plus the file and event handler that contained the error. If a particular error message is too long to fit in the window, resize the window so that you can see the whole message.



The icons indicate the severity of the error; an exclamation mark indicates an error that prevented completion of the building of the target, and an "i" indicates a warning of possible problems that do not prevent building. An error may be followed by notes that further explain the problem. Notes are identified

by a bent arrow icon.

JLT files

When PowerJ builds a Java target, it creates a special file with the file extension `.JLT`. The file is stored in a subfolder of the target folder.

The purpose of the JLT file is to preserve a time stamp for when the target was last built. A JLT file is just a "housekeeping" file for PowerJ. You will never have a reason to work with it directly, but will see it appear in various contexts. For example, when PowerJ is building a target, the progress bar frequently reports that PowerJ is building the JLT file.

Running different types of targets

PowerJ runs different types of targets in different ways. For example, if a target is an applet, the default behavior is to run that applet using the Microsoft Applet Viewer program (`jview`). Applet Viewer can run a compiled applet, without requiring the applet to be invoked by an `<applet>` command in a web page. On the other hand, if the target is a standalone Java application, the default behavior is to open a Java console, then run the program under control of that console.

You can change the default behavior for running a program by specifying appropriate *run options*. For example, you can specify your own command line for invoking the program. Different types of targets allow different types of run options. For further information, see Run options and Target types.

Build strategy

PowerJ can build targets using two different strategies:

- The **Build all files** strategy rebuilds all the class files associated with the target when any source file has been changed since the last build.
- The **Build modified source files only** rebuilds class files whose source files have changed since the class file was last built.

Build modified source files only can reduce the amount of time required to build the target. However, there is a chance that you will introduce error conditions that are not detected by the build operation. For example, suppose that you edit the definition of class `A` to remove the definition of a particular function. Also suppose that class `B` uses that function. When you rebuild the target, PowerJ rebuilds the class file for class `A` (since it has recently changed), but doesn't look at class `B`. Therefore, PowerJ doesn't notice that class `B` tries to use a function that no longer exists. This can lead to run-time errors.

If you choose **Build modified source files only** as your build strategy, you should occasionally do a full build of the target to detect errors of the kind discussed above. You can do this by clicking **Rebuild All** in the PowerJ **Run** menu.

Important: Whatever build strategy you choose, you should always use **Rebuild All** to build the final version of your target (when you are ready to deploy the program).

You set the build strategy for a target through the properties of that target. For more information on setting target properties, see Options for building targets and Properties for Java applet targets.

Debugging facilities

PowerJ offers a number of facilities for debugging your programs. These facilities let you examine the program throughout the course of execution, making it easier to investigate what happens as the program runs.

For a complete description of PowerJ debugging facilities, see Debugging.

The Java console

PowerJ can open a console which displays information about running programs.

- **To open the PowerJ console:**

1. On the **Tools** menu of the main PowerJ menu bar, click **Show Console** so that it is marked with a check.

The console window displays messages when Java programs are running. It displays information written to the standard output and allows you to type information to be read on the standard input.

Running with multiple targets

Even if your project contains more than one target, PowerJ only runs one target at a time. If you run a project with several targets that could be run, PowerJ asks which target you want to run. Some targets, such as classes and archive files, can not be run.

Run options

The run options for a target control the way that PowerJ runs the target. By specifying run options, you can override the default method of running the target.

- **To specify run options for a target:**

1. From the **Run** menu of the main PowerJ menu bar, click **Run Options**.
2. If you have more than one target in this project, PowerJ displays a list of the targets. Click the name of the target whose run options you want to set, then click **OK**.
3. Set the run options as desired, then click **OK**.

Different types of targets have different types of run options. For more information, see the individual target descriptions in Target types.

Batch files for running targets

Whenever you run a target, PowerJ creates a batch file (`.BAT`) which gives a command line suitable for running the target outside of PowerJ. This batch file has the same base name as the target. For example, if the target's name is `MyTarget`, the batch file would be called `MyTarget.bat`.

Note: If you are running in Debug mode, the batch file is stored in the target's `Debug` folder. If you are running in Release mode, the batch file is stored in the target's `Release` folder. For more information on these modes, see Debug and release versions of a target.

The command lines in the batch file contain the full `CLASSPATH` settings that PowerJ uses for running the program. This is useful when you want to check exactly what `CLASSPATH` PowerJ is using for the run. It is also useful when you are getting ready to deploy the program, since it tells you what `CLASSPATH` may be needed when users want to run the program on their own systems. For more on `CLASSPATH`, see CLASSPATH and CODEBASE and How PowerJ uses the CLASSPATH environment variable.

When you run an applet, the batch file that is created contains two command lines:

- A command line for running the applet as an applet (either in a web browser or with the applet viewer, depending on the run options that you've set).
- A command line for running the applet as a standalone application. In the batch file, this command line is commented out; however, it shows you what command line would be required if you wanted to run the program standalone. To successfully run an applet as an application, the applet cannot depend on methods in the java.applet.Applet and related applet helper classes.

PowerJ gives absolute pathnames for all the files and folders used in the batch file. This means that you can copy the batch file to any other location on your system and run the program just by invoking the batch file.

Target folder contents

You will seldom need to know how PowerJ organizes your project into files and folders; PowerJ is designed so you can ignore the underlying mechanics. This section is supplied for the rare occasions when you must refer to source or *intermediate files* directly.

Intermediate files are files that may be produced when a target is built (in addition to the target itself). For example, PowerJ may compile Java source code files to produce class files (with the extension `.class`) while building a Java archive (JAR) target.

PowerJ creates a *target file* for each target in a project. Target files have the file extension `.WXT`. This is a text file in the target folder listing information about the target, including all the files that are needed to build the target. The target file can be regarded as a summary of the entire target.

Suppose that you have saved an applet target called `target`. The following list shows the files that you may see in the target folder. (Some types of targets do not have all the files shown in the list.)

`target.html`
: An HTML file which can form the basis for a web page containing the applet. This file contains the minimal HTML for setting up the applet, including an appropriate `<applet>` directive for invoking the applet. You can add extra HTML code to this file to make a more suitable web page; for example, you might add instructions for using the applet form before the `<applet>` directive that actually displays the form.

  You can edit this file using any appropriate software on your system. This can be anything from a simple text editor to an HTML authoring system like Microsoft Front Page.

`target.wxj`
: The project file.

`target.wxt`
: The target file containing information about the target. This information controls the steps that PowerJ takes when it builds an executable file for the target.

`target.wxu`
: Contains *user settings* for the project. It holds information on the state of PowerJ when you last edited this project (for example, the size of the main PowerJ window, the view windows you had open, the size of any debugger windows, and so on). This lets PowerJ restore all those windows to the same positions, the next time you start working on the project.

  Since the WXU file contains your personal preferences for the project, and may have absolute file paths that only make sense on your computer, you should not share this file with others working on the project.

`*.wxc`
: *Managed class files* containing a complete description of *managed classes* in your program. The description stored in a class file includes all the Java source code related to the class.

`*.wxf`
: *Form files* containing a complete description of each form in your program; for example, you may see `form1.wxf`, `form2.wxf`, and so on. The description stored in a form file includes the properties of the form and its objects, the position of each object on the form, and all the Java source code related to the form.

`Debug`
: A folder used when you create a debugging version of your target. The folder may contain compiled Java class files (`.class` extension) and other files created in the process of building your program.

`Release`

A folder containing files for a version of the target suitable for release to end users. Debug and release versions of a target explains how to prepare this version.

The `Release` folder also holds a number of intermediate files generated in the course of building any version of the target, whether the debug version or the end-release. This includes Java source code files generated from the WXF and WXC files for every form in the target. Don't edit these files, since any changes will be lost when they are regenerated.

If you want to save disk space, you may delete all the contents of the `Debug` and the `Release` folders by clicking **Clean** in the **Run** menu. These files are all built from other source files, so they can be rebuilt if you delete them. (Of course, you then have to wait for PowerJ to rebuild all these files the next time you run the target.)

The source files necessary for building a target are all stored directly in the target folder (except for any outside libraries that are used by these source files). Therefore, if you want to archive a target in such a way that it can be completely restored later, you only have to keep the files that are directly under the target folder.

Changing names

The name of a WXF form file matches the name of the form at the time that the file was first saved. For example, if you save the project when the form's name is `Form1`, the name of the file will be `Form1.wxf`. However, if you change the name of the form later, the name of the associated form file does not change automatically. If you want to change the name of the form file as well as the form itself, follow these steps:

1. Change the name of the form first.
2. Open the Classes window and use the right mouse button to click on the form whose form file you want to rename.
3. Click **Properties** in the resulting menu.
4. Enter a new name for the form file under **File Name**, then click **OK**.

The same principle applies to changing the name of managed classes. If you change the name of the class, the name of the WXC file for the class does not change unless you change it manually.

Important: When you use the right mouse button to click on an object in the Classes window, the resulting menu has a **Rename** item. Using this item renames the object in the Classes window, and renames the associated files as well. If you rename a class or function that is also displayed in an editor, you will need to close and reopen the editor to see the name change.

Backup files

Backup files

PowerJ creates backup files in various situations. For example, suppose you run a target but have made changes since the last time you saved the project. PowerJ overwrites your saved files with the new (modified) contents of the target, but keeps backup files of the original versions to use if you close the project without saving changes.

Backup files are identified by having a tilde (~) character in the middle of the file name extension. For example, the backup file for `Form1.wxf` is named `Form1.w~f`.

Managing projects and targets

PowerJ can do more than just prepare single programs; it can help you create a complete software application containing multiple targets. This section explains how to add targets to projects and how to add files to targets.

PowerJ lets you add new or existing source files to a target. Source files may be of several types, including the following:

- Java source code files.
- Java class files.
- Files containing images (for example, GIF files).
- Managed class definitions (either new ones or ones created for other targets).
- Forms (either new ones or ones created for other targets).
- Text files (for example, HTML files).
- Files containing CORBA interface definitions (IDL files).
- Existing targets.

You can also delete targets and source files from projects and targets.

Adding a target to a project

There are two types of targets that can be added to a project:

- A new target.
- An existing target (obtained from another PowerJ project).

◆ **To add a new target to a project:**

1. From the **File** menu of the main PowerJ menu bar, point to **New** and click **Target**. This opens the *Target Wizard* which takes you step-by-step through the creation of the new target.
2. The Target Wizard displays a list of targets that PowerJ can create. This includes several types of applets and applications. Click the type of target you want to create, then click **Next**.
3. If you are adding either a Java - Applet or a Java - Application target, the Target Wizard asks you if you wish to use the PowerJ framework. The framework allows you to create forms and use the drag and drop programming features. For more information, see Using the PowerJ framework. Select if you'd like to use the framework or not, then click **Next**.
4. If you are adding either a Java - Applet or a Java - Application or Java - Classes target, the Target Wizard asks you which class library you'd like to use. Select the preferred library, then click **Next**.
5. If you are creating a Java CAB target, the Target Wizard asks which kind of CABinet you'd like to create. Select either Applet or Class Library then click **Next**.Under **Drives**, choose a drive for the target and under **Folder** choose the folder where you want to save the target file. Under **Target name**, type in a name for the new target. The target file will be saved and a target folder will be created with the same name (without extension) in the folder specified by **Drives** and **Folder**. A target name must begin with a letter and cannot contain spaces. You may want to save targets under the PowerJ `Projects` folder. Click **Next** to proceed.
6. Depending on the type of target you have chosen, the Target Wizard may ask you to choose what type of form should be used for the new target's initial form. You will also be asked to type in a name for this form as well as a name for the file where the form is saved.
7. If you are creating a target with database capabilities, the Target Wizard requires further information in order to create the target. See Chapter 8. A simple database application for information about creating database applications with the Target/Form Wizard.

When you have finished entering this information, PowerJ creates the new target and all the necessary source files.

Note: Whenever you start your PowerJ session, PowerJ creates a default target named `untitled`. If you create a new target, and the existing target is an unmodified version of the default target, PowerJ automatically deletes the original (default) target.

◆ **To add an existing target to a project:**

1. From the **File** menu of the Files window, click **Add File**.
2. PowerJ starts an Open File dialog. Use this dialog to locate the WXT file for the target you want to add. Click **Open**.
3. You will be asked if you want the current target to depend on the new target. Click **Yes** if you want the current target to be rebuilt any time the new target changes; click **No** if changes in the new target will not affect the current target.

Using the PowerJ framework

With some types of targets, you will be asked if you want to use the PowerJ framework when you create the target. The PowerJ framework consists of generated code which PowerJ uses to handle the startup and shutdown of your program. This code creates and displays the main form, and terminates the application when you close the main form.

The primary purpose of the framework is to allow PowerJ to take care of the coding details, and to remove the need to write startup code. Under most circumstances, you should use the PowerJ framework when creating a target. See below for more information on when to use non-framework targets.

Applications and the PowerJ framework

When you create an application target with the PowerJ framework, PowerJ automatically generates the following:

- A managed form class which extends `java.awt.Frame`, and is set as the main form.
- A managed class (with the same name as your target) which contains the **main** method. The **main** method is read-only code, but the PowerJ framework provides three methods which you can use as entry points: **StartApp**, **RunApp**, and **EndApp**.

Applets and the PowerJ framework

Similarly, when you create an applet target with the PowerJ framework, PowerJ automatically generates the following:

- A managed form class which extends `java.applet.Applet` and is set as the main form.
- A managed class (named the same as your target) which extends the main form class and contains the applet **init** method. PowerJ also generates a **main** method which allows you to run your applet as an application, if desired.
- A managed class (named `_targetname_frame`) which, by default, extends `java.awt.Frame`. This is the frame which the **main** method uses to create the applet.
- Sample HTML code to load the applet into the applet viewer.

You can see these classes in the **Classes** window by clicking **Classes** on the **View** menu.

Using non-framework targets

When you create a non-framework target, PowerJ makes no assumptions about your target, and lets you handle all of the details. Under some circumstances, this is desirable. For example, you should not use a PowerJ framework target when you:

- Import Java files as managed classes and have no need for PowerJ design time form capabilities. See Importing from other Java environments for more information on importing source files into PowerJ.
- Want to write your own startup code, and use your own class hierarchy.
- Write non-visual (command line) applications or applets. See Using Java tutorials for more information.

Adding source files to a target

You can add files using the Files window, the Targets window or the **File** menu on the main PowerJ menu bar. This section describes procedures for the following types of source files:

- Java source code files.
- Java class files.
- Files containing images (for example, GIF files).

Note: You should use the **Import Java Files** option under the **Tools** menu to add existing source files (`.java` files) to a target. It is better to use the import tool than the **Add File** command, because the importer adds the code as forms or as managed classes, rather than `.java` files. This provides you with more powerful editing capabilities. For more information on importing sources files, see Importing from other Java environments.

You can use the following procedure to add new Java source code and include files.

◆ **To add a new code file to a target:**

1. In the Files window, click the target to which you want to add the source file.
2. From the **File** menu of the Files window, point to **New** and click **File**.
3. PowerJ prompts you to type in a name for the new file. This should not be the name of an existing file, because PowerJ will overwrite the file if it already exists. PowerJ uses the extension of the file name that you type to determine the file type.
4. Click **OK** when you have typed the file name.

PowerJ adds the file to the target and opens a code editor for it.

◆ **To add an existing file to a target:**

1. In the Files window, click the target to which you want to add the source file.
2. From the **File** menu of the Files window, click **Add File**. PowerJ opens a file dialog window that lets you specify the file you want to add.
3. At the bottom of the file dialog window, click the arrow for **Files of type**. From the resulting list, select the type of file you want to add.
4. Use the file dialog to find the file you want to add. Click **Open** when you have specified the file.

PowerJ adds the file to the target. If you add a class file to a collection target, the file will be added to the set of class files used to create the target. If you add a PowerJ form file, the form and all its associated code will be added to the current target. PowerJ determines the type of file by looking at the file extension; for example, if the file has the extension `.WXF`, the file is assumed to be a form file.

With target files, PowerJ adds the target to the current project. For more information, see Adding a target to a project.

With all types of files, PowerJ offers you the choice of working from the original file, or making a copy and working from the copy. If you make a copy, the copy is stored in the target folder. When a target uses a copy of a file, any later changes to the original file will *not* be inherited by the copy.

Adding classes to a target

You can add a new class to a target by using the **File** menu of the main PowerJ menu bar, or the Classes, Files, Objects and Targets windows, to open the *Class Wizard*. The Class Wizard guides you through the steps of adding a class to a target.

Classes created through the Class Wizard are called *managed classes* because PowerJ has records of what function(s) the classes contain. Managed classes are stored in files with the extension `.WXC`.

The Class Wizard lets you define the following types of managed classes:

Visual class
A class that is seen at design time but not at run time. For further information, see Visual classes.

List Resource
A skeleton class derived from the JDK 1.1 ListResourceBundle class for creating resource tables. If you create such a class, PowerJ opens a code editor window where you can enter code for the class. This code editor window contains comments which provide further information about creating a List Resource class. For more information, see Resource bundles and localization.

BeanInfo
A skeleton class for providing information on a JavaBeans component. For more information on creating a BeanInfo class, see BeanInfo classes.

Standard Java
Any other type of class.

Note: For **List Resource** and **BeanInfo**, the Class Wizard only lets you specify the class name. You can still specify the package name and other properties after the Class Wizard completes by using the property sheet for the class. You can open the property sheet for a class in the Classes Window.

- **To define a new managed class:**

1. From the **File** menu of the Classes window, click **New** and then **Class**. This opens the Class Wizard.
2. If your project contains more than one target, the Class Wizard asks which target should contain the new class. Click a target, then click **Next**.
3. Click the type of class you want to create, then click **Next**.
4. If the Class Wizard asks for a **Package Name**, type a name for the Java package that will contain this class. You can also leave the name blank, in which case the class will be added to the same package as the target. If the Class Wizard does not prompt you for a class name, or you later wish to change it, you can use the property sheet for the class in the Classes window.
5. Under **Class name**, type a name for the new class.
6. If the Class Wizard offers an **Inherits from** field, type the name of a class that the new class will be based on. (The default is Object.)
7. If the Class Wizard offers an **Implements** field, and if this class implements any interfaces, type the names of the interfaces separated by spaces or commas.
8. Specify any other information allowed by the Class Wizard, then click **Finish**.

Once you have followed the above steps, you can use the Classes window to further define the class. Click on the name of the new class in the left pane of the Classes window, then define the contents of the class by double-clicking items in the right part of the window. See Adding new member functions for

instructions on adding member functions.

PowerJ automatically creates a constructor and destructor for any managed class; however, you can delete either of these if it is not needed.

To add a class that is defined in an existing file, add the file itself to the target (see Adding source files to a target).

You can rename a managed class from the Classes window. Use the right button to click the class name, then click **Rename**. You can then enter a new name for the class. When you change a class name, PowerJ automatically changes all occurrences of the old name to the new name (except for occurrences in string constants and comments).

For more information on using classes, see Working with classes.

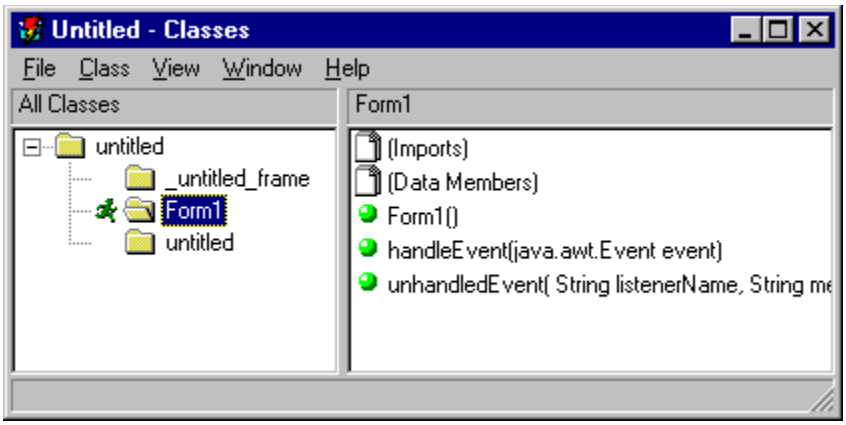
Adding forms to a target

If you are adding a new form to a target, the *Form Wizard* helps you to create the new form. For further information, see Defining forms.

To add a form that is defined in an existing file, add the form's WXF file to the target. The steps for adding an existing file to a target are outlined in Adding source files to a target.

Setting the main form

If a target has forms associated with it, one of the forms is designated the *main form*. The main form is created automatically when the target begins running; it is the first form that the user sees. In the Classes window, the main form is marked with a green running figure:



In the above example, `Form1` is the main form.

- **To designate a different form as the main form:**

1. In the Classes window, use the right mouse button to click on the name of the form that you want to designate as the main form, then click **Main Form**. The green running figure appears beside the new main form.

Targets that depend on each other

A project may contain one target that depends on another. For example, the project may contain an applet and a JAR file that contains its classes. In this case, you have to tell PowerJ about the dependency, so that PowerJ will remake the JAR file whenever the applet changes.

- **To establish that target A depends on target B:**

1. Create each target separately.
2. Save both targets.
3. In the Targets window, click target A so that it is selected.
4. From the **File** menu of the Targets window, click **Add Target Dependency**. This displays a list of targets on which A may depend.
5. Click target B in the Available Targets list, then click **Add**.
6. Click **OK**.

From this point onward, if you make a change in target B, PowerJ will update both B and A.

- **To remove an existing target dependency:**

1. In the Targets window, click the target from which you want to remove the dependency.
2. From the **File** menu of the Targets window, click **Add Target Dependency**.
3. In the Selected Targets list, click the name of the target whose dependency you want to remove, then click **Delete**.
4. Click **OK**.

Target dependencies are recorded in the project file (`.WXJ` extension), not in the target file (`.WXT` extension). Therefore, if you add an existing target to a new project, the target dependencies of the target are not recorded in the new project. If you want the same target dependencies in the new project, you must define them again.

If the same target dependency exists in more than one project, when you remove that dependency in one project it will also be removed from the other projects when you next open them.

Deleting targets and source files

You can delete targets and source files from a project, using the Files window. Deleting a target from a project does *not* remove the file from the disk; it simply tells PowerJ that the file is no longer needed by this project. Deleting any other type of file *does* delete the file from disk.

- **To delete a file from the current project:**

1. In the Files window, use the right mouse button to click the name of the file you want to delete, then click **Delete**. Before deleting the file, PowerJ verifies that you really want to go ahead with the deletion.

Note: The same source file may be used as input for more than one target. For example, several targets may use the same ZIP file as a source file. If you delete a file from the source file list of one target, it has no effect on the source file lists of other targets.

You can also delete forms and managed classes from the Classes window, and delete files from the Targets window. The process is the same as deleting items from the Files window.

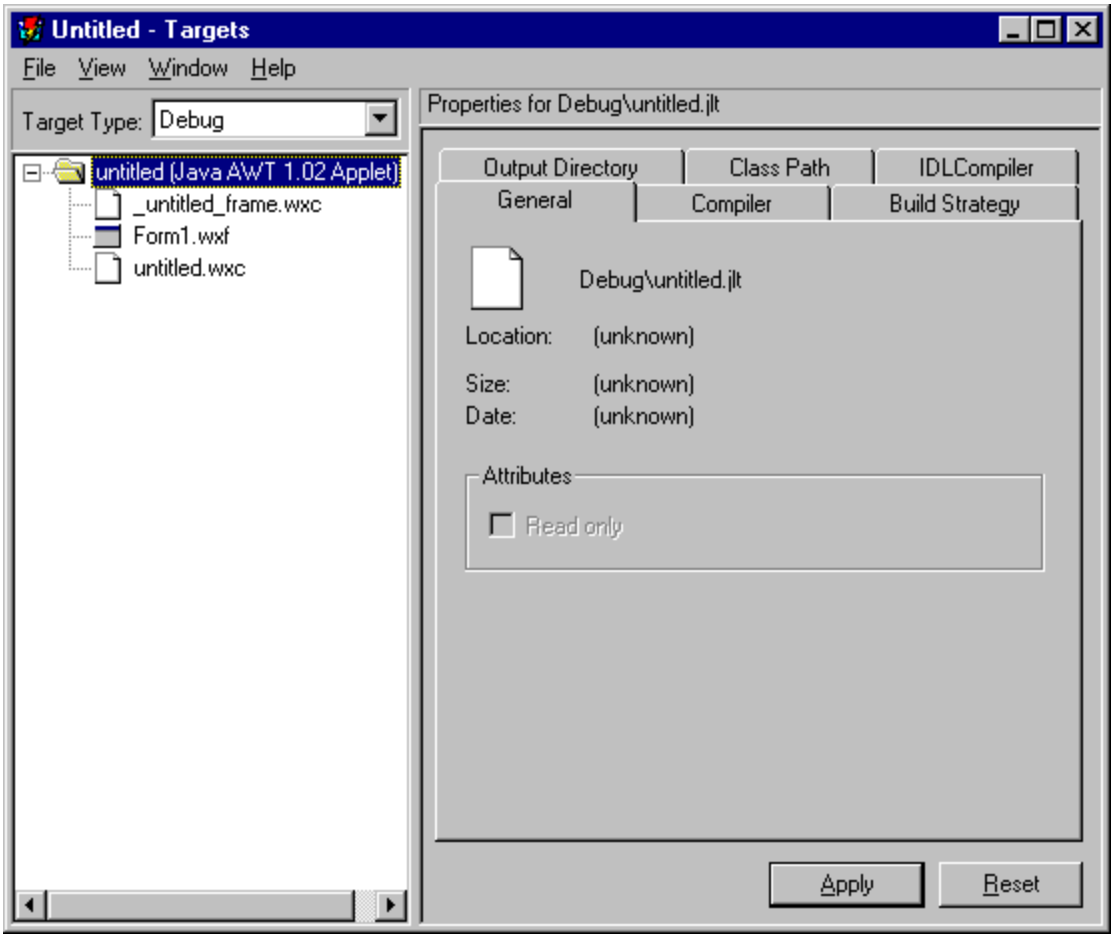
Read only folders

The concept of read only folders is only supported for compatibility with Power++. They have no effect in PowerJ.

Options for building targets

The PowerJ Targets window provides access to the facilities that control how targets are built from their source files.

In the Targets window, you can use the right mouse button to click any of the file names displayed and obtain a *property sheet* for the file. You may also click **Show Property Sheet** on the Targets window's **View** menu to display property sheets as part of the Targets window itself:



Target properties
Source file properties
Debug and release versions of a target

Target properties

For a target, the property sheet controls the way in which PowerJ attempts to build the target. This includes:

- Any specific options for building the target
- What folders will be searched for class files needed to build the target

For information on any of the options on the property sheet, use the online help facilities.

Source file properties

For a source file, the property sheet controls the actions that PowerJ takes when processing the file. This includes any specific options for processing the source file.

Note: For information on any of the options on the property sheet, use the online help facilities.

Debug and release versions of a target

When you are debugging a target, PowerJ prepares a test version of the target under the `Debug` folder in the target folder. Debug versions of a target are larger and slower than release versions, since debugging information is stored in them and compiler optimizations are turned off. When you are ready to create a version of the target for end users, you should build a *release version*.

Release versions do not contain any of the debugging information that is provided with debug versions of the program. They are also optimized in various ways, making the executable file smaller and suitable for distribution to end users.

- **To build a release version of your target:**

1. From the **View** menu of the main PowerJ menu bar, click **Targets**. This displays the Targets window.
2. Click the **Target Type** drop down list, then click **Release**.

From this point on, PowerJ prepares release versions of your target. These are placed in the `Release` folder in the target folder.

If you later need to do more debugging on the target, you can use the Targets window to change back to the debug target type.

For more information on deploying an application, see Collecting and deploying.

Build options for Debug and Release versions

The Debug and Release versions of a target may have different build options. Therefore, if you change the build options for one version of the target, you usually have to change the build options for the other version too. To do this, you set the build options, change the target type, then set the build options again. This applies to build options for source files as well as targets.

Clean builds

PowerJ makes it easy to get a clean build of all the targets in the current project.

- **To build all the targets in the project:**

1. From the **Run** menu of the main PowerJ menu bar, click **Rebuild All**.

This operation first cleans out all intermediate files previously used in building or running targets, then rebuilds all the targets as required.

Target types

This section looks at types of targets that can be created with PowerJ. In particular, it looks at the properties that control the nature of each type of target. To examine and change these properties, open the Targets window and click **Show Property Sheet** in the Targets window **View** menu.

The types of targets include:

- Java applet
- Java application
- Java classes (class library)
- WebApplication
- Java JAR (archive) file
- Java CAB file
- Java ZIP file
- Java Dynamo Server application (servlet)
- Java WWW server application (Web servlet or server extension)
- User-defined targets (using target templates)

The following sections describe the run options available for running each type of target and the properties available to control how these targets are built.

For a description of how to set the run options for a target, see Run options. For a description of how to set properties for a target, see Options for building targets.

Java applets

A Java applet target initially has the following source files:

- A basic HTML file which contains an appropriate `<applet>` directive to invoke the applet.
- A managed class file (`.WXC` extension) defining global information for the applet; this class has a name of the form `_appletName_frame`.
- A managed class file (`.WXC` extension) for the applet object itself.
- A form file (`.WXF` extension) for the applet's initial form.

Important: When you create an applet target, you must choose whether the applet will be based on JDK 1.02 or JDK 1.1. For more information, see JDK 1.02 vs. JDK 1.1.

When you run a Java applet, the default is to invoke the Applet Viewer program to execute the applet. Another way to execute the applet is to open the target's HTML file with a web browser (for example, Internet Explorer or Netscape Navigator). This lets you see how these browsers will handle the applet.

In theory, an applet should have the same behavior no matter which browser you use to run it. In practice, however, there are small differences between browsers which may lead to different applet behaviors.

Note: PowerJ generates code that allows you to launch your applet as an application if it does not depend on methods in the java.applet.Applet and related applet helper classes. For more information on how to run an applet as an application, see Batch files for running targets.

Run options for Java applets

The following run options apply to Java applets:

Use Microsoft's Java interpreter [**General** page]
Run the applet under control of the Applet Viewer program, using the Microsoft implementation of the Java virtual machine. This is the default.

Use Sun's Java interpreter [**General** page]
Run the applet under control of the Applet Viewer program, using the JavaSoft implementation of the Java virtual machine.

Use a web browser [**General** page]
Runs the applet under the control of a specified web browser. In this case, you must specify how to invoke the web browser. Click the **Configure** button. This opens a dialog box where you specify the web browser to be used.

Initial URL [**General** page]
The URL that the web browser or applet viewer should open in order to execute the applet. Typically, you will use the HTML file that PowerJ creates when it creates the applet target.

Codebase [**General** page]
Specifies a folder to be used in the `CODEBASE` parameter when the applet is invoked. For more information on `CODEBASE`, see CLASSPATH and CODEBASE.

Specify name=value pairs on separate lines for applet parameters [**General** page]
This lets you specify named parameter values that will be passed to the applet when it is run. For example, if your applet expects parameters called `firstName` and `lastName`, you could type in the following:

```
firstName=Jane
lastName=Doe
```

This option is only available if your applet uses the PowerJ framework.

Don't use the debugger [**Debug** page]
Prevents the use of the debugger while running the applet.

Run with the debugger [**Debug** page]
Runs the program under control of the PowerJ debugger.

Class for debugging [**Debug** page]
Specifies a particular class that you want to debug. If you leave this blank, PowerJ begins debugging with the main form for the applet.

Select an initial breakpoint [**Debug** page]
Specifies an initial breakpoint to be used when debugging the applet under control of the PowerJ debugger. For more information, see Breakpoints.

Properties for Java applet targets

Properties for Java applet targets are specified on the property sheet available through the Targets window. The properties include:

Which Java compiler do you want to use? [**Compiler** page]
Current choices are PowerJ's built-in Java compiler, the Sun Java compiler, or the Microsoft Java compiler.

Class Library [**Compiler** page]
You may choose which Java class library you wish to use. Current choices are the Microsoft library or the Sun library.

Output Directory [**Output Directory** page]
You may choose where the application stores its class files: in the target folder, in the project folder (the folder containing the `WXJ` file), or in some other folder.

Class Path [**Class Path** page]
This is a list of folders where the running applet should look for any class files that it needs; this is used to create a `CLASSPATH` environment variable whose value is the list of folders. For more information, see How PowerJ uses the CLASSPATH environment variable.

The `--- Java Class Library ---` entry in the list stands for the location of the JDK classes that you specify in the **Class Library** property.

The arrow buttons on the **Class Path** page let you change the order of the folders. Click a folder name, then use the arrow buttons to change the position of the folder in the list.

The names of folders are typically specified using *build macros*. For more information about build macros, see Build macros.

IDL Compiler [**IDL Compiler** page]
IDL (Interface Definition Language) is used in conjunction with CORBA targets. IDL specifies an interface to an object, independent of the object's actual implementation. For example, the same IDL interface description might be suitable for a Java implementation or a C++ implementation. An IDL interface description can be compiled with an IDL compiler and then linked with an object implementation. The **IDL Compiler** properties let you choose which compiler is used to compile IDL (if your project includes IDL source files). For more information, see Distributed computing.

Global Package Name [**IDL Compiler** page]
Lets you override the default global package name if no module is explicitly specified.

Build Strategy [**Build Strategy** page]
Specifies which files should be rebuilt when any source file changes. The options are to build all

files associated with a target or only the ones that have changed since the last build. The default is **Build all files**. For more information, see Build strategy.

Java applications

A Java application is simply a standalone program written in Java. A Java application target initially has the following source files:

- A managed class file (`.WXC` extension) defining global information for the application.
- A form file (`.WXF` extension) for the applet's initial form.

When you run a Java application, the default is to open a Java console and then run the application. The console reports on certain actions of the Java application; for example, if the application attempts to open a URL, the request is displayed on the console.

Run options for Java applications

The following run options apply to Java applications:

Use Microsoft's Java interpreter [**General** page]
Run the application using a Java console and the Microsoft implementation of the Java virtual machine. This is the default. To specify command-line parameters, use the **Run with the following parameters** option.

Use Sun's Java interpreter [**General** page]
Run the application using a Java console and the JavaSoft implementation of the Java virtual machine. To specify command-line parameters, use the **Run with the following parameters** option.

Use the following command line [**General** page]
Run the application by executing the given command line. The default command line invokes the program `java.exe`. To specify parameters, you can either type them as part of the command line here or use the **Run with the following parameters** option.

Run with the following parameters [**General** page]
This lets you specify the parameter part of the command line that will be passed to your application when it is run.

Add the following files/folders to the classpath when executing [**General** page]
Specifies a number of files and/or folders to be added to the `CLASSPATH` that PowerJ uses to execute the application. For more information on the `CLASSPATH` environment variable, see How PowerJ uses the CLASSPATH environment variable.

Don't use the debugger [**Debug** page]
Prevents the use of the debugger while running the application.

Run application with debugger [**Debug** page]
Runs the program under control of the PowerJ debugger.

Class for debugging [**Debug** page]
Specifies a particular class that you want to debug. If you leave this blank, PowerJ begins debugging with the main form for the application.

Select an initial breakpoint [**Debug** page]
Specifies an initial breakpoint to be used when debugging the application under control of the PowerJ debugger. For more information, see Breakpoints.

Properties for Java application targets

Properties for Java application targets are specified on the property sheet available through the Targets window. They are the same as for Java applets. For more information, see Properties for Java applet targets.

Java classes

A Java classes target creates a set of Java class files. It is similar to an object file library, in that it cannot be executed on its own, but it is not a single file. You should use this type of target when you want to compile Java source files into class files but the class files do not form an applet, application or servlet.

For example, if you want to create a JavaBeans component you can use this type of target, and add managed classes to it. For more information on creating JavaBeans components, see Creating JavaBeans Components.

When you first create a Java classes target, it only contains the WXT file that defines the target. You can then add existing Java source files (`.java` extension) to the target (using **Add File** from the Target window's **File** menu), or you can create new forms, classes and files for the target (using **New** on the context menu for the target in Target Window).

When you build a Java classes target, PowerJ compiles all the Java source files into class files.

Run options for Java class targets

The run options for Java class targets are similar to those for Java application targets. However, they also contain the following options which control what PowerJ does when you actually try to run the target:

Do nothing [**General** page]
Builds the classes in the target if necessary, but does not try to execute them in any way.

Select a class to run from this target [**General** page]
Executes a single class contained by the target. At run time, PowerJ will prompt to ask which class you want to run.

Run an external Java application [**General** page]
[JDK 1.1 only] Executes a Java application which presumably uses some or all of the classes in the target. On the run options page, you must enter the name of the Java application that you want to invoke.

Attach to a remote virtual machine [**General** page]
[JDK 1.1 only] Runs the class within a Sun Virtual Machine that is already running. Additional information for this process is specified on the **Remote Debug** page of the run options.

Remote machine name or IP address [**Remote Debug** page]
Specifies the machine for running the class. The default is `localhost`, referring to your local computer.

Agent password for Sun Virtual Machine [**Remote Debug** page]
When a Sun Virtual Machine begins execution, it displays a password (usually five or six characters) which must be specified by Java classes that should be attached to that virtual machine. You must copy the password when the virtual machine starts executing and fill it into the field provided on this page. For more information on this process, see Starting the debugging server.

Properties for Java class targets

Properties for Java class targets are specified on the property sheet available through the Targets window. They are the same as for Java applets. For more information, see Properties for Java applet targets.

WebApplication targets

A WebApplication target can be used to aid deployment of Java applications. This type of target includes the concept of *publishing*, where all the files of the application can be copied to a specified location when you run the application. This makes it much easier for you to manage and debug large web applications. It can bring together all the parts of your web site, including Java applets, HTML files, graphics files, and web server extensions. For further information, see WebApplication targets.

Collection targets: CAB, JAR, and ZIP

Collection targets make it easy to package a complete application in a single file, simplifying the job of distributing the application to other sites. For more information, see Using collection targets: CAB, JAR, and ZIP.

Java web server applications

A Java web server application is sometimes called a servlet or web server extension. PowerJ has target types to help you create CGI, NSAPI and ISAPI servlets. For further information, see Server extensions.

Dynamo server applications

A Dynamo server application makes use of NetImpact Dynamo running in conjunction with a web server. The most important use of NetImpact Dynamo is accessing databases on the Server system. For more information, see NetImpact Dynamo server applications.

Migration from JDK 1.02 to JDK 1.1

PowerJ makes it possible to change a JDK 1.02 target into a JDK 1.1 target.

- **To change the type of a target:**

1. On the **File** menu of the main PowerJ menu bar, click **Save Project As**.
2. PowerJ asks you where you want to save each target folder. When you reach the target whose type you want to change, type a new name under **Folder name** and set **File type to save as** to `PowerJ AWT 1.10 target file`.
3. Click **Save**.
4. Continue to save the project in the usual way.

From this point on, the target will be considered a JDK 1.1 target and will use the JDK 1.1 versions of all Java components. Note, however, that this does not make any changes in the code you have written. For example, if you have used JDK 1.02 methods that are considered obsolete in JDK 1.1, your old code will *not* be changed. You must change such code manually.

Note: If you click **Save Project As** to save your AWT 1.02 target as an AWT 1.1 target, but click **Cancel** when PowerJ prompts you to save the project file, PowerJ still converts your target to JDK 1.1, but does not save the project file to disk.

Targeting both JDK 1.02 and 1.1

If you want to write an application that will work under both JDK 1.02 and JDK 1.1, write your program according to the JDK 1.02 standard. JDK 1.1 is (mostly) backward compatible with JDK 1.02. However, you should recognize that this is only a stopgap measure, since JDK 1.1 deprecates some JDK 1.02 methods and support for those deprecated methods may eventually be discontinued.

If you want your 1.02-compatible program to use JDK 1.1 functionality if it is available, put the 1.1 method calls inside a `try` block followed by a `catch` block for `java.lang.NoSuchMethodError`. That error will be generated when a 1.1-specific method is called under a Java 1.02 VM.

Build macros

PowerJ makes it possible to define *macros* to be used as part of file path names. For example, suppose that a target depends on a number of class files from `C:\MyProject\Class`. You could define a macro named `classfolder` to refer to this folder. Then you could specify library search paths similar to the following:

```
$(classfolder)
$(classfolder)\sub1
$(classfolder)\sub2
```

Defining path names in this form contributes to the portability of targets and projects. When you define a build macro, it is saved in a file called `Optima.mac`, in your PowerJ `System` folder. It thus applies to all projects on your computer, and must be set on any other computer that is used to build a project that uses the macro. If you move the target to a computer that contains the libraries in a different folder, all you have to do is redefine the `classfolder` macro to the new folder and PowerJ can find all the libraries again.

Tip: The `Optima.mac` file can be edited with a text editor. If you want to move a set of build macros to a new computer, you can copy the file and then edit the paths in it.

Macro names follow the usual rules for Java symbols: they can consist of alphanumeric characters and the underscore character.

Note: Build macros are case-sensitive. Therefore, a macro named `Win32SDK` is not the same as one named `WIN32SDK`.

- **To define a macro for use in path names:**

1. From the **Tools** menu of the main PowerJ menu bar, click **Options** and then **Build Macros**.
2. Click **New**. This produces a dialog box to take information about the new macro.
3. Type a macro name into **Name**.
4. Type the associated pathname into **Value**.
5. Click **OK** in the **Build Macro Details** dialog box, then click **OK** in the main dialog box.

Once you have defined the macro, you can use it in path names anywhere in the property sheets of the Targets window. As shown above, you use a macro in the form

```
$(macroname)
```

When PowerJ comes across such an expression in the property pages for a target or source file, PowerJ replaces the expression with the value of the macro.

- **To change an existing macro:**

1. From the **Tools** menu of the main PowerJ menu bar, click **Options** and then **Build Macros**.
2. In the list of macros, double-click the name of the macro you want to change.
3. Type a new value under **Value**.
4. Click **OK** in the **Build Macro Details** dialog box, then click **OK** in the main dialog box.

You can delete an existing macro by clicking the macro name and then clicking **Remove**.

On the **Build Macros** page of the Options dialog box, you can click **Show system-defined macros** to see the build macros automatically defined by PowerJ.

- Referring to environment variables
- How PowerJ uses the CLASSPATH environment variable

Referring to environment variables

In the property pages for a source file or target, you can refer to environment variables using the syntax

```
$(%envnam)
```

For example,

```
$(%TMPDIR)
```

stands for the value of the environment variable `TMPDIR`.

How PowerJ uses the CLASSPATH environment variable

The CLASSPATH environment variable tells a Java VM where to locate class files used when running a Java program. Usually the CLASSPATH is set as an environment variable, though for the Microsoft VM it can also be set in the Windows registry. The value of this variable is a list of folders where the VM can search for the class files it needs. Folder names in the list should be separated with semicolons.

There are two different times when CLASSPATH is important:

1. When you are running a program under the control of PowerJ. This is discussed in the current section.
2. When you are preparing a program for final deployment. This is discussed in CLASSPATH and CODEBASE.

When you are using PowerJ to run your Java programs, PowerJ sets the CLASSPATH for your program independently of CLASSPATH environment variable setting for your computer. In other words, PowerJ overrides your computer's normal CLASSPATH when PowerJ runs a Java application or applet.

For standalone testing of applications and applets, where you are not using PowerJ to run the program, you may need to set the CLASSPATH environment variable yourself.

Generally, you only need to modify the PowerJ CLASSPATH if your program uses class files that are not controlled by PowerJ. If all your class files are created with PowerJ or are known to PowerJ (because they're part of the PowerJ library or the standard JDK library), you should not need to worry about CLASSPATH.

When a browser searches for class files under a CLASSPATH, the browser goes one at a time through the list of folders named in the value of CLASSPATH. The folders in the class path should give the name of the "top" of the class file hierarchy. For example, suppose that an application uses the class

```
abc.def.MyClass
```

If the first folder named in the CLASSPATH is C:\MyFolder, the browser will look for the file

```
C:\MyFolder\abc\def\MyClass.class
```

As shown, the parts of the package name are interpreted as subfolders under the folder names given in the CLASSPATH list.

When the Java program needs to find a class file, it searches through the folders named in the CLASSPATH list, in the order they appear. The program uses the first appropriate class file that it finds. Therefore, the order of the list is important if the set of folders contain more than one class file with the same name.

Setting the CLASSPATH

If you need to set the value of CLASSPATH as an environment variable for a program that you are going to run outside of PowerJ, you can execute a command of the following format in a DOS window before starting the application in the same DOS window:

```
set CLASSPATH="folder1;folder2;folder3"
```

You can modify the PowerJ CLASSPATH using the property sheet for your target.

- **To specify that a folder should appear in the CLASSPATH of a target:**

1. Open the Targets window.
2. In the Targets window's **View** menu, click **Show Property Sheet** to make sure it is marked with a

check.

3. In the list of targets, expand the entry for the target whose `CLASSPATH` you want to set. Click the target entry to select it.
4. In the property sheet for the target, click the **Class Path** tab.
5. In the **Folder** field, type the name of the folder whose name should appear in the `CLASSPATH`, then click **Add**.
6. Click **Apply**.

Folder names are often specified using build macros. For more information, see Build macros.

If you make target X depending on target Y, all the classpath dependencies for target Y are automatically added to the classpath dependencies for target X. This simplifies the process of making test runs within PowerJ itself. However, when you finally deploy target X, you should make sure that the classes needed by target Y are included in the list of classes needed by target X.

Class paths for WebApplication targets

WebApplication targets automatically create a `CLASSPATH` variable that references the folder where the application is published. For more information, see WebApplication targets.

Global classpath settings

PowerJ makes it possible for you to specify a set of folders to be included in the PowerJ `CLASSPATH` whenever PowerJ runs a target. For further information, see Classpath options.

Note: To see the exact `CLASSPATH` that PowerJ uses for running a particular target, check the batch file that PowerJ generates during the run. For more information about these batch files, see Batch files for running targets.

Resource bundles and localization

Resource bundles were introduced in JDK 1.1 to support localization. A resource bundle is a group of classes containing strings and other objects that may differ from locale to locale. For example, you might store all the strings that are displayed for the user in a resource bundle. Then, if you are preparing a program for English-language users, you would create a resource bundle with all the appropriate strings in English; for French-language users, you would create another bundle with all the appropriate strings in French. At run-time, your program would fetch all its strings from the appropriate bundle, thereby presenting itself to the user in the language of the user's choice.

Note: Most programs will only store strings in resource bundles. However, it is also possible to store other data, such as sounds, pictures, or video sequences.

The most important part of a resource bundle class is a data member named `contents`. This is an array giving a list of pairs:

- The first element in each pair is a String giving a label for the resource.
- The second element is an Object giving the resource itself.

A program using the resource bundle can specify the string label and obtain the corresponding resource object from the resource bundle.

Each class within a resource bundle contains a version of the same data prepared for a different locale; for example, there might be an English class, a French class, a Japanese class, and so on. All of these classes have parallel entries in the `contents` table. For example, they might all have a resource where the first element in a pair is the label `"message1"`. In each resource class of the bundle, the second element would be the local version of `message1`: a message in English, a message in French, and so on.

A program using the bundle can request the version of a resource for a given locale, and the Java virtual machine will retrieve the `contents` table from the most appropriate resource bundle class that is available.

Naming resource bundle classes

The classes in a resource bundle have names that follow a standard naming convention. The Java virtual machine relies on the naming convention to determine which class to use for a particular locale.

The naming convention can be summarized as follows:

- A resource bundle class name must use a base name and optional extensions.
- The base name and extensions are separated by underscore characters.
- The base name must be the same for all classes in a particular bundle. It can be any valid Java symbol that does not contain underscore characters.
- The first optional extension is a lower-case, two letter ISO-639 code that identifies the language of the resources in the class. For example, `en` for English, `fr` for French, `de` for German, `ja` for Japanese and `zh` for Chinese. For a complete list of codes, see:

  `http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`
- The second optional extension consists of two upper-case letters that identify the country that is appropriate for the resources in the class, following ISO-3166. For example, `US` for United States, `GB` for United Kingdom and `JP` for Japan. For a complete list of codes, see:

  `ftp://ftp.ripe.net/iso3166-countrycodes`

If a class name has only one extension it is assumed to contain information that is valid for all speakers of that language. If a class name has no extensions it will be used if there is no class with more appropriate extensions.

Here are some examples of resource bundle class names for a resource bundle called `foo`:

`foo_en_us`
: This class will be used for resources from the `foo` bundle for English users in the United States.

`foo_en`
: This class will be used for resources from the `foo` bundle for all English users if there is no class for the country of the current locale.

`foo`
: This class will be used for resources from the `foo` bundle for all users if there are no classes for the language of the current locale.

If the above three classes are in the same bundle, `foo_en_us` will be used for an English-speaking American locale, `foo_en` will be used for an English-speaking Canadian locale and `foo` will be used for a French-Canadian locale.

Creating a resource bundle

In a JDK 1.1 application, PowerJ makes it easy to create a resource bundle class.

- **To create a resource bundle class:**

1. In the **File** menu, click **New**, then click **Class**. This opens the Class Wizard.
2. In the list of class types, click **ListResource** then click **Next**. (If **ListResource** does not show up in the list, check that you have a JDK 1.1 target.)
3. Under **Class name**, type a name for the new resource bundle class. The name must conform to the standard naming convention for Java resource bundles, described above.
4. Click **Finish** to open the code editor with commented code skeletons for your resource bundle class.
5. Read the comments in the skeleton code and edit the code to complete your class.
6. Use the property sheet for the class to change the package name or other properties.

You should repeat the above steps to create a class for each locale that you want to support.

Tips for using resource bundles

An application may use several different resource bundles. For example, you might have one bundle for error messages, another bundle for strings that are frequently used by the program, and so on. By separating low-use resources (like error messages) from high-use resources, you may be able to reduce an applet's download time. Separating different types of resources into different bundles may also be convenient in large projects, where different translators may work on different parts of the application.

You can use functions in the JDK 1.1 library to determine the current locale and the preferred format for data like dates and currency. For more information, see:

`http://java.sun.com/docs/books/tutorial/intl/datamgmt/formatting.html`

For general information on Java internationalization, see:

`http://java.sun.com/docs/books/tutorial/intl/index.html`

You can also check the references in the Bibliography.

Chapter 3. Using PowerJ

This chapter explains the mechanics of using PowerJ: placing objects on forms, setting the initial properties of design objects, and editing the source code associated with various events.

This chapter assumes that you have read Basic concepts of PowerJ.

General usage notes

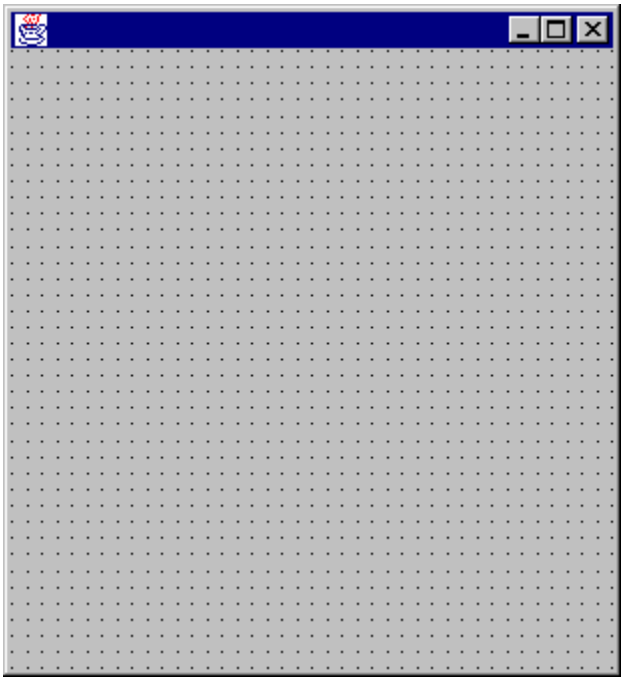
PowerJ often offers several different ways to perform the same task. For example, if you want to see an object's property sheet, you can do any of the following:

- Double-click the object.
- Use the right mouse button to click the object, then click **Properties**.
- Go to the Objects window, and click **Show Property Sheet** in the **View** menu.

This chapter does not attempt to list all the possible ways to perform tasks.

Using the form design window

The *form design window* is the design time representation of a form. You can lay out the form by resizing it, changing its properties and adding components to it. When you run your program, the form will have the size, properties and controls that you have designed.



By default, the form is marked with a grid of dots to help you position objects. When you run your program, these dots will not appear.

You can change the size of the form design window by dragging the window frame. The size that you set for the form will be its initial size when you run the program.

If your application has several forms, each form has its own form design window. To save on screen space, you can close form design windows that you don't currently need.

- **To close a form design window:**

1. If the form has a close button (upper right corner), you can click the button.
2. If the form has a system menu (upper left corner), click **Close** in that menu.
3. If the form has neither a close button nor a system menu, press ALT+F4.

The **Window** menu of the main PowerJ menu bar contains a list of all open form design windows.

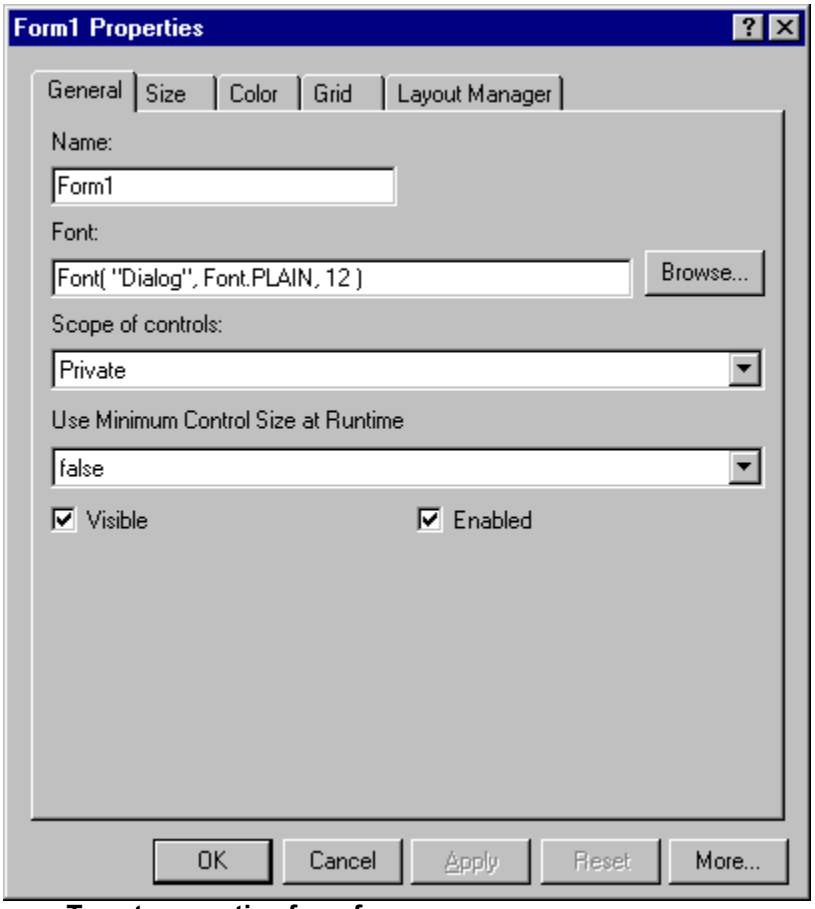
- **To open a closed form design window:**

1. In the Objects window, use the right mouse button to click the name of the form.
2. On the context menu, Click **Open** to display the form.

Changing a form's properties

Every form has a set of associated properties. These control the appearance and behavior of the form when it is displayed during program execution. Some properties, such as grid options, only apply to the form at design time.

When you create a form, PowerJ sets the form's properties to default values. You can assign different properties to the values using the form's *property sheet*.



- **To set properties for a form:**

1. In the form design window, use the right mouse button to click a blank area of the form (one that does not have any buttons, boxes, or other objects). This displays a menu of possible actions.
2. Click **Properties** on this menu. This displays the form's property sheet.
3. Use the property sheet to set values for any properties you want to change. Click the tabs at the top of the main area of the property sheet to see different types of properties.
4. Click **OK** when you have assigned values to the properties you want to change.

The changes you have made may or may not be visible in the form design window.

The *PowerJ Component Library Reference* provides more information on form properties.

Tip: You can also change properties through the Object Inspector. Since the Object Inspector displays properties in alphabetical order, it may be easier to find a specific property through the inspector rather

than going through the form's property sheet.

- The form grid
- Reopening a closed form design window

The form grid

The **Grid** page of a form's property sheet lets you control the grid of dots that appears in the form design window. The **Grid** page contains the following items:

Display the grid

If this box is blank, the form will not have a grid of dots when it is displayed in the form design window.

Grid Size

The **Width** and **Height** boxes specify the distance between dots in the grid. Distances are specified in *dialog units*, a measure of screen distance that is less device-dependent than pixels. For a discussion of the difference between dialog units and pixels, see Pixels vs. dialog units.

Align objects to the grid

If this box is marked, all objects on the form have their size and position adjusted to coincide with the grid. For example, suppose that the dots appear every 10 dialog units; then every object on the form has its size and position adjusted so that its corners exactly coincide with grid dots.

If this box is blank, PowerJ does not adjust the size and position of objects on the form. For example, you can drag the edge of an object so that it falls between dots.

Reopening a closed form design window

You can close a form design window by clicking the close button (in the upper right corner of the window). You can also reopen the form design window later on.

- **To open a form design window for a form:**

1. Open the Classes window (click **Classes** in any **View** menu).
2. In the left part of the Classes window, use the right mouse button to click the name of the form you want to open, then click **Open Form**.

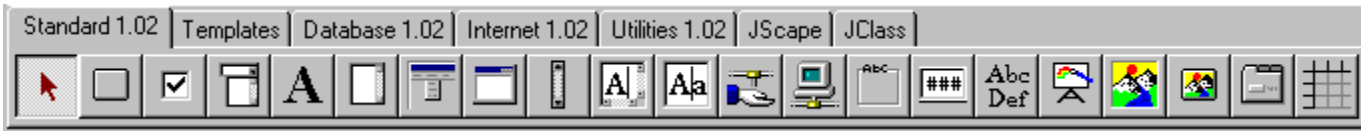
This opens the form design window for the form. If you want to open a code editor to edit the form's code, click **Open** instead of **Open Form**.

Adding objects to a form

The first step in creating a program with PowerJ is to design one or more forms. When you start a new project, PowerJ displays a blank form where you can design the first form for that project. The design process is simply a matter of adding *objects* to the blank form.

Component palette

The *component palettes* are tabbed toolbars whose buttons represent the components that you can add to your form. The leftmost button for each tab is for the selection tool and the rest are for components.



To see the name of a component on the palette, move the cursor to the button and wait a second or two. After a few moments, PowerJ displays a *tooltip* telling what the button means. By default, PowerJ displays the following component palettes:

- If your current project only contains Java 1.02 targets, PowerJ only shows you the Java 1.02 palette.
- If your project only contains Java 1.1 targets, PowerJ only shows you the Java 1.1 palette.
- If your project contains both types of targets, PowerJ shows you both palettes.

You can also manually specify which component palettes should be shown. For more information, see JDK component palettes.

For an explanation of how to use each item in the component palette, see Programming standard objects.

◆ **To add an object to your form from the component palette:**

1. On the component palette, click the button for the type of component you want to add to the form.
2. Move the cursor to the form design window and click the location where you want to place the component. PowerJ adds a component of default size, with its top left corner at the location you clicked.
3. Resize the component if necessary, by dragging the component's sizing handles.

Once you have added a component to your form, the result is called an *object*. This terminology helps distinguish between "components" (which are abstract buttons on the component palette) and "objects" (which are real items on a form).

Tip: You can combine steps 2 and 3 above by moving the cursor to the form design window, holding the button down, and dragging across the form until the object reaches the desired size.

Adding several objects of the same type

If you click a button on the component palette, PowerJ lets you place a single object of that type onto the form design window. After you have placed one object, the button on the component palette turns itself off; you need to click another component button before you can place another object.

In some cases, you may want to place several objects of the same type on a form. For example, you might want to place several option buttons on the form.

◆ **To place several objects of the same type on a form:**

1. Hold down the SHIFT key and click the appropriate button on the component palette.
2. Move the cursor to the form design window and click the location where you want to place the first component. PowerJ adds a component of default size, with its top left corner at the location you clicked.

3. Repeat the above step to place other objects on the form.
4. Resize the objects if necessary, by dragging their sizing handles.

In other words, if you SHIFT-click a component palette button, the button stays clicked until you click a different button, or until you select a different page in the palette.

Positioning an object

If you place an object in one position on a form, then decide to put it somewhere else, you can move it.

- **To change the position of an object on a form:**

1. Click the object in the form design window, then drag the object to its desired position.

Tip: If you hold down the CTRL key and press an arrow key, PowerJ moves the active object in the direction of the arrow.

Sizing an object

PowerJ makes it easy for you to change the size of an object that you have already placed on the form.

- **To change the size of an object:**

1. Click the object in the form design window. You will see sizing handles appear on the edges of the object.
2. Drag a sizing handle. You will see the outline of the object grow or shrink as you move the handle.

The true size of the object may or may not be apparent to the user during program execution. For example, the size of a text box is obvious, but the size of a label doesn't make much difference, as long as it is big enough to hold the text of the label.

Tip: Holding down SHIFT and pressing an arrow key resizes the currently selected object(s) in the direction of the arrow key.

Deleting an object

If you decide to get rid of an object, you can easily delete it from the form design window.

- **To delete an object from a form:**

1. Use the right mouse button to click on the object you want to delete. This displays a menu of actions you can perform on the object.
2. Click **Delete** in this menu. The object will disappear from the form design window.

Deleting an object also deletes any event handlers you may have associated with the object.

Tip: You can also delete objects by selecting them, then pressing the DEL key.

Deleting forms

The previous section showed how to delete an object from a form. To delete the form itself, you use the Objects window:

- **To delete a form:**

1. In the Objects window, click the name of the form you want to delete. (The name should appear as a source file for one of the targets.)
2. Use the right mouse button to click the name, then click **Delete**.

Deleting the form in this way states that the form will not be used in the target. If the form is used for other executables, it will be removed from the selected target but its source files will not be deleted. However, if the form is not needed by any other target, PowerJ deletes the form. In the process, PowerJ deletes all objects and member functions associated with the form.

Copying an object

There are many situations where you want the form to contain a set of similar objects. For example, you may want several option buttons lined up with each other in a vertical list. You can save yourself some typing by creating the first such object from scratch, then creating the other objects by copying the first.

- **To copy an object:**

1. Click the right mouse button on the object you want to copy, then click **Copy**. This writes a copy of the object into the Windows clipboard.
2. Click the right mouse button anywhere on the form, then click **Paste**. This places a new copy of the object onto the form.

The copy is given an appropriate symbolic name, based on its type. For example, if the original is named `cb_1`, the copy may be named `cb_2`.

The copied object has the same properties and event handlers as the original, except that event handler names are changed appropriately. For example, if you have defined an **Action** event handler for the original object, the copied object has an identical **Action** event handler, except that it is named `cb_2_Action` instead of `cb_1_Action`. After the copy operation has taken place, it's a good idea to review the properties and the event handlers of the copied object, just to make sure that they're what you want.

Cut operations

When you use the right mouse button to click an object, the resulting menu has a **Cut** command. **Cut** copies the object to the clipboard, then deletes it from the form. You can then paste the copied object elsewhere in the form design window. You can use cut and paste operations to move an object from one form to another.

Copy, cut, and paste shortcuts

You can use the following standard keyboard shortcuts for copy, cut, and paste operations:

CTRL+C	Copies the selected object(s).
CTRL+X	Cuts the selected object(s).
CTRL+V	Pastes onto the selected form.

Aligning objects

PowerJ makes it easy for you to align one object with another in the form design window.

- **To align one object with another:**

1. Drag the mouse until the framing rectangle surrounds all the objects you want to align. When you release the mouse, PowerJ places sizing handles on all the objects. One object has solid sizing handles, and the rest are hollow.
2. Click an object whose alignment you want to match. PowerJ places solid sizing handles on this object.
3. Use the right mouse button to click the solid-handled object, then click **Align**. This produces another menu of possible alignments, shown as pictures.
4. Click the type of alignment you want. The objects with hollow handles move to match the object with solid handles.

In any alignment operation, the hollow-handled objects are moved to match the one solid-handled object.

Matching object sizes

PowerJ also lets you change the size of one object to match another.

- **To change the size of one object to match another:**

1. Drag the mouse until the framing rectangle surrounds all the objects whose sizes should match. When you release the mouse, PowerJ places sizing handles on all the objects. One object has solid sizing handles, and the rest are hollow.
2. Click an object whose size you want to match. PowerJ places solid sizing handles on this object.
3. Use the right mouse button to click the solid-handled object, then click **Same Size**. This produces another menu of possible operations to match sizes in various ways (height, width, or both). This menu shows possible operations with pictures.
4. Click the type of adjustment you want. The objects with hollow handles change size to match the object with solid handles.

Selecting multiple objects

You can select several objects at once by holding down the SHIFT key and clicking each object you want to select. The first time you SHIFT+click an object, it receives solid sizing handles. When you SHIFT+click a new object, the new object receives solid sizing handles and previously clicked objects receive hollow sizing handles.

Once you have selected a set of objects, you can align or resize them using the techniques mentioned in previous sections.

When several objects are selected, the object with solid sizing handles is the primary object. For example, if you select several objects, then use **Align** to give all the objects the same alignment, the objects with hollow handles move to match the alignment of the object with solid handles.

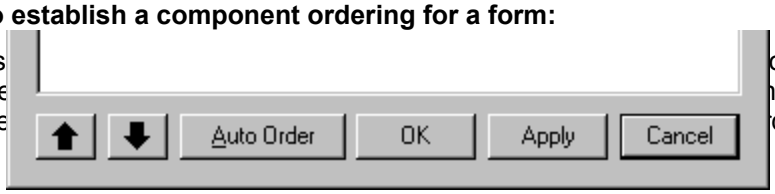
Component order editor

It is often useful to establish an ordering on the components for a form. For example, suppose a form contains a number of text fields where the user will be asked to enter various types of information. If you establish an ordering between these text fields, the user can fill in the first field, then move to the next field by pressing the TAB key. In this way, users can TAB from one field to another without having to remove their hands from the keyboard. Without a component ordering, users would have to use the mouse to switch from one text field to the next.

Important: Component ordering only works with Java 1.1 applications. Java 1.02 has a bug in AWT which prevents tabbing from one component to another.

- **To establish a component ordering for a form:**

1. Us ot have a component or, if you are the form. On the resulting me der Editor.



2. To order the components according to their position on the form (moving from top to bottom and from left to right), click **Auto Order**.
3. To change the position of an individual component in the ordering, click the component and then use the arrow buttons (or press CTRL+UPARROW or CTRL+DOWNARROW on your keyboard) to move the component up or down in the component list.
4. Click **OK** when you have finished setting the component order.

Once you have established a component ordering, users will be able to tab from one marked component to the next marked component on the list. If the user is currently at the last marked component on the list, pressing TAB goes to the top of the list again (the first marked component).

Tip: Remember that component ordering only works for Java 1.1 programs, not Java 1.02.

Component ordering in container objects

A form may include objects that contain other objects. For example, a panel on a form may contain objects of its own. In this case, objects can be ordered within the container as well as on the form as a whole.

Visual classes

A *visual class* can be thought of as a form that is never displayed. You can also think of a visual class as a class that is:

- Visible at design time
- Not visible at run time

In other words, visual classes let you visualize objects that are not displayed during execution.

For example, suppose you are making a web server servlet: one that doesn't display forms. However, you want the program to access a database. The easiest way to do this is to use transaction and query objects (as discussed in Accessing databases). Therefore, you could make a visual class object, then place transaction and query objects on this object in the same way that you'd place them on a form.

- During design time, you can work with the objects on the visual class as if they were objects on a normal form.
- At run time, the visual class is not displayed. However, when you construct a visual class object, PowerJ automatically constructs all the objects that you have placed on the visual class "form". This saves you the trouble of constructing such objects explicitly. Similarly, when you call the **create** method to initialize a visual class object, PowerJ automatically calls **create** on each object contained in the visual class, thereby initializing all the contained objects.

As with any Java class, a visual class must inherit from Object, or a class which descends from Object.

◆ **To define a visual class:**

1. On the **File** menu of the main PowerJ menu bar, point to **New** and then click **Class**. This opens the Class wizard.
2. If your project has more than one target, the Class Wizard asks where you want to define the new class. Click the target where you want to define the new class, then click **Next**.
3. Under **What type of class do you want?** click **Visual Class**, then click **Next**.
4. Under **Package Name**, type a name for the Java package that will contain this class.
5. Under **Class name**, type a name for the new class.
6. If you do not want this class to inherit directly from Object, type the name of a different class under **Inherits from**.
7. If this class implements an interface, type the name of the interface under **Implements**.
8. If this will be a public class, make sure **Public** is checked.
9. If this will be an abstract class, make sure **Abstract** is checked.
10. If this will be an interface, make sure **Interface** is checked.
11. Click **Finish**.

PowerJ displays a design window for the visual class similar to a form design window. You can define data members within the visual class by placing non-visual PowerJ objects (for example, timers, transaction, or query objects) onto the visual class's design window.

When PowerJ constructs an object of the visual class type, it automatically constructs all the member objects that have been placed on the visual class. In the process it triggers **objectCreated** events for those objects. Similarly, when PowerJ destroys the object, it triggers **objectDestroyed** events for all the objects on the visual class.

- Visual class properties
- Visual classes at run time

Visual class properties

The design-time property sheet for a visual class has the following entries on the **General** page:

Name
The name of the class.

Scope of Controls
This combo box lets you choose the scope of the controls that you place on the visual class. For example, if you choose **Public**, all controls placed on the visual class will be `public`.

Note: In most cases, you will want the scope of controls to be `public`.

The **Grid** page of the design-time property sheet controls the grid of dots displayed on the visual class design window. The options on this sheet are similar to the options for the grid on a true form.

Visual classes at run time

In order to use a visual class at run time, you typically construct the class object, then initialize it using **create**, as in:

```
MyVisualClass mvc = new MyVisualClass();
mvc.create();
```

You must call **create** in order to initialize the visual class and all the objects it contains. Otherwise, the class and its objects will not have the property settings that you specified at design time.

Templates

A *template* is a saved copy of one or more objects, including their event handlers. Form design templates provide a sophisticated form of copy and paste facility, letting you create shortcuts for handling component combinations that you use frequently.

For example, you might create a template containing a button labeled **OK** and a button labeled **Cancel** (including **Action** event handlers for the two buttons). If you place that template onto a form, PowerJ places the two buttons onto the form. If necessary, you can then change the position of the buttons or the code in their **Action** event handlers.

- **To create a template:**

1. In the form design window, drag the mouse to surround the objects that will make up the template. When you release the mouse, PowerJ marks the objects with hollow sizing handles.
2. Use the right mouse button to click on one of the objects, then click **Copy to Template**.
3. PowerJ prompts you for a name you want to assign to this template. Type in a name.
4. You may also type in a description of the template (to be used when someone requests help information about the template).
5. If you click **Edit Icon**, PowerJ calls up the Image Editor to create an icon for this template.
6. Click **OK**.

PowerJ adds your chosen icon to the **Templates** page of the component palette. From this point on, you can place the template onto a form in the usual way: click the icon in the **Template** page, then drag across the form design window.

Once you place a template onto the component palette, it remains there for all future PowerJ sessions.

- **To delete a template from the component palette:**

1. Use the right mouse button to click on the template you want to delete, then click **Delete**. PowerJ checks to make sure you really want to delete the template.

Form templates

You can create a template form in much the same way that you create a template made up of individual components. For example, you could create a template form that had a standard set of menus or a predefined set of buttons.

- **To create a template form:**

1. Design the form in the form design window.
2. Use the right mouse button to click a blank area of the form, then click **Copy to Template**.
3. PowerJ prompts you for a name you want to assign to this template. Type in a name.
4. You may also type in a description of the template (to be used when someone requests help information about the template).
5. If you click **Edit Icon**, PowerJ calls up the Image Editor to create an icon for this template.
6. Click **OK**.

PowerJ records the template form you have just defined. The next time you use the Form Wizard to make a new form (for example, by clicking the **New Form** button in the PowerJ toolbar), you will see the template form as one of the possible choices. Click on the template and follow the usual steps to create a new form. When the form is created, it will start with the properties, objects, and event handlers you specified in the template form.

Note: It is important to recognize the difference between the name of the template and the name of a form defined by that template. The template name is displayed by the Form Wizard to identify the template. The form name is used in the definition for the form class.

If you create a new form from a form template, and the form name associated with that template matches an existing form name, PowerJ gives the new form a unique name by appending an underscore and a number to it (for example, `TemplateForm_3`).

- **To delete a template form:**

1. In the Form Wizard, use the right mouse button to click the template you want to delete, then click **Delete**. PowerJ checks to make sure you really want to delete the template.

Target templates

A *target template* is a user-defined target type containing predefined forms, objects, and code for a target. For example, you might make a target template containing three types of forms you use frequently. When you want to make a new application that uses some or all of those forms, you would create the application using your target template.

Before you can create a target template, you must create the target that you will use as the template. This means defining the forms, objects, event handlers, and other code that will make up the template. Once you have done this, you are ready to create the target template.

- **To create a target template:**

1. Open the Targets window by clicking **Targets** in the **View** menu.
2. In the Targets window, click the target that you want to use as a template.
3. From the **File** menu of the Targets window, click **Save As Template**.
4. Type a short name for the target under **Template Name**.
5. Type a description of the target under **Description**.
6. If you want to specify a different palette image for the target, click **Edit** (to edit the existing icon with the Image Editor) or click **Browse** to select a palette image from some other file.
7. Click **OK**.

This creates a target template based on the selected target. The next time you create a target with the Target Wizard, your template will appear as one of the possible target types.

- **To delete a target template:**

1. In the Target Wizard, use the right mouse button to click the template you want to delete, then click **Delete**. PowerJ checks to make sure you really want to delete the template.

The templates folder

By default, the templates you create are stored in a folder named `Template` under the main PowerJ folder. In some situations, you may want to specify a different folder. For example, if you have a team of programmers working on a project, you may wish to store templates under a common folder on your local network.

- **To change the location where templates are stored:**

1. From the **Tools** menu on the main PowerJ menu bar, click **Options**.
2. On the **Folders** page, click **Let me specify a folder**, then type the name of the folder under **Folder**.
3. Click **OK**.

From this point on, PowerJ will search for templates in the specified folder and will also store new templates in that folder.

JDK configuration options

The JDK configuration options let you tell PowerJ to use a different version of one of the standard Java development kits (JDKs). PowerJ also helps you to convert your PowerJ project from JDK 1.02 to JDK 1.1; for further information, see Migration from JDK 1.02 to JDK 1.1.

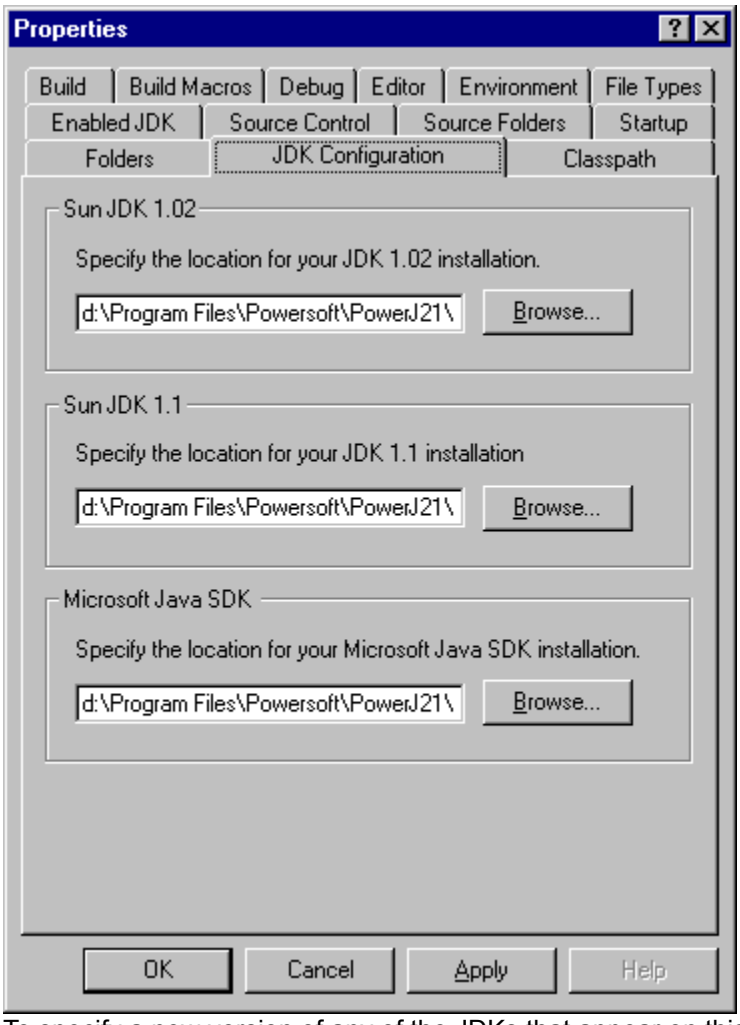
Do not change the JDK configuration options unless you have installed your own version of a JDK on your system. For example, if Sun releases a new version of JDK 1.1 and you want to use that instead of the version that comes with PowerJ, you should install the new version on your system, then instruct PowerJ to use the new version rather than the old one.

Important: Do not overwrite PowerJ's version of a JDK with a new version; install the new version in a different folder. This makes it possible to revert to the original version of the JDK if you encounter compatibility problems.

- **To tell PowerJ to use a different version of a JDK:**

1. On the **Tools** menu on the main PowerJ menu bar, click **Options** and then **JDK Configuration**.

This displays the following:



To specify a new version of any of the JDKs that appear on this page, type the folder containing the JDK

in the appropriate blank, or use the **Browse** button to specify the folder. If you have the Sun JDK 1.1 or Microsoft SDK for Java 2.0, you do not need to specify the location, since PowerJ can automatically determine their locations from the Windows registry.

Important: Sybase cannot guarantee that PowerJ will work with different versions of the JDK or the Microsoft SDK for Java. New versions may introduce incompatibilities that prevent this version of PowerJ from performing correctly.

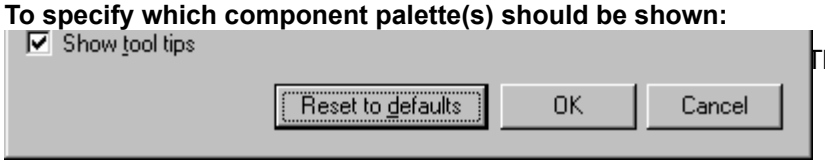
Newer JDK versions

By the time you read this, Sun may have released new versions of the JDK. If you wish to try using a newer JDK version, you can specify the newer JDK as your "JDK 1.1 installation". If the newer JDK is sufficiently backward compatible, PowerJ should be able to use it instead of JDK 1.1.

JDK component palettes

You can have PowerJ automatically determine which component palette to show, or you can manually specify which component palettes to specify.

- **To specify which component palette(s) should be shown:**

1. 

This displays the Toolbar

2. Under **Component palettes**, click the **Let me select palettes manually** option so that it is selected.
3. Click on the list items to check the palettes that you want to display.
4. Click **OK**.

The palettes that you have selected will be shown.

Enabling and disabling JDKs

Enabling and disabling JDKs

PowerJ lets you choose which JDKs will be enabled at design time. For example, you can choose to disable JDK 1.02 at design time. This simplifies the task of creating new targets, since it is automatically assumed that all targets are JDK 1.1; PowerJ doesn't have to ask which JDK you want to use. Furthermore, the Target Wizard will only display target types that are compatible with JDK 1.1. You get similar simplifications if you disable JDK 1.1 at design time—PowerJ omits various choices that are not compatible with JDK 1.02.

- **To enable or disable a JDK at design time:**

1. On the **Tools** menu on the main PowerJ menu bar, click **Options** and then **Enabled JDK**.
2. In the selection box, choose one of the entries (either a single JDK or both).
3. Click **OK**.

Remember that disabling a JDK only reduces the choices that PowerJ offers at design time. It does not, for example, delete the JDK files from your disk.

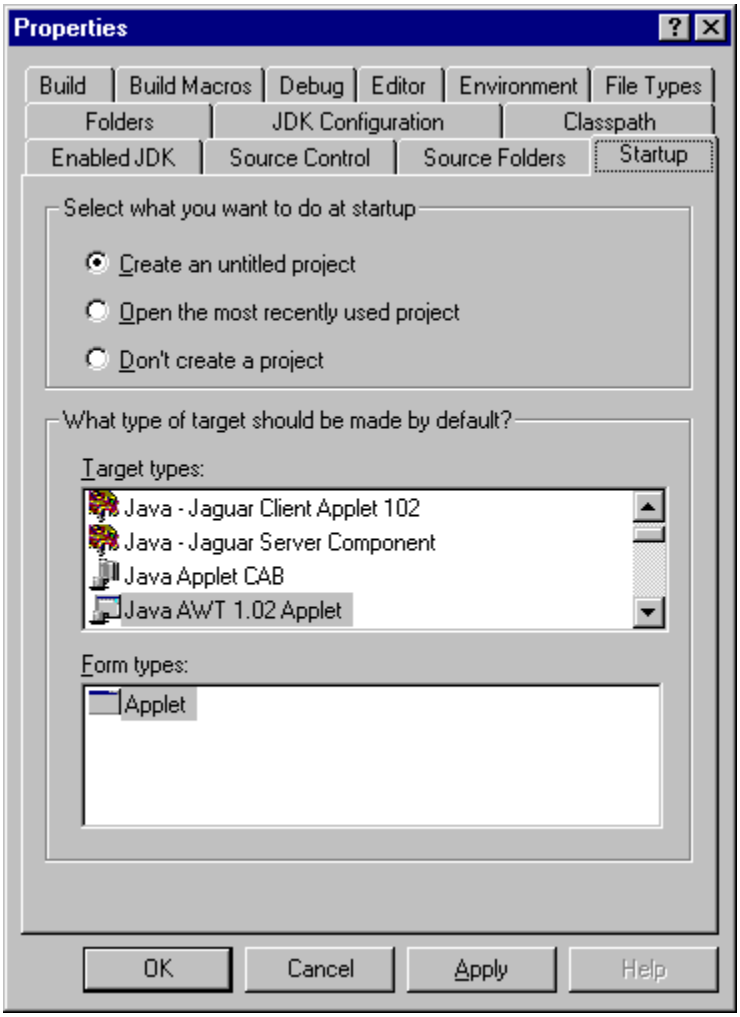
Startup options

The *startup* options for PowerJ specify what PowerJ does when it first starts up.

- **To see the current startup options:**

1. On the **Tools** menu on the main PowerJ menu bar, click **Options** and then **Startup**.

This displays the following:



The default selection is **Create an untitled project**. If this is marked, PowerJ creates an untitled project every time it begins execution. If **Don't create a project** is marked, PowerJ does not create such a project; you must explicitly load an existing project or create a new one. Finally, if **Open most recently used project** is marked, PowerJ opens the project you were working on at the end of your last session.

The **Default Target** list lets you specify a target to be created by default in any newly created project. When you click the target type in the **Target** list, the **Form** list changes to reflect the types of form that may be created in such a target.

Once you choose a target and a form, PowerJ will use those as the initial target and form for any new project.

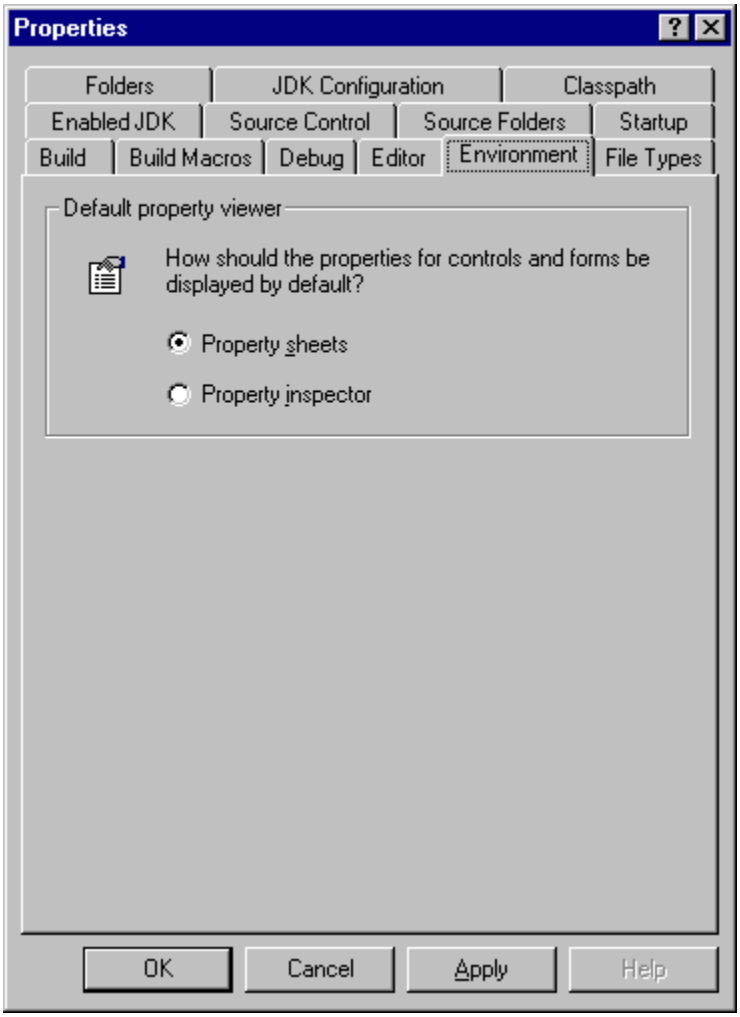
Property viewer options

The property viewer options of PowerJ control the default way in which object properties are displayed.

- **To see the current property viewer options:**

1. On the **Tools** menu on the main PowerJ menu bar, click **Options** and then **Environment**.

This displays the following:



If **Property sheets** is marked, PowerJ displays property values using standard property sheets. If **Property inspector** is marked, PowerJ displays property values using the Object Inspector.

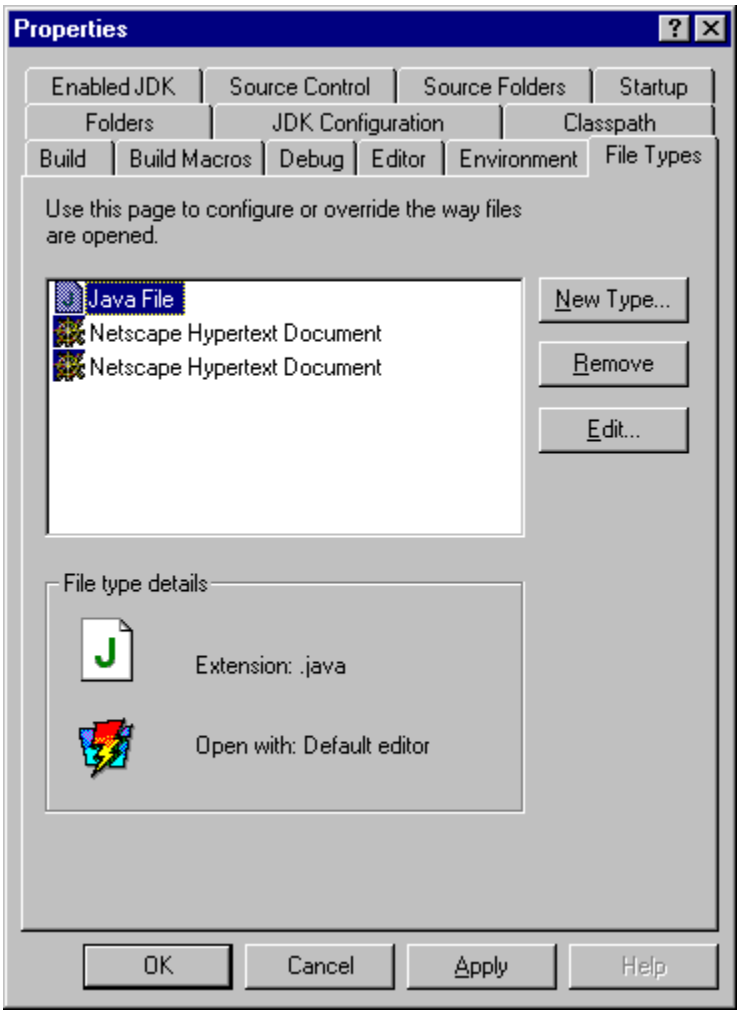
File type options

The *file type options* for PowerJ dictate how PowerJ opens various types of files. PowerJ determines the type of a file by examining the file name extension of the file. For example, files whose names end in `.WXC` are taken to be managed class definitions; when PowerJ opens such a file, it uses a code editor to display the results.

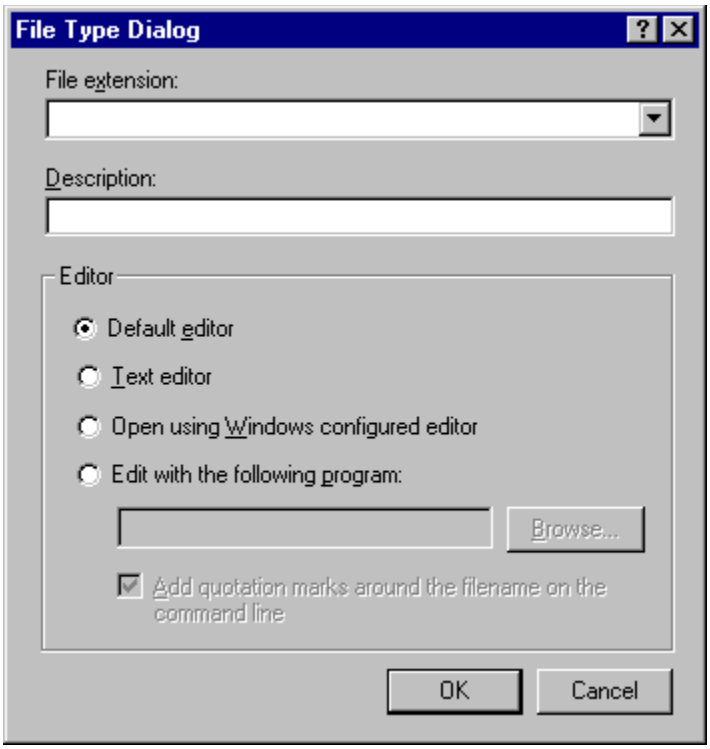
- **To see the current file type options:**

1. On the **Tools** menu on the main PowerJ menu bar, click **Options** and then **File Types**.

This displays the following:



If you want to create a new file type definition, click **New Type**. This displays the following dialog:



This dialog box lets you specify a file extension and the technique used to display the contents of files with that extension. In particular, the possibilities for HTML files include:

Default editor
PowerJ's own HTML editor. This editor is similar to the usual Java code editor; for example, it uses color to distinguish between different types of syntactic elements (constants, keywords, and so on). It also performs automatic indentation. Most importantly, the default editor lets you set breakpoints in some types of HTML code (for example, DynaScript).

Text editor
PowerJ's text editor. This is a simple text editor without any special HTML-based features. It does not let you set breakpoints.

Open using Windows configured editor
Uses the editor given by the system registry. For example, if you have Microsoft Front Page configured to be your "official" HTML editor, this option would invoke Front Page whenever you told PowerJ you wanted to edit an HTML file.

Edit with the following program
Gives a specific program that should be used when opening such files.

The **Description** box is supplied so that you can give a brief description of what the given file extension means.

If you are invoking a program to do the editing, it may be necessary to enclose file names in quotations marks on the command line that invokes the editor. If so, click **Add quotation marks around the filename on the command line** so that the box is marked with a check.

Classpath options

PowerJ makes it possible to specify a standard set of folders that are added to your `CLASSPATH` every time PowerJ runs a target.

Note: For more information about the `CLASSPATH`, see CLASSPATH and CODEBASE.

- **To see the current classpath options:**

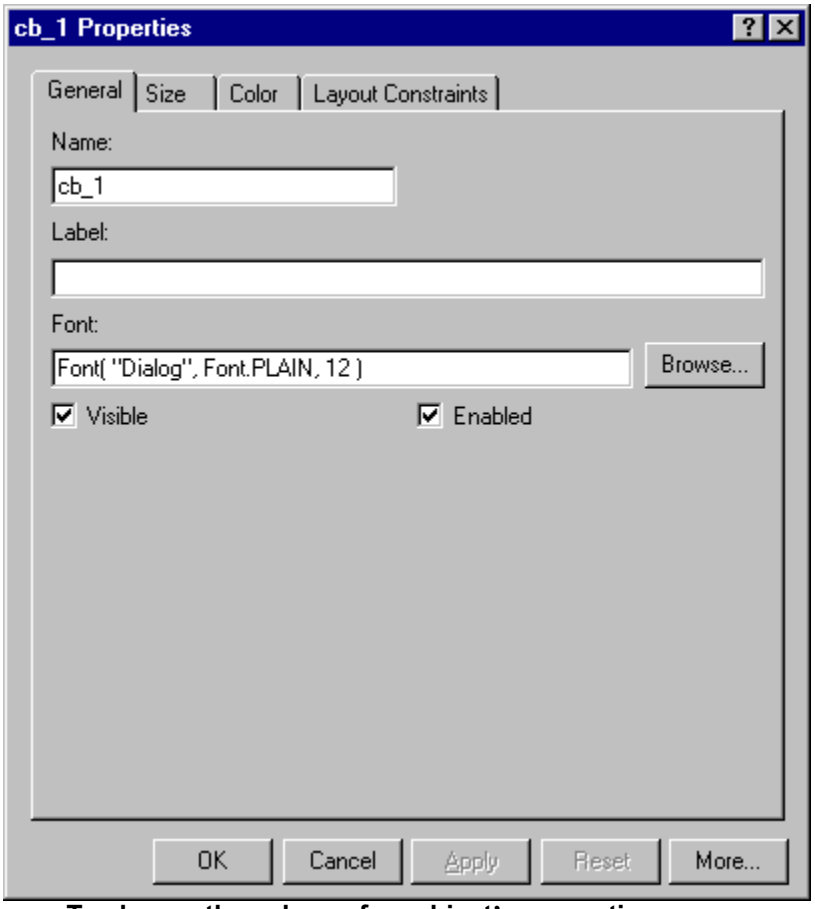
1. On the **Tools** menu on the main PowerJ menu bar, click **Options** and then **Classpath**.

The **Classpath** options page lets you specify any number of folders to be added to your `CLASSPATH` list. These folders are added to the `CLASSPATH` every time PowerJ runs a target. This process is transparent to the user; for example, the folders specified in the classpath options do *not* appear in the Class path properties for a particular target. In other words, the **Classpath** options page specifies folders which are used globally by PowerJ, not specifically by any particular target or project.

Changing an object's properties

Every form and component has a set of associated properties. These properties affect the appearance and the behavior of the object. Different types of objects have different properties.

When you place an object on a form, PowerJ assigns default values to the object's properties. You can assign new values to these properties using the object's *property sheet*. For example, here is a typical property sheet for a command button:



- **To change the values of an object's properties:**

1. In the form design window, use the right mouse button to click the object whose properties you want to change, then click **Properties**. This displays the object's property sheet.
2. Use the property sheet to set values for any properties you want to change. You can click the tabs at the top of the main area of the property sheet to see different types of properties.
3. Click **OK** when you have assigned values to the properties you want to change.

The changes you have made may or may not be shown in the form design window. For example, if you turn off an object's **Visible** property, the object remains visible in the form design window, so you can see that it is still part of the form. However, when you run the program, the object will be invisible, as specified.

Note: The property sheet for an object does not display all the object's properties—it only displays the properties that are most commonly used. Clicking the **More** button on the property sheet opens the

Object Inspector, which allows complete access to all the object's properties.

- Changing an object's name
- Changing the class associated with an object

Changing an object's name

PowerJ automatically assigns a name to each object that you place on the form. You can give the object a different name if you want. For example, you may choose to change a command button's name from `cb_1` to `okButton` so that the name provides a description of what the button does.

If you change the name of an object, PowerJ changes the corresponding names of all associated event handlers. For example, if you change the command button's name from `cb_1` to `okButton`, PowerJ automatically changes the button's **Action** handler from `cb_1_Action` to `okButton_Action`.

PowerJ also changes references to the object in your code. For example, PowerJ automatically changes

```
cb_1.setLabel( "Hello" );
```

into

```
okButton.setLabel( "Hello" );
```

However, PowerJ does not make any changes to:

- Comments
- Quoted strings
- Compiler directives (e.g. `import`)
- User-defined structures that contain elements with the same name as the object

For example, if you have the comment

```
// change the caption on cb_1
```

PowerJ will *not* change the text of the comment to use the new name. Similarly, PowerJ will not change code like

```
System.out.println( "This used to be called cb_1" );
```

PowerJ only changes the code for the form that contains the object. It does not check for occurrences of the old name in the code for other forms or in resources.

If you want to change the items that PowerJ doesn't touch, you must make the changes yourself. For example, you can use the **Find/Replace** item of the code editor's **Search** menu to make a global replacement.

Tip: If you intend to change the name of an object, it is best to change the name as soon as you place it on the form. If you change the name later, you may have to do extra work revising your existing code to use the new name.

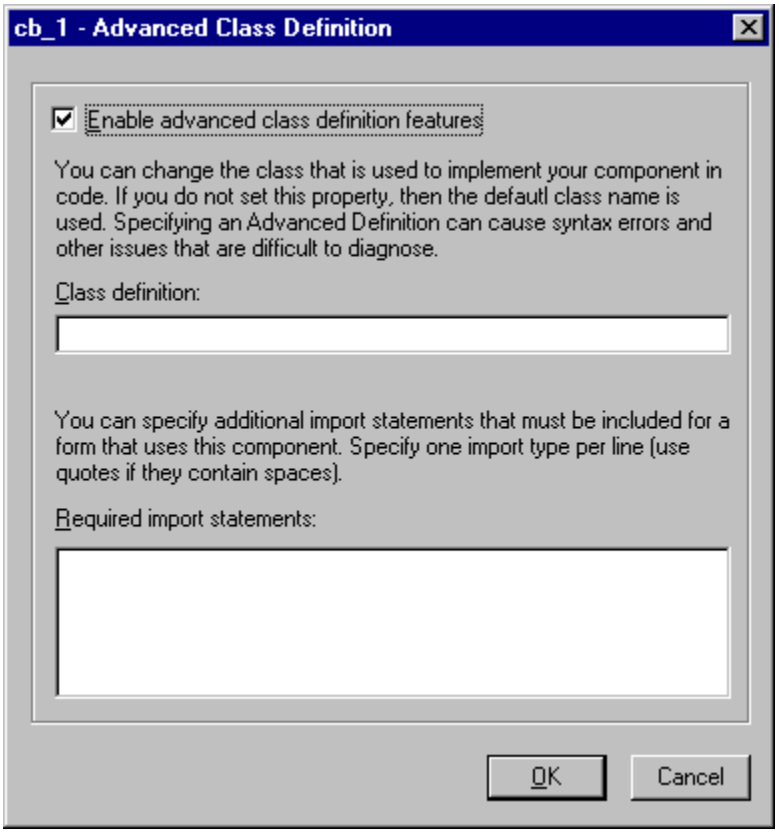
Changing the class associated with an object

Warning: The advanced class definition feature discussed in this section should only be used by experienced PowerJ programmers!

PowerJ associates a class with each type of component. When you build your project, PowerJ generates code to declare and instantiate each object using the associated class.

You can override the default class that PowerJ associates with an object, by using the *advanced class definition* feature of PowerJ, although doing so can lead to errors in your project that are very hard to diagnose. If you are not careful when using this feature, you may create forms that will not compile or that will operate incorrectly.

The advanced class definition feature provides a way to create custom versions of the PowerJ components. It is available under the **Class** property of the Object Inspector; if you click on the ellipsis in the right column, the Advanced Class Definition dialog box for the object opens.



You can use this dialog box to associate a different class with the object. PowerJ will use that class when generating code to declare and instantiate the object. The rest of the design environment is not affected. If you implement the new class in a library and save the modified object as a template, you can reuse the modification while only having to maintain one copy of its source.

Important: For a visual object, the new class must extend the Component class or one of its subclasses. For non-visual objects, this is not necessary. All new classes must have a default constructor.

◆ **To change the class associated with an object**

1. Add the object.
2. Create the new class file.
3. Edit the **Class** property in the Object Inspector to open the Advanced Class Definition dialog box.
4. Use the Advanced Class Definition dialog box to specify the new class and import files.

For example, you might want to add a couple of business rules (some actions you commonly perform) to a list box. First add a list box object to your form. If you run the application at this point and look at the generated source code (in `Release\Form1.java` under the target folder), it should look like:

```
class Form1 extends java.applet.Applet
{
    ...
    /***************************************
     * data members
     ***************************************/

    private java.awt.List lb_1 = new java.awt.List();
    ...
};
```

Next you would use the Classes window to add a new class called MyList that inherits from `java.awt.List` and has a default constructor. Define your business rules in that class as you would for any Java class.

Now open the Object Inspector for `lb_1` and use the **Class** property to open the Advanced Class Definition dialog box. This property is only available from the Object Inspector. Then:

- Under **Class Definition**, type `MyList`.
- Under **Required Import Statements**, type any classes that should be imported by `Form1.java` in order for the form's code to use the MyList object. You only type the names of the classes to import; you do not type complete `import` statements. For example, you might have to type the full name of the MyList class so that `Form1` imports the appropriate class definition. (This is only necessary if MyList is not defined in the same package as `Form1`.)

Now you can rebuild your application. The code that gets generated would now look something like this:

```
class Form1 extends java.applet.Applet
{
    ...
    /***************************************
     * data members
     ***************************************/

    private MyList lb_1 = new MyList();
    ...
};
```

It's almost the same as before, except that the reference to `java.awt.List` has been replaced with `MyList`. The design environment is unchanged, so you can set all the normal list view properties and events, but at run time your code (your constructor, any of the methods you've overridden, etc.) will be used.

Advanced class definition and serializable objects

If you start with an object that is serializable, then use Advanced Class Definition to specify your own class based on the serializable class, you will get serialization errors when you run the target.

This is because the PowerJ design-time environment generates a `.ser` file for the original serializable

object. This `.ser` file usually conflicts with the object that is actually used at run time (the object that you specified with Advanced Class Definition), since the advanced class typically contains more properties and/or methods than the original class described by the `.ser` file.

Adding and modifying event handlers

Every object may have a number of associated *event handlers*. An event handler contains Java code that is executed when a specific event occurs. For example, a command button should have an associated routine specifying what happens when the user clicks the button. This is called the button's **Action** event handler.

Once you have placed an object on a form, you must write routines to handle events that may happen to that object.

- **To create an event handler routine for an object:**

1. Use the right mouse button to click an object in the form design window, then click **Events**. This displays a short menu of events that may occur on the selected object. The menu also contains an entry named **More**.
2. If you click a specific event in the list of events, PowerJ displays a *code editor* which you can use to write a routine to handle that event.
3. If you click **More**, PowerJ opens the Object Inspector on the **Events** page. This lists all the events that can be triggered on the selected object. Double-clicking any of these events opens a code editor to edit an existing event handler or create a new one. For more information on how to use the code editor to edit event handlers, see The PowerJ code editor.

The list of events has check marks beside events which already have event handlers. Some types of objects have event handlers predefined by PowerJ.

The Object Inspector lists all the possible events that can be triggered on an object. In most cases, you will ignore the majority of these events. For example, there are a large number of events that may be associated with a command button. In most programs, however, the only event you care about is when the user clicks on the button. You only need to write a **Action** event handler for the button, and ignore all the other possibilities.

Deleting event handlers

Deleting event handlers

You can delete an event handler using the Classes window.

- **To delete an existing event handler:**

1. Open the Classes window.
2. In the left part of the Classes window, click the form that contains the event handler you want to delete.
3. In the right part of the Classes window, use the right mouse button to click the event handler that you want to delete, then click **Delete**.
4. When PowerJ asks if you are sure you want to delete the event handler, click **Yes**.

You can use a similar technique to delete any user function defined in a form or class.

The PowerJ code editor

The PowerJ code editor helps you write Java source code. PowerJ opens a code editor window whenever you ask to examine or modify an event handler. The same editor is used to edit many other parts of your project, including managed classes.

When you begin writing a new event handler, the code editor displays the skeleton of the routine. For example, the initial code for an **Action** event handler for a command button object, the initial code is:.

```
public boolean cb_1_Action(java.awt.Event event)
{

    return false;
}
```

The most important part of this skeleton is the prototype for the handler. This gives several pieces of information:

- A function name constructed from the name of the object (`cb_1`) and the name of the event (**Action**).
- An argument to the function—this argument is explained in Standard events.
- The type of value returned by the function: a boolean value. This value can be `true`, indicating that the function completely handled the event on its own, or `false`, indicating that the function wants to let the default event handler finish handling the event.

The code editor places the cursor on the blank line after the opening brace (`{`) so that you can begin typing source code for this routine. It also places

```
return false;
```

at the end of the routine. This is the normal way to end an event handler, letting the default event handlers take care of any clean-up and other technical details associated with the event.

Color coding

As you type in your source code, the editor uses color to indicate various elements in your code. For example, if you begin to type a character string, as in

```
"Hello
```

the editor displays the partial string in bright red. When you add the closing quote, as in

```
"Hello"
```

the editor changes the string to a different color. This use of color helps you to remember the closing quote.

For similar reasons, the editor displays comments in bright blue. Reserved words (such as `if` and `while`) are shown in green.

If you prefer a different color coding scheme, the **Tools** menu of the code editor window lets you change the colors for various program elements. Once you have chosen your colors, they remain the same for all projects.

The **Tools** menu also lets you change the tab stops and character font used in displaying code. For more information, see the **Tools** menu and press F1 to obtain a complete description of these facilities.

Line codes

The code editor marks each source code statement with a symbol at the beginning of the line. The following symbols are used:

- Red octagon (stop sign): a debugger breakpoint has been set on this statement.
- Small green circle (like green traffic light): ordinary statement.
- Gray octagon (gray stop sign): there is a breakpoint on this statement, but it is currently disabled.
- White X in red circle: compilation error.
- Black exclamation mark in yellow triangle: compiler warning.
- Small red triangle: line cannot be changed by the user (because it is generated by PowerJ itself).
- No marking: a line that can be changed by the user but currently does not contain executable code (for example, blank lines or comments).

Double-clicking on a green circle sets a breakpoint on the statement. Similarly, double-clicking on a red stop sign removes a breakpoint from the statement. For more information on breakpoints, see Breakpoints.

Saving source code

The usual method of saving source code is to click the **Save Project** button on the main PowerJ toolbar, or to click **Save Project** on the **File** menu of the main PowerJ menu bar. This saves your entire project, including the source code you have just been editing.

The code editor can also *export* the current function to a text file. This saves a copy of the function in a specified file. For example, if you wanted to export a single function from one project to another, you could use this feature to create a file that contains a copy of the function.

Undo and Redo

The **Edit** menu contains an **Undo** command (CTRL+Z) for undoing the most recent editing action. Using this several times in a row undoes the same number of editing actions. There is no fixed limit on the number of steps you may undo; the number is only restricted by the amount of memory available.

The **Edit** menu also contains a **Redo** command which repeats the last step that you undid. If you use **Undo** several times and then the use **Redo** the same number of times, you get back to where you started.

Bookmarks

The code editor automatically places *bookmarks* into your code. For example, it puts bookmarks at the beginning of each event handler routine. This makes it easier to move about your code: if you want to edit a particular routine, go to the appropriate bookmark.

◆ **To go to a bookmark:**

1. Click the arrow beside **Bookmark** at the top of the code editor window, then click the name of the bookmark.

You can also define your own bookmarks, if there are lines of code that you may want to find quickly later.

◆ **To define your own bookmark:**

1. From the **Search** menu of the code editor, click **New Bookmark**.
2. Enter a name for your bookmark, then click **OK**.

The big editor vs. the small editor

The editor can work in two different modes:

- *Small editor mode* in which each code editor window shows the code for a single event handler.
- *Big editor mode* in which a code editor window can show all the code associated with a form.

By default, PowerJ starts off in big editor mode. Big editor mode makes it easier to do global operations. For example, if you want to change the name of a variable, you can go into big editor mode and do a global replace operation, changing the variable's name wherever it appears. On the other hand, small editor mode helps you concentrate on a single routine.

◆ **To change to small editor mode:**

1. From the **Tools** menu of the main PowerJ menu bar, click **Options** and then click **Editor**.
2. Click **Edit each Event with a new editor**.
3. Click **OK**.

When you switch from one mode to another, PowerJ may close code editor windows that you currently have open.

Read-only code

If you are in Big editor mode, you may see code generated by PowerJ as well as code that you wrote yourself. Code that is generated by PowerJ is *read-only*; you cannot change it with the editor. Read-only code is displayed in gray to distinguish it from normal code.

Files not to be edited

A number of files associated with a project begin with a warning that the files should not be edited. In particular, the `.java` source files stored in the `Release` subfolder of a target folder have this kind of warning.

The warning is present because the files are generated automatically from other files whenever the target is built. For example, `.java` source files are generated automatically from form description files (`.wxf`) or managed class files (`.wxc`). If you change a `.java` file, the changes will simply be lost the next time the target is built, since the `.java` files will be freshly generated again.

If you want to change Java source code, you must edit the appropriate form file or class file instead of the `.java` file. For example, if you want to add a method to a Java class, add the method through the Classes window rather than directly editing the Java source file.

Working with classes

This section describes how to add member functions to classes, how to modify existing functions, how to define new data members, and how to import classes that are needed by the code in a class.

Adding new member functions

Until now, this guide has concentrated on writing functions that are directly related to handling events on objects. However, you may need to define other functions within a form class or a managed class.

For example, if several event handlers have to perform the same set of actions, it makes sense to create a separate function which performs those actions. Then all the event handlers can call that function to perform the appropriate work.

You may also need to define an event handler for an object that doesn't exist at design time. For example, if you add a new object to the form during execution, you can't create event handlers for that object in the normal way. Instead, you create a special member function within the form; then when you create the object during execution, you can register the member function as an event handler for the new object.

Member functions can be divided into two classes:

- A *method* is a member function that you want to make available for use outside the class. Methods are always `public`. There may be several overloaded versions of a method.
- A *user function* is a member function that is intended for the internal use of the class itself. User functions are typically `private` or `protected`, although they can also be `public`. Usually, you do not overload user functions.

The difference between the two is primarily the way in which you intend to use the function, not necessarily the way they are defined in the source code. In general, this guide uses the generic term "function" or "routine" to refer to both methods and user functions.

◆ **To create a new method for a class:**

1. In the Classes window, use the right mouse button to click on the class; click **Insert**, then click **Method**.
2. Under **Name**, type the name for the new method (only the name, not the prototype).
3. Under **Prototype(s)**, type one or more prototypes for the method.
4. Click **Finish**.

PowerJ opens a code editor window so that you can begin typing the definition of the new method.

The process for creating a user function is similar.

◆ **To create a new user function for a class:**

1. In the Classes window, use the right mouse button to click on the class; click **Insert**, then click **User Function**.
2. Type one or more prototypes for the function in **Function Prototype**.
3. Under **Scope**, click the scope of the new function (private, protected, or public).
4. Click **OK**.

PowerJ opens a code editor window so that you can begin typing the definition of the new user function.

Renaming a member function

If you add a method or user function to a form class, you can change the name of the function through the Classes window.

- **To rename a member function previously added:**

1. In the Classes window, use the right mouse button to click on the name of the function you want to change, then click **Rename**.
2. Edit the existing function name to change it to the new name.
3. Press ENTER when you have changed the name.

This technique only works for methods and user functions. You cannot rename event handler functions (although the name of an event handler will change automatically if you change the name of the associated object).

When you change the name of a member function, the change is *not* made in any of your program's source code (except for the heading of the function definition). For example, if you change a function name from `myfunc` to `yourfunc`, any existing calls to `myfunc` do not change. You must change the code explicitly, typically with **Find/Replace** from the **Search** menu of the code editor. Another way to find references to the old name is just to compile the project, then check for error messages resulting from uses of the old name.

Adding data members to a form definition

The Classes window lets you declare new data members inside a form class or a managed class. If you look at the window, you will see an entry named **Class Declarations**. To declare a new data member for the current form, double-click the **Class Declarations** entry. PowerJ displays a window where you can type an appropriate declaration for the new data object. For example, you might type

```
private int i;
public double x;
```

to declare extra data members of the form class.

Deleting events, classes, objects, and other items

PowerJ uses the same approach for deleting all items used in your session: use the right mouse button to click the item, then click **Delete** in the resulting menu. The following list shows examples:

- To delete an existing user function or event handler from a form definition, use the Classes window to display the items defined in the form class. Use the right mouse button to click the user function you want to delete, then click **Delete** in the resulting menu.
- To delete an object from the current form, use the right mouse button to click on the object in the form design window, then click **Delete** in the resulting menu. You can also delete the object using a similar technique in the Objects window.

Importing classes

The Classes window lets you specify classes which you want to easily access in your code. By *importing* a class, you can refer to it in your code without prefixing it with its package name.

- **To import a class:**

1. In the Classes window, click the name of the class where you want to import another class, then double-click **Imports** for that class.
2. In the resulting code editor window, enter an appropriate `import` statement to import the class or all classes in a package.

For example, if you want to refer to the `java.util.Random` class simply as `Random` in your code, type:

```
import java.util.Random;
```

If you want to be able to refer to all classes in the `java.util` package without using the package name, type:

```
import java.util.*;
```

Specifying package names

You can specify a package name for a managed class, form, or other type of class that you create with PowerJ.

- **To specify a package name for a class:**

1. In the left part of the Classes window, use the right mouse button to click the name of the class, then click **Properties**. This opens a dialog box specifying the current properties for the class.
2. Under **Package Name**, type a package name for the class.
3. Click **OK**.

Important: The package for a form must match the package for the class corresponding to the name of the target itself. (These classes both appear in the Classes window.) If you specify different packages for the two classes, you will get compilation errors when you try to build the target.

Using drag-and-drop programming

Drag-and-drop programming makes it easy to construct expressions that refer to objects on a form. For an example of drag-and-drop programming in action, see the *Getting Started* guide.

Principles of drag-and-drop programming

In drag-and-drop programming, you can start the drag operation from a form design window or the Objects window.

- **To use drag-and-drop programming:**

1. Position your windows so that you can see the code editor window and the form design window or Objects window.
2. Drag the cursor from an object on the form (or in the Objects window) into the code editor window. This opens the Reference Card, positioned at the methods for the appropriate type of object.
3. Examine the various categories for the object in the Reference Card. Expand a category to list the methods and properties in that category.
4. Click the method or property you want to use.
5. If there are several overloaded versions of the same function, click the version you want from the list at the bottom of the Reference Card.
6. Click the **Parameters** button. This opens the Parameter Wizard.
7. Fill in the blank entries displayed by the Parameter Wizard. Click **Finish** when you're done.

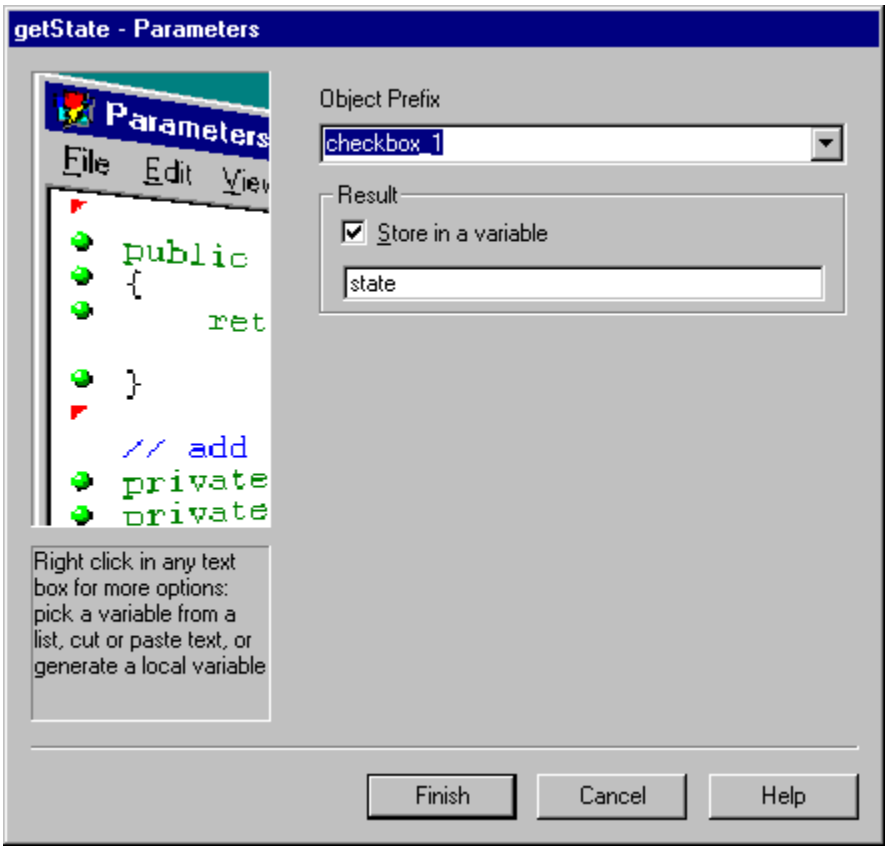
This places an appropriate expression or statement into your source code at the position indicated by the cursor.

Note: If you do not have a code editor window open, you cannot open the Parameter Wizard. When you select an entry from the Reference Card, the **Parameters** button is grayed out so that you cannot press it. There is no point in calling the Parameter Wizard if it has nowhere to place the code that it constructs.

Dragging from the Objects window is often more practical than dragging from a form design window, especially if you are running short of space on your monitor screen: these windows often take up less space than a form design window. Furthermore, you can't drag from the form itself to a code editor window. Therefore, if you want to use a method on the form as a whole, the only way to use drag-and-drop programming is to drag from the Objects window.

Methods that return values

If the method you select returns a result value, the Parameter Wizard asks if you want this result assigned to a variable. If you click **Store in a variable**, the Parameter Wizard creates a suitable variable declaration as well as a statement that assigns the method result to that variable. For example, suppose you use **getState** to determine if a check box is checked. The Parameter Wizard displays the following:



When you click **Finish**, the Parameter Wizard generates the following:

```
boolean state;
state = checkbox_1.getState();
```

This calls **getState** method to determine if the button is checked and returns the result to a boolean variable named `state`.

The object prefix

The Parameter Wizard always has an area where you can enter an *object prefix*. This specifies the object to which you want to apply the method.

When you are using drag-and-drop programming, PowerJ fills in the object prefix with the name of the object where the drag-and-drop operation started.

If you use the Reference Card to generate code directly, without using a drag-and-drop operation from the form design window, you can select the object prefix by clicking the arrow under **Object Prefix** and choosing an object name from the resulting list.

You can also type an object name directly into **Object Prefix**. The Parameter Wizard generates a function call of the form

```
object.function()
```

Note: If you are performing a method on the form itself, the object prefix will be blank. This is because methods are assumed to act on the form if you do not specify an object explicitly. For example,

```
texta_1.setBackColor( Color.blue );
```

sets the background color of the specified text area, but

```
setBackColor( Color.blue );
```

sets the background color of the form itself.

PowerJ command line options

When you start PowerJ, there are several options that can be specified on the command line:

`-bt "Debug" -b file.wxj`
Immediately builds a `Debug` version of the project specified by the `.WXJ` file.

`-bt "Release" -b file.wxj`
Immediately builds a `Release` version of the project specified by the `.WXJ` file.

`-b file.wxj`
Immediately builds the project specified by the WXJ file. Each target will either be a `Debug` or `Release` version: whichever type was selected the last time each target was built.

`-c file.wxj`
Deletes all the generated files associated with the project specified by the WXJ file. This option is a good way of cleaning out target folders in preparation for a complete rebuild.

For example, the following command line starts PowerJ and immediately builds a `Release` version of a project:

```
optima -bt "Release" -b c:\PowerJ\projects\myproj\myproj.wxj
```

If you are invoking PowerJ from a console, you should add `start` at the beginning of the command line to avoid leaving the console busy and to allow PowerJ to be found if it's not in the path:

```
start optima -c c:\Power\projects\myproj\myproj.wxj
```

Note: In the `-bt` options, you must type either `"Release"` or `"Debug"` as shown; for example, you cannot use all lowercase letters.

Batch processing with Batch PowerJ

The Batch PowerJ facility lets you build PowerJ projects in a batch environment. For example, suppose that you have several programmers working on a shared PowerJ project. You might designate a particular computer as the project's "build machine": the site where you combine the source code produced by the programmers and build it into a unified project. Batch PowerJ lets you do this using batch processes rather than the usual PowerJ development environment. In this way, you can automate the build process with .BAT files, making it unnecessary for someone to interact manually with PowerJ.

Batch PowerJ reads the files of your project and compiles your code using the same options and tools that are used by the full PowerJ environment. All output from the building process is printed to the standard output.

Installing Batch PowerJ

Batch PowerJ is installed as part of a normal PowerJ installation. If you are setting up a build machine where Batch PowerJ will be run but PowerJ will not, you can save disk space by following these directions:

Do not install Sybase SQL Anywhere, Sybase SQL Tools, Art Gallery or Component Gallery.

Do not install Sample Programs, PowerJ Help or JDBC Drivers. These options are located on the dialog displayed when you choose the **Options** button on the **Select Components** dialog in the installation procedure.

After the installation finishes, you may delete the following files:

```
system\optima.exe
system\dt*.dll
system\wedit*.dll
templates\*.*
```

Preparing a target for use with Batch PowerJ

Batch PowerJ is not as powerful as PowerJ. It requires more files than PowerJ to build a target. To determine the files required by Batch PowerJ to build your target, follow these steps.

1. Load your project into PowerJ and open the Targets window.
2. Under **Target type** in the Targets window, click **Debug**.
3. Save the project.
4. In the **Run** menu of the main PowerJ menu bar, click **Clean**.
5. Close the project.

You can now determine which files are required by Batch PowerJ by examining the target folder. Batch PowerJ needs:

- All files from the target folder except the .WXU file.
- All files from the target's `Release` subfolder except the .IDX files.

If these files are not present, Batch PowerJ cannot build the target.

Batch PowerJ command line options

Batch PowerJ accepts command lines of the following form:

```
boptima options projectfile options
```

The `projectfile` is the name of the .WXJ or .WXP file for the project to be built. This must always be specified. As shown above, options may be specified before and/or after the project file name.

The `options` may be a combination of the following:

`-d`
: Builds the debug version of the targets in the project.

`-r`
: Builds the release version of the targets in the project.

  You may not specify both `-r` and `-d`. If you do not specify either, Batch PowerJ builds the release targets.

`-t=targetfile`
: Only builds the specified target and any targets on which it depends. The `targetfile` value should be the name of the .WXT file for the target.

  If you do not specify a `-t` option, Batch PowerJ builds all the targets in the project.

`-a`
: Builds files even if they appear to be up to date.

`-c`
: Cleans the project instead of building it. Cleaning deletes any files that Batch PowerJ may have generated in previous build operations.

Chapter 4. Debugging

This chapter discusses the debugging facilities of PowerJ, and provides several suggestions for improving general program performance.

Introduction to debugging

Finding bugs is an art, not a science. However, PowerJ offers a variety of ways to examine your program as it runs, making it easier to track down where things are going wrong. Staring at source code in the hope that you notice a mistake is not nearly as productive as stopping your program in the middle of execution and looking at data values to make sure they're correct. Even better is the ability to mark potential trouble spots in your code and have your program automatically report when something goes wrong.

This chapter examines the debugging tools of PowerJ and suggests a few simple ways that you can use them. However, no book can cover all the possible tricks a programmer might use in tracking down a particular bug. We strongly recommend that you experiment with the debugging facilities on simple programs, to see how the tools work and how you can use them productively.

Most of this chapter describes debugging of Java code in PowerJ or debugging facilities that are language-independent. PowerJ can also debug DynaScript, the scripting language used by NetImpact Dynamo. For specific information on debugging DynaScript in PowerJ, see A Dynamo WebApplication and Setting DynaScript breakpoints.

Specifying your interpreter

The debugging facilities of PowerJ interact with the Java virtual machine (interpreter) that you are using to run your Java code.

- The Microsoft SDK for Java 1.5 virtual machine provides debugging services for JDK 1.02 targets.
- The Microsoft SDK for Java 2.0 virtual machine provides debugging services for both JDK 1.02 and JDK 1.1 targets. (This virtual machine is not included with PowerJ, but if you have it installed PowerJ will automatically use it in place of the 1.5 VM. You can install the Microsoft SDK for Java 2.0 by installing Microsoft Internet Explorer 4.)
- The Sun virtual machine only provides debugging services for JDK 1.1 targets.

To specify which virtual machine you want to use, set the run options for the target you are debugging.

◆ **To set run options for a target:**

1. From the **Run** menu of the main PowerJ menu bar, click **Run Options**.
2. If you have more than one target in the current project, click the target whose run options you want to set, then click **OK**.

You specify the virtual machine for an applet or application on the **General** page of the run options dialog.

Debugging limitations of the Sun virtual machine
Debugging consoles

Debugging limitations of the Sun virtual machine

Because of limitations in Sun's Java virtual machine, you may encounter some limitations to the debugging facilities that can be provided for applications using that interpreter:

- If you set a breakpoint on a line with no code, the debugger will just ignore it.
- Sometimes the application hangs up and refuses to run.
- Tracing a `for(;;)` statement traces over the whole loop.
- Debugging applications that run under the Sun virtual machine tends to be very slow.
- The Assembly window is disabled when running with the Sun virtual machine.

Debugging consoles

If you do any testing with Internet Explorer and need to look at the console, you may find the Java Console Logging utility useful. If you have Java logging enabled, anything written to `javalog.txt` is displayed on the console.

You can download this utility from:

`http://www.trevorharmon.com/visualj/javaconsole/index.html`

There is a different Internet Explorer console available from:

`http://www.obsolete.com/people/cansdale/java/java_console/index.htm`

This console has a number of useful features, including Thread display and applet information.

Important: The utilities discussed above are not supported or endorsed by Sybase. As with any software that you download from the Net, use it at your own risk.

Netscape Navigator also provides some useful console commands:

```
c:   clear console window
d:   dump applet context state to console
f:   finalize objects on finalization queue
g:   garbage collect
h:   print this help message
m:   print current memory use to console
q:   hide console
s:   dump memory summary to "memory.out"
t:   dump thread info to "memory.out"
x:   dump memory to "memory.out"
X:   dump detailed dump memory to "memory.out"
k:   checkpoint memory to "mem.out" (Windows only)
w:   dump GlobalAlloc blocks to "heaps.out" (Windows only)
0-9: set applet debug level to <n>
```

These commands are supported under Netscape Navigator 3.02 and later.

Debugging session options

- PowerJ offers you several options for controlling your debugging sessions.

◆ **To view your current debugging options:**

1. From the **Tools** menu on the main PowerJ menu bar, click **Options** and then **Debug**.

The **Debug** page shows the following options:

Make editors read-only while debugging
If you turn on this option, PowerJ will not let you change code while you are running a target.

Use a single editor while debugging
If you turn on this option, PowerJ only uses a single code editor window while you are running a target. When the code editor is displaying one source file and the debugger needs to show a different file (for example, because a breakpoint has been triggered in the second file), PowerJ closes the first file and opens the second in the same code editor window. This avoids the confusion of having multiple code editors open at the same time.

Hide form painters while debugging
If you turn on this option, PowerJ temporarily makes all form design windows invisible when you run a target. This reduces the number of windows on the screen during a run and prevents the possibility of confusing design-time forms with forms that are used by the running program. When the run is finished, PowerJ makes the form design windows visible again.

Breakpoints

A *breakpoint* is a point in your executable code where you want to break (interrupt) the normal execution of your program while you are debugging. For example, you might set a breakpoint at the start of a function, so that the program temporarily stops executing whenever that function is called.

While your program is stopped at a breakpoint, you can examine data objects used by the program. PowerJ also lets you change data values, examine your program as assembler code, and investigate your program in other ways. You can resume executing the program from the breakpoint by stepping through the code or letting it run until completion or the next breakpoint.

Setting a simple breakpoint

The code editor makes it easy to set a breakpoint on any code instruction. The following rules apply:

- You cannot set a breakpoint on a blank line or a comment line.
- If you set a breakpoint on the prototype that begins a function, the break occurs when the function is called.
- If you set a breakpoint on the closing brace that marks the end of the function, the break occurs when the function returns.
- If you set a breakpoint on any other line, the break occurs immediately *before* the line is executed.

◆ **To set or remove a breakpoint for a line of code:**

1. In the code editor, use the right mouse button to click on the line.
2. Click **Toggle Breakpoint**.

You can also toggle a breakpoint on or off for an executable line by double clicking on the icon at the beginning of the line.

When PowerJ has a breakpoint on a line, it places a red stop-sign icon at the start of the line. Lines without breakpoints have small green circle icons, suggesting green traffic lights. A line with a disabled breakpoint has a gray stop sign icon.

You can control the behavior of a breakpoint by specifying commands in the *Breakpoints* dialog box. For further details about the Breakpoints dialog box, see Breakpoints dialog box.

Note: If you set a breakpoint in the middle of a statement that is broken over several lines, the breakpoint moves to the first line of the statement when you actually run the program. For example, if you have the statement

```
i = j +
    k;
```

and set the breakpoint on the second line, the breakpoint moves to the first line when you run the program.

When a breakpoint is encountered

If the executing program reaches a breakpoint, the following steps occur:

1. If you have specified a count (with the **After 'n' times** option), a counter is increased by one. If it is then less than the count you specified, the following steps are skipped and program execution resumes.

   If no condition or count is set, or the condition and count are met, the breakpoint action is *triggered* by proceeding to the next step.
2. Program execution is suspended.

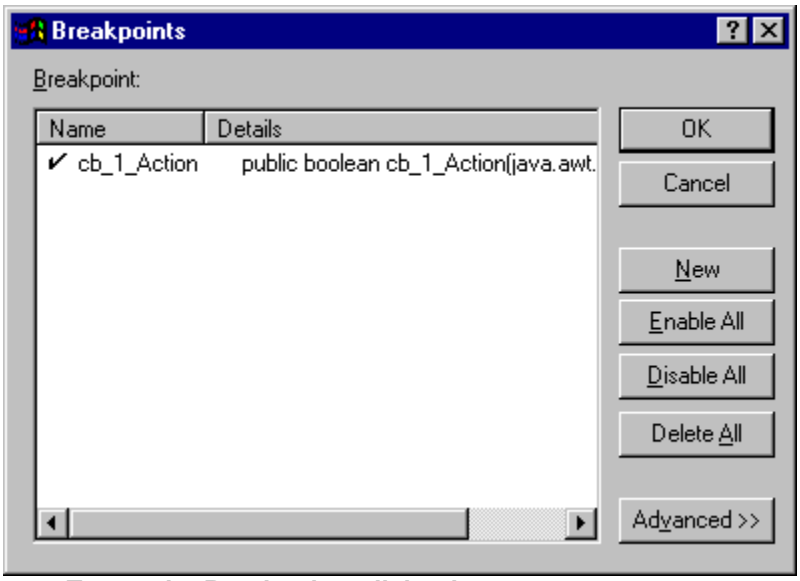
For information on breakpoint conditions and code patches see Advanced breakpoint options.

When a breakpoint is encountered that suspends execution, PowerJ opens a code editor window to show the source code that contains the breakpoint. The red stop sign icon marking the line with the breakpoint now has a yellow pointer in it, pointing to the line where the breakpoint is triggered. This makes it easier to identify where you are if you set several breakpoints in the same region of code.

While execution is suspended at a breakpoint, you can access numerous debugging tools through the **Debug** menu of the code editor. Some of these tools are also available through buttons on the toolbar, either in the main PowerJ window or in the code editor window.

Breakpoints dialog box

The *Breakpoints* dialog box displays all the breakpoints that are currently set in your program.



- **To see the Breakpoints dialog box:**

1. From the **Run** menu of the main PowerJ menu bar, click **Breakpoints**.

When you use the right button to click the name of an entry in the breakpoint list, PowerJ opens a menu of operations that can be performed. For example, clicking **Show Code** in this menu opens a code editor which displays the code that contains the breakpoint.

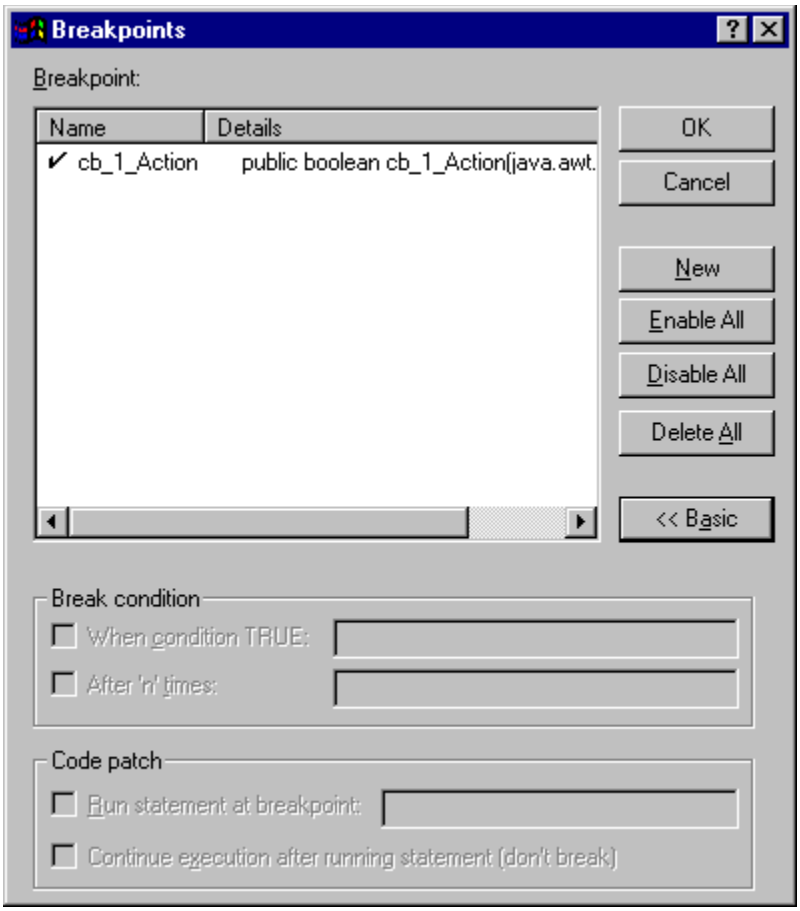
As another example, **Disable** in this menu disables the breakpoint without removing it. This means that the breakpoint will not go off if execution ever reaches that point. However, you can enable the breakpoint just by clicking **Breakpoint is Enabled** again (so that the check box is checked). Enabling or disabling breakpoints through the Breakpoints dialog box can be faster than searching for the corresponding statement in a code window.

Note: Another way to disable a breakpoint is to click on the check mark beside the breakpoint. This makes the check mark disappear. You can enable the breakpoint again by clicking on the blank area where the check mark was.

A disabled breakpoint has a gray stop sign icon in the code editor window.

Advanced breakpoint options

If you click the **Advanced** button of the Breakpoints dialog box, PowerJ expands the dialog box to offer more options.



When condition TRUE:

If you check this option and type a Java expression in this box, you create a *conditional breakpoint*. Whenever execution reaches the given point, PowerJ evaluates the specified expression. If this expression is `TRUE` (non-zero), PowerJ triggers the breakpoint normally. However, if PowerJ finds the expression is `FALSE` (zero), PowerJ does not trigger a breakpoint; it resumes normal execution immediately. This makes it possible to create a breakpoint that only has effect when certain criteria are met.

After 'n' times:

If you check this option and type a value in this box, you create an *occasional breakpoint*. Each time PowerJ passes through the breakpoint position, it increments a counter by `1`. When the counter finally reaches the value *n*, PowerJ triggers the breakpoint. For example, if you type a value of `10` for *n*, PowerJ only triggers the breakpoint after ten passes through the breakpoint position. Typically, you set this kind of breakpoint inside a loop in your code, making it possible to check on the progress of the loop without stopping the program for every iteration.

If you specify **After 'n' times** and **Break when condition TRUE** together, the breakpoint only happens after *n* executions when the condition is true. For example, suppose you specify

```
Break when condition TRUE: p != NULL
After 'n' times: 5
```

Each time execution passes through the break location, PowerJ checks whether *p* is null. If *p* is null, this execution does not count toward the count of *n*=5. The breakpoint only suspends execution on the fifth time that PowerJ finds a non-null *p*.

Run statement at breakpoint

If you check this option and type a Java statement in this box, PowerJ executes the given statement when the breakpoint is triggered. For example, you could type a statement that automatically sets a data object to a certain value if the breakpoint is reached.

Continue execution after running statement (don't break)

If you check this option, the breakpoint doesn't stop execution when it is triggered. It simply performs the statement specified in the previous line, then resumes execution. Combining this selection with the previous one, you can change the value of a variable when the breakpoint is hit, then resume execution. For example, you might do this to simulate the effects of a function that you haven't written yet.

Setting DynaScript breakpoints

As well as Java code, PowerJ can also debug DynaScript, the scripting language used by NetImpact Dynamo. For an example of how to debug DynaScript in PowerJ, see A Dynamo WebApplication.

If you want to set breakpoints in DynaScript `<!-- SCRIPT -->` tags, you must be using PowerJ's default editor to view HTML code. To determine whether you have PowerJ configured to use the default editor, consult PowerJ's File Types options.

- **To ensure that you are using the default HTML editor:**

1. On the **Tools** menu of the main PowerJ menu bar, click **Options**. This opens the Options dialog.
2. On the **File Types** page, click **.html file**, then click **Edit**.
3. Under **Editor**, click **Default editor**.
4. Click **OK** to close the File Types dialog, then click **OK** to close the Options dialog.

If you are not using the default editor, you will not be able to set breakpoints in DynaScript.

Debug windows

This section examines a number of the windows of debugging information available under the **Debug** menu of the code editor.

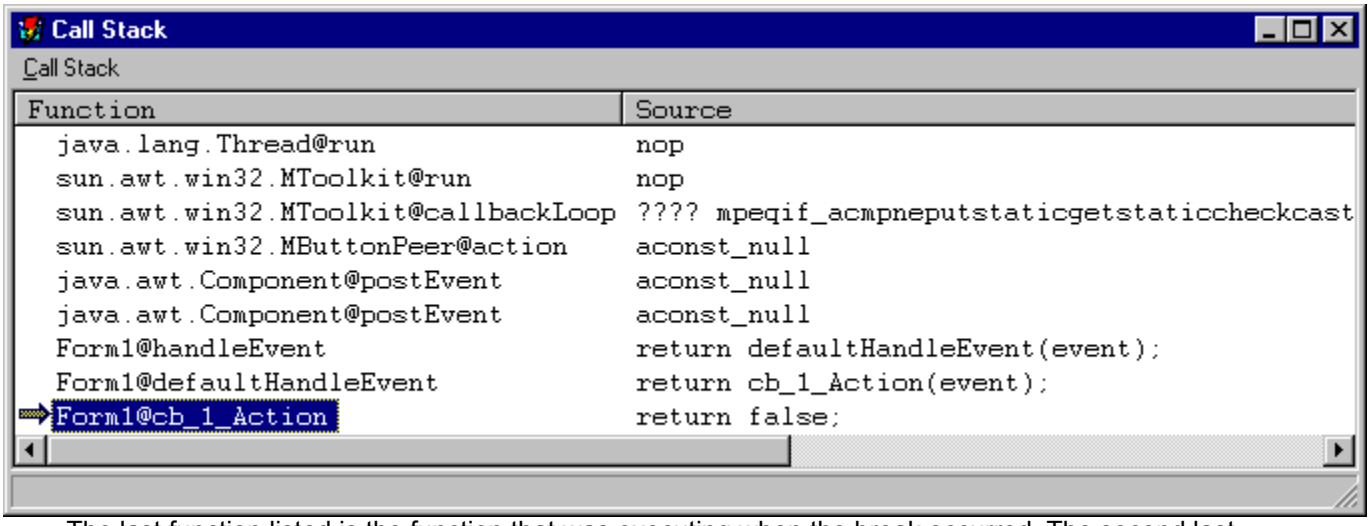
Note: These elements are only available while you are running your program under control of the PowerJ debugging facilities. Therefore, all the elements are defined relative to a particular point in program execution. For example, if your program is stopped at a breakpoint, these elements can show you information about the program relative to that breakpoint.

Call Stack window and commands

The **Debug** menu contains several entries that let you track the sequence of function calls at this point in execution.

Call Stack

Displays the sequence of function calls leading up to the function that was executing at the time of the breakpoint.



The last function listed is the function that was executing when the break occurred. The second last function is the one that called the last function; the third last function called the second last, and so on up the list. A pointer in the Call Stack list indicates the function you are currently examining.

You can also use items on the **Debug** menu to examine different functions in the call stack.

Up Call Stack

Displays information about the next function up the Call Stack. For example, if you stop at a breakpoint inside the function `cb_1_Action`, **Up Call Stack** displays information about the function that called `cb_1_Action`. If possible, PowerJ opens a code editor window to display the function; however, if PowerJ does not have Java source code for the function, PowerJ shows an assembly code version of the function. (Note that the assembly code version is only meaningful with the Microsoft virtual machine.)

Down Call Stack

Displays information about the next function down the Call Stack. For example, if you have used **Up Call Stack** to display the function that called `cb_1_Action`, **Down Call Stack** moves back down to `cb_1_Action` again.

Bottom Call Stack

Returns to the bottom of the call stack. This is the function that was executing at the time the break occurred (for example, `cb_1_Action`).

When you change positions in the call stack, all of your debugging windows change to display information about the new function. For example, the Locals window changes to show the local variables in the new function.

If you use the right mouse button to click an entry in the Call Stack, PowerJ displays a menu of actions you can perform on that entry:

Toggle Breakpoint

Sets a breakpoint at the instruction immediately following the point where the entry called the next routine on the call stack. Therefore, the breakpoint goes off as soon as execution returns to that function. This may be in the middle of a Java statement.

If there is already a breakpoint at that location, **Toggle Breakpoint** turns off the breakpoint.

Run to Cursor

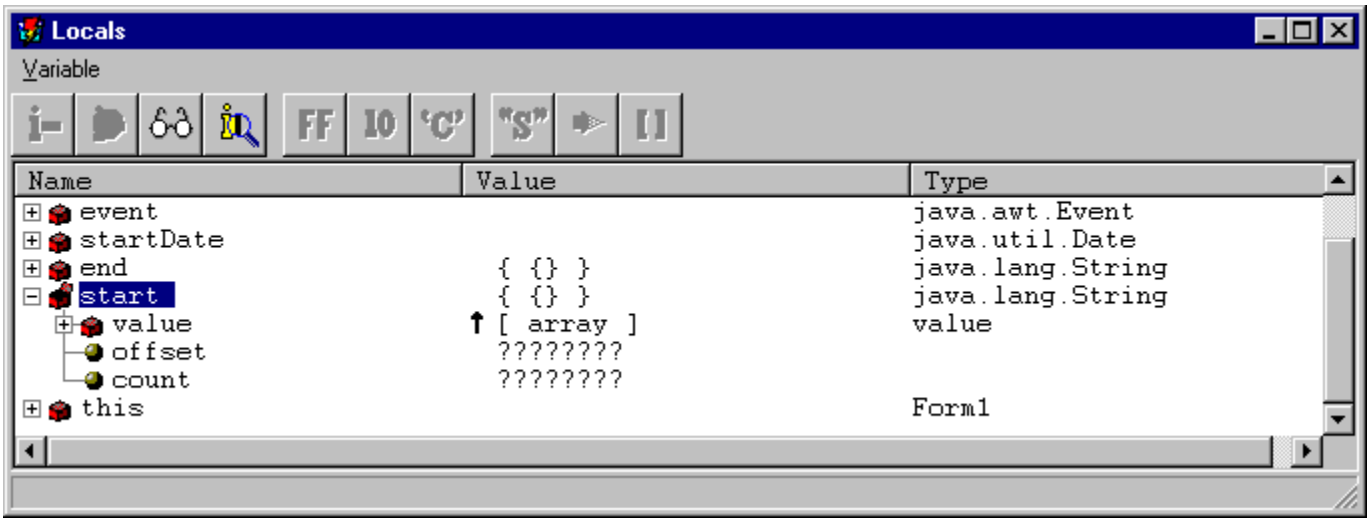
Runs the program up to the instruction immediately following the point where the entry called the next routine on the call stack. Therefore, execution stops as soon as it returns to the specified function. This may be in the middle of a Java statement.

Up Call Stack to Cursor

Moves up the call stack to the routine that you have clicked. This has the same effect as a sequence of **Up Call Stack** operations, until you reach the selected routine.

Locals window

The *Locals* window displays all the local variables defined in the function that is currently executing.



- **To see the Locals window:**

1. From the **Debug** menu of the code editor, click **Locals**.

Each line in the Locals window gives the name and value of a local variable. Variables with special types are marked with appropriate symbols:

- Objects are marked with boxes: red for member objects defined in the current class, and blue for base classes.
- Simple values like integers are marked with dark yellow balls.

The Locals window is organized as a tree view. If you click the **+** sign beside an object, the window displays the contents of the object.

The Locals window shows all the variables declared in the current scope, even if those variables have not been encountered by the time of the current breakpoint. For example, suppose your code contains

```
int i = 1;
int j = 2;
```

and suppose that you set a breakpoint on the declaration of `i`. At this point in the code, you might think that `j` doesn't exist because execution has not reached the declaration of `j`. However, the Locals window lists `j`, because it is declared in the current scope (even if it has not been declared yet). Since `j` has not been assigned a specific value at the time of the breakpoint, the value shown in the Locals window is not meaningful.

For performance reasons, the Locals window does not display variables defined in a parent class. To see variables defined in a parent class, you can use the Watches window. For information on the Watches window, see Watches window.

The Variable menu

The **Variable** menu of the Locals window offers a number of operations that can be performed on the variables shown in the window. Before you can use any of these operations, you must click on a

specific variable name; then you have a number of possible options.

Modify
Prompts you to type a new value for the variable. When you resume program execution, the variable will have the new value.

Inspect
Opens a new window containing a tree view of the specified variable. The new window is similar to the Locals window but only displays information about the single variable or data member.

You can also open an Inspect window for a variable by using the right mouse button to click on the variable in the code editor and choosing **Inspect '***variable***'** from the context menu.

Watch
Opens the *Watches List* to set up the variable as a *watch expression*. For further information, see Watches window.

Show
Displays a section of memory.

Type
Can display the value of the variable in different formats. If you ask to display the value as an array, PowerJ pops up a dialog box to ask which array elements you want to see.

Class
Lets you specify what kind of information you want to see when the Locals window displays class objects. This only applies to objects with the same class as the selected item. For example, if you select one variable and click **Class/Show Functions**, the Locals window displays functions associated with the selected variable and any other variables of the same class.

Field on Top
Specifies the "key" value or values for a data object. When the Locals window displays an unexpanded version of the object or of pointer to the object, it shows the on-top value as the "value" of the object. For example, consider a String object: this object contains a number of data members, but usually, you're most interested in the text of the string. Therefore, the data member containing the text is marked as the **Field on Top**; when the Locals window displays an unexpanded version of the string, it shows the text as the string's value.

Inside any object, you can click any data member and mark it as the **Field on Top**. The value of the data member is marked with a dark arrow and is displayed as part of the value of the object that contains the data member.

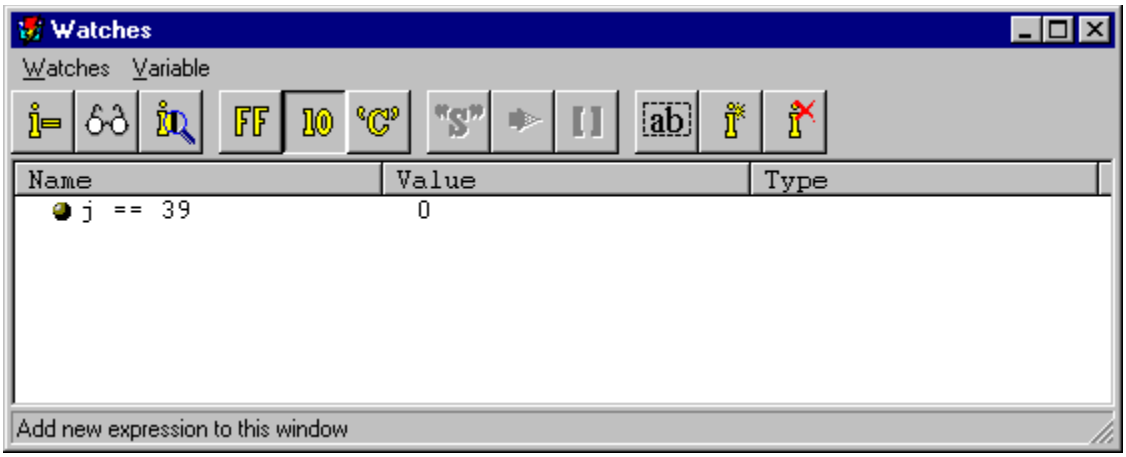
When object A contains object B, object A does not inherit the **Field on Top** settings from object B. You must set the **Field on Top** setting for the item in the tree of A which has type B.

The toolbar of the Locals window supplies buttons that perform the same actions as certain items from the **Variable** menu.

If you use the right mouse button to click an entry in the Locals window, PowerJ displays a context menu offering the same choices as the **Variable** menu.

Watches window

A *watch expression* is a Java expression whose value is displayed by PowerJ during program execution. The *Watches* window displays the current value of all the watch expressions.



- **To see the Watches window:**

1. From the **Debug** menu of the code editor, click **Watches**.

When you first begin program execution, the Watches window is empty. You can add watch expressions to the list by copying them from the Locals window (described previously) or by adding a new expression directly.

- **To add a variable to the Watches window:**

1. Use the right mouse button to click on the variable in the code editor.
2. From the context menu, click **Watch '***variable***'** where *variable* is replaced with the text that you clicked on with the right mouse button.

PowerJ opens the Watches window and adds the variable to the end of the list. The list also displays the current value of the variable.

- **To add a new watch expression to the Watches window:**

1. From the **Watches** menu of the Watches window, click **Add**. PowerJ displays a dialog box where you can type the expression.
2. Type the watch expression. This can be any Java expression; if it is not valid in the current program context then question marks are shown for the value.
3. Click **OK** when you have typed the watch expression.

Another way to add a new watch expression is to select an expression in a code editor window, use the right mouse button to click the expression, and then click **Watch**.

PowerJ adds new watch expressions to the end of the Watches window. The list also displays the current value of each expression.

Other entries in the **Watches** menu let you modify or delete expressions that are currently in the Watches window.

The Watches window has a **Variable** menu with a **View** menu; these duplicate the menus of the Locals window. If you use the right mouse button to click an entry in the Watches List, PowerJ displays a

context menu offering the same choices as the **Variable** menu.

Parent class instance variables

Unlike the Locals window, the Watches window can display variables defined in a parent class. For example, consider the following code:

```
class superClass
{
    int superField;
};

class subClass extends superClass
{
    int subField;
    void SubMethod()
    {
        // ==> breakpoint here
    }
};
```

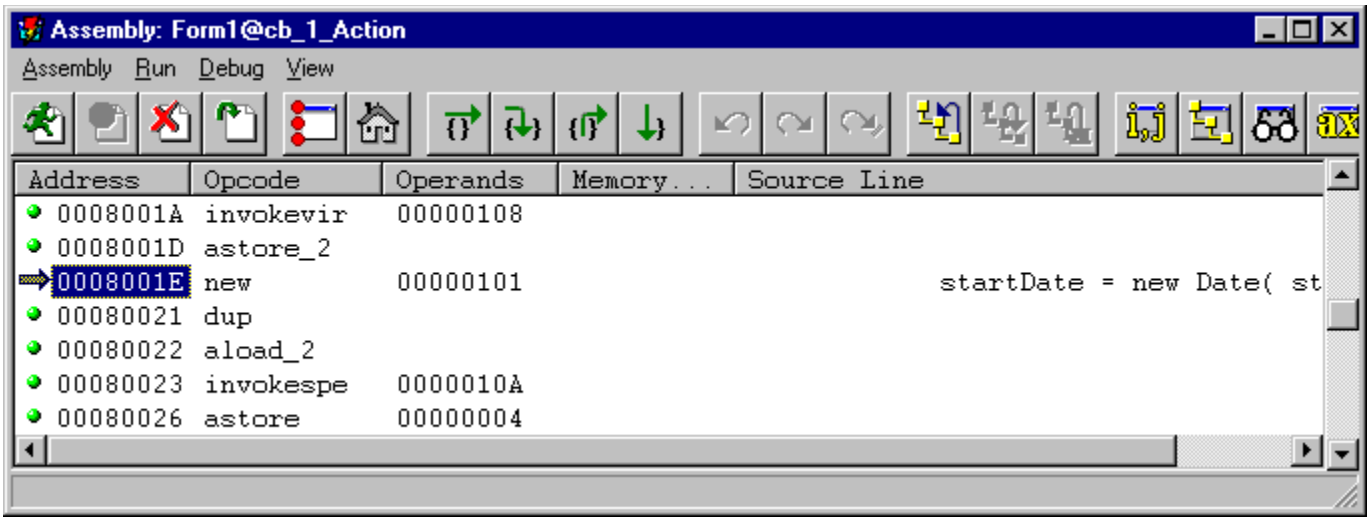
When the breakpoint goes off, the Locals window will not display `superField` in the list of local variables, but you can still obtain the value of such parent-defined variables. In the Watches window, specify that you want to watch:

```
this.superField
```

The Watches window will display the value of that specific variable as defined in the object's parent class.

Assembly window

The *Assembly* window displays an assembly language version of executable code in your program. This only provides useful information if you are using the Microsoft virtual machine for Java.



- **To see the Assembly window:**

1. From the **Debug** menu of the code editor, click **Assembly**.

If your program is stopped at a breakpoint, the Assembly window marks the breakpoint with a red stop sign, overlaid by a yellow arrow. The Assembly window begins with the breakpoint at the top of the instructions shown in the window.

The Assembly window has a vertical scroll bar to let you scan back and forth through your executable code. The scroll indicator for this scroll bar cannot be dragged directly; you must click the arrows of the scroll bar or the empty areas on either side of the scroll indicator.

- **To look at a new code location:**

1. From the **Assembly** menu in the Assembly window, click **Show Address**. PowerJ displays a dialog box, asking the address that you want to look at.
2. Type the new address in this box, using the same format as the addresses shown in the Assembly window or any Java expression that evaluates to an address.
3. Click **OK**.

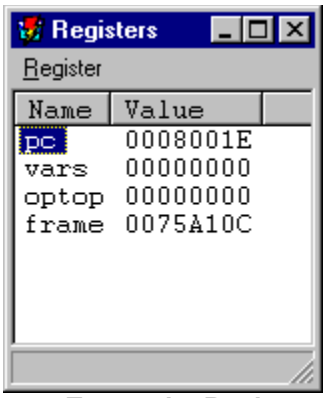
If you use the right mouse button to click a line in the Assembly window, PowerJ displays a context menu offering the same choices as the **Assembly** menu. These are **Show Address**, **Run to Cursor** (see Run to Cursor), **Skip to Cursor** (see Skip to Cursor) and **Toggle Breakpoint**.

The Run menu

The **Run** menu of the Assembly window offers a number of alternative ways for resuming execution of your program. For further information, see Stepping through your code.

The Registers window

The *Registers window* displays the contents of the Java interpreter's virtual registers at the time execution was suspended.



- **To see the Registers window:**

1. From the **Debug** menu of the code editor, click **Registers**.

In some situations, the Registers window is displayed automatically when PowerJ displays an Assembly window.

Changing register values

The Registers window lets you change the value of a hardware register.

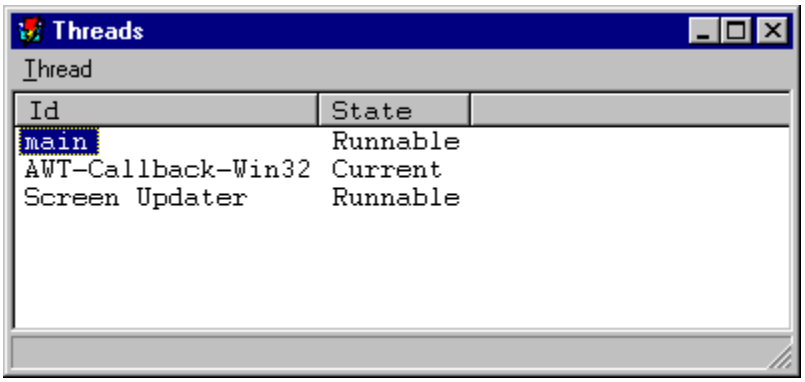
- **To set the value of a hardware register:**

1. In the Registers window, double-click the name of the register you want to change. (Note that you double-click the name, not the value.) PowerJ displays a dialog box for you to type the new value.
2. Type the value you want to store in the register, using the same format as the current value.
3. Click **OK**.

When you resume execution of the program, the specified register will contain the assigned value.

Threads window

The *Threads* window displays information on the execution threads of your program.



- **To see the Threads window:**

1. From the **Debug** menu of the code editor, click **Threads**.

For more information about the Threads window, see Debugging threads.

Memory window

The *Memory* window displays the contents of a region of memory, using a variety of formats.

Note: The Memory window is only of limited use in Java. You really only get meaningful results when you are viewing the contents of a single array.

- **To see the Memory window:**

1. From the **Debug** menu of the code editor, click **Memory**. PowerJ displays a dialog box asking the address that you want to examine.
2. Type the starting address of the memory area you want to examine. You can specify the address with any Java expression referring to a memory location (for example, `&var` or the name of a function).
3. Click **OK**.

The Memory window has scroll bars to let you scroll forward and backward in memory.

The Stack window

The Stack window is not enabled in this version of PowerJ.

Drag-and-drop in debugging windows

The debugging windows offer extensive drag-and-drop facilities. For example, if you click a variable name in the Locals window and drag it to an empty area of the Watches List, the variable is added to the Watches List. Similarly, you can drag variables or expressions from a code editor window into the Watches List.

You can also perform patching operations with drag-and-drop. For example, if you drag a value from the Memory window and drop it on a variable in the Locals window, PowerJ assigns the value to the variable. Similarly, you can drag a value from one location in the Memory window and drop it on another location. In this case, PowerJ asks you to confirm that you want to assign the dragged value to the new location.

In general, if you drag a value and drop it on some other object, PowerJ assigns that value to the object. This works for all relevant debugging windows: the Locals window, Watches window, Assembly window, Registers window, and Memory window. You can also use drag-and-drop between debugging windows and code editor windows.

Stepping through your code

When a breakpoint is encountered, PowerJ suspends your program so that you can examine its code and data. At this point, PowerJ offers several ways in which you can resume program execution:

- **Run**: PowerJ lets the program run normally from the point it was suspended. If the program hits another breakpoint, it will suspend execution again.
- **Restart**: PowerJ starts the program again from the beginning. If you choose this, you will lose any unsaved data that you may have provided for the program, since the program makes a completely clean start. PowerJ confirms that you really want to do this before restarting the program.
- **Terminate**: PowerJ kills the program entirely, returning to your original PowerJ session. If you choose this, you will lose any unsaved data that you may have provided for the program. PowerJ confirms that you really want to do this before the program is killed.
- *Stepping*: With this facility, you can move through your program one step at a time, pausing after every action so that you can examine the results of that action. This is an extremely powerful debugging ability, helping you examine the flow of control within your program and the effects of your code.

There are several different types of steps. The rest of this section examines these types in detail.

Run to Cursor

The **Run to cursor** action is related to the stepping actions. Suppose you are stopped at a breakpoint and looking at a code editor window or the Assembly window. If you use the right mouse button to click on a statement, then click **Run to cursor**, PowerJ runs the program until it reaches the statement that contains the cursor. Execution stops just before executing that statement.

- **To perform a Run to cursor action:**

1. Use the right mouse button to click the statement where you want the run to end.
2. From the context menu, click **Run to Cursor**.

Run to cursor is a quick way to step through several statements at a time.

Skip to Cursor

The **Skip to cursor** action doesn't actually run any statements. It simply moves the execution point to a different source code statement. The statement that you skip to becomes the next statement to be executed.

- **To perform a Skip to cursor action:**

1. Use the right mouse button to click the statement where you want to skip.
2. From the context menu, click **Skip to Cursor**.

Important: You should exercise caution when using **Skip to cursor**, since it skips code that would normally be executed. For example, if you skip a statement that initializes a local variable, the variable will not have a meaningful value in subsequent code. If you try to skip to a completely different function, your program will probably crash.

Step over

The **Step Over** action steps through your code one source statement at a time. In the code editor or Assembly window, PowerJ displays a yellow pointer, showing the statement that will be executed next (or that is currently being executed).

- **To perform a Step Over action:**

1. Press F10; or
2. From the **Run** menu, click **Step Over**; or
3. Click the **Step Over** button on the code editor's tool bar.

Step Over was given its name because it "steps over" function calls. For example, suppose the next source code line contains

```
x = func(y);
```

If you execute a **Step Over**, PowerJ executes the entire statement in a single step, even though the function call may consist of many source code statements in itself. This is the main difference between **Step Over** and the next type of stepping action, **Step Into**.

Step into

Like **Step Over**, the **Step Into** action steps through your code one source statement at a time. In the code editor window or Assembly window, PowerJ displays a yellow pointer, showing the statement that will be executed next (or that is currently being executed).

- **To perform a Step Into action:**

1. Press F8; or
2. From the **Run** menu, click **Step Into**; or
3. Click the **Step Into** button on the code editor's tool bar.

Step Into was given its name because it "steps into" function calls. For example, suppose the next source code line contains:

```
x = func(y);
```

If you execute a **Step Into**, PowerJ does not execute this statement as a single step. Instead, it steps into the function `func`: a code editor displays the code of `func`, with the yellow pointer positioned at the beginning of the function. Additional **Step** actions will step through `func` as displayed in the code editor window. When `func` eventually returns to its caller, the code editor window switches back to show the statement that contained the original function call.

In the code editor, **Step Into** only steps into functions for which the source code is available. It does *not* step into library functions, including methods from the PowerJ component library, unless you have the source code for those included in your project. **Step Into** may step into code that has been automatically generated by PowerJ, such as the code that constructs and initializes a new form.

In the Assembly window, **Step Into** will always step into functions.

If you use **Step Into** to walk through your program one step at a time, the worst that can happen is that you step into a function whose contents you don't want to see. If so, you can quickly step to the end of the function using **Step Out** (see Step out). On the other hand, if you get into the habit of using **Step Over**, you may accidentally step over a function you really wanted to examine in more detail.

Nested function calls

Suppose a line of code contains a function call of the form

```
f( g(x) )
```

This is called a *nested* function call. To evaluate the expression, Java evaluates `g(x)` first, places the result in temporary storage, then calls function `f` using `g`'s result as the argument for `f`. Therefore, suppose you use **Step Into** on the line of code shown above:

1. First, you step into `g`, since the program evaluates `g(x)` first.
2. When `g` finishes, you return to the original function: the one containing the code `f(g(x))`.
3. If you perform another **Step Into**, you step into `f` to execute the final result.

In other words, **Step Into** tracks the line of execution from function to function, showing the order in which the code is actually executed. This may be surprising the first time you see it, but it corresponds to the order that the functions have to be called.

Step out

The **Step Out** action executes the rest of the current function, beginning with the statement indicated by the yellow pointer and ending when the function returns (either because of a `return` statement or because the function reached the end of its code).

- **To perform a Step Out action:**

1. From the **Run** menu, click **Step Out**; or
2. Click the **Step Out** button on the toolbar.

Step next

The **Step Next** action tells PowerJ to execute until reaching the next line of source code. It is typically used when the yellow pointer is at the end of a loop construct. For example, consider

```
for (i = 0; i < 100; i++) {
    array[i] = i;
}
// other instructions
```

Suppose you have used **Step Over** to step through the `for` loop once. If you use **Step Over** again, it will go back to the top of the `for` loop, since that is the next statement to be executed. However, if you use **Step Next** when the yellow pointer is at the end of the `for` loop, PowerJ starts executing the program and keeps going until it reaches the next line of source code after the loop. This is a quick way of avoiding stepping through the loop a hundred times.

- **To perform a Step Next action:**

1. From the **Run** menu, click **Step Next**; or
2. Click the **Step Next** button on the toolbar.

Warning: Step Next actually works by setting a breakpoint at the beginning of the next line of source code, then running the program until it hits the breakpoint. This may lead to surprising results in an `if-else` construct. For example, suppose the yellow pointer points to the `if` statement of:

```
if ( condition ) {
    statement1;
} else {
    statement2;
}
```

PowerJ places a breakpoint on the next line of source code (`statement1`), then starts the program running until that breakpoint is encountered. If the `condition` expression is false, execution skips `statement1` and you may never get back there.

Breaking program execution

Even if you haven't set any breakpoints, you can break program execution at any time by issuing a **Break** command. This temporarily suspends the program in the same way that a breakpoint does.

- **To break program execution:**

1. From the **Run** menu, click **Break**; or
2. Click the **Break** button on the toolbar.

Break is particularly useful if you notice the program behaving incorrectly in a place where you have not set a breakpoint. However, you have less control over where the program pauses—you can set a breakpoint at a particular location, whereas with **Break**, you just have to click and hope the program will stop at a recognizable point in your code.

When you pause a program, PowerJ displays the code that was executing at the time of the pause. If the program was executing user-written code, the code is shown in a code editor window. Often, however, the program was executing code from a library, so there is no source code available. In this case, PowerJ displays an Assembly window showing the assembly code that was being executed.

Resuming normal program execution

After a **Break** or a breakpoint, you can resume normal program execution with a **Run** command.

- **To resume normal program execution:**

1. Press F5; or
2. From the **Run** menu, click **Run**; or
3. Click the **Run** button on the toolbar.

Execution begins with the statement marked by the yellow pointer in the code editor (or with the statement marked in the Assembly window if you paused the program outside recognized source code).

Source code folders

The **Source Folders** page of the **Options** dialog lets you specify folders where PowerJ should look for source code, other than the folders referenced in the class files of your project. For example, if your program uses classes whose source code is given in some other folder, you can use **Source Folders** to specify the pathname of that folder. This lets PowerJ find the source code, so that you can debug at the source-code level instead of the virtual machine-language level.

The debug log

You can use the *debug log* to display diagnostic messages from running programs. Many of the PowerJ classes write diagnostic messages to the debug log, and your program can use the Debug class to write messages to the debug log. This can be very useful, particularly if you need to use a VM that does not support debugging.

By default, you are asked if you want to display the debug log if it is not being displayed when your program sends messages to it. If you choose to turn that option off, you can open the debug log manually when you want to display it.

- **To view the debug log:**

1. On the **View** menu of the main PowerJ menu bar, click **Debug Log**.

This opens the debug log as a separate window.

Note: By default, the debug log is cleared before each run of a target. Therefore, the contents of the debug log represent the results of the most recent run.

The Debug class, described in the next section, provides many options for sending messages to the debug log.

Important: In order for the debug log to work properly, you must have Client for Microsoft Networks installed on your system.

The Debug class

The Debug class (`powersoft.powerj.util.Debug`) offers a number of methods that are useful for debugging your program. Most of these methods are static methods, so you can execute them without creating a Debug object. For example, you can use

```
Debug.assert( i > 0 );
```

to invoke the **assert** method of Debug directly; you do not need to create a Debug object first.

Note: In order to use the Debug methods, you must import the appropriate class definition:

```
import powersoft.powerj.util.Debug;
```

Assertions

The **assert** method is a simple way to check for bugs while your program executes. The method takes a single argument expression which must evaluate to a boolean result. If the result is `true`, nothing happens. If the result is `false`, **assert** writes an *assertion failure* message to the debug log.

For example, suppose that a variable `var` is expected to fall within a certain range of values at some point in the program. The function

```
Debug.assert( (var > min ) && (var < max) )
```

tests whether `var` actually does fall into the range. If the assertion expression is not true, **assert** writes a message to the debug log to let you know something has gone wrong.

Note: The **assert** method only has an effect for Debug targets. In Release targets, **assert** does nothing.

Assertions with explanations

The **assert** method can take a second form:

```
// boolean expression;
// String message;
Debug.assert( expression, message );
```

The `expression` is an assertion expression and `message` is a string that should be displayed in case of an assertion failure. For example,

```
Debug.assert( (var > min) && (var < max),
     "var out of bounds" );
```

displays the given message if `var` falls outside the acceptable range. The message is written to the debug log when the assertion failure takes place.

Release assertions

The **verify** method is like **assert**, except that it is active for both Debug and Release targets. Therefore, you can use **verify** to check for "should-not-happen" conditions in the end-release version of your program. The **verify** can take either of the following forms:

```
// boolean expression;
// String message;
Debug.verify( expression );
Debug.verify( expression, message );
```

In order for **verify** to work in Release mode, you must explicitly turn on debug logging as described in Debug log mode.

Throwing exceptions with Debug methods

The Debug class offers a number of methods that can throw specific types of exceptions. For example, the **checkArgument** method makes it easy for your code to check whether a particular argument has a valid value. If not, **checkArgument** throws an IllegalArgumentException.

The simplest form of **checkArgument** is

```
// boolean exp;
Debug.checkArgument( exp );
```

If the expression is `true`, the argument is assumed to be valid. If the expression is `false`, **checkArgument** throws the IllegalArgumentException.

There is a second form of **checkArgument** that accepts a String argument:

```
// String msg;
Debug.checkArgument( exp, msg );
```

This includes the `msg` as part of the IllegalArgumentException. Typically, the software that catches the exception will display the string as part of an error message.

There are similar methods for throwing other kinds of exceptions:

```
Debug.checkState( exp );     // throws InvalidStateException
Debug.checkState( exp, msg );

Debug.checkCreated( exp );   // throws NotCreatedException
Debug.checkCreated( exp, msg );
```

Displaying the run-time stack

The **printStackTrace** method of Debug writes out a trace of the current call stack. This trace tells the function that is currently executing, the function that called that function, and so on back up the call stack.

The simplest form of the method is

```
Debug.printStackTrace();
```

This writes the trace to the debug log. You can also use

```
// PrintStream stream;
Debug.printStackTrace( stream );
```

This writes the trace to the specified output stream.

Displaying thread information

The **logThreadInfo** method writes information about a specified thread group to the debug log:

```
// ThreadGroup group;
Debug.logThreadInfo( group );
```

The information written out describes the threads in the given group. For more information about threads, see Using threads.

General log methods

The Debug class offers a number of methods for writing to the debug log. The simplest of these is:

```
// String msg;
Debug.log( msg );
```

This writes the given string to the debug log. (A new-line character is automatically added after the string, in order to end the line.)

The second form is:

```
// byte bmsg[];
Debug.log( bmsg );
```

This is similar, except that the message is given as a byte array instead of a String.

The **log** method may also take the form

```
// Throwable err;
Debug.log( err );
```

In this case, the argument is an object derived from the Throwable interface. This includes all standard exceptions. For example, if you write code to catch an exception, you could use the exception object as an argument to **log**. The **log** method will display whatever error message is associated with the exception.

Note: All of the above versions of **log** are called *general* log messages.

Group log messages

In addition to general log messages, the Debug class lets you create special log messages which have a group identifier. For example, you might identify some log messages as "Database" messages, some as "Number-crunching" messages, and so on. Groups are identified by strings, and you can make up your own string identifiers.

You write group log messages with the **log** method of Debug. The methods for writing group log messages are similar to those for writing general log messages:

```
// String msg;
// byte bmsg[];
// Throwable err;
// String groupID;
Debug.log( groupID, msg );
Debug.log( groupID, bmsg );
Debug.log( groupID, err );
```

Enabling groups

The **setGroupLog** method of Debug enables a named group of log messages. For example,

```
Debug.setGroupLog( "Database", true );
Debug.setGroupLog( "Number-crunching", false );
```

enables group log messages with the identifier "Database" and disables those with the identifier "Number-crunching". By enabling or disabling groups, you can control which log messages actually appear in the debug log. (The **LogMode** property also affects which log messages appear as discussed in the next section.)

Debug log mode

The **LogMode** property of Debug controls which log messages are actually written to the debug log. You set this property with

```
Debug.setLogMode( code );
```

where `code` is one of the following:

`Debug.LOG_NONE`
No log messages are written. In other words, the **log** method has no effect.

`Debug.LOG_GENERAL`
Only general log messages are written. Group log messages are not sent to the debug log.

`Debug.LOG_GROUPS`
Only group log messages are written. General log messages are not sent to the debug log.

`Debug.LOG_ALL`
The debug log shows all general log messages and all group messages for groups that have been enabled. Messages from disabled groups are not sent to the debug log.

`Debug.LOG_FORCEALL`
The debug log shows all general and group log messages, even from groups that are disabled.

Detailed logs

The **DetailedLog** property controls the amount of information that is included in log messages (both general and group). If the **DetailedLog** property is `true`, the **log** method adds a trace of the call stack to every log message written out. You can control this with the **setDetailedLog** method:

```
Debug.setDetailedLog( false );
```

The default value of **DetailedLog** is `false`.

Alternate log streams

By default, log messages are sent to the standard PowerJ debug log. However, you can change the destination either to a PrintStream or a URL.

The **setLogStream** method of Debug tells PowerJ to write log messages to a specific print stream instead of the debug log:

```
// PrintStream stream;
Debug.setLogStream( stream );
```

The PrintStream argument must be open.

The **setLogURL** method of Debug tells PowerJ to write log messages to a given URL instead of the standard debug log:

```
// String url;
Debug.setLogURL( url );
```

The default is `"Localhost:7777"` which writes to port 7777 on your system. This corresponds to the PowerJ debug log.

Debugging techniques

The preceding sections describe the basic debugging tools of PowerJ. The remainder of this chapter suggests some approaches to using those tools to find bugs, including a general discussion of where to look for bugs when programming in the Windows environment.

Important: Always run your program with the debugging facilities until you believe you have removed all the bugs. To make sure that debugging is active, look at the Targets window. In **Target Type**, you should see the word `Debug`.

Modular testing

It is easier to test a small piece of code than a large one. Therefore, you should make an effort to break down your program into small sections which can be tested independently. For example, it makes sense to create each form of your program separately and to do as much testing as possible on a single form before integrating it with other forms.

When necessary, you can create simple *stubs* to stand in for other forms in the early phases of testing. For example, suppose that clicking on a button in `Form1` is supposed to open `Form2`. In the early phases of testing, you can replace the creation of `Form2` with code like this:

```
System.out.println( "Stub!" );
```

Instead of displaying the actual `Form2`, this replacement code displays the given message. This gives you feedback that the button is being activated, without introducing the complexity of creating a new form.

Similarly, if you want to test the creation of `Form2`, you might create a very simple version of `Form1` which only consists of a command button that calls `Form2`. In this way, you can test the creation process without having to worry about the complexities of a full `Form1`.

Searching for bugs

Finding the location of a bug can be a time-consuming activity. It helps if you take a methodical approach to the job, rather than skipping about hit-or-miss.

The following section discusses several possible approaches. In some situations, one approach may be more effective than others, but starting out with a sensible plan for finding a bug is better than leaping in at random.

Start at the beginning

One approach is simply to set a breakpoint at the beginning of the code you think contains the bug, then single-step through until something goes wrong. It may help to keep the Watches window or the Locals window visible on the screen as you single-step, so that you can see the effects of each step.

The binary search

If you have to examine a large section of code, you may find a binary search is faster than starting at the beginning:

1. Place a breakpoint at the approximate midpoint of the code where you suspect the error may be occurring, then start the program running. When the break occurs, check your data to see if the bug has occurred yet.
2. If the bug has already occurred, place a breakpoint halfway between the beginning of the suspicious code and the current breakpoint. Run the code again.
3. If the bug hasn't occurred yet, place a breakpoint halfway between the current breakpoint and the end of the suspicious code. Continue running the program until you reach the next breakpoint.

If you keep placing a breakpoint at the halfway point of the code you think you should examine, you cut the search area in half with each test run. In this way, you can quickly narrow down the section of code that you need to examine. When you have reduced the suspicious code to a small number of lines, a step-by-step search should find the problem much more quickly.

The hypothesis/test approach

Sometimes you suspect what might be causing the problem, but have trouble making that situation arise in a normal test run. For example, you think that a particular piece of code is misbehaving if it receives erroneous data, but you have difficulty feeding erroneous data to that piece of code. In such a case, you might set a breakpoint at the beginning of the code and then set up some erroneous data directly, using patch operations or assigning values to appropriate variables. Once you have finished setting up the data, you can run or step through the program to test your suspicions.

Chapter 5. Collecting and deploying

This chapter discusses how you can manage and deploy your Web site. You can use a WebApplication target to collect and publish all the files of your Web site. You can use ZIP, CAB and JAR file targets to facilitate the deployment and downloading of your Web site files.

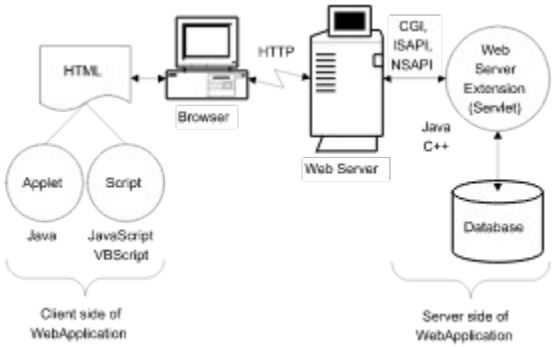
WebApplication targets

A WebApplication target ties together a set of other targets into a single manageable package. For example, suppose that you are creating a software package that consists of software running on User systems (such as Java applets) as well as software running on a Server system (web servlets). You can create a WebApplication target which represents all this software as a single object. This offers a number of advantages:

- When you build the WebApplication target, PowerJ can build everything as needed. You don't have to worry about different parts of the project getting out of synch with each other.
- WebApplication targets can include various types of source files, including HTML files, GIF/JPEG/AVI files, and HTML templates (such as those used by NetImpact Dynamo), as well as sub-targets like Java applets and CGI, NSAPI, or ISAPI web services. These files may be stored in any folders you find convenient.
- As with any other PowerJ target, WebApplication targets can be managed with a source control system (for example, RCS, PVCS, Object Cycle, and so on).

Note: WebApplication targets provide a useful mechanism for distributing programs even if they are not intended for use on the Web. A WebApplication target gathers all the files needed to run a particular program (applet, application, or some other type of target) and copies those files to a designated location.

The following diagram represents some of the client-side and server-side components that you might have in a WebApplication:



Web projects

A *web project* consists of the following:

- A WebApplication target
- Any number of dependent targets—these are targets such as Java applets and web services that PowerJ builds as separate "sub-targets" of the WebApplication:
- A staging web site
- A source control archive

The files associated with the WebApplication target represent all the files in a web site. The WebApplication target also contains a small number of extra PowerJ generated files that are not needed in the final production web site. PowerJ uses these to maintain information about the structure of the target.

Dependent targets: Dependent targets are built in their own folders, separate from the WebApplication target. As with normal PowerJ targets, dependent targets have a target folder, a `Release` folder and a `Debug` folder.

Whenever PowerJ builds the WebApplication, it also builds any dependent targets that need to be built.

Note: A WebApplication may not contain another WebApplication as a dependent target.

Staging web site: The staging web site is a web site where the developer can run the WebApplication. This web site may be:

- A web site on a different machine
- A folder on the same machine as the WebApplication target
- The folders used by the WebApplication target itself.

The process of copying files from the WebApplication target folders to the staging web site is called *publishing*. When a WebApplication target is published, PowerJ publishes files associated with the WebApplication itself as well as files associated with the dependent targets of the WebApplication.

Source control archive: The source control archive contains the master revision control copies of the files in the WebApplication target and the dependent targets. The archive can be implemented with any source control system PowerJ supports.

WebApplication publishing

One of the main reasons for creating a WebApplication target is the ability to *publish* the contents of a WebApplication. Publishing a WebApplication means copying the folders and directories under the WebApplication's `Debug` or `Release` folder to some other location. This can be done with normal disk copying operations or by invoking the WebPost API to do a recursive copy operation.

For example, if you are publishing a packaged Java class file with a name like

```
mycompany.mypackage.myapp.myclass
```

the publishing process would write the class file to

```
mycompany\mypackage\myapp\myclass.class
```

in the folder where you want to publish the WebApplication. If any of these subfolders do not already exist, PowerJ automatically creates them.

The following rules apply during the publishing process:

- If a class file to be published is already in a target's folder (`Debug` or `Release`), PowerJ publishes the file to a corresponding location in the destination directory (including any subfolders). For example, if the class file is originally found in

  ```
  Debug\mydir1\mydir2\myclass.class
  ```

  PowerJ publishes the file to

  ```
  mydir1\mydir2\myclass.class
  ```

  under the destination folder.
- If a class file to be published is not in the target's folder, PowerJ examines the fully qualified name of the class being defined and copies the class file to a destination based on that name. For example, if PowerJ sees that a class file defines

  ```
  mycompany.mypackage.myapp.myclass
  ```

  the publishing process would write the class file to

  ```
  mycompany\mypackage\myapp\myclass.class
  ```

  in the destination folder.

Notice that class files which are already in the target folder are presumed to be in the correct location, even if this conflicts with the fully qualified name of the class.

Publishing happens whenever you **Run** a WebApplication. Therefore, if you want to test a program as it runs on a particular web site, you can configure the run options for the WebApplication to publish the application files to the folder associated with that web site.

Creating a WebApplication target

If you have already created one or more dependent targets, you can create a WebApplication target in the same project.

- **To create a WebApplication target:**

1. From the **File** menu of the main PowerJ menu bar, click **New** and then **Target**.
2. In the Target Wizard, click **WebApplication**, then click **Next**.
3. Enter a name for the WebApplication's target folder, then click **Next**.
4. Click the names of the targets that will be the dependent targets for this WebApplication (or click **Select all** if all the targets listed will be dependent targets).
5. Click **Finish**.

PowerJ creates an HTML file named `index.html` to serve as the starting web page for this target.

The contents of the HTML file is a typical skeleton for a web page. For the purposes of testing, you might put HTML links on this page which allow you to go to other HTML pages that belong to this application. For example, you might put in an applet tag for a Java applet that you want to test.

Note: The generated applet HTML file is only published as part of the WebApplication when you are building targets in release mode.

Adding targets to a WebApplication target

Typically, you create the WebApplication target after you have created one or more targets that you want to package as a WebApplication. In this way, the WebApplication is the last target you create.

If you create the WebApplication target before the target(s) that you want to include in the WebApplication, you must add the targets to the existing WebApplication.

- **To add a target to an existing WebApplication target:**

1. Open the Targets window and click on the name of the WebApplication target.
2. On the **File** menu of the Targets window, click **Add File**. This opens a file selection dialog.
3. Open the **Files of type** list in the file selection dialog, and click **Targets**.
4. Use the file selection dialog to locate the target file (WXT) for the target that you want to add to the WebApplication. When you have located the desired target file, click **Add**.

Although you add a WXT target file to the WebApplication target, the Targets window actually shows that the WebApplication target contains a JLT file. For more about JLT files, see JLT files.

Actions on a WebApplication target

Publishing the WebApplication means copying all files in the target folder to the staging web site. If the staging web site is the same as the WebApplication target folder, publishing the WebApplication does nothing. Otherwise, the publishing process copies the files to a specified location, either on the local system or on another system.

A WebApplication target supports the following actions (available through the **Run** menu):

Build
Building a WebApplication means building all the dependent targets, and copying all appropriate files to the WebApplication's `Debug` or `Release` folders. This includes copying files from dependent targets as well as copying files associated with the WebApplication itself. If necessary, the Build process constructs subfolders under the WebApplication's `Debug` or `Release` folder in parallel to the directory structure under the dependent targets.

Run
Running a WebApplication means performing a **Build** operation, then publishing the WebApplication. After the WebApplication has been published, you have the option of invoking a web browser or server program to test some aspect of the application (for example, an applet or servlet).

Rebuild All
Performs a full clean build on the WebApplication target and all its dependent targets. You should always perform this operation when you are preparing to deploy the final version of an application —it can clean up unwanted intermediate files that may have been produced in previous builds.

These actions are controlled through the run options for the WebApplication. For example, the run options dialog has a **Publish** page where you can specify a folder that will serve as the staging web site. Similarly, the **General** page lets you specify the following options for running the application:

Just publish the target files
Presumably, you will then invoke a web browser and/or web server manually.

Publish, then run a web browser
You may specify the command line for invoking the web browser.

Publish, then run a web server
You may specify the command line for invoking the web server.

If you wish to run a web browser or web server, you can also specify configuration options for the software being invoked. For example, you could browse the files directly or use a web server. You can start a browsing session by specifying a local file or a URL.

Note: The run options let you run either a web browser or web server, but not both. If you want to test an application that needs both, you might set the run options to handle the server side, then manually open a web browser to test that side of the application.

The WebPost wizard

The **Publish** page of a WebApplication's run options lets you publish the application using the Microsoft WebPost wizard (also called the Web Publishing wizard). The WebPost wizard is available from Microsoft as part of the Internet Explorer Starter Kit.

Debugging WebApplication targets

You set debugging options for a WebApplication target using the **Debug** page of the Run options dialog for the target. This page offers a number of choices, described below.

Note: You can only debug one member of the WebApplication at a time. For example, you can debug an applet on the user side or a servlet on the system side, but not both at the same time.

Debug the following applet

This lets you choose an applet to be debugged. The applet should be one of the targets which are members of the WebApplication. You can specify more information (for example, an initial breakpoint) by changing the run options for the applet target itself.

Debug NetImpact Dynamo script, using this Linked Folder

This lets you specify the name of a Dynamo Linked Folder. The name should be the same as the name defined in Dynamo to link to a disk folder. You must specify this folder name to debug DynaScript, since PowerJ needs disk files to debug, and cannot access scripts within databases accessed by Dynamo.

Debug PowerJ Java Servlet

This lets you debug a servlet belonging to the WebApplication.

To debug, PowerJ will first publish any files. Then, PowerJ runs the Java server under the control of the PowerJ debugger and invokes your browser. You then need to use your browser to request a page that loads the servlet.

You must provide PowerJ with information about the servlet to be debugged, and the Java server that will be loading the servlet. When you click the **Configure** button associated with this option, PowerJ displays a dialog box.

- You can select the servlet to be debugged from the servlet targets in the WebApplication.
- You can select the default Java server. The source code is provided with PowerJ as part of the targets created by PowerJ when you create a new Dynamo or web server servlet.
- You can specify a command that will run a Java virtual machine corresponding to the debug options set for the servlet, and eventually load your servlet.
- You can specify that the server should be executed within a Java console (DOS box).

You will have to run the web server and/or Dynamo yourself. The extension DLLs that extend the web server and/or Dynamo are provided with PowerJ. For more information, see Basic server support architecture.

Properties of a WebApplication target

The properties of a WebApplication target can be examined and changed through the Targets window.

- **To examine the properties of a WebApplication target:**

1. In the left part of the Targets window, use the right mouse button to click the name of the WebApplication target, then click **Properties**.

The resulting property sheet has three pages:

General

This page gives general information about the WebApplication target, and also offers a button that will build the WebApplication.

Applet Classes

This page controls whether the WebApplication automatically publishes the Java classes used by the applets. The choices are:

- To publish only the classes defined in the applet itself (the classes in the applet target folder).
- To publish all the classes needed by the applet, including any library classes that are found by analyzing the applet classes and looking for references to other classes.
- To publish all the classes needed by the applet, with the exception of classes that belong to specified packages. For example, if you know that users will already have PowerJ installed on their system, add the package name `powersoft.powerj`. This indicates that you don't need to publish classes from the standard PowerJ library.

  To add a package to the list of packages whose classes should not be published, type the name of the package in the blank labeled **Package**, then click **Add**.

Note: Classes that are loaded by name (such as the JDBC drivers) must be added directly to the WebApplication target as class files or else manually copied to your web site before running your applets. The PowerJ publishing operation currently has no way of detecting places where your application loads classes by name.

Applet Files

This page controls whether non-class files should be published with the applet. For example, if the target folder contains HTML, GIF or JPEG files, the **Applet Files** page controls whether these files should be published.

Note: The HTML file that PowerJ generates for an applet target will not be copied.

To keep track of the information supplied through the property sheet, a WebApplication target creates a file in the `Debug` or `Release` folder with the file name extension `.WWA`. This file contains information about publishing the applet.

Using the Microsoft Java VM Plugin for Netscape

Microsoft's Virtual Machine for Java plugin makes it possible for Netscape Navigator to run Java programs using the Microsoft VM 1.5 for Java. Since the Microsoft VM offers debugging facilities that are not available with the VM normally used by Netscape Navigator, using the plugin may make it easier to debug applications under the Netscape browser.

The WebApplication target makes it easier for you to use the Microsoft VM Plugin by automatically adjusting your HTML tags during the publishing process.

In order to use the Microsoft VM Plugin, all the `<applet>` directives in an HTML file must change to `<embed>` directives. For example, if you are testing an HTML file which runs an applet with:

```
<applet code=test.class width=400 height=420 myparam=abc>
```

For the Microsoft VM Plugin, the HTML tag needs to be:

```
<embed code=test.class width=400 height=420 myparam=abc>
```

- **To specify that you want to test a WebApplication using the Microsoft VM Plugin:**

1. Open the run options dialog for the WebApplication target, and click **General**.
2. Click **Publish, then run a web browser**, then click **Configure**. This opens a web browser configuration dialog.
3. In the web browser configuration dialog, click **Netscape Navigator with Microsoft VM Plugin**.
4. If necessary, choose an initial URL for the browser, then click **OK**.

When you are running with Netscape Navigator and the Microsoft VM Plugin, the publishing process automatically converts `<applet>` directives into corresponding `<embed>` directives. Therefore, you don't have to edit the HTML yourself.

CLASSPATH and CODEBASE

When you run a Java program, the Java VM searches for class files relative to an environment variable named `CLASSPATH`. The value of `CLASSPATH` is a list of folders or archive files (ZIP or, for Java 1.1, JAR) that hold the class files. The order in the list specifies the order that they will be scanned by the VM.

For Java applications (including servlets), you often need to change the `CLASSPATH` environment variable so that the Java VM can locate all of the classes used by your program.

For Java applets, instead of changing the `CLASSPATH` environment variable, you can add a `CODEBASE` parameter to the `<applet>` tag in the HTML file. If the browser cannot locate a class that your applet needs in the browser's system `CLASSPATH`, it tries to download the class from a location relative to the folder given in the `CODEBASE` parameter. In the HTML `<applet>` tag that invokes your applet, you may specify a `CODEBASE` parameter of the form:

```
codebase=FolderName
```

For example:

```
<applet code=app.class codebase="myfolder">
```

The class files need to be in locations below the folder given in the `CODEBASE` parameter. Using the `<applet>` tag from above, if your applet needs the class

```
powersoft.powerj.ui.ResizePercentLayout
```

then your browser will try to download the class from the following location on your web server:

```
myfolder\powersoft\powerj\ui\ResizePercentLayout.class
```

If you do not specify a `CODEBASE` parameter in the `<applet>` directive, then the path to the folder of the HTML file that the applet is in is used as the codebase.

If you are creating a number of applets that have a number of library classes in common, you can set up a WebApplication target that publishes the applets to a single codebase folder. The publishing action can copy the shared library classes to the appropriate subfolder location under this codebase folder.

When you specify a codebase, all the required class files are assumed to be found relative to the codebase folder. For example, if you are using the `com.sybase.jdbc.*` jConnect classes and your codebase is `myfolder`, the jConnect classes should be under:

```
myfolder\com\sybase\jdbc
```

Similarly, archive files (such as ZIP and JAR files) must also be under the codebase. For example:

```
<applet codebase="myfolder" code=app.class
    archive=powerj.zip ...>
```

This specifies that the standard PowerJ class library was found in:

```
myfolder\powerj.zip
```

In this case, the web browser would request class files from the specified ZIP file as required. The `archive=` parameter of an `<applet>` directive may specify a list of archive files, as in:

```
archive="powerj.zip, mylib.jar, otherstuff.zip"
```

For more information about archive files, see Using collection targets: CAB, JAR, and ZIP.

Important: All the class files needed by an applet must be stored in the proper folder under the

codebase, or in an archive file specified by the `archive=` parameter in the `<applet>` directive.

Global CLASSPATH options

PowerJ makes it possible for you to specify a set of folders to be included in the PowerJ `CLASSPATH` whenever you run a target from PowerJ. This saves you from having to specify the `CLASSPATH` individually for each new target. For further information, see Classpath options.

Note: To see the exact `CLASSPATH` that PowerJ uses for running a particular target, check the batch file that PowerJ generates during the run. For more information about these batch files, see Batch files for running targets.

Using collection targets: CAB, JAR, and ZIP

Collection targets make it easy to package a complete application in a single file, simplifying the job of distributing the application to other sites. Typically the files are compressed to reduce size and hence download time. There are three types of collection targets:

JAR targets
: A Java Archive (JAR) is a platform-independent file format that can collect multiple files into a single file. By combining class files, images, and so on into a JAR file, you make it possible to download all the files for an application in a single HTTP transaction. This greatly improves the download speed. The JAR format can reduce download times even further by compressing the contents of the JAR file. Finally, you can place digital signatures on the individual entries of a JAR file to authenticate their origin.

CAB targets
: A Java Cabinet (CAB) is a file containing a collection of Java class files. CAB is a general purpose compressed archive format from Microsoft.

ZIP targets
: A Java ZIP target creates a single ZIP file containing a collection of Java class files. ZIP is a general purpose archive format from PKWare, commonly used to distribute Java class libraries. For example, the class libraries of PowerJ itself are collected into a ZIP file named `powerj.zip`. Many Java Virtual Machines cannot read ZIP archives that use compression, so the ZIP archives that PowerJ creates are not compressed.

Note: If you are creating a library of Java class files, you may want to use a PowerJ Java Classes target rather than a collection target.

Setting up collection targets

When you first create a collection target, the target is empty. You must add other items to the JAR, CAB, or ZIP target so that the collection target actually contains something.

- **To add a target to a collection target:**

1. Open the Targets window and click on the name of the collection target.
2. On the **File** menu of the Targets window, click **Add File**. This opens a file selection dialog.
3. Open the **Files of type** list in the file selection dialog, and click **Targets**.
4. Use the file selection dialog to locate the target file (WXT) for the target that you want to add to the collection. When you have located the desired target file, click **Add**.

Although you add a WXT target file to the collection target, the Targets window actually shows that the collection target contains a JLT file. For a description of JLT files, see JLT files.

You can also add individual files to a collection target (for example, Java class files). However, it is usually better to add whole targets to a collection, rather than individual files. For example, if you want to create a collection target containing a number of Java class files, it is usually best to create a PowerJ Java Classes target first, then add that target to a collection target. In this way, if you add or delete classes in the Java Classes target, the collection target will automatically change too the next time you rebuild it. Generally, it is better to let PowerJ manage the contents of the collection target, rather than adding or deleting files by hand.

Note: You can only add PowerJ Java applet, Java application and Java Classes targets to a collection target.

As an example of how this works, suppose that you want to package an applet named `MyApp` using a JAR file.

- First, create and debug `MyApp` as an applet target.
- Next, create a JAR target in the same project as `MyApp`.
- Next, add the file `MyApp.wxt` to the JAR target. The Targets window will show `MyApp.jlt` as the contents of the JAR target.
- Finally, build the JAR target. PowerJ will rebuild `MyApp` if necessary, then create a JAR file containing all the files needed for running `MyApp`. This includes any class files needed, plus extra files like GIFs and JPEGs.

The final JAR file will only contain the files needed to run the applet. For example, it will *not* contain Java source files, nor will it contain intermediate files like IDX, JLT, or LST files.

Note: If a collection target will contain a complete package of class files, the directory structure under the target folder should match the package structure. For example, if you have a package with names like `package.subpack.sub2.stuff`, the corresponding class file should be `package\subpack\sub2\stuff.class` under the target folder. (You only have to pay attention to this when you add files to the collection target manually; if you let PowerJ gather the collection target from other targets, PowerJ takes care of the package structure automatically.) If you use target properties to add additional class files to the collection target, PowerJ determines the appropriate location for each class file by examining the fully qualified name of the class that the class file defines.

Collection target properties

Collection targets do not have run options, since they cannot be executed on their own. All collection targets have the following property sheet pages (obtained through the Targets window):

Applet Classes

This page controls which classes will be collected into the collection file. The choices are:

- Only the classes defined in the applet itself (the classes in the applet target folder) and any serialization files (with the extension `ser`) for those classes.
- All the classes needed by the applet, including any library classes that are found by analyzing the applet classes and looking for references to other classes.
- All the classes needed by the applet, with the exception of classes that belong to specified packages. For example, if you know that users will already have PowerJ installed on their system, there is no need to publish classes in the PowerJ library (`powerj.zip`).

  To add a package to the list of packages whose classes should not be published, type the name of the package (e.g. "`powersoft`" or "`powersoft.powerj`") in the blank labeled **Package**, then click **Add**.

  When you change the options for including/excluding files, do a full clean build of the target before rebuilding it. If you do not perform a clean build first, any old files that are left behind will still be included in the target.

Note: Classes that are loaded by name (such as the JDBC drivers) must be added directly to the collection target as class files. PowerJ currently has no way of detecting places where your application loads classes by name.

Applet Files

This page controls whether non-class files should be collected into the collection file. For example, if the target folder contains HTML, SER, GIF or JPEG files, the **Applet Files** page controls whether these files should be added to the collection.

Note: ZIP targets do not have the **Applet Files** page, since non-class files will never be included in ZIP collections.

Additional properties for CAB targets

CAB targets have the following additional properties:

Signature [**Signature** page]

CAB files may include digital signatures. In this case, you can have PowerJ reserve space to hold the signature, but you must use third party signing software later to add the signature to the CAB file.

Class Identification [**CAB Properties** page]

Lets you assign a unique identifier to a class library in a CAB. (This property is not available for applet CABs.)

Trusted [**CAB Properties** page]

Indicates whether a CAB class library should be downloaded as a trusted or untrusted library. (This property is not available for applet CABs.)

Additional properties for JAR targets

JAR targets have the following additional properties:

Compression [**JAR Properties** page]
: JAR files may use ZIP compression to decrease the memory requirements. If you compress a JAR file with ZIP, your users must have appropriate software to "unzip" the JAR.

Manifest [**JAR Properties** page]
: If a JAR file contains a JavaBeans component, the component may have an associated *manifest file*. This option lets you specify whether the JAR process should create a default manifest file, should use a user-specified manifest file, or should omit the manifest file.

  For information on specifying information about a JavaBeans component in the manifest file, see Specifying which classes are Beans.

As with CAB files, JAR files may have digital signatures. However, PowerJ has no way of setting the digital signature for a JAR file—you must specify the signature in the JAR's manifest file. For more information on this subject, see:

`http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/javakey.html`

Look for the section titled "JAR Files and Digital Signatures".

Accessing a CAB file from a web page

CAB files may contain applets or Java libraries. In order to access CAB files, you must use the appropriate HTML directives in the web page files.

CAB files containing applets

In a Java applet, you can access a CAB file by using the CABBASE parameter in connection with the <APPLET> directive. Here is a simple example:

```
<APPLET CODE="myapp.class" WIDTH=300 HEIGHT=400>
<PARAM NAME="cabbase" VALUE="mycab.cab">
</APPLET>
```

If the applet is not already present on the user's system, the web browser downloads the CAB file, extracts the contents, and starts the applet.

Using CABBASE does not conflict with CODEBASE or any other parameters necessary for other browsers. For example, you might create a CODEBASE folder that contains all the class files required by an applet, plus a CAB file that collects all those class files together. This means:

- If the user's browser supports CAB files, it can get the applet's classes by loading the CAB file.
- If the user's browser does not support CAB files, it can download individual class files from the same folder in the usual way.

CAB files containing Java libraries

For Java libraries, you use the <OBJECT> directive to point to the CAB file. The syntax should be:

```
<OBJECT CLASSID="classId"
     CODEBASE="CABpathname#Version=a,b,c,d">
</OBJECT>
```

Here is a typical example:

```
<OBJECT
    CLASSID="clsid:0123456789-vendorname"
    CODEBASE="cabs/mycab.cab#Version=1,0,0,3">
</OBJECT>
```

If the classes are not already present on the user's system, the browser downloads the CAB file, extracts the contents, and places them in the appropriate location on the user's system.

As shown above, the CODEBASE can specify a version number. This ensures that the libraries are only downloaded if the version on the user's system is out of date. Both the CLASSID and the version number are also stored in the INF file inside the outer cabinet file. The version number is optional, but highly recommended.

After a library has been installed using <OBJECT>, <APPLET> directives can refer to the classes placed in the library. Such applets work in a normal way, bringing in their classes using CABBASE or CODEBASE.

Note: The <OBJECT> directive installs the classes permanently on the user's system, so it should only be used for libraries.

Accessing JAR and ZIP files from a web page

JAR and ZIP files are accessed using the ARCHIVE parameter of the HTML <APPLET> directive.

Note: JAR files were introduced in Java 1.1, so you must have a browser that supports JDK 1.1 or newer to use JAR files.

Accessing a JAR file

When an applet is stored in a JAR file, the <APPLET> directive in a web page must contain an ARCHIVE parameter specifying the location of the JAR file relative to the web page's HTML file. Here is a typical example:

```
<APPLET CODE=myapp.class ARCHIVE="jarfiles/myjar.jar"
        WIDTH=300 HEIGHT=400>
<PARAM name=abc value="xyz">
</APPLET>
```

The usual CODE=appletname.class must still be present. The CODE parameter identifies the name of the class where execution of the applet should begin. However, the class file for the applet and all of its helper classes are loaded from the JAR file.

Once the archive file is identified, it is downloaded and separated into its components. During the execution of the applet, when a new class, image or audio clip is requested by the applet, the browser begins by searching the archives associated with the applet. If the desired file is not found in the archives, the browser searches on the server system where the applet was first obtained. This search is made relative to the applet's CODEBASE (as in JDK 1.02).

Accessing a ZIP file

ZIP files are also accessed through the ARCHIVE parameter of the <APPLET> directive:

```
<APPLET ARCHIVE="myzip.zip" CODE="myapp.class"
     WIDTH=300 HEIGHT=400>
</APPLET>
```

The above directive downloads myzip.zip to the user's system. The browser will then search the ZIP file for the myapp class and other classes that the applet requires. In the ARCHIVE parameter, the name of the ZIP file should be specified relative to the CODEBASE of the applet; the ZIP file must not be compressed.

If the applet requires class files that are not found in the ZIP file, the browser searches for those class files on the server system in the usual way (relative to the CODEBASE).

Accessing multiple JAR and/or ZIP files

The ARCHIVE parameter may specify multiple JAR and/or files, with the file names separated by commas:

```
<APPLET CODE=Animator.class
    ARCHIVE="classes.jar , gifs.zip , other.jar"
    WIDTH=300 HEIGHT=400>
<PARAM name=abc value="xyz">
</APPLET>
```

There can be any amount of white space between entries with the value of the `ARCHIVE` parameter.

Deployment notes

This section provides a number of tips for preparing programs that you intend to distribute to other users.

Clean builds

Before the final deployment of any target, you should always perform a full clean build by clicking **Rebuild All** in the **Run** menu. This operation can clean up unwanted intermediate files that may have been produced in previous builds. This is particularly important with JAR, CAB, and ZIP files, to make sure the final version of the target does not contain unnecessary material.

Case sensitivity in file names

Some file systems are sensitive to the case of letters in file names. Under UNIX, for example, `MyFile.java` and `myfile.java` are two separate files. On Windows systems, however, the file system is not case-sensitive, and the two file names above refer to the same file.

If you are preparing an application for general use on the World Wide Web, you must remember that the application may be used by users with case-sensitive file systems. Therefore, you must use case consistently in the names of Java source files, class files, package names, and so on.

Note: Some Java compilers issue error messages if the name of the class file does not match the name of the Java file exactly, including the case of letters. This is another reason why case is important in file names and package names.

Dynamically loaded classes

When using a WebApplication or collection target, you can have PowerJ compute the classes used by the target. PowerJ can then automatically extract those referenced classes, and add them to the target.

However, Java lets you load arbitrary classes dynamically, and it is impossible for PowerJ to determine these classes. You must manually add these classes yourself to the WebApplication or collection target.

Working with the JDBC drivers

The JDBC drivers are designed to be loaded dynamically in the way just described. Therefore, if you use JDBC in an applet that is part of a WebApplication or collection target, you must include the driver's class files in the list of files to be packaged with the target. This can only be done manually.

- **To add a file manually to a target:**

1. Open the Files window and click the name of the target where you want to add the file.
2. In the **Files** menu of the Files window, click **Add File**.
3. Use the file dialog box to specify the file that you want to add. Click **Add**.

For a list of the files that must be included to support JDBC drivers, see JDBCPackage and deployment.

The sunw classes

Under JDK 1.02, Powersoft components like the grid component are created as *transitional beans*. This means that they follow an event model similar to the one for JDK 1.1 JavaBeans components. As a result, they use a set of utility classes defined by Sun, including:

```
sunw.util.EventListener
sunw.util.EventObject
sunw.util.Serializable
```

If you have a JDK 1.02 target that uses one of these Powersoft components, you must include the `sunw.*` classes when you deploy your application. PowerJ will look for this classes in the automatic extraction process, so the will be automatically included if you use that option with a PowerJ collection target. You can also find these classes in the same directory as `powerj.zip` and deploy them like any other classes that must be included with an application.

Note: Some browsers automatically include the `sunw.*` classes in their classpath, while other browsers do not. Therefore, if you do not deploy the `sunw.*` classes with your application, you will get the confusing situation where your application works with some browsers but not others.

Testing with a local web server

As you write more complicated applets, you'll probably find that you want to test them from a web server before you make the applet "live" on your real web server. You can do this using the NetImpact Dynamo Personal Web Server that comes with SQL Anywhere. You should have this installed if you installed the **Sybase Tools** option when you installed PowerJ. The following instructions assume that you have it installed.

The following steps show you how to use the NetImpact Dynamo Personal Web Server to test a WebApplication. For more complete information on using the Personal Web Server with PowerJ, including an overview of Dynamo, see A Dynamo WebApplication. You can use the same steps for other collection targets (ZIP, JAR or CAB files) as well as for WebApplication targets.

To use the NetImpact Dynamo Personal Web Server (PWS), you need to define mappings between URLs and files on your local disk drive. By default, the PWS has a mapping that maps `/` to the `c:\` folder and one that maps `/Site` to the Dynamo Demo database. This means that for a URL of the form

```
http://localhost/site/
```

the PWS will get the file from the database, but for any other URL it will get the file from the `C:` drive. For example, you might have your browser request the following URL:

```
http://localhost/temp/index.html
```

The PWS would send the following file back to your browser:

```
c:/temp/index.html.
```

You need to synchronize where PowerJ creates your WebApplication and where the PWS looks for it. While you could set up a mapping for each WebApplication, you will probably find it simpler to create a mapping to a folder above where you usually publish your WebApplications. You can then use a single mapping for many WebApplications; you just have to specify which application in the URL.

The rest of this section assumes that you publish all your WebApplications under the folder:

```
C:\WWWTest
```

This section also assumes that you wish to access the WebApplications using URLs that start with:

```
http://localhost/test
```

- **To configure the Personal Web Server:**

1. If you have not already done so, open the **Sybase SQL Anywhere 5.0** window, found in the same program group as PowerJ.
2. In the SQL Anywhere window, double-click **Dynamo Configuration** to open the Dynamo Configurator program (which is also available as a property sheet for the Personal Web Server when it is running).
3. As it is starting, the Dynamo Configurator displays a message indicating that configuration changes will not take place until Dynamo applications are restarted. Click **OK** to finish opening the NetImpact Dynamo Configurator.
4. On the **Mappings** page of the Dynamo Configurator, click **Add**. This opens the Mapping Properties dialog box.
5. Under **1. Type in a URL prefix that Web Browsers will reference**, type `/test`.
6. Under **2. Pick a method to access the resource on your machine**, click **FILE**. This specifies that the PWS will use the file system instead of a database for the site.

7. Click **Browse** and select the folder that you want to map to the site:

   `C:\WWWTest`

8. Click **OK**. Note that there is now an entry for `/test` in the list that maps to `c:\WWWTest`. Click **OK** to close the NetImpact Dynamo Configurator.

Now when the Personal Web Server gets a request for a URL starting with `http://localhost/test`, it will look for files under the `C:\WWWTest` folder.

Now you can start the web server.

- **To start the Personal Web Server:**

1. Open the **Sybase SQL Anywhere 5.0** window, found in the same program group as PowerJ.
2. Invoke the Personal Web Server by double-clicking **Personal Web Server** in the SQL Anywhere window.

The Personal Web Server indicates that it is running by placing a small computer icon in the task notification area, or system tray, of your taskbar.

Later you can terminate the Personal Web Server by using the right mouse button to click on the computer icon and then clicking **Exit**. You can also close the SQL Anywhere window.

Setup PowerJ

Now you can change the run options for your WebApplication to use the URL that you have set up with the Personal Web Server.

- **Set the run options to use a URL:**

1. In the Targets window, use the right mouse button to click on your WebApplication target and then click **Run Options**.
2. On the **General** page of the Run Options dialog, click **Publish then run a web browser**. Click **Configure** to open the dialog that lets you choose the web browser and initial URL.
3. Choose the browser that you want to use.
4. Under **Initial URL for the browser**, type the URL for the web page that that you want the browser to start with. You can use the HTML page that PowerJ automatically generates for an applet. For example, if the applet in your WebApplication is called MyApplet and you want to use `MySite` in the URL, you could type:

   `http://localhost/test/MySite/MyApplet.html`

   When you request this URL from the Dynamo Personal Web Server, it will use the folder `C:\WWWTest\MySite\MyApplet.html`.

5. On the **Publish** page of the Run Options dialog, click **Copy the files to a folder**. Under **Folder**, type the folder under `C:\WWWTest` that corresponds to your URL and the mapping you set up. Continuing the example from the previous steps, you would type:

   `C:\WWWTest\MySite`

6. Click **OK**, then click **OK** again to close the property sheet.

Run the program

Once you have set up the run options, you should run the WebApplication target.

Note: If you use the automatically generated HTML file, be sure that you run the release version of your WebApplication. Because the HTML file is generated in the `release` subfolder of your target, it

will not be included in the debug version of your WebApplication. To avoid this limitation, copy the applet tag from the generated HTML file into the WebApplication's `index.html` file, and use `index.html` instead of the generated HTML file.

When you run, PowerJ will build the WebApplication's applet, figure out all the classes that the applet depends on and copy them into the folder that you specified. Then PowerJ starts the browser using the URL that you specified.

At this point you should see your applet working. If it does not work, then you should check the log for the Personal Web Server.

- **To open the log:**

1. In the Windows system tray or task notification area, use the right mouse button to click on the icon for the NetImpact Dynamo Personal Web Server.
2. In the context menu, click **Log**. This opens the log for the web server.

The log contains lines like:

```
GET /test/MySite/MyApplet.html
GET /test/MySite/MyApplet.class
```

If your applet is missing a file, you will see a line that ends in an error and you will see the URL that the browser is using to fetch the file. You can then adjust your deployment accordingly.

Once your WebApplication is working properly with the web server, you may want to change to deploy using a JAR, ZIP, or CAB file. For example, you can add a JAR target to your WebApplication as follows: Change the target dependencies so that the JAR target depends on the applet target and the Web Application target depends on the JAR target. Clean out the `C:\WWWTest` folder. Now change the applet tag in your HTML file to use the JAR file. For example, if the applet tag was:

```
<applet code=MyApplet.class> </applet>
```

you would change it to:

```
<applet code=MyApplet.class archive=MyJar.jar> </applet>
```

If you run the WebApplication target again and look at the web server log, you will see that the JAR file is downloaded but no class files are downloaded (as long as everything is inside the JAR).

Once you have things working locally you can just change the publishing options on the WebApplication target to send things over to the "live" site on your real web server.

Trusted applets

In the interests of security, certain operations (for example, running an ActiveX control) can only be executed by *trusted applets*. There are two ways for an applet to be considered a trusted applet:

- The `.class` files for the applet must all be found in the `CLASSPATH` on the user's machine; or
- The applet must be packaged using a CAB or JAR file that has a digital signature.

Putting class files in the CLASSPATH

If the user downloads the applet and its class files over the Web, the classes are obviously not in the user machine's `CLASSPATH`—the classes must already be present on the user's machine before the applet is invoked. This generally means that support for the applet has to be installed in an independent way. For example, you might give your users an installation CD containing the necessary class files; once the users have installed the files, they can connect with your web site and run your applets.

The `CLASSPATH` for Internet Explorer is given by the registry entry for

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Java VM
```

Your installation program can install the class files under any of the folders listed in the value of `CLASSPATH` or can add a new folder name to the existing list.

Note: For more information about `CLASSPATH`, see CLASSPATH and CODEBASE.

Deploying using a CAB or JAR file

CAB and JAR files are described in Using collection targets: CAB, JAR, and ZIP. As discussed in that section, CAB and JAR files may have associated digital signatures that can be used to authenticate the files.

Chapter 6. Team programming

This chapter describes how to use PowerJ with various source code control systems.

Source code control in PowerJ

This section explains the PowerJ facilities for performing source control in cooperation with various source management systems.

Source control support

Source code control systems are programs that help you manage your source files. They let you make sure that files are accessed in an orderly manner in a group environment, and also store old versions of files.

Within the PowerJ design environment, you can do the following:

- Check files in
- Check files out
- Get the latest revisions of files
- Undo checkout operations
- See the checked-in/checked-out state of files
- Open a new source control project
- Set source control options

For other source control operations and for configuration of your source control system, use the tools that came with your system.

Important: When you open a project in PowerJ, it is important to let PowerJ handle all of the source control activities. For example, if you want to check in a file, do so from within PowerJ. Do not use a tool provided with your source control system to check in a file. If you must perform some source control action on your project files from outside PowerJ, close the project first. If you do not do this, PowerJ may overwrite newer files with older versions, or may not be able to save your project at all.

Supported source control systems

PowerJ can work with the source control systems listed below:

- Powersoft ObjectCycle (version 1.0 and later)
- INTERSOLV PVCS (version 5.1 and later)
- MKS RCS (version 6.2 and later)
- MKS Source Integrity (version 3.2 and later)
- Any system that supports the Microsoft Source Code Control (SCC) interface. These include Microsoft Visual SourceSafe (version 4.0 or later), MKS Source Integrity (version 7.2 or later) and Powersoft ObjectCycle (version 2.0 or later).

If you don't have any of these source control systems, you can configure PowerJ to use a generic source control system in which you specify the commands PowerJ should execute to check files in or out.

You should be familiar with your source control system before configuring PowerJ to use it. The rest of this discussion assumes your source control system is already installed.

Registered systems

When you begin configuring PowerJ to use a source control system, PowerJ attempts to determine which systems are currently installed on your computer. For example, PowerJ may check the registry on your machine and see that both Visual SourceSafe and ObjectCycle are currently installed. PowerJ will let you choose which of these two packages you want to use to control your PowerJ projects.

In some situations, PowerJ may not be able to detect that you have a particular source control package installed on your machine. If this happens, make sure that the package has been installed correctly. In the case of Visual SourceSafe, you should make sure that the software was installed to include Visual Basic support.

Important: Many operations related to source control are disabled until you have configured PowerJ to use a particular source control package. For example, menu items related to checking files in or out are disabled if you have not configured PowerJ to use a source control system.

Configuring PowerJ for ObjectCycle

ObjectCycle stores files in collections called projects. Files within a project are stored hierarchically. Each PowerJ target has its own node in the ObjectCycle project. Target files are stored under the target's node.

You are free to store PowerJ files under any ObjectCycle project you like. You could create one ObjectCycle project for every PowerJ project, or you could create a single ObjectCycle project to hold all PowerJ projects in your organization.

Once you have created an ObjectCycle project for PowerJ, you do not have to worry about creating nodes in ObjectCycle; PowerJ takes care of that for you.

The following steps assume that you want to create a new ObjectCycle project to hold your PowerJ projects. You can also use an existing ObjectCycle project, if you prefer.

- **To set up PowerJ to use ObjectCycle:**

1. Using the *ObjectCycle Manager* program, create a new ObjectCycle project to hold your PowerJ projects.
2. Start PowerJ.
3. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click the appropriate version of ObjectCycle that you are using.
4. Click **OK**.

The next time you open a project, you will be prompted to log in to your ObjectCycle server.

Configuring PowerJ for PVCS

PowerJ is compatible with PVCS versions 5.1 and 5.2. If you are using version 5.2 or later, we recommend you install the Version Manager Interface to Microsoft Development Environments. PowerJ can use this interface to communicate with PVCS. See your PVCS documentation for information on how to install this interface.

- **To set up PowerJ to use PVCS:**

1. Start PowerJ.
2. On the **Tools** menu, click **Options** and then click the **Source Control** tab.
3. If you are using PVCS 5.2 or later, and have installed the Interface to Microsoft Development Environments, click **PVCS Version Manager** under **Source control system**, then click **OK**. This finishes setting up PowerJ to use PVCS, so you can skip to the last step listed below.
4. If you are using PVCS 5.1, click **PVCS** under **Source control system**.
5. If you are using PVCS 5.1, make sure **Show commands as they are executed** is checked.
6. If you are using PVCS 5.1 and you want PowerJ to pause after executing each command, make sure **Pause after each command** is checked.
7. Click **OK**.

Note: If you do not check **Show commands as they are executed**, PVCS may wait indefinitely for input from you when you perform a source control command. If this happens, it will appear as if PowerJ has stopped executing.

By default, PVCS 5.1 creates archive files that have the same name as the original checked-in file, except that the last character of the extension is replaced with the letter `v`. Since some file extensions used by PowerJ differ only in the last letter, PVCS will not be able to check in two files with the same name but different extensions if they are in the same folder. For example, PVCS would try to store both `TEST.WXJ` and `TEST.WXT` in a single archive called `TEST.WXV`. As a result, one of the files would be lost.

There are two ways to resolve this problem:

1. Move the project file out of the target folder.
2. Edit your PVCS configuration file to change the template that PVCS uses when forming the file extensions for archives. For an explanation of how to do this, see the **ArchiveSuffix** directive in your PVCS Reference Guide.

If you are using PVCS version 5.2 or later with the Version Manager Interface to Microsoft Development Environments, you must configure Version Manager for PowerJ to work with it. To do this, follow these steps:

1. Start the PVCS Version Manager program.
2. From the **Project** menu, click **Configure Project**. This opens a configuration dialog box.
3. Under **Categories**, click **New Archives**.
4. Under **Archive Suffix**, replace `"??v___"` with `"?v?___"`. This changes the default file extension for archive files to have a `"v"` in the second character of the extension rather than the last. You have to do this because PowerJ file extensions differ only in the last character.
5. Under **Locking in New Archives**, click **Limit to One Lock per Archive**. This ensures that a file

can be checked out by one person at a time.

6. Click **OK** and then click **Save** to save the changes to the PVCS configuration file.

Configuring PowerJ for Visual SourceSafe

Visual SourceSafe stores files in collections called projects. You may create Visual SourceSafe projects using the Visual SourceSafe Explorer program. If you do not create a project explicitly, PowerJ prompts you to create such a project the first time you check in files for a given PowerJ project.

PowerJ uses the read-only file attribute to determine whether a file needs to be checked out or not. There is a Visual SourceSafe option that makes local files read-only if they are not checked out. Make sure this option is turned on before using PowerJ with Visual SourceSafe.

PowerJ assumes the following Visual SourceSafe settings:

- If you did a custom installation of Visual SourceSafe, you must make sure that the **Visual Basic and Visual C++ Registration** support was installed.
- **Remove local copy after Add or Check In** should be *off*.
- **Use read-only flag for files that are not checked out** should be *on*.

◆ **To set up PowerJ to use Visual SourceSafe:**

1. Start PowerJ.
2. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **Microsoft Visual SourceSafe**. If the **Microsoft Visual SourceSafe** option is not shown, make sure that you installed the **Visual Basic and Visual C++ Registration** option when you installed Visual SourceSafe.
3. Click **OK**.

The next time you open a project, you will be prompted to log on to Visual SourceSafe.

Configuring PowerJ for RCS

- **To set up PowerJ to use RCS:**

1. Start PowerJ.
2. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **RCS**.
3. PowerJ can display RCS commands as they are executed. If you want to see the commands, make sure **Show commands as they are executed** is checked.
4. If you want PowerJ to pause after executing each command, make sure **Pause after each command** is checked.
5. Click **OK**.

Configuring PowerJ for Source Integrity

Source Integrity stores files in collections called projects. PowerJ assumes that strict locking is used for all archive files. This means that files which are under Source Integrity's control must be read-only when not checked out.

Note: You will need to know which version of Source Integrity you are using. For version 7.2 and above, make sure that you installed the optional Visual Basic interface when you installed Source Integrity.

- **To set up PowerJ to use Source Integrity:**

1. Start PowerJ.
2. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click either **Source Integrity** for Source Integrity version 7.1c or earlier, or **MKS Source Integrity SCC extensions** for Source Integrity version 7.2 (with the Visual Basic interface installed).
3. Click **OK**.

Configuring PowerJ for a generic system

If you do not have any of the above source control systems, you can tell PowerJ what command(s) to execute in order to check a file in or out. When you tell PowerJ to use a generic source control system, it calls an external batch file to perform source control operations. You just add whatever commands your source control system needs to the batch file.

- **To set up PowerJ to use a generic source control system:**

1. Using a text editor such as Notepad, open the file `OPT_GEN.BAT` in the `system` subfolder of your main PowerJ folder.
2. Look for the `:doCheckIn` label in `OPT_GEN.BAT`. After this line, add command line directives to check in a file. For example, if your source control system uses a program called `checkin.exe` to check in files, your copy of `OPT_GEN.BAT` should read:

   ```
   :doCheckIn
       c:\bin\myrcs\checkin %_rcsfile%
       goto :done
   ```

   When the batch file runs, the `%_rcsfile%` will be replaced by the name of the file you want to check in. If your checkin program is already in your search path, you do not have to specify the full pathname.
3. Look for the `:doCheckOut` label in `OPT_GEN.BAT`. After this line, add command line directives to check out a file. For example, if your source control system uses a program called `checkout.exe` to check out files, your copy of `OPT_GEN.BAT` should read:

   ```
   :doCheckOut
       c:\bin\myrcs\checkout %_rcsfile%
       goto :done
   ```
4. Look for the `:doRefresh` label in `OPT_GEN.BAT`. After this line, add command line directives to get the current version of a file. For example, if your source control system uses a program called `get.exe` to retrieve files (without locking them), your copy of `OPT_GEN.BAT` should read:

   ```
   :doRefresh
       c:\bin\myrcs\get %_rcsfile%
       goto :done
   ```
5. Save the batch file.
6. Start PowerJ.
7. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **Generic**.
8. Click **OK**.

Out of environment space errors

When you try to check out a file using `OPT_GEN.BAT`, you may get an error that reads:

```
Out of environment space
```

(This should not happen on Windows NT but may happen on Windows 95.) There are two possible ways to correct this problem:

1. You can set the properties for the `OPT_GEN.BAT` file to allow for more environment space. (Look on the **Memory** page of the file's property sheet, under **Initial environment**.) Increase the **Initial environment** size to allow more space for environment variables.
2. Alternatively, you can increase the environment size for all batch files run on your system. To do this, open `CONFIG.SYS` (by running `sysedit`) and edit it to contain the line

   ```
   SHELL=c:\windows\command.com /e:4096 /p
   ```

   The `/e:4096` argument increases the size of the environment space to 4096 bytes. If you use this approach, you must reboot after editing the file in order for the change to take effect.

Generic source control assumptions

If you are using the **Generic** source control system setting, PowerJ must make some assumptions about the nature of the files you are working with.

- If a file is read-only, it is assumed to be checked in.
- If a file is read/write, it is assumed to be checked out by you.

These assumptions are simplistic, but they are the most sensible when PowerJ is unfamiliar with the source control system you are using.

Local files

PowerJ needs to work with a local copy of a project; it cannot load files directly from the source control system's archive.

In addition, the local files that are under source control must be read-only. Most source control systems have an option to make local files read-only; you should turn on this option for your project.

PowerJ does not create the initial local copy of a project's folders and files. You must do this by using the tools that come with your source control system. Once you have created your local copies, you can use the facilities of PowerJ to check out files, check in modifications, and refresh files from the source control archive.

Checking in a project for the first time

If you are using ObjectCycle or Source Integrity, and you're checking in files for a new target, you may have to create a new source control project. This section describes how and when you have to do this. If you are not using a system that requires special setup for new source control projects, you can skip this section.

Some source control systems store files in a hierarchical database. The top level is usually referred to as a project. Don't confuse source control projects with the projects you create with PowerJ. Source control projects are an abstraction used by your source control system to help organize checked in files. PowerJ projects are collections of PowerJ targets.

The following sections give some tips for creating source control projects.

Creating an ObjectCycle project

You can use a single ObjectCycle project to hold files for all of your PowerJ targets. If you have already created an ObjectCycle project, you can skip the following steps. To create a new ObjectCycle project to hold your PowerJ files, do the following:

1. Start the ObjectCycle Manager program. Log in to ObjectCycle as an administrator.
2. On the ObjectCycle Manager's **File** menu, click **New**, then **Project**. The ObjectCycle Manager will prompt for a project name. Type a name, then press **OK**.

Once you have created the ObjectCycle project, you are ready to run PowerJ. Note that you do not need the ObjectCycle Manager to check in or check out files; you should do that from within PowerJ itself. See Opening a new source control project for more details.

Creating a Source Integrity project

To check in a project for the first time, follow the steps below. (These steps assume you are using version Source Integrity version 7.2.)

1. Start the MKS Source Integrity program.
2. On the **File** menu, click **Create Project**.
3. You will be prompted for a project location. You are free to put the project (`.PJ` extension) file wherever you want.
4. Source Integrity will open a **Create Project** dialog. Enter the location of the sandbox (where you want your local files) and select the PowerJ files to add to Source Integrity. Then press **OK**.

Source Integrity will create local copies of the selected files in your sandbox folder. From now on, when you open the project in PowerJ, you should open the WXJ project file in the sandbox folder.

Creating a PVCS project

The following notes on creating PVCS projects apply only if you have installed the Version Manager 5.2 or later, plus the Version Manager Interface to Microsoft Development Environments.

The first time you check in a project, PVCS prompts you for a PVCS project name and an archive directory. Enter the full path of the target folder with `"\pvcs"` at the end. For example, if your project name is

```
pvcsTest
```

you might type

```
c:\Program Files\Powersoft\PowerJ\Projects\pvcsTest\pvcs
```

(depending on the name of the folder where you installed PowerJ). This tells Version Manager to store its archive files in a subfolder called `pvcs`. It's important that the archive files be stored in a different folder than the actual working files.

Checking in files

What to check in

In general, you should check in all of the source files for each target that you want to keep under source code control. If your project has a single target, all of these files are in the target folder and are visible from the PowerJ Files window. A list of the files extensions is given below:

Extension	File Type
wxf	Form file
wxc	Class file
wxt	Target file
wxj	Project file
java	Java source file
htm, html	HTML file
gif	GIF graphic file
jpg, jpeg	JPEG graphic file

If you keep other types of files in your target folders, such as documentation or text files, you are free to check them in too.

Some source control systems are restricted to handling text files—they cannot check in non-text files such as GIF or JPEG files. All of the files listed above are text files except for GIF and JPEG. Check the documentation for your source control system to see if it can handle non-text files.

What not to check in

You don't need to check in any of the following:

- WXU (user setting files). These files hold information unique to you and your computer. They are not designed to be shared between developers.
- PowerJ backup files. When PowerJ is running, it makes a backup copy of the files you're working with. Backup files have the same name as the original except that the second character of the file extension is replaced with a tilde. For example, the backup file for `FORM1.WXF` is `FORM1.W~F`.
- Anything in the `Release` or `Debug` folders. Although you can find Java source files here, they are all generated from other files found directly under the target folder. If you check out files in the `Release` or `Debug` folders, PowerJ may not be able to keep these generated files in sync with the real source files.

How to check in files

Normally you will want to check in all files that you have checked out to ensure a consistent version of

the project in the repository; however, you can also select which files to check in.

Important: You must close your project in PowerJ before using a tool other than PowerJ to check in files.

- **To check in all the files that you have checked out:**

1. In the main PowerJ **File** menu, click **Check In Project**.

This checks in all of the files that you have checked out. PowerJ then scans the target folders for files that are not currently under source control. If it finds any, it will display the list of file names and ask if you want to check in the files.

In the Files window, a picture of a closed padlock appears to the left of all files that are checked in.

- **To check in a file or a number of files:**

1. Open the Files window
2. Select the file or files you want to check in
3. Use the right mouse button to click one of the files you have just selected, then click **Check In**.

Note: PowerJ lets you check in any file that appears in the Files window. However, not all source control systems can handle all types of files. For example, some systems let you check in binary files, while others do not. Check your source control system manual for details.

Checking out files

PowerJ has an auto-checkout feature which simplifies the checkout process. If you do something in the design environment that would change the contents of a read-only file, PowerJ asks if you want to check out the file.

Suppose you are working on a target which contains a checked-in form file called `MYFORM.WXF`. The first time you change a property of this form or add a control to it, PowerJ asks if you want to check the file out. If you don't check out the file, any changes you make will be lost when you close the project.

You can also check out files from the Files window.

Important: You must close your project in PowerJ before using a tool other than PowerJ to check out files.

- **To check out a file or a number of files:**

1. Open the Files window.
2. Select the file or files you want to check out.
3. Use the right mouse button to click one of the files you have just selected, then click **Check Out**.

A picture of an open padlock appears to the left of all files that are checked out.

If the latest copy of a file is different from your current local copy, PowerJ automatically reloads the file in the design environment.

Removing write protection

By default, PowerJ removes the write protection on a file during the check-out process. However, some source control packages issue a warning message when this happens; the package may or may not offer you the option of suppressing this warning. If you don't want to see the warning every time you check out a file and your package offers no way to suppress the warning, you can prevent PowerJ from removing the write protection. To do this, you need to add a new registry string value under the `HKEY_LOCAL_MACHINE\Software\Powersoft\Optima\2.3` key. The name of the string should be `Remove Write Protect Before Checkout` and its value should be zero to prevent PowerJ from removing the write protection.

Undoing a check out operation

After checking out files and changing them, you may decide to throw away the changes you have made. PowerJ can discard your changes if you have not checked them in. PowerJ can also unlock the files and get the latest versions of the files.

Important: You must close your project in PowerJ before using a tool other than PowerJ to undo a check out operation.

- **To undo a check out:**

1. Open the Files window.
2. Click the checked-out files that you want to unlock.
3. Use the right mouse button to click one of the files you have just selected, then click **Undo Check Out**.

The open padlock turns back to a closed padlock, indicating that the file is no longer checked out.

If the latest copy of a file is different from your current local copy, PowerJ automatically reloads the file in the design environment.

Getting the latest revision of a file

Important: You must close your project in PowerJ before using a tool other than PowerJ to get the latest revision of any project file(s).

- **To replace your local copy of a file with the latest revision of that file:**

1. Open the Files window.
2. Click the files you want to get.
3. Use the right mouse button to click one of the files you have selected, then click **Get Latest Version**. PowerJ replaces your local copies of the selected files with the latest versions.

If the latest version of a file differs from your current local copy, PowerJ automatically reloads the file in the design environment.

The **Refresh Project** item of the main PowerJ **File** menu refreshes each file in each target of the current project with the most recent version of the file. However, **Refresh Project** does *not* overwrite any files that you currently have checked out.

Some source control systems (for example, ObjectCycle) support the ability to get the latest copies of all checked in files, regardless of whether you currently have local copies of the files. If you are using such a system, the **File** menu of the Files window will contain the item **Refresh Target**. As with **Refresh Project**, **Refresh Target** does not overwrite any files that you currently have checked out.

Opening a new source control project

When a new programmer is added to an existing development project, the programmer can download all the files of the existing project to the local system in order to begin development.

Note: This feature is not supported with some source control packages—it depends on functionality that the SCC specification regards as optional. Packages that do not implement this optional functionality (for example, ObjectCycle) cannot use this feature. The feature is also unavailable with source control packages that are not SCC-compliant.

On the **File** menu of the main PowerJ menu bar, **Open New Source Control Project** performs the following actions:

- Closes the current project.
- Prompts you to enter the name of a source control archive.
- Downloads files from that archive to your local disk.
- Opens the project

In other words, PowerJ does everything needed to bring the project to your system and set things up so you can begin development on the project.

Source control options

PowerJ makes it possible for you to control various options provided by your source control package. These options are only supported for source control systems that are SCC compliant (for example, Visual SourceSafe, ObjectCycle, or MKS Source Integrity).

- **To set source control options:**

1. On the **Tools** menu of the main PowerJ menu bar, click **Source Control Options**.

Note: The **Source Control Options** item is disabled until you have configured PowerJ to use a particular source control package.

The options are different for each source control package. With some source control packages, you can only set source control options if the current project is controlled by the specified package.

Chapter 7. Coexisting with other tools

This chapter discusses the ways in which PowerJ can be used in connection with other software tools.

Integrating PowerJ and Microsoft FrontPage

A web site is simply a collection of files. Some of these files contain only content: HTML, JPG and GIF files are examples of content files. Some files in a web site are executable, such as Java applets or servlets. Creating and maintaining the two types of files requires different skills and tools.

Those responsible for the content files need a sense of aesthetics and layout. They need tools that help them maintain the links between the content files that make up the web site. They need editors that let them modify the various files. Those responsible for the executable files need an understanding of computer programming. They need tools that help them build source code into executables and debug those executables.

The integration between FrontPage and PowerJ is based on this division of tasks. FrontPage and PowerJ may be used as different views on the same web site. Both those responsible for content and those responsible for executable files will work on the same files but they will use different tools because they have different needs.

You would likely use FrontPage to edit the web site's content files. It contains file management facilities that let you view the links between the files in the web site and functions that let you verify that all the links are valid. It also contains a graphical HTML editor.

You would likely use PowerJ to work on the executable files in the web site. It contains facilities that let you manage the dependencies between source files and the executable files built from them. It allows you to build executable files from source files and debug them.

While either tool can be used to view the web site, you may not edit the web site in both tools simultaneously on one machine. If you do, the two tools may overwrite each other's files and the revision control support in PowerJ will not work correctly.

Control Files

Each product creates its own control files, used to store information about the files in the web site. PowerJ generates WXT, WXJ and WXU files. FrontPage stores control files in folders that begin with `_vti_`. Neither PowerJ or FrontPage will read or write the other's control files. Information about the state of the web site will not be shared between the two tools.

Creating a Web Site

The web site can be initially created using either PowerJ or FrontPage. You simply follow the usual procedure to create a new web site using either tool.

Before the web site can be edited using the other tool, you must go through the process of creating a new web site using that tool. When asked for the folder for the new web site, you enter the folder created by the first tool for the web site.

Both PowerJ and FrontPage may display dialog boxes warning that there are already files in the folder you have chosen. If so, choose the dialog box option that continues without deleting files. Any files created by the tool that initially created the web site will not be disturbed.

File Management

FrontPage considers all files in the web site folder to be part of the web site. Any executable files added to the web site by PowerJ appear as part of the web site when it is viewed with FrontPage.

PowerJ maintains its own list of the files in the web site. When files are added to the web site using FrontPage, they only appear in PowerJ Files window. They will not appear in the Targets window unless you specifically add them to the PowerJ web target.

Revision Control

Different skills are needed to edit the content and executable files in the web site. In most cases, separate people will work on these two types of files. A revision control system provides the tools needed to let several people share access to the various files in the web site. It lets you set up a central repository for the web site files where everyone can retrieve the most up-to-date copies of the files.

PowerJ has built in support for many popular revision control systems including Visual Source Safe. It also has a generic batch file scheme that will allow it to work with almost any revision control system. When editing the web site with PowerJ you will be able to use the built in revision control support.

If you use both tools to view the project you will be able to do revision control operation using the facilities available in either environment. For example, you could check a file out using whatever revision control support is available in FrontPage, then check that file in later using PowerJ. However, you must not have the same web site open in both tools on one machine at a time.

Publishing

Publishing is the process of copying files from the folder where PowerJ or FrontPage edits them to the folder where a web server accesses them. Both FrontPage and PowerJ will recognize and publish files added to the web site by the other. Publishing can be accomplished using either tool.

PowerJ recognizes control files created by either tool and does not publish them. FrontPage publishes control files created by PowerJ.

FrontPage supports publishing by copying all the web site files to another folder. PowerJ can also publish using the WebPost API.

HTML Editing in PowerJ

The **File Types** page of the PowerJ Options dialog box lets you choose the editor used to edit a file with a particular file name extension. For example, you can specify that you want to edit HTML files with the FrontPage HTML editor. For further information, see File type options.

Alternate versions of the JDK

The PowerJ package includes the standard versions of JDK 1.02 and JDK 1.1; the package also includes Microsoft's SDK for Java 1.5. However, it is possible that you will want to use different versions of these in your applications. For example, Sun is expected to release updated versions of the JDK from time to time. When Sun creates a new release of the JDK, you may wish to use that release instead of the one that you received with PowerJ.

PowerJ makes it easy to use your own version of the JDK. For further information, see JDK configuration options.

Importing from other Java environments

PowerJ can import Java source files from other Java environments, such as Symantec Visual Café. The importer adds the Java source files to a target as forms or managed classes. This allows you to make full use of the PowerJ environment with the imported code. For more information on managed classes and how to edit them, see Adding classes to a target.

You must import source files to a specific target. You can choose an existing target or create a new one during the importing process.

Note: You can also use the **Add File** command to add source files to a target. The difference between *adding* and *importing* is that the importer adds the code as forms or as managed classes, rather than `.java` files, providing you with more powerful editing capabilities. For more information on adding files to a target, see Adding source files to a target.

- **To import one or more Java source files:**

1. From the **Tools** menu of the main PowerJ menu bar, click **Import Java Files**. This opens a dialog box where you can specify which files you want to import.
2. In the **Enter the files(s) you want to import** field, specify the files that you want to import. These files will typically have `.java`, as their file name extension. Visual Café files (`.vpj`) are also supported, in which case, PowerJ adds all the `.java` files from the Visual Café project. You can select files using the **Browse** button or you can type the file names directly.

   If you choose to type file names, wild cards are accepted (for example, `*.java`). In this case, you may want to use the **Search subfolders for matching files** option, so that PowerJ also searches subfolders.
3. If you want to import the files into an existing target, click **Add to an existing target** and click the desired target name in the list of targets.
4. If you want to import the files into a new target, click **Create a new target**, type a name for the target under **Name**, and click the type of target you want to create under **Type**.

   For more information about target types, see the section Target types.
5. Click **Options**. The Import Options dialog box appears.

   By default, PowerJ imports any class that has a form type base class (Applet, Panel, Frame, or Dialog) as a form. If you want to import such a class as a managed class rather than a form, you must import this class separately, and use the **Always generate managed classes** option. Select the preferred import option and click **OK**.
6. Click **Import**. PowerJ displays a progress bar which shows the progress of the importing operation.
7. When PowerJ has finished the importing operation, it displays a dialog box listing the imported files and indicating success or failure. Click **OK**.

Note: The import procedure is not complete until you have verified the imported code; as described in the next section.

For a detailed description of how the PowerJ importer works, see How importing works.

Completing the import

The final step of the import process is to verify the imported code and confirm that all controls, properties, and events are imported and set properly.

You may encounter the following conditions which require you to manually adjust the imported code.

Unrecognized code

- If the importer does not recognize any property initialization code, it puts the code into a method named **__extra_init**. The code is labeled with the source file line numbers and is commented out. Examine this function for code which needs to be added to the **ObjectCreated** event or the constructor.
- If the importer does not recognize any event dispatch code, it puts the code into a method named **__extra_event**. The code is labeled with the source file line numbers and is commented out. You should look in this method for event code that you still want to include. You can then use PowerJ to create event handlers and copy and paste the code into those handlers.
- The importer adds any class fields which were recognized (and possibly used in the property initialization) but not imported as form controls to the data members section. This code is commented out. Review the data members section and un-comment any required fields.

Forms and dialog boxes

- You may need to reset the main form after importing. For more information on setting the main form, see the section Adding forms to a target.
- The form size may not always be exactly the same as the original project. Confirm the size and position of all forms and controls, and adjust if required.
- By default, the importer imports dialog boxes as modal dialog boxes. If you would rather have a modeless dialog box, modify the dialog box's constructor by setting the modal parameter in the superclass constructor call to `false`.

How importing works

The importer uses pattern recognition to import native Java files. The importer scans the classes in each Java file specified, and imports each as either:

- a form

or:

- a managed class

Note: The PowerJ importer assumes that the Java code is syntactically correct (that is, it will compile successfully).

By default, a class is imported as a form if the base class is a form type—either Applet, Panel, Frame, or Dialog. If you wish to import such classes as managed classes instead, click **Always generate managed classes** under **Options** so that it is checked.

If the base class is not a form type, the class is imported directly (without modification) as a managed class.

Importing forms

When a class is recognized as a form, the PowerJ importer takes several additional steps. The importer scans:

- instance data for control declarations
- property initialization code
- event handling methods

Detailed information about each of these steps is provided below.

Control declarations

The importer scans the instance data for control declarations, and adds all recognized controls to the form.

Property initialization methods

The importer scans the constructor and, for applets, the **init** method for property initialization code.

The importer assumes that each line in the property initialization code has one of the following forms:

```
setmethod( someArgs );
control.setmethod( someArgs );
control.field = someArgs;
control = new className( someArgs );
```

In addition, the importer only recognizes literal values or objects containing literal values as arguments.

The importer puts any code not in this form or which uses unrecognized methods or controls in an **__extra_init** method, and comments it out.

For example:

```
button1.reshape( 10, 10, 40, 20 );
button1.setFont( new Font( "Dialog", Font.PLAIN, 12 ) );
```

This code is recognized by the importer. The importer creates code to set the **X**, **Y**, **Width**, and **Height** properties of the `button1` control to 10, 10, 40, and 20 respectively, and the **Font** property to be set to 12 point, plain, and to use the Dialog font. In contrast, the importer does not recognize the following code because `getXCoord()` and `getYCoord`() are not literal values:

```
button1.reshape( getXCoord(), getYCoord(),10, 10 );
```

The importer adds this code to the **__extra_init** method as comments.

Event handling methods

The PowerJ importer scans the **handleEvent** method and, for JDK 1.1, event listeners, to determine control events and event handling code. In **handleEvent**, the importer assumes that the code has one of the following forms:

```
if( event.target == control  &&
    event.id == Event.event_id ) {
    // event handling code goes here
    return true;    // optional
}
```

or:

```
if( event.id == Event.event_id ) {
    // event handling code goes here
    return true;    //optional
}
```

In event listener methods, the importer assumes that the code has the form:

```
if( source == control ) {
    // event handling code goes here
    return true;    //optional
}
```

The importer puts any code not in this form in an **__extra_event** method, and comments it out.

Additional event handling methods, such as **action**, are imported directly.

Customizing the import process

The file `system\jimport.txt` under the main PowerJ folder contains information that controls the import process. You will probably never need to touch this file. However, if there are special conversions that you want to perform during an import process, you may be able to do this automatically by placing an appropriate entry in this file.

For more information, see the file itself. It contains comments describing how to create an entry that performs an import conversion.

Using Java tutorials

If you are a new Java user, you may find it useful to do the beginner programming exercises provided in an introductory Java book, such as *Thinking in Java* by Bruce Eckel.

Similar to the *PowerJ Getting Started* guide, these books are designed to walk you through simple Java programs, explaining each step along the way. Most of these books, however, are designed to be used with a command line Java compiler, while PowerJ is a RAD tool, designed to help you create windowed applications and applets. PowerJ supplies a framework to help you create such programs with ease by taking care of a lot of the details.

When you create a new target with the PowerJ framework, PowerJ assumes that you wish to create a windowed application or applet, and automatically generates a main form and startup code for your target. For more detailed information about the PowerJ framework, see Using the PowerJ framework.

In order to set up PowerJ to handle non-windowed applications, such as those suggested in *Thinking in Java*, you need to create a new target which does *not* use the PowerJ framework. You specify whether or not you would like to use the framework during the target creation process.

For example, suppose you wish to do the first exercise in Chapter 2 of *Thinking in Java*. This exercise asks you to write a simple Java application which outputs the text "Hello, World". The first step is to create Java code. Instead of using a text editor and typing all the code, you create a new target in PowerJ and only type the body of the code.

- **To create the Hello World application:**

1. On the **File** menu of the main PowerJ menu bar, point to **New** and click **Target**. This opens the Target Wizard which takes you step-by-step through the creation of the new target.
2. The Target Wizard displays a list of targets that PowerJ can create. Click the **Java - Application** icon as the type of target you want to create.
3. The Target Wizard asks you if you wish to use the PowerJ framework. Click the **Use the PowerJ framework** checkbox so that it is *not* selected, then click **Next**. The framework allows you to manage forms and use the drag and drop programming features. However, the *Hello World* application is a simple command line program which does not require visual components such as forms.
4. Select which class library you'd like to use by clicking either Powersoft **Java AWT 1.02** or **1.10**, then click **Next**. You can create the application with either class library.
5. Type `Hello` as the name for the new target, and specify where to store the files associated with the new target. Typically, you will create a new folder under the PowerJ `Projects` folder, using the name you just chose for the new target. Specify a location, then click **Finish**.

   PowerJ creates the new target and all of the necessary source files. PowerJ automatically creates a class named `Main` which contains the **main** method, and opens a code editor window which displays this class and method. You can now enter the remainder of the Java code for the *Hello World* application.
6. After the opening brace of the **main** method, start a new line and type:

```
System.out.println("Hello, World");
```

Note that you only need to add this line at the appropriate location in the file because PowerJ has created the main method for you. The final code should look like this:

```
public class Main
{
```

```
        public static void main( String[] args )
        {
            System.out.println("Hello, World");
        }
    }
```

Note: Whenever you start your PowerJ session, PowerJ creates a default target named `Untitled`. If you create a new target, and the existing target is an unmodified version of the default target, PowerJ automatically deletes the original (default) target.

Now you can use PowerJ to save the application. In order for a Java file to be compiled, the file must have the same name as the public class within that file, with the extension `.java`. PowerJ handles the file naming for you. When you build or run your project, PowerJ creates a `.java` file which it places in the `Release` folder of the target. This file contains your Java code, which PowerJ automatically generates each time you save your project. You can view this file in the **Files** window by double-clicking on the file name, however, if you edit this file your changes will be lost the next time you build your target.

- **To save the Hello World application:**

1. On the **File** menu of the main PowerJ menu bar, click **Save Project**. This displays the contents of the `Projects` folder in your PowerJ folder.
2. Under **Folder name**, type the name `Hello` and click **Save**. This creates a folder, called `Hello`, for the target.
3. Under **File name**, leave `Hello.wxj` as the project file name, and click **Save**. PowerJ creates a project file named `Hello.wxj`, in the folder named `Hello`, along with other target files.

   If you open the **Files** window and expand the `Hello` target, you can see a file named `Main.java` in the `Release` folder. PowerJ has created this file to correspond with the public class named `Main` in your application.

Now you are ready to run your application. Clicking the **Run** button in PowerJ is equivalent to compiling your Java code with a command line compiler (e.g. `javac.exe`) and then executing the application (e.g. by typing `java classname`). Each time you modify and run the target, PowerJ compiles the Java code and generates a class file which contains the Java bytecodes. PowerJ places this file in the `Debug` folder within the main target folder, and regenerates this file each time it recompiles the target.

- **To run the Hello World application:**

1. On the **Run** menu of the main PowerJ window, click **Run**.
2. If there is more than one executable target in your project, Power J asks you which program you would like to run. Select the `Hello` target, and click **OK**.

PowerJ runs the target. A Java Console appears, displaying the text "Hello, World". If you open the **Files** window, and expand the `hello` target folder, you can see a file named `Main.class` in the `Debug` folder. This file contains the bytecodes for your application which is read by the Java Interpreter when your target is run.

Applets and the PowerJ Framework

The creation process for a non-framework applet is much the same as for an application.

When you create a non-framework target, PowerJ automatically creates a class named `Main` which contains the **init** method, and opens a code editor window which displays this class and method. PowerJ also generates an HTML file for the new applet called `main.html`. This file is stored in the target folder and can be opened for editing through the **Files** window.

- **To create and run a non-framework applet:**

1. Create a **Java - Applet** target which does *not* use the PowerJ framework.

When you have entered all the required information, PowerJ creates the target and all the necessary source files.

2. Add the Java code for your applet in the code editor window.
3. On the **File** menu of the main PowerJ menu bar, click **Save Project**. This displays the contents of the `Projects` folder in your PowerJ folder.
4. Under **Folder name**, type a name for your applet and click **Save**.
5. Under **File name**, leave the given name or type a new name for the project file, and click **Save**. PowerJ creates a project file along with other files and folders of the project.

   If you open the **Files** window, and expand the applet target, you can see a file named `Main.java` in the `Release` folder. PowerJ has created this file to correspond with the public class named `Main` in your applet.
6. Run the applet by clicking **Run** on the **Run** menu of the main PowerJ window.

   PowerJ builds the project, and executes the applet in the Applet Viewer, depending on the run options you have set. For more information about Run options for Java applets, see Java applets.

Part II. Programming fundamentals

This part describes fundamental PowerJ programming topics: standard types, the basics of using Java components, working with standard objects, creating your own menus, and using forms, graphics and multiple threads.

Chapter 8. Standard types

This chapter examines a number of basic classes used by the PowerJ component library. Many of these come from the AWT library, while others are unique to PowerJ.

Frequently used types

The PowerJ component library provides a class for every kind of object that can appear on a form. For example, text boxes are represented by the TextField and TextArea classes and command buttons are represented by the Button class.

The PowerJ library also defines a number of general purpose data types which are used in a variety of contexts. This section examines the types which you are most likely to see when you begin programming with PowerJ. This includes some standard Java library classes as well as PowerJ classes.

For complete details on all PowerJ classes, see the *PowerJ Component Library Reference*.

Fully qualified Java names

Fully qualified Java names

The code you write in PowerJ draws on many different libraries (for example, the standard PowerJ library and AWT). In some cases, there are conflicts between names used in these libraries.

Java lets you resolve these conflicts by specifying *fully qualified names* for various items. A fully qualified name specifies where the name is defined. For example, the following classes are all related to labels:

```
java.awt.Label                // AWT Label
powersoft.powerj.ui.DBLabel   // Label with database
                              //   capabilities
```

The names specify whether the class is defined as part of the Java AWT itself or by PowerJ.

All fully qualified names in the PowerJ library start with `powersoft.powerj`. Within this library are several subsets:

`db`	Database-related classes
`event`	Event-related classes
`net`	Internet components
`server`	Web server components
`ui`	User interface components
`util`	Utility classes

To identify where a class is defined, look under the `Java` subfolder of your main PowerJ folder. The class hierarchy is defined in subfolders of `Java`. You can check these subfolders to find exactly where a particular class is defined.

Fully qualified names are also important when placing `import` statements in your source code. For example,

```
import powersoft.powerj.util.*;
```

imports information about all the utility classes defined for PowerJ. (Remember that names used in `import` statements are case-sensitive.)

You will see fully qualified names in much of the code generated by PowerJ and in many of the examples in this guide.

Classes vs. interfaces

In addition to object classes, Java supports the concept of *interfaces*. As the name suggests, an interface definition lists a set of methods that can be used on a particular object, but does not actually define those methods. (It may, however, define symbolic constants to be used in connection with the listed methods.) An interface definition looks much like a class definition, except that the methods within the interface are not actually defined with Java code.

One simple example of an interface is the Adjustable interface in JDK 1.1 (`java.awt.Adjustable`). This specifies a set of methods for working with objects which have an adjustable value that lies within a range of values. The Adjustable interface specifies methods like **getValue/setValue**, **getMaximum/setMaximum**, **getMinimum/setMinimum**, and so on that should be supported by any object that has an appropriately adjustable value.

You cannot create objects of an interface type. For example, you cannot declare an object to have the Adjustable type. Instead, you must create a class that *implements* an interface type. For example, you might create a class definition that begins with

```
public class MyAdjustable implements Adjustable
{
    // class definition
}
```

When a class implements a definition, it provides actual definitions for all the methods listed in the interface. For example, MyAdjustable would have to define methods named **getValue**, **setValue** and so on with the same prototypes given in the Adjustable interface definition. While the interface definition just lists the prototype for a particular method, the class definition will have to provide actual Java code implementing the method.

In general then, an interface specifies a number of operations that can be carried out on a particular kind of object. When a class implements the interface, it must define actual code for each of the operations that the interface specifies.

Object class

The Object class (`java.lang.Object`) is the base class for all other AWT classes in Java and PowerJ. This means that Object can serve as a "generic" class in many contexts. For example, you can declare a function to take an Object argument if you want the function to accept several different kinds of objects.

Since all other classes are derived from Object, the methods defined for Object can be applied to an object of any standard Java or PowerJ class.

The standard constructor for an Object is

```
Object obj = new Object();
```

This creates an object with no contents.

Note: C and C++ programmers should note that Java typically uses the Object class in situations where C/C++ would use void *.

Comparing Object values

The **equals** method compares two Object values to determine if they are identical.

The **equals** method is not very useful in the Object class itself, rather it is intended for classes which extend Object. Each class should define its own version of **equals**, to reflect what it means for two objects of that type to be equal. In particular, most classes define **equals** to identify two objects as equal if their data members have equal values.

The standard form for using **equals** is:

```
boolean result = object1.equals( object2 );
```

The result is `true` if both objects are deemed equal and `false` otherwise.

For the Object class itself, which has no data members, **equals** only returns `true` if `object1` and `object2` are the *same* object.

You can always check if two objects are the same object with a comparison:

```
object1 == object2
```

No object is equal to the special `null` object.

The clone method

The **clone** method creates a duplicate of the original object, as in

```
// Object oldObj;
Object newObj = oldObj.clone();
```

The new object is a bit-for-bit duplicate of the original.

By default, the **clone** method is `protected`; if code outside the class definition tries to **clone** an object, the code receives an exception. However, a class can make **clone** available to outside code by writing a new `public` definition of **clone** and implementing the Cloneable interface. For more information, see one of the references on the Java language.

The toString method

The **toString** method returns a String representation of an object. Since the **toString** method is defined in the Object class, all Java objects have a **toString** method. This means that you can display all Java objects in a text string form. The following shows an example:

```
// int i;
System.out.println( i.toString() );
```

This uses **toString** to obtain the value of `i` as text, then prints out the result.

When you create Java classes of your own, you should consider overriding the basic **toString** method of Object with a version that is specific to your class. For example, if you create a class representing a data structure, you might define a **toString** method for that class which displays the data structure in some easy-to-read form. This gives you a simple way to "dump" the contents of the structure during debugging.

Simple classes based on Object

The Java library defines a number of classes which are based on Object but correspond to simple data types. For example, the Integer class corresponds to the `int` data type but is based on Object.

These classes are commonly used when a function requires an Object argument and you want to pass a simple value. For example, suppose that `func` takes an Object argument. You cannot call the function with

```
func( 0 );
```

because the `int` value zero cannot be converted to the Object class. However, you could use:

```
func( new Integer(0) );
```

This creates a new Integer object with the value zero, which *can* be converted to Object.

The simple classes of this type are:

```
Boolean
Byte
Character
Double
Float
FloatingDecimal
Integer
Long
Number
Short
```

These are all defined in the `java.lang` package (for example, `java.lang.Boolean`).

Note: Objects of the above classes are *immutable*. Once they have been assigned a value (in the constructor), the value cannot be changed. Objects that can be changed after the initial assignment are *mutable*.

String class

The String class (`java.lang.String`) represents a string of characters. All string constants (for example, `"Hello"`) are created as objects of the String class.

The String class is a standard Java class, defined fully in the *Java SDK*. String is derived from Object, so all String values are also Object values.

All String objects are immutable; their values cannot be changed once they are created. However, you can change the value of a String variable by assigning it a new String object value. If you want to work with strings whose contents can be changed, use the StringBuffer class (described in StringBuffer).

Note to C and C++ programmers: The text stored in a String object may include any character, including null characters.

Creating a string

The simplest constructor for a String object is simply

```
String str1 = new String();
```

This creates a string that contains no characters.

Another simple constructor is

```
String str2 = new String( "abc" );
```

which creates a String object containing the given character string.

You can create String objects from char arrays, as in

```
char arr1[] = { 'a', 'b', 'c' } ;
String str3 = new String( arr1 );
```

You can also create String objects using substrings of char arrays:

```
char arr2[] = { 'H', 'e', 'l', 'l', 'o' } ;
String str4 = new String( arr2, offset, count );
```

In this example, `offset` gives the character position where the substring starts (the first character in the string has a position of zero) and `count` gives the number of characters in the substring.

There is also a copy constructor that creates a new String with the same contents as an existing String:

```
String str5 = new String( str4 );
```

This is actually the format used when you specify a string constant as the argument for the constructor.

Finally, you can create a String with the same contents as an existing StringBuffer:

```
StringBuffer strBuf = new StringBuffer();
String str6 = new String( strBuf );
```

There are several other constructors defined for String but the ones just listed are the most commonly used forms. For more information about String constructors, see the *Java SDK*.

Simple String methods

The **length** method returns the length of a string:

```
int len = str.length();
```

This means the number of characters in the text of the String.

The **concat** method creates a String object consisting of one string concatenated on the end of another:

```
String str1 = "abc";
String str2 = "def";
String str3 = str1.concat( str2 );  //abcdef
```

It's important to note that **concat** does not change the contents of the String object. Instead, it creates a third String that is the concatenation of the original two.

Important: If you are concatenating a number of character strings, it is much more efficient to use the StringBuffer class than String.

The methods **toUpperCase** and **toLowerCase** return new strings which match the original string converted to upper or lower case:

```
String str = "String Sample";
String upr = str.toUpperCase(); //STRING SAMPLE
String lwr = str.toLowerCase(); //string sample
```

The **trim** method returns a new string which matches the original string with white space removed from the beginning and the end:

```
String str = "  Hi! ";
String trm = str.trim();      // "Hi!"
```

For the purposes of **trim**, a character is considered white space if its value is less than or equal to `\u0020` (the space character).

The **replace** method returns a new string with all the occurrences of one character replaced by another:

```
String oldStr = "Robin" ;
String newStr = oldStr.replace( 'i', 'y' );
```

produces `"Robyn"`.

Comparing strings

The String class has a number of methods for comparing strings. The **equals** method compares two strings for equality:

```
boolean result = str1.equals( str2 );
```

returns `true` if the two strings represent the same sequence of characters. Note that this is not the same as the version of **equals** in Object. For two Object values to be equal, they must refer to the same object; for two String values to be equal, they only have to refer to the same sequence of characters.

The **equalsIgnoreCase** method is similar to **equals** but ignores the case of characters that have upper or lower case:

```
String str1 = "Hello!" ;
String str2 = "HELLO!" ;
boolean result = str1.equalsIgnoreCase( str2 );  // true
```

In the above example, `result` is `true` because the two strings are identical except for the case of the letters.

The **compareTo** method determines the "sorting order" of two strings:

```
int i = str1.compareTo( str2 );
```

returns a positive value if `str1` is lexicographically greater than `str2`, returns zero if the two strings are equal, and returns a negative value if `str1` is lexicographically less than `str2`:

```
String str1 = "abc" ;
String str2 = "def" ;
int i = str1.compareTo( str2 );  // i < 0
```

The **endsWith** method determines whether a string ends with a specified sequence of characters. For example,

```
boolean result = str.endsWith( "xyz" );
```

returns `true` if the string ends with the given characters and `false` otherwise. Similarly, the **startsWith** method

```
boolean result = str.startsWith( "abc" );
```

returns `true` if the string begins with the given characters and `false` otherwise.

There is a second form of **startsWith** that specifies an integer offset value:

```
boolean result = str1.startsWith( str2, offset );
```

This returns `true` if `str1` contains the string `str2` beginning at the given `offset`. The beginning of the string has an offset of zero, the next character has an offset of `1`, and so on. Therefore,

```
String str = "Smile!"
boolean result = str.startsWith( "il", 2 ) ;
```

returns `true`.

Obtaining substrings of strings

The **substring** method returns a new String object containing a substring of the original String. There are two forms:

```
String sub1 = str.substring( offset );
```

returns the substring beginning at the given offset and extending to the end of the string. The offset of the beginning of the string is zero, the next character is `1`, and so on.

```
String sub2 = str.substring( start, end );
```

returns the substring beginning at the given `start` offset and ending at the `end` offset minus one. The following code shows an example:

```
String str = "Hi there!";
String sub = str.substring(3,8);  // "there"
```

Notice that the `start` and `end` arguments are specified so that

```
start - end
```

is the length of the substring.

Comparing substrings

The **regionMatches** method of String compares selected substrings of two strings. The simplest form of **regionMatches** is

```
boolean result =
     str1.regionMatches( offset1, str2, offset2, len );
```

where `offset1` is the integer offset of a region in `str1`, `offset2` is the integer offset of a region in `str2`, and `len` is an integer giving the number of characters to be compared. For example,

```
boolean result = str1.regionMatches( 0, str2, 0, 10 );
```

compares the first 10 characters in both strings.

Another form of **regionMatches** is

```
boolean result =
  str1.regionMatches(ignoreCase, offset1, str2, offset2, len);
```

`ignoreCase` is a boolean argument. If it is `true`, **regionMatches** ignores the case of letters when comparing the two substrings.

Searching strings

The **indexOf** method searches for the first occurrence of a character or substring in a String. The simplest form is

```
int pos = str.indexOf( ch );
```

where `ch` is a char value. The result is the position of the first matching character in `str`. The position of the beginning character of the string is zero, the next character is `1`, and so on.

You may also use the form

```
int pos = str.indexOf( ch , start );
```

where `start` is an integer value giving the position where the search should start.

There are two similar forms for searching for substrings:

```
// String sub;
int pos1 = str.indexOf( sub );
int pos2 = str.indexOf( sub, start );
```

These find the position of the first matching substring in `str`.

In all cases, **indexOf** returns `-1` if there is no matching character or substring.

The **lastIndexOf** method is similar to **indexOf**, but looks for the *last* matching character or substring in the string. This method has the same four forms:

```
int pos1 = str.lastIndexOf( ch );
int pos2 = str.lastIndexOf( ch, start );
int pos3 = str.lastIndexOf( sub );
int pos4 = str.lastIndexOf( sub, start );
```

Again, **lastIndexOf** returns `-1` if there is no matching character or substring.

Converting an object to a string

The **toString** method of the Object class creates a String object whose contents represent an Object:

```
// Object obj;
String str = obj.toString();
```

The string that results from **toString** depends on the type of object. For example, AWT components include the name of the class and the settings of various properties in the string. On the other hand, **toString** in the `java.lang.Integer` class simply returns the string representation of its numeric value.

Tip: The **toString** method is often used for debugging purposes, when you want a quick and dirty way of displaying the current "value" of an object.

Converting data into strings

The **valueOf** method of the String class produces a string that provides a textual representation of another type of data value. For example, this lets you "convert" a number into a string containing the number in text form. There are several forms of **valueOf**, and all of them are static methods within the String class. Therefore, the methods are commonly used in the form

```
String str = String.valueOf( value );
```

As shown, you use the name of the class as a whole (String) rather than the name of a specific String object.

The `value` argument of **valueOf** can have any of the following types:

boolean
: The resulting string is either `"true"` or `"false"`.

char
: The resulting string contains a single character equal to the argument character.

char [ ]
: The resulting string contains the same sequence of characters as in the character array.

double
: The resulting string contains a textual representation of the value. This may be normal decimal notation or scientific notation, depending on the size of the value. For more information, see the *Java SDK*.

float
: The resulting string contains a textual representation of the value. This may be normal decimal notation or scientific notation, depending on the size of the value. For more information, see the *Java SDK*.

int
: The resulting string contains a textual representation of the value as a decimal integer.

long
: The resulting string contains a textual representation of the value as a decimal integer.

Object
: The resulting string is `"null"` if the object is null. Otherwise, the string is the same as the result of **toString** on the object.

Here are some examples:

```
String.valueOf( 3 );         // "3"
String.valueOf( 3.14159 );   // "3.14159"
String.valueOf( 'x' );       // "x"
String.valueOf( 1 < 2 );     // "true"
```

The **valueOf** method can also take the form

```
// char charArr[ ];
String.valueOf( charArr, index, count );
```

where `index` is the index of a character in the character array and `count` is an integer. This creates a String consisting of `count` characters beginning at `charArr[index]`.

StringBuffer

The StringBuffer class (`java.lang.StringBuffer`) supports character strings whose contents can be changed. String buffers are safe for use by multiple threads, since the support library prevents two different threads from modifying the same StringBuffer simultaneously.

Note: StringBuffer objects are mutable, as opposed to String objects which are immutable.

Every string buffer has a *capacity*. This is the number of characters that the buffer can contain at its current size. If you perform an operation for which the buffer is too small, the library automatically reallocates new internal storage big enough to hold the desired amount of text. However, this reallocation process imposes a certain amount of overhead work; therefore, you may be able to improve program performance by specifying a reasonable capacity for the buffer when it is created.

Many StringBuffer methods change the contents of the buffer, then return the changed StringBuffer object as the method's result. For example, the **reverse** method reverses the order of characters in a string buffer. In the code,

```
StringBuffer buf1 = new StringBuffer( "abc" );
StringBuffer buf2 = buf1.reverse();
```

both `buf1` and `buf2` end up with the contents `"cba"`. The **reverse** method changes `buf1` in place, then returns the result.

String buffers are used as intermediate data objects in string concatenation. For example, in the expression

```
"Hello " + "world"
```

the compiler creates a temporary StringBuffer object containing `"Hello "`, appends `"world"`, then converts the result to String.

Note: If you intend to concatenate a number of character strings, it is much more efficient to work with StringBuffer objects rather than String objects.

Creating a string buffer

The simplest constructor is

```
StringBuffer buf1 = new StringBuffer();
```

This creates an empty string buffer with an initial capacity of 16 characters.

You can specify the initial capacity explicitly with

```
StringBuffer buf2 = new StringBuffer( cap );
```

where `cap` is an integer giving the initial capacity.

One way to convert a String to a StringBuffer is to use the constructor

```
// String str;
StringBuffer buf3 = new StringBuffer( str );
```

This creates a string buffer whose initial contents are equal to the characters in `str`. The initial capacity of the buffer is 16 plus the number of characters in `str`. Since string constants are String values in Java, the following code is valid:

```
StringBuffer buf4 = new StringBuffer( "abc" );
```

Converting a string buffer to a string

The **toString** method produces a String value whose contents match the string in a string buffer:

```
StringBuffer buf = new StringBuffer( "abc" );
String str = buf.toString();    // "abc"
```

Simple string buffer methods

This section lists a number of simple methods supported by StringBuffer.

The **capacity** method returns the current capacity of a string buffer:

```
int cap = buf.capacity();
```

The **ensureCapacity** method makes sure that the buffer can hold a specified number of characters. The method takes a single integer argument giving the minimum number of characters desired:

```
buf.ensureCapacity( min );
```

If the current capacity is less than `min`, the library allocates a new internal buffer for the StringBuffer object. The capacity of the new buffer is `min`, or twice the old capacity plus 2, whichever is larger. For example,

```
buf.ensureCapacity( 50 );
```

makes sure that the buffer can hold at least 50 characters. The new capacity of the buffer may be considerably larger than 50, depending on the buffer's original size.

The **length** method returns the current length of the string in the string buffer:

```
int len = buf.length();
```

Note that length will always be less than or equal to the capacity. The capacity is the maximum number of characters that the buffer can currently hold, while the length is the number of characters that the buffer actually holds.

The **setLength** method sets a new length for the buffer:

```
buf.setLength( newLength );
```

The `newLength` argument must be an integer greater than or equal to zero. If the new length is less than the current length, the string in the buffer is truncated to contain the specified number of characters. If the new length is greater than the current length, null characters (`'\u0000'`) are added to the end of the string until the string has the desired length.

The **charAt** method returns the character at the given offset position in the buffer. For example,

```
StringBuffer buf = new StringBuffer( "Cat" );
char ch = buf.charAt( 0 );
```

returns the character `'C'`. The **setCharAt** function changes a specific character:

```
buf.setCharAt( 0, 'B' );
```

changes `"Cat"` to `"Bat"`.

Appending text to a string buffer

The **append** method appends text to the end of the current contents of a string buffer. Different versions of **append** take different types of arguments; **append** converts these arguments to text before appending them to the string. Therefore, you can use successive **append** calls to build up a string, as in

```
// double X;
// int J;
StringBuffer buf = new StringBuffer();
buf.append("The value of J is ");
buf.append( J );
buf.append(", and the value of X is ");
buf.append( X );
buf.append( '.' );
```

The result of **append** is the StringBuffer object after it has been changed. Because of this, calls to **append** can be chained as in

```
buf.append("The value of J is ").append( J ).append('.');
```

The text conversions performed by **append** are similar to those performed by the **valueOf** method of String. For more information, see Converting data into strings.

Inserting text into a buffer

The **insert** method inserts text into an existing string buffer. The **insert** method is similar to **append**, but takes an argument specifying the offset position where the text is to be inserted. For example,

```
buf.insert( 0, 'X' );
```

inserts the character `'X'` at the beginning of the string buffer (offset zero). Similarly,

```
buf.insert( 1, 3.14159 );
```

inserts the text string `"3.14159"` after the first character in the string buffer.

If the given offset value is equal to the current length of the string buffer, the text is inserted after the last character of the current string.

The result of **insert** is the modified StringBuffer object. Therefore, calls to **insert** may be chained as with **append**. For more information, see Appending text to a string buffer.

Vector class

The Vector class (`java.util.Vector`) represents a one-dimensional dynamic array of objects. It provides various methods for working with such an array; for example, it has a method that searches to see if the array contains a particular element.

The *size* of a vector is the number of elements it currently contains. The *capacity* of a vector is the number of elements it can contain without having to grow. A vector's size is always less than or equal to its capacity.

The constructor

```
Vector v = new Vector( N );
```

creates a vector object with an initial capacity of `N` elements (where `N` is an integer). Vector objects automatically grow if necessary as you add more elements; however, you can avoid extra performance overhead by creating the vector big enough right from the start.

If you omit the initial capacity, as in

```
Vector v2 = new Vector();
```

the default is an initial capacity of 10 elements.

Elements in a vector are numbered with indexes, beginning at 0.

Controlling size and capacity

The **size** and **capacity** methods determine a vector's current size and capacity:

```
// Vector vec;
int s = vec.size();
int c = vec.capacity();
```

The **trimToSize** method of Vector shrinks the capacity of a vector to match its current contents:

```
vec.trimToSize();
```

The **ensureCapacity** method makes sure that a vector has sufficient capacity to hold a given number of elements:

```
vec.ensureCapacity( N );
```

The above function ensures that the vector has a capacity for at least `N` elements. If the vector has more than the given capacity, the vector does not change.

Finally, the **setSize** method sets the size of a vector to a given number of elements:

```
vec.setSize( newSize );
```

If the new size is bigger than the current size, the vector's capacity is increased, as with **ensureCapacity**. If the new size is smaller than the current size, elements are discarded from the end of the vector until the vector has the given length.

Searching vectors

The **contains** method of Vector determines if a vector contains a specified object:

```
// Object obj;
boolean b = vec.contains( obj );
```

The result is `true` if the vector contains the object and `false` otherwise.

The **indexOf** method returns the index of the first occurrence of an object in a vector.

```
int index = vec.indexOf( obj );
```

If the given object cannot be found in the vector, **indexOf** returns `-1`.

There is a second version of **indexOf** which begins the search at a specified position in the vector:

```
// int pos;
int index = vec.indexOf( obj, pos );
```

The search begins at the element with the index `pos` and continues until it finds the next element that matches `obj`. If the search reaches the end of the vector without finding a match, **indexOf** returns `-1`.

The **lastIndexOf** method is similar to **indexOf** but searches backward through the vector:

```
index = vec.lastIndexOf( obj );
index = vec.lastIndexOf( obj, pos );
```

Accessing vector elements

The **firstElement** and **lastElement** methods of Vector obtain the corresponding elements from the vector:

```
Object objFirst = vec.firstElement();
Object objLast  = vec.lastElement();
```

The **elementAt** method obtains the value of the element that has a specified index:

```
// int index;
Object obj = vec.elementAt( index );
```

The **setElementAt** method stores a value in a given element of the vector:

```
vec.setElementAt( obj, index );
```

The **removeElementAt** method deletes a given element of the vector:

```
vec.removeElementAt( index );
```

All elements with an index greater than the one given are shifted down (their index decreases by one).

The **insertElementAt** method inserts a new element at a given position:

```
vec.insertElementAt( obj, index );
```

All elements with an index greater or equal to the one given are shifted up (their index increases by one).

The **addElement** method adds a new element to the end of the vector's current contents:

```
vec.addElement( obj );
```

The **removeElement** method removes the first element that matches the value of a given object:

```
boolean success = vec.removeElement( obj );
```

The result is `true` if an element is actually removed, and `false` otherwise. If the vector contains several occurrences of the given element, only the first is removed.

The **removeAllElements** method clears out the contents of the vector so that it becomes empty:

```
vec.removeAllElements();
```

Point class

Positions on a window are represented by Point objects (`java.awt.Point`). Point values represent positions using X and Y coordinates, with (0,0) indicating the upper left corner of the window. X values increase as you move right, and Y values increase as you move down. Sizes are given in pixels.

The coordinates of a Point object are publicly accessible int values named `x` and `y`. The code below shows some common operations with Point objects:

```
Point p = new Point(200,300);  // creates an object
int Xval = p.x;                // obtains X coordinate
int Yval = p.y;                // obtains Y coordinate
p.x = 100;                     // sets X coordinate
p.y = 200;                     // sets Y coordinate
p.move(50,50);                 // moves point to (50,50)
p.translate( 50, 100 );        // adds 50 to X, 100 to Y
```

Dimension class

The Dimension class (`java.awt.Dimension`) represents rectangular dimensions: width and height. Dimension contains two public members describing the dimensions:

`int width;`
A width in pixels.

`int height;`
A height in pixels.

The most commonly used constructor is

```
Dimension dim = new Dimension( width, height );
```

which creates a Dimension object with the given width and height.

Rectangle class

The Rectangle class (`java.awt.Rectangle`) represents rectangles. For example, you could use a Rectangle object to specify a frame around a graphic or to describe the size of a rectangular region in a window. Sizes in a Rectangle object are always given in pixels.

Rectangle contains several public members describing the rectangle:

`int x;`
: The X coordinate of the upper left corner of the rectangle, relative to the window itself. The leftmost edge of the window has an X coordinate of zero and values increase going to the right across the window. The X coordinate is measured in pixels.

`int y;`
: The Y coordinate of the upper left corner of the rectangle, relative to the window itself. The upper edge of the window has a Y coordinate of zero, and values increase going down the window. The Y coordinate is measured in pixels.

`int width;`
: The width of the rectangle in pixels.

`int height;`
: The height of the rectangle in pixels.

The **equals** method determines whether two rectangles specify the same size and position:

```
boolean result = rec1.equals( rec2 );
```

The result is `true` if the two rectangles have equal `x`, `y`, `width`, and `height` values. The result is `false` otherwise.

Constructing rectangles

One common constructor for a Rectangle object is

```
Rectangle r1 = new Rectangle( x, y, width, height );
```

which defines a rectangle at the specified position with the given width and height. If you omit the position arguments, as in

```
Rectangle r2 = new Rectangle( width, height );
```

the rectangle uses the position (0,0).

There are two constructors that use Dimension arguments:

```
// Dimension dim;
// Point p;
Rectangle r3 = new Rectangle( p, dim );
```

creates a rectangle at the given point with the given dimensions, and

```
Rectangle r4 = new Rectangle( dim );
```

creates a rectangle with the given dimensions at the point (0,0).

Changing a rectangle's size and position

The Rectangle class supports a number of methods that change the rectangle's size and/or position.

The **reshape** method

```
rec.reshape( x, y, width, height );
```

changes the rectangle `rec` to have the new dimensions.

The **move** method

```
rec.move( x, y );
```

changes the position of the rectangle to the given (`x`,`y`) coordinates, without changing the rectangle's size.

The **translate** method changes the position of the rectangle by adding given increment values to the `x` and `y` coordinates:

```
rec.translate( incrX, incrY );
```

If the original rectangle has a position of (`x`,`y`), the translated rectangle has a position of (`x+incrX`, `y+incrY`). The increment values may be negative.

The **resize** method

```
rec.resize( width, height );
```

changes the size of a rectangle to the specified width and height.

The **grow** method has the format

```
rec.grow( incrHorz, incrVert );
```

This changes the size of the rectangle by pushing each side outward by `incrHorz` pixels, and pushing both the top and bottom out by `incrVert` pixels. The center of the rectangle remains in the same position. For example, suppose a rectangle is initialized with

```
Rectangle rec = new Rectangle( 150, 150, 100, 100 );
```

This is a 100x100 rectangle centered on (200,200). The borders are:

```
left:      x = 150
right:     x = 250
top:       y = 150
bottom:    y = 250
```

If you use **grow** with

```
rec.grow( 20, 50 );
```

the result is still a rectangle centered on (200,200). The borders become:

```
left:      x = 130
right:     x = 270
top:       y = 100
bottom:    y = 300
```

The width of the rectangle is now 140, and the height is now 200.

Points and rectangles

The **inside** method of Rectangle determines whether the rectangle contains a specified point:

```
boolean result = rec.inside( x, y );
```

The result is `true` if the point (`x`,`y`) is inside the rectangle and `false` otherwise. Points on the boundary of the rectangle are considered inside for the purposes of this function.

The **add** method of Rectangle changes the rectangle to the smallest rectangle that contains both the original rectangle and a given point:

```
rec.add( x, y );
```

You can also specify the point as a Point value:

```
//Point pt;
rec.add( pt );
```

Unions and intersections of rectangles

The **intersects** method determines whether two rectangles intersect:

```
boolean result = rec1.intersects( rec2 );
```

The result is `true` if they intersect and `false` otherwise. The rectangles intersect if they have any point(s) in common, even if the intersection is a single corner point.

The **intersection** method determines the rectangle that lies at the intersection of two other rectangles:

```
Rectangle interRec = rec1.intersection( rec2 );
```

The result is only valid if the rectangles actually intersect. If the rectangles do not intersect, **intersection** still returns a value but the value does not represent a valid rectangle (at least one of the dimensions will be negative).

The **union** method determines the smallest rectangle that contains both rectangles:

```
Rectangle unionRec = rec1.union( rec2 );
```

Date and time classes

The fundamental class for representing a date/time is the Date class (`java.util.Date`). This can represent a date/time to the nearest millisecond.

The JDK 1.02 implementation of Date has a number of inadequacies. The most important are:

- It is poorly suited for internationalization.
- The most commonly used constructor cannot construct date/times before January 1, 1970.

The JDK 1.1 implementation of Date was improved, but at the cost of incompatibility with JDK 1.02. For example, the JDK 1.02 version of Date offered a method for producing a formatted date/time string; the JDK 1.1 version does not. (In JDK 1.1, date formatting is performed using methods of the DateFormat class, `java.text.DateFormat`.)

Important: Because of the incompatibility in date handling between JDK 1.02 and JDK 1.1, you should take care to localize (isolate) any date manipulations a JDK 1.02 program may do. This will simplify the conversion to JDK 1.1 later on.

Dates in PowerJ

The design time facilities of PowerJ are based on JDK 1.1. This means, for example, that the database query editor analyzes date values in a database using the JDK 1.1 definition of Date, even if you intend to generate a JDK 1.02 target. As a result, the query editor can obtain pre-1970 dates from a database; however, a JDK 1.02 target program may fail to handle such dates correctly.

Constructing a date

There are two simple approaches to constructing a date object.

The first version of the Date constructor takes no arguments:

```
Date d = new Date();
```

This constructs a Date object representing the current date/time to the nearest millisecond.

The second version of the Date constructor takes a long integer argument giving a date/time as a number of milliseconds since January 1, 1970:

```
// long dateTime;
Date d = new Date( dateTime );
```

In JDK 1.02, a negative value for `dateTime` produces invalid results. In JDK 1.1, a negative value produces a date/time before January 1, 1970.

JDK 1.02 offers a number of other constructors for Date, but all those constructors are deprecated in JDK 1.1.

Comparing dates

The Date class offers a number of simple methods for comparing dates.

```
// Date d1, d2;
boolean b = d1.before( d2 );
      // true if d1 comes before d2
b = d1.after( d2 );
      // true if d1 comes after d2
b = d1.equals( d2 );
      // true if the dates are equivalent
```

Simple date formatting

JDK 1.02 and JDK 1.1 take different approaches to formatting dates into strings.

JDK 1.02: The following methods are available:

```
// Date d;
String s = d.toString();
      // format: "Sat Jul 12 14:30:00 EDT 1997"
s = d.toLocaleString();
      // intended to be a format commonly used
      //   in the user's locale
s = d.toGMTString();
      // format: "12 Jul 1997 14:30:00 GMT"
```

JDK 1.1: The **toLocaleString** and **toGMTString** methods are deprecated under JDK 1.1 and should not be used. Instead, you are advised to use methods of the DateFormat class. The simplest is the **format** method:

```
// Date d;
DateFormat formatter = DateFormat.getDateInstance();
String s = formatter.format( d );
```

The above code uses the static **getDateInstance** method of DateFormat to obtain a DateFormat object, then uses the **format** method of that object to format the given date. If you do not specify an argument for **getDateInstance**, the function returns a DateFormat object corresponding to the current locale. You can also specify an argument for **getDateInstance**, specifying a different locale as in:

```
DateFormat df = DateFormat.getDateInstance(
     DateFormat.DEFAULT, Locale.ENGLISH );
String s = df.format( d );
```

In this case, the DateFormat object will format Date values in a style suitable for the given locale.

For more information about DateFormat, see the standard JDK 1.1 documentation.

Breaking up dates into components

In many situations, you want to obtain the components of a date object. For example, you may want to obtain the month and year corresponding to a given Date object.

JDK 1.02: The following methods are defined in Date:

```
// Date dd;
int y = dd.getYear();    // year - 1900
int m = dd.getMonth();   // month: 0-11
int d = dd.getDate();    // month date: 1-31
int w = dd.getDay();     // day of week: 0-6 (Sunday=0)
int h = dd.getHours();   // hours: 0-23
int i = dd.getMinutes(); // minutes: 0-59
int s = dd.getSeconds(); // seconds: 0-60
long t = dd.getTime();   // milliseconds since
                         //    January 1, 1970
```

There are corresponding **set** methods to set a component of the date.

JDK 1.1: Under JDK 1.1, all of the above methods are deprecated and should not be used. Instead, you use the FieldPosition class (`java.text.FieldPosition`) and the **format** method of the DateFormat class.

A FieldPosition object represents a position in a date/time string. For example, if you create such an object with

```
FieldPosition f = new FieldPosition( DateFormat.YEAR_FIELD );
```

the object represents the beginning and ending positions of the year field in a date/time string. You can obtain these positions with

```
f.getBeginIndex()
f.getEndIndex()
```

Here is a code sample showing how these can be used in conjunction with the **format** method:

```
// Date dd;
// StringBuffer s;
DateFormat df = DateFormat.getInstance();
FieldPosition f = new FieldPosition( DateFormat.MINUTE_FIELD );
s = df.format( dd, s, f );
String minutes = ( s.toString() ).substring( f.getBeginIndex(),
    f.getEndIndex());
```

This uses the **substring** method of String to extract the minutes portion of the date `dd`. The result is stored in the String `minutes`. Note that the FieldPosition object is specified as the third argument of **format**.

FieldPosition codes may be locale-specific, depending on the date format; however, the following are predefined by DateFormat.

```
ERA_FIELD
YEAR_FIELD
MONTH_FIELD
DATE_FIELD
HOUR_OF_DAY1_FIELD   // 1-24 (24-hour clock)
HOUR_OF_DAY0_FIELD   // 0-23 (24-hour clock)
MINUTE_FIELD
```

```
SECOND_FIELD
MILLISECOND_FIELD
DAY_OF_WEEK_FIELD
DAY_OF_YEAR_FIELD
DAY_OF_WEEK_IN_MONTH_FIELD
WEEK_OF_YEAR_FIELD
WEEK_OF_MONTH_FIELD
AM_PM_FIELD
HOUR1_FIELD          // 1-12 (12-hour clock)
HOUR0_FIELD          // 0-11 (12-hour clock)
TIMEZONE_FIELD
```

Parsing date/times

In many situations, you want to read a date/time as a string and convert it into an equivalent Date value.

JDK 1.02: The **parse** method of Date is a static method that reads a string containing a date and returns the number of milliseconds between the given date and 00:00:00 GMT on January 1, 1970. You can use this number to create a Date object:

```
long i = Date.parse( "Sat, 12 Jul 1997, 14:30:00 GMT" );
Date d = new Date( i );
```

The **parse** method recognizes a number of date formats and time zone abbreviations.

JDK 1.1: The **parse** method of Date is deprecated and should not be used. Instead, you should use the **parse** method of DateFormat:

```
DateFormat df = DateFormat.getDateInstance();
Date d = df.parse( "Sat, 12 Jul 1997, 14:30:00 GMT" );
```

Color class

The Color class (`java.awt.Color`) represents colors. The standard way of specifying a color is the RGB format: a combination of red, green, and blue component values. Each of these component values is represented by an integer in the range from 0 through 255.

Note: This method of representing colors is called the *RGB* color model. It is the default color model for the Java component library.

An entire color can be represented as a single 32-bit integer with the form:

```
0xffRRGGBB
```

where `RR`, `GG`, and `BB` are hexadecimal values representing the red, green, and blue component values. For example, solid blue is the hexadecimal number

```
0xff0000ff
```

The **equals** method determines whether two Color values specify the same combination of red, green, and blue:

```
boolean result = color1.equals( color2 );
```

The result is `true` if the two colors have equal red, green, and blue component values; the result is `false` otherwise.

Note: The color black has a value of zero for the red, green, and blue components. The color white has a value of 255 for the red, green, and blue components.

Constructing colors

The simplest constructor for a color is

```
Color c1 = new Color( red, green, blue );
```

where `red`, `green`, or `blue` are integers in the range 0 through 255. There is also a form that uses float values:

```
// float fRed, fGreen, fBlue ;
Color c2 = new Color( fRed, fGreen, fBlue );
```

In this case, the three argument values should be float numbers in the range 0.0 through 1.0.

The final constructor is

```
Color c3 = new Color( intColor );
```

where `intColor` is an integer of the form

```
0xRRGGBB
```

where `RR`, `GG`, and `BB` are hexadecimal values representing the red, green, and blue component values. The constructor creates a Color integer that can be represented by the value

```
0xffRRGGBB
```

In other words, the specified argument `intColor` is used for the bottom 24 bits of the final Color integer.

Color components

The following methods determine the component colors in an existing Color value:

```
// Color col;
int red   = col.getRed();
int green = col.getGreen();
int blue  = col.getBlue();
```

There are no corresponding **set** methods.

Darker and brighter colors

The **darker** method returns a darker version of a given color:

```
// Color original;
Color darkCol = original.darker();
```

This method returns a new Color object with decreased values for each of the red, green, and blue components; **darker** does not change values in the `original` color object. Repeated applications of **darker** eventually approach the color black.

The **brighter** method returns a brighter version of a given color:

```
// Color original;
Color brightCol = original.brighter();
```

This method returns a new Color object with increased values for each of the red, green, and blue components; **brighter** does not change values in the `original` color object. Repeated applications of **brighter** eventually approach the color white.

The HSB color model

Although the RGB color model is the default model for the Java component library, the library also supports a second model: the *HSB color model* standing for *hue*, *saturation*, and *brightness*. If you have software that uses the HSB model, you can use the methods **HSBtoRGB** and **RGBtoHSB** to convert between one color model and the other. Both of these methods are static methods.

The usual way of converting from HSB to RGB is

```
int rgbColor = Color.HSBtoRGB( hue, saturation, brightness );
```

(Notice that `Color` is used in the call to **HSBtoRGB** instead of naming a specific Color object; this is the usual form for invoking static methods.) The `hue`, `saturation`, and `brightness` arguments are all float values in the range 0 to 1.0. The result is a color integer in the form

```
0xffRRGGBB
```

where `RR`, `GG`, and `BB` are hexadecimal values representing the red, green, and blue component values.

You can create a Color object directly from HSB values with

```
Color col = Color.getHSBColor( hue, saturation, brightness );
```

Again, `hue`, `saturation`, and `brightness` are all float values in the range 0 to 1.0.

The usual way of converting from RGB to HSB is

```
// float hsbVals[];
float hsb[] = Color.RGBtoHSB( red, green, blue, hsbVals );
```

The `red`, `green`, and `blue` arguments are integers in the range 0 through 255. The `hsbVals` argument is a float array where **RGBtoHSB** can store appropriate float values from 0.0 to 1.0, representing the hue, saturation, and brightness of the color.

Color names

The Color class defines a number of named Color values that can be used in Java code. For example,

```
Color.white
```

is a Color object that represents the color white. This is a static object within the Color class. The following list gives all the named Color values:

```
Color.white
Color.lightGray
Color.gray
Color.darkGray
Color.black
Color.red
Color.pink
Color.orange
Color.yellow
Color.green
Color.magenta
Color.cyan
Color.blue
```

Font class

The Font class (`java.awt.font`) represents a character font. Fonts are specified using the following values:

Name
: The name of a font (for example, `"Courier"`). This is expressed as a String.

Style
: Can be one of the following static constants:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
```

: These can be added together to make mixed styles (for example, `Font.BOLD+Font.ITALIC`).

Size
: The point size.

The **equals** method determines whether two Font values specify the same fonts:

```
boolean result = font1.equals( font2 );
```

The result is `true` if the two fonts have the same name, style, and size; the result is `false` otherwise.

Font name and family

The name of a font is intended to be system-independent; it is sometimes called the *logical name* of the font. There is a corresponding platform-specific name which is called the *family* of the font.

For example, if you are creating a Java application to run on another system, you would create the font by specifying the logical name. When the application actually runs, the user's system will choose an appropriate native font corresponding to the specified logical font. Your program can use

```
String family = font.getFamily();
```

to determine the name of the specific font chosen on the user's system.

Font name differences between JDK 1.02 and 1.1

Certain fonts have different names under JDK 1.02 and 1.1. These are summarized in the table below.

JDK 1.02	JDK 1.1
Helvetica	Sans Serif
TimesRoman	Serif
Courier	Monospaced

Other fonts have the same name in both versions of the JDK.

Constructing fonts

There is only one constructor for fonts:

```
Font f = new Font( name, style, size );
```

where **name** is a String, and `style` and `size` are integers. For example,

```
Font f = new Font( "Dialog", Font.BOLD+Font.ITALIC, 14 );
```

represents a 14-point bold italic font with the logical name Dialog.

Simple font methods

The following list shows a number of simple methods defined in the Font class:

```
String family  = font.getFamily();
String logical = font.getName();
int style      = font.getStyle();
      // PLAIN, BOLD, ITALIC or sum
int pointSize  = font.getSize();
boolean plain  = font.isPlain();
boolean ital   = font.isItalic();
boolean bold   = font.isBold();
```

The boolean functions return `true` if the font has the given style and `false` otherwise.

The FontMetrics class

The FontMetrics class is used to provide information about how a particular font is rendered on a particular monitor screen. For example, a FontMetrics object can tell the maximum width of any character in the font, the maximum height of any character above the baseline, the maximum descent of any character below the baseline, and so on. FontMetrics is a standard Java AWT class; for further information, see the *Java SDK*.

Toolkit class

The Toolkit class (`java.awt.Toolkit`) provides methods for a variety of purposes (for example, loading graphic images from GIF or JPEG files). Loosely speaking, Toolkit is a grab-bag of general purpose methods that have all been grouped together in a single class.

In order to execute a Toolkit method, you need a toolkit object. The standard way to create one is

```
Toolkit tk = Toolkit.getDefaultToolkit();
```

This executes a static method within the Toolkit class to obtain a "default" toolkit object. You can then use this object to execute other Toolkit methods, as in

```
Image img = tk.getImage("file.gif");
```

URL class

The URL class (`java.net.URL`) represents a Uniform Resource Locator (URL): a reference to an object available through the Internet. A simple URL has the form

```
protocol://host:port/file
```

where:

`protocol`
: Specifies a protocol for transferring information over the net. Possible protocols include `http`, `ftp`, `nntp`, and many others.

`host`
: Specifies the system to which you want to connect in order to obtain information.

`port`
: Specifies a protocol entry point on that system.

`file`
: Specifies a file on the system.

URLs may contain more information than the parts discussed above, but the ones listed are the most basic.

Note: URLs may also be written with components separated by colons, as in

```
jdbc:sybase:Tds:jdbc.sybase.com:4444
```

A URL object cannot be changed once it is created. For example, you cannot change the file part of a URL once the URL object has been constructed. Instead, you may create a new URL object with the same protocol, host, and port but a different file name.

Constructing a URL

The standard constructors for a URL object are:

```
// String protocol, host, file;
// int port;
URL u1 = new URL( protocol, host, file );
URL u2 = new URL( protocol, host, port, file );
```

You can also create a URL object from a single absolute URL string, as in

```
URL u3 = new URL( "http://www.abc.com/info/file1.html" );
```

Obtaining information from a URL

The following methods obtain information from a URL object:

```
// URL u;
String pclName  = u.getProtocol();
String hostName = u.getHost();
int portNum     = u.getPort();
String fileName = u.getFile();
```

There are no comparable **set** methods. URL objects cannot be changed once they have been constructed.

Converting a URL to a string

The **toString** method of URL returns a human-readable string representing the URL:

```
// URL u;
String urlString = u.toString();
```

Types of URLs

There are several types of URLs:

- *Code-based* URLs specify a location relative to the folder that contains the code for the current applet.
- *Document-based* URLs specify a location relative to the folder that contains the current HTML document.
- *Absolute* URLs specify a location in absolute terms.

Using an absolute URL in an applet may violate the security model for an untrusted applet. Therefore, some web browsers may refuse to open absolute URLs when running some applets.

AudioClip class

The AudioClip class of AWT (`java.applet.AudioClip`) represents audio clips that can be played on systems that have appropriate sound support.

The usual way of creating an AudioClip object is to load it using the **getAudioClip** method of Applet. For more information, see Playing audio clips.

Once you have loaded an audio clip from a file, you can play it with

```
// AudioClip clip;
clip.play();
```

This plays the clip once from the beginning.

The **loop** method plays the clip over and over again:

```
clip.loop();
```

To stop playing an audio clip, use the **stop** method:

```
clip.stop();
```

This interrupts a looping clip, or a clip that started playing with **play**.

Applet class

The Applet class of AWT (`java.applet.Applet`) provides information that pertains to the applet as a whole. For example,

```
// Applet app;
URL docURL = app.getDocumentBase();
```

returns the URL of the HTML document where the applet is embedded. Similarly,

```
URL appURL = app.getCodeBase();
```

returns the URL of the applet itself.

Obtaining an Applet object

The main form of a PowerJ applet target inherits from the Applet class. Therefore, you can execute any Applet method on the main form itself.

Other forms in the application do not inherit from Applet. If you want to use Applet methods in the code for another form, the code must reference the main form (for example, using **getParent**).

In non-applet targets (for example, Java application targets), the main form does not inherit from Applet.

Resizing an applet

The **resize** method attempts to change the current size of the applet. There are two versions:

```
// Dimension d;
app.resize( d );
app.resize( newWidth, newHeight );
```

The first expresses the desired size as a Dimension value, while the second specifies a separate width and height.

The size change request is passed on to the web browser which is displaying the applet form. The browser may reject the request if the size change is not appropriate (for example, too big for the user's monitor screen).

Applet parameters

The **Parameter** property makes it possible for the applet to refer to parameters specified when the applet was invoked:

```
// String name;
String value = app.getParameter( name );
```

The argument to **getParameter** is the name of a parameter, and the result is the value of that parameter.

Loading images

The **getImage** method of Applet loads an image from a specified URL:

```
// URL url;
Image img = app.getImage( url );
```

Typically, the URL refers to a GIF or JPEG image.

The **getImage** method returns an Image object immediately, and does not actually wait for the image to be loaded from the specified URL. The image is only loaded when it is actually used (for example, when you draw the image in a graphic context). For more information about images and graphic contexts, see The Image class.

There is a second version of **getImage** which loads an image relative to a URL:

```
// String name;
Image img2 = app.getImage( url, name );
```

For example, if `url` refers to the current document base,

```
Image img2 = app.getImage( url, "picture.gif" );
```

loads the file `picture.gif` from the same folder as the original document.

Playing audio clips

The **play** method of Applet plays an audio clip that is obtained from a specified URL:

```
// URL url;
app.play( url );
```

If the specified URL cannot be found or played, nothing happens.

There is a second version of **play** which plays an audio clip obtained relative to a URL:

```
// String clipName;
app.play( url, clipName );
```

As an alternative method, you can use the **getAudioClip** method of Applet to obtain an audio clip object:

```
AudioClip ac1 = app.getAudioClip( url );
AudioClip ac2 = app.getAudioClip( url, clipName );
```

After you have obtained an audio clip object, you can play the clip using the **play** method of AudioClip.

AppletContext interface

The methods of the AppletContext interface provide information about the program that is running an applet. The interface is implemented by a class provided by the browser (or applet viewer) that is running your applet.

To obtain an object that implements the AppletContext interface for your applet, use the **getAppletContext** method of Applet:

```
// Applet app;
AppletContext appContext = app.getAppletContext();
```

The AppletContext interface shares a number of the same methods supported by Applet. These include

```
// URL url;
AudioClip ac = appContext.getAudioClip( url );
Image img    = appContext.getImage( url );
```

For further information on these, see Applet class.

Opening a new document

The **showDocument** method of AppletContext opens a new document in the browser or applet viewer. The simplest form is

```
// URL url;
appContext.showDocument( url );
```

This opens the specified document.

A second form of **showDocument** takes an additional argument specifying where you want the document to be opened. The argument is a String, and is often a keyword:

`appContext.showDocument( url, "_self" );`
Shows the new document in the current frame (overwriting the applet).

`appContext.showDocument( url, "_parent" );`
Shows the new document in the parent of the current frame.

`appContext.showDocument( url, "_top" );`
Shows the new document in the topmost frame.

`appContext.showDocument( url, "_blank" );`
Shows the new document in a new unnamed top-level window.

If the string argument is anything other than the special strings shown above, it is taken to be a name that you want to attach to the new document window. Therefore, **showDocument** opens a new top-level window with the given name.

Note: The browser or applet viewer which is running your applet may ignore any **showDocument** request.

Range class

The Range class (`powersoft.powerj.ui.Range`) is defined by PowerJ to represent ranges of integers. A Range object has the following public members:

`int start;`
The starting value of the range.

`int end;`
The ending value of the range.

The range reaches from the `start` value to the `end` value, inclusive.

The Range class has two constructors. The simplest is

```
Range r = new Range();
```

where `start` and `end` are both set to zero. The other constructor is

```
Range r = new Range( s, e );
```

where the arguments `s` and `e` specify the `start` and `end` values for the Range object.

Exception objects

Many Java methods are designed to throw *exceptions* when an error occurs. For example, the method for opening a file throws a null pointer exception if the given file name is a null string.

Throwing an exception creates an exception object which contains information about the exception. For example, throwing a null pointer exception creates a NullPointerException object.

With some types of exceptions, the exception object contains information describing the cause of the exception. Exception objects may also have an associated string describing the error; this might be used for displaying an error message to the user. You can obtain the associated string (if there is one) using the **getMessage** method on the exception object.

The following code attempts to open a file whose name is given by the String object `fileName`. If an exception occurs, the code does something to handle the problem:

```
// String filename;
try
{
    File f = new File( filename );
}
catch ( NullPointerException e )
{
    // Send message that filename was null
}
```

Notice that the `catch` part receives a NullPointerException object if the exception occurs. The code inside the `catch` construct can use this object if appropriate to obtain information about the error.

Throwing exceptions in user code

In many implementations of Java, the process of throwing and catching an exception requires a substantial amount of overhead. For this reason, you should avoid throwing exceptions in your own code, except in situations that are truly "errors" of some sort.

As an example, suppose you are writing a function that searches for an item in a table of data.

- In some applications, it might be a significant error if the function cannot find the desired item. In that case, you would be justified in throwing an exception.
- In other applications, it may be a common occurrence for the item not to be found. In this case, the function should return a `null` value or use some other simple technique for reporting that the item wasn't found, rather than throwing an exception.

Checked vs. unchecked exceptions

Checked vs. unchecked exceptions

The exceptions thrown by methods fall into two classes:

- Exceptions declared with a `throws` clause in the method declaration. These are sometimes called *checked* exceptions. A checked exception must be either caught or passed on. When you write a method that calls a method having a checked exception declared with a `throws` clause, your method must either use a `try/catch` block, or pass the exception up the calling stack by declaring the exception in its own declaration. If you do not enclose the call in a `try/catch` block, or declare the exception, you get a compilation error.
- Exceptions that are not declared with a `throws` clause. (Such exception classes must extend either `Error` or `RuntimeException`). These are sometimes called *unchecked* exceptions. An unchecked exception may be ignored. Typically, unchecked exceptions are used for infrequently occurring errors, where it would be a nuisance to always have to catch or declare them. A method that throws an unchecked exception may also return a special error value (for example, `null`) to indicate that a problem occurred.

If you generate a method call through the PowerJ Reference Card, the Parameter Wizard automatically creates `try/catch` blocks around the call if the method throws a checked exception.

PowerJ Programmer's Guide

Part II. Programming fundamentals

Chapter 9. Using JDK and PowerJ components

This chapter discusses basic principles of programming with the standard components that appear on the component palette. In particular, it discusses properties that are common to the standard components, and the way in which events are triggered. It also offers some general guidelines for writing Java classes.

Base classes in PowerJ

The Component class (`java.awt.Component`) is the root class underlying all PowerJ components. Classes like Button, List, and so on are all derived from Component.

The Container class (`java.awt.Container`) is based on Component. Container is the root class for all objects that can contain other objects; this includes forms, frames, and so on.

The Panel class (`java.awt.Panel`) is based on Container, and is the root class for all containers that can be contained by other containers.

The Window class (`java.awt.Window`) is also based on Container. It represents a simple window with no borders and no menu bar.

The Frame class (`java.awt.Frame`) is based on Window. It represents a window with a title bar, and possibly a menu bar.

The Dialog class (`java.awt.Dialog`) is also based on Window. It represents a window that can take input from the user.

You will almost never have to create objects of these classes. However, you may see these classes used in various methods that need to refer to "lowest common denominator" classes. For example, the **getParent** method determines the object that contains a particular object; the result of **getParent** is a Container value, since Container is the lowest common denominator class for objects that contain other objects.

Object properties

The properties of an object determine its appearance and behavior. Some properties are unique to a particular type of object, while other properties are associated with many different types of objects. This section examines a few properties that apply to most objects. For information on the mechanics of setting object properties at design time, see Changing an objects properties.

Properties at run time

Objects have separate **get** and **set** methods for each property. For example, to manipulate the **Text** property, there are separate **getText** and **setText** methods. To manipulate the **Checked** property, there are separate **setChecked** and **getChecked** methods. A read-only property will only have a **get** method.

Aside: In PowerJ, **get** and **set** are written in lower case letters. Contrast this with Powersoft Power++ (the C++ version) where the corresponding functions begin with upper case letters (**Get** and **Set**).

This guide does not explain all the properties associated with objects. For full details, see the *PowerJ Component Library Reference*.

Note: The *PowerJ Component Library Reference* documents the **get** and **set** methods for a property in the entry for the property itself. For example, **getLabel** and **setLabel** are documented in the entry for the **Label** property.

Label

The **Label** property is a string of text displayed on a visual object. For example, the label of a button is the caption that is displayed on the button itself.

An object's initial **Label** can be set in the **General** page of the object's property sheet. The property sheet also lets you choose the font in which the label is displayed: click the **Browse** button to choose a font.

Changing the label at run time

During execution, the **setLabel** method changes the **Label** property of an object. For example, suppose that `cb_1` refers to a command button; then

```
cb_1.setLabel( "New label" );
```

changes the label on the command button to the given string.

Font

The **Font** property specifies the font of text displayed in an object. For example, the following code increases the point size of the font for text in text field `textf_1`:

```
Font oldF = textf_1.getFont( );
Font newF =
 new Font( oldF.getName(), oldF.getStyle(), oldF.getSize()+2 );
textf_1.setFont( newF );
```

Notice that this code creates a new Font object that is the same as the old font except that the size is 2 points bigger. For more information about fonts, see Font class.

Visible

The **Visible** property controls whether the user can actually see an object. For example, you may wish to make a set of check boxes disappear in contexts where the associated options are irrelevant.

When you place an object on a form, PowerJ makes it visible by default. During program execution, your code can use methods to make an object appear or disappear. The methods for doing this depend on which version of AWT you are using:

Java 1.02: Use the **show** method:

```
object.show( true );          // now you see it
object.show( false );         // now you don't
```

Java 1.1: Use **setVisible**:

```
object.setVisible( true );    // now you see it
object.setVisible( false );   // now you don't
```

With both versions of Java, the **isVisible** method determines whether an object is currently visible:

```
boolean vis = object.isVisible();
```

The result is `true` if the object is visible and `false` otherwise.

Deferring properties

When you turn off the **Visible** property for an object at design time, the object remains visible in the form design window. If PowerJ immediately made the object invisible, you might not know it was there. PowerJ defers making the object invisible until the program actually runs.

The same principle applies to a few other properties: PowerJ doesn't put them into effect until your program actually runs. If you set a property and click **Apply**, but don't see any change, assume that the setting has been deferred. When you run the program, you should see the setting take effect.

Enabled

Enabled controls whether the user can actually perform actions on an object. For example, if you turn off the **Enabled** property for a command button, the button still appears on the form, but nothing happens if the user clicks on it. This means that the button will not respond to being clicked, even if you have written an **Action** event handler for the button.

When an object is disabled, its appearance changes to show that it is unavailable. For example, a command button typically has its label grayed out to show that it is disabled. The actual effect depends on the behavior of the user's implementation of Java.

When you place an object on a form, PowerJ turns on **Enabled** by default. During execution, your program can call methods to enable or disable the object on the fly. The methods for doing this depend on which version of AWT you are using:

Java 1.02: Use the **enable** method, as in

```
object.enable( true );        // now it's enabled
object.enable( false );       // now it's disabled
```

Java 1.1: Use the **setEnabled** method, as in

```
object.setEnabled( true );    // now it's enabled
object.setEnabled( false );   // now it's disabled
```

With both versions, the **isEnabled** method determines whether an object is currently enabled:

```
boolean enab = object.isEnabled();
```

The result is `true` if the object is enabled and `false` otherwise.

Transient

For Java 1.1 components, the **General** page of an object's property sheet has a design-time property called **Transient**. This property controls whether or not the `transient` keyword will be added to the code that PowerJ generates for the object. You should enable this property if you are going to make the form serializable but do not want this object to be serialized.

The `transient` keyword was introduced in Java 1.1, so this property is not available for Java 1.02 components.

For more information on serialization, see Persistence and serialization.

Color

Many types of object have two associated colors: a *foreground color* and a *background color*. In a text field, for example, the field's interior is filled with the background color, and text is displayed in the foreground color.

The **Color** page of an object's property sheet contains a **Preview** box at the top, showing the current foreground color on top of the current background color.

- **To change a color:**

1. Click the arrow button of **Foreground** or **Background**. This displays a list of possible colors.
2. Click any color in this list.

The sample at the top of the **Color** page changes to show you the new color combination.

Note: In a JDK 1.1 application, the list of colors includes colors defined in the `java.awt.SystemColor` class. These colors correspond to system color settings like `activeCaptionBorder`; such settings are taken from the user's current system colors (for operating systems where system colors are defined). The system color setting `textText` does not appear in the offered list; use `controlText` instead.

You can also change colors by clicking the **Browse** button beside **Foreground** or **Background**. This displays a color selection dialog that lets you choose a color from a palette or create a custom color of your own.

At run time, you can change colors using **setForeground** and **setBackground**, as in

```
textf_1.setForeground( Color.blue );
textf_1.setBackground( Color.white );
```

Note: In Java 1.02, a color change may not be visible until the program is forced to refresh its display.

For further information about these properties, see the *PowerJ Component Library Reference*.

Parent

Objects may contain other objects. For example, a form contains all the buttons, text fields, and so on which have been placed on the form. In this situation, the form is called the *parent* of all the objects that the form contains.

The **Parent** property specifies the parent of a particular object. For example,

```
Container par = textf_1.getParent( );
```

returns the parent of `textf_1` as a Container object. Usually you would cast this to a form object, as in

```
Form1 par = (Form1) textf_1.getParent( );
```

Size and position properties

This section discusses properties related to the size and position of an object on a form.

Pixels vs. dialog units

Distances on the monitor screen can be measured in pixels or in dialog units.

- *Pixels* are dependent on size and resolution; for example, a line two pixels wide will be much thinner on a screen with high resolution than on a screen with low resolution, even if the screen has the same physical size.
- *Dialog units* are semi-independent of the size and resolution of the screen. If two screens have the same physical size, a line that is two dialog units wide will be close to the same width on both screens, even if the screens have different resolutions.

At design time, PowerJ measures all screen distances in dialog units. For example, the **Size** page of an object's property sheet shows the object's size and screen position in dialog units. Similarly, the status bar of the form design window shows the current cursor position in dialog units. This lets you design forms in a manner that is relatively independent of screen resolution on the system where the program finally runs.

At run time, however, PowerJ measures screen distances in pixels. This gives you finer control over what is actually displayed on the screen. You can, for example, change the size or position of an object by a single pixel, making it possible to exploit the precision of a high-resolution monitor, if the user has one.

Since PowerJ uses pixels to measure the screen at run time, run-time data objects like Point and Rectangle always express measurements in pixels.

When writing code, you should take into consideration that design-time units are different from run-time ones. For example, suppose that at design time you specify that a particular object is 100 units high by 100 units wide. Since this is design time, the units you use are dialog units. At run time, you *cannot* assume that the object will still measure 100 units by 100 units. Since run-time measurements are given in pixels, the run-time numbers will not be the same as the design-time numbers; in fact, the run-time numbers may be different on different systems. To determine the size of an object at run time, you must use a method like **getSize** (described later in this section). You cannot just assume that the numbers will be the ones you set at design time.

Note: PowerJ automatically converts dialog units to pixels when it runs your program. Some round-off may occur in this conversion.

Setting size and position at design time

At design time, the size and position of an object are given on the **Size** page of the object's property sheet. By adjusting these values, you can change the size and position of the object.

The entries on the **Size** page are:

Left
The X coordinate of the left side of the object relative to the form that contains the object.

Top
The Y coordinate of the top of the object relative to the form that contains the object.

Width

The width of the object.

Height
The height of the object.

Note: At design time, all of the above dimensions are specified in dialog units. If you want to see the process of converting dialog units to pixels, see the code generated when you run the program. For example, you might run the program and then look in the file `Release/Form1.java` under the target folder.

You can also change the object's size and position at design time by clicking the object, then dragging the object's sizing handles. In this case, the values on the **Size** page are automatically updated to show the object's new size and position.

Note: If you want to specify the size of an object in pixels, and are not satisfied with the way that PowerJ converts design-time dialog units to run-time pixels, you can write code in the object's **ObjectCreated** event handler to override the size and position set at design-time. The **ObjectCreated** event is triggered after the object has been initialized with size values given at design-time but before the object is actually displayed.

Enforcing the AWT minimumSize property

All components based on `java.awt.Component` have a read-only **minimumSize** property, which the component developer uses to specify a preferred minimum size for the component.

When you specify an object's position and size at design time, PowerJ will generate code to automatically set that size and position. Depending on the size you specify, PowerJ may generate code that makes a component smaller than the size returned by **getMinimumSize** on the component.

You can have PowerJ generate additional code that prevents objects from being smaller than the size specified by the **minimumSize** properties of the objects. To do this, enable the **Enforce minimum control size constraints at runtime** option on the **General** page of the container's property sheet.

Setting size and position at run time

Java 1.02 and Java 1.1 have different methods for dealing with size and position at run time. In both cases, measurements are given in pixels and positions are given relative to the object's parent (typically the form that contains the object).

Java 1.02: The following methods are available for determining the current size and/or position of an object:

```
Point p = object.location();    // current location
Dimension s = object.size();    // current size
Rectangle r = object.bounds();  // size and location
```

The location of the object is given by specifying the upper left corner.

The following methods are available for changing the current size and/or position:

```
object.move( newX, newY );
object.resize( newWidth, newHeight );
object.reshape( newX, newY, newWidth, newHeight );
```

Java 1.1: The following methods are available for determining the current size and/or position of an object:

```
Point p = object.getLocation();    // current location
Point p = object.getLocationOnScreen();
Dimension s = object.getSize();    // current size
Rectangle r = object.getBounds();  // size and location
```

The location of the object is given by specifying the upper left corner. For **getLocation**, this is the

position relative to the object's parent (typically the form that contains the object). For **getLocationOnScreen**, this is the position of the object relative to the entire screen.

The following methods are available for changing the current size and/or position:

```
// Point p;
// Dimension d;
// Rectangle r;
object.setLocation( newX, newY );
object.setLocation( p );
object.setSize( newWidth, newHeight );
object.setSize( d );
object.setBounds( newX, newY, newWidth, newHeight );
object.setBounds( r );
```

Note: If you are using one of the `java.awt` layout managers, you should not set the size or position of objects at run time—the layout manager does this for you. For more information on layout managers, see Layout managers.

FDX properties

The **FDX** properties of an object are used to exchange data between different forms. For further information, see FDX.

Database properties

The **Database** properties of an object come into play when the object is used as a *bound control* for a database. For further information, see Bound controls.

Layout manager properties

The **Layout Manager** properties of a container object determine which layout manager should be used. This property is only for container objects, such as Panel and GroupBox components. For further information, see Layout managers.

Layout constraints properties

The **Layout Constraints** properties of an object are determined by the layout manager being used on the form. For example, with the `ResizePercentLayout` manager, these properties determine how the size and position of the object change when the size of the form changes. For further information, see Resize percentages property.

Notes on writing PowerJ code

This section offers a few hints on writing code for PowerJ programs.

Object names

PowerJ automatically associates names like `cb_1` or `label_2` with the objects that you place on a form. These are the names of objects created from PowerJ components. You therefore use the `.` notation to refer to properties and methods associated with the objects, as in:

```
cb_1.setLabel( "OK" ); // change label
```

By default, the objects are `private` variables defined in the form class. This means that other forms cannot use these variables. If you want other forms to have access to these variables, you must make them `public`.

- **To make form variables public:**

1. Open the property sheet for the form that contains the objects.
2. Under **Scope of Controls** on the **General** page, click **Public**.
3. Click **OK**.

This specifies that the variables associated with the objects on the form should all be `public`.

Note: Many experts on object-oriented programming disapprove of using `public` data members within an object. These experts contend that an object's data members should only be directly accessed by the object itself; the data members should be hidden from other objects, for the sake of data abstraction.

Object initialization

Some objects require initialization while the program is executing. For example, some properties can be set at run time but not design time. Therefore, if you want to set such a property for an object, you must do so during execution.

There are two common approaches for initializing objects during execution:

1. You can initialize an object in the **objectCreated** event handler for the object. The **objectCreated** event takes place just before the object is displayed to the user. For example, the **objectCreated** event for a list takes place just before the list is displayed. Therefore, you might write an **objectCreated** event handler to initialize the list (for example, to set up the initial items displayed).
2. You can initialize an object in the **objectCreated** event handler for the form that contains the object. Again, this event takes place before the form is displayed.

You should not depend on the **objectCreated** events for separate objects taking place in any particular order. For example, you cannot be sure that the **objectCreated** handler for one object is executed before the **objectCreated** handler for another object on the same form. Therefore:

- If the initialization process for a particular object is independent of all other objects, it makes sense to put the initialization instructions in the **objectCreated** handler for the object itself.
- If it is important to initialize a set of objects in a specific order, it makes sense to put the initialization instructions for those objects into the **objectCreated** handler for the form. That way you can write the initialization instructions in the required order.

The **objectCreated** event for a form takes place after the **objectCreated** events for each object on the form. Usually, these **objectCreated** events take place before any other events for the form. This ensures that the objects on the form are properly created and initialized before other user code is executed. However, certain events related to databases may take place before **objectCreated** events; for more information, see Event timing.

Referring to the form

In the code for an event handler routine, the Java keyword `this` stands for a pointer to the form. For example,

```
if ( textf_1.getParent() == this ) ...
```

determines if the current form is the parent of `textf_1`. Similarly,

```
this.setForeground( Color.red );
```

sets the foreground color for the form. The preceding function would usually be written

```
setForeground( Color.red );
```

If a method function call isn't explicitly associated with an object, it is assumed to be applied to the current object. Since all event handlers are member functions within a form class, all methods used in event handlers are assumed to refer to the current form unless they are explicitly associated with some other object. Therefore,

```
textf_1.setBackground( Color.blue );
```

sets the background color of the given text field, but

```
setBackground( Color.blue );
```

sets the background color of the whole form.

Note: After setting the background or foreground color of the form, you must call

```
repaint();
```

before the form actually changes color.

Focus

Note: Focus is only implemented reliably in Java 1.1. Java 1.02 contains some support for focus, but focus events are not triggered reliably enough to be very useful.

Focus is the ability to receive input from the keyboard or from the mouse. Only one object can have focus at any time. For example, if a form contains three text fields, only one of them can have the focus. When the user types input from the keyboard, the resulting text appears in the text field that has the focus. In order to type into another text field, the user must move the focus from the old field to the new one.

The most common way for the user to move the focus is to click an object. This moves the focus to the clicked object. Your program can explicitly set the focus for an object using the **requestFocus** method. For example, the following code moves the focus to a text field named `textf_1`:

```
textf_1.requestFocus( );
```

Implementations of Java typically mark the object that currently has focus by making the object's outline slightly different from other objects. When the focus shifts to a different object, the special outline is moved to the new object.

Objects that may receive the focus can recognize an event named **GotFocus**. This event is triggered when the object gains the focus. Such objects also recognize an event named **LostFocus**. This event is triggered when the object loses the focus. The object that is losing the focus receives the **LostFocus** event first; then the object that is getting the focus receives the **GotFocus** event.

Note: If the user moves from your program to another application, the active form in your program receives a **LostFocus** event. Similarly, if the user moves back to your program from another application, the active window in your program receives a **GotFocus** event.

Mouse movement events

Moving the mouse across an object may trigger events. For example, the **MouseEnter** event may be triggered when the mouse pointer begins to cross an object, while the **MouseDown** event is triggered when the user clicks a mouse button.

In general, the AWT objects of JDK 1.02 do *not* trigger mouse events. However, in JDK 1.1, standard AWT objects do trigger such events.

Standard events

When you create a program using PowerJ, you write source code based on the *events* that can happen to objects.

PowerJ makes it possible to associate many different events with the same object, but in most cases, there are only a few events that you really care about. For example, consider a simple command button (also called a push button). There are actually a number of events that can be triggered on a command button, but in a normal program, the only event you care about is the user clicking the button. Therefore, your program will respond to the **Action** event and ignore everything else.

If you do not respond to a particular event on an object, the event will receive default handling from the PowerJ run-time environment. In most cases, the default handling simply cleans up after the event, making it look like the event was ignored.

Note: PowerJ uses the JavaBeans component event model, including the JDK 1.02 versions of the classes in the `powersoft.powerj` package. For more information, see Underlying mechanisms.

Event objects

Information about events is conveyed in *event objects*. An event object belongs to one of the following classes:

```
java.awt.Event
java.util.EventObject [JDK 1.1 only]
powersoft.powerj.event.EventData
```

or a class derived from one of the above classes. For example, when an **Action** event occurs on a JDK 1.02 command button, the PowerJ library creates an event object of the `java.awt.Event` class, providing information about the **Action** event.

Different types of events may have different types of information associated with them. For example, events that happen in connection with database queries have event information stored in a QueryEvent object; the QueryEvent class is derived from EventData but may contain extra information specific to query-related events.

Event objects based on java.awt.Event

The Event class (`java.awt.Event`) provides information through public data members. The most important of these data members are listed below:

`public Object target;`
: The object that received the event. For example, if an **Action** event is triggered when the user clicks command button `cb_1`, the Event object describing this event will have `target` equal to `cb_1`.

`public int id;`
: An identifier specifying what event occurred. The Event class defines a number of static integer constants which are used to specify different events. For example, in an **Action** event, `id` is set to `java.awt.Event.ACTION_EVENT`. For a complete list of possible `id` values, see the *PowerJ Component Library Reference*.

`public int x, y;`
: X and Y coordinates for the position where the event took place.

`public int key;`
: If the event occurred because the user pressed a key, `key` specifies which key the user pressed.

`public int modifiers;`
: Specifies modifiers in keyboard events. For example, if the user presses CTRL+S, `key` will have the value of the Unicode character `'S'`, and `modifiers` will be `java.awt.Event.CTRL_MASK`.

`public Object arg;`
: An arbitrary argument associated with the event. Different types of events may use this to hold different types of information.

The Event object contains additional data members that are less commonly used. For more information, see the *PowerJ Component Library Reference*.

Event objects based on EventData

This section examines features which are common to all event objects based on the EventData class.

The Source property

The **Source** property of EventData specifies the object that originally received the event. For example, suppose that

```
EventData event;
```

contains event information for the user clicking a command button. Then

```
Object source = event.getSource();
```

returns the Button object corresponding to the command button that was clicked.

The RawEvent property

Many (but not all) of the events triggered in PowerJ result from an AWT event. If a PowerJ event results from an AWT event, the **RawEvent** property of EventData obtains the event information associated with the AWT event:

```
Object awtEvent = event.getRawEvent();
```

If the PowerJ event did not result from an AWT event, **getRawEvent** returns `null`.

Using **getRawEvent** is only necessary if you are doing low level programming that interacts directly with the AWT library. Most PowerJ programmers will never have to worry about this, because PowerJ automatically handles the low level details for you.

The Handled property

The **Handled** property of EventData indicates whether an event has been completely handled by an event handler. In most cases, your code will never set **Handled** explicitly. However, if you define multiple event handlers for an event, you should use **setHandled** to specify whether an event has been handled, or **getHandled** to determine if the event has been previously handled by another event handler. For more information, see Adding a new event listener.

Event handlers

The code that responds to a particular event is called an *event handler*. Event handlers are usually methods defined in the class that represents a form.

All JDK 1.02 event handler routines have a function prototype similar to:

public boolean *object_eventName*(EventData *event*)

All JDK 1.1 event handler routines have a function prototype similar to:

public void *object_eventName*(EventData *event*)

In both cases, *object* is the name of the object and *eventName* is the name of the event that the function handles. For example, a JDK 1.02 event handler that responds to an **Action** event on a button named `cb_1` would normally have the prototype

```
public boolean cb_1_Action( java.awt.Event event )
```

The `event` argument always has a class type derived from one of the possible event object types. Different types of events use different types of `event` arguments. Many event handlers do not need any of the information stored in the `event` object. However, the information is available if necessary.

As the prototype shows, a JDK 1.02 event handler is expected to return a boolean result. A result of `false` indicates that PowerJ can proceed with normal default handling; `true` indicates that the handler completely handled the event, and no further handling is necessary.

In most cases, your event handlers should return `false`. This makes sure that default handling takes place after any specialized event handling you might want to do yourself. For example, the default handling does some simple clean-up after certain types of events; if your own event handler returns `true`, PowerJ assumes that you have already done all the clean-up necessary. Returning `false` lets you ignore underlying technical details, so that you can concentrate on matters which are directly relevant to your program.

Note: If your event handler returns `true`, the run-time environment uses **setHandled** to indicate that your event handler has handled the event. You do not have to call **setHandled** in your own code.

A JDK 1.1 event handler does not return a result.

Default handling

PowerJ and the run-time environment provide default handling for all events. In the majority of cases, the default handling just does simple clean-up; for example, the default handling tells the run-time environment that the event has been handled properly.

Certain events may have specialized default handling. For example, if the event is serious enough, the default handling may issue an error message and terminate your program. The default handling may also transfer the event from the object that originally received the event to the parent of that object.

Underlying mechanisms

The underlying mechanisms for event handling depend on the JDK version you are using.

Note: The event handling mechanisms conform with the delegation model as defined in the JavaBeans component specification. Some special handling is required for JDK 1.02 targets, but JDK 1.1 targets use the delegation model completely.

JDK 1.02 mechanisms

The JDK 1.02 event interface conforms with the old AWT model of event handling. This means that the interface uses a function named **handleEvent** to handle events. The **handleEvent** function is defined in a form class and handles all events on the form. When an event occurs, **handleEvent** is invoked; in turn, **handleEvent** calls the appropriate user-written event handler to deal with the event.

For example, suppose an **Action** event occurs because the user clicks a command button, `cb_1`. The form's **handleEvent** routine is invoked. This routine receives an Event argument providing information about the event. The **handleEvent** routine examines the contents of Event and determines that it corresponds to an **Action** event on `cb_1`; **handleEvent** therefore invokes the user-written `cb_1_Action` to handle the event.

Control eventually returns from `cb_1_Action` to PowerJ-generated code. This generated code performs default handling, if the user-written event handler returns `false`.

JDK 1.1 mechanisms

The JDK 1.1 event interface conforms with the JavaBeans component interface based on `java.util.EventListener`. This model lets you define an object as an implementation of an *event listener*. An event listener object must define a method named after the event; this method is invoked if the particular event is triggered on the event listener object.

For example, suppose that a particular object wants to receive **ActionPerformed** events. In PowerJ, the declaration of the object must state that it implements the ActionListener interface (`java.awt.event.ActionListener`). This indicates that the object intends to listen for **Action** events. The object must also include a method named **actionPerformed**. This function is invoked when an **Action** event is received by the object. The argument to the **actionPerformed** function is an ActionEvent object (`java.awt.event.ActionEvent`) providing information about the action.

Important: An event listener is an *object*, not a method. The event listener object contains a method which is invoked when the event listener is notified of that type of event. In most cases, the event listener will be the form that contains the component which receives the event.

If you define event handlers for PowerJ objects at design time, PowerJ does all this work for you. For example, suppose you define an **ActionPerformed** event handler for a command button on a form. PowerJ declares that the form implements the appropriate listener class and creates the appropriate **actionPerformed** function to be invoked when the event occurs. This **actionPerformed** function invokes the event handler that you define (`cb_1_ActionPerformed`) and also takes care of any default handling that might be appropriate.

If a form contains several objects that accept **ActionPerformed** events, the same **actionPerformed** function deals with all of the objects. When an **ActionPerformed** event takes place, the PowerJ-generated **actionPerformed** function identifies which object should receive the **ActionPerformed** event, then invokes the appropriate user-written event handler.

Important: Do not change the method name that PowerJ provides for the event handler. The name must follow the convention *object_eventName*; if you change the provided name, your code will not compile properly.

Adding a new event listener

Under JDK 1.1, you may define more than one event listener for an object. When an appropriate event is triggered on the object, all of the event listeners are notified. The JDK 1.1 standard specifically states that there is no guaranteed order for invoking the event listeners—all of the appropriate listeners will be invoked, but you cannot be sure which will be invoked first.

The process of adding a new event handler has several steps:

- Defining a new event listener class if necessary; you may also use one of the existing event listener classes.
- Adding a new event listener object to the current list of event listeners.

Tip: In order to understand this process, it may be useful to examine the Java code that PowerJ generates in order to create the first event handler for any event.

Defining a new event listener class

To add a new event handler for an object, you often need to create a new event listener for that object. The normal process of creating event handlers at design time only lets you specify a single event listener for each object. Therefore, you must define a new event listener for the object involved.

The following example shows a format for defining a new **ActionPerformed** listener class. Listeners for other events follow a similar format. For the sake of example, this **ActionPerformed** listener is intended to invoke a method named `cb_1_NewAction` defined for a command button on Form1:

```
class MyActionListener implements java.awt.event.ActionListener
{
    public MyActionListener( Form1 form )
    {
        _form = form;
    };

    public void actionPerformed(java.awt.event.ActionEvent ev)
    {
        _form.cb_1_NewAction( ev );
    };

    private Form1 _form;
};
```

Important: In order to use standard JavaBeans component tools (for example, "introspection"), the name of the class should end in the string `Listener`.

The constructor for MyActionListener receives a Form1 argument indicating the form that contains the button. The constructor saves a copy of this argument so that this MyActionListener can refer to the form later.

The **actionPerformed** method invokes `cb_1_NewAction` for the specified form. It passes the ActionEvent data as an argument to `cb_1_NewAction`.

Note: In order to use the above code, you would have to define a method named `cb_1_NewAction` inside Form1. This method would perform whatever operations you wished to take when the **ActionPerformed** event was triggered on `cb_1`.

Adding an event listener to the list

Every PowerJ component object may have a list of event listeners which listen for events on that object. For example, a button may have a list of listener objects listening for **ActionPerformed** events on that button.

When an object is established as an event source, the object has a method for adding a new event listener to the list of objects listening for that event. For example, a command button has a method named **addActionListener**.

The following example shows how to add a new **ActionPerformed** listener for a button named `cb_1`:

```
cb_1.addActionListener( new MyActionListener( this ) );
```

Notice that this function call specifies an argument of `this` when creating the new MyActionListener object. This creates an **ActionPerformed** listener which refers to the current form. Once the MyActionListener object has been added to the list of **ActionPerformed** listeners for `cb_1`, it waits for an **ActionPerformed** event to happen. If the event happens, the run-time environment invokes the **actionPerformed** method in the **ActionPerformed** listener which in turn invokes `cb_1_NewAction` for the current form. Presumably this method handles the **ActionPerformed** event in the context of the current form.

Note: Event source objects also have "remove listener" functions to remove an existing event listener. For example, command buttons have a **removeActionListener** method to remove **ActionPerformed** event listeners. For more information, see the *PowerJ Component Library Reference*.

Triggering events manually

At present, the specifications for Java and JavaBeans components do not address techniques for triggering an event manually (for example, simulating a click action on a button). Therefore, there is no standardized way for performing such an operation.

If you need to perform this kind of operation, you must make your own provisions for it. For example, suppose you want to be able to simulate pushing the button `cb_1` on Form1. To do this, you could define a public method in Form1; calling this method would invoke the event handler `cb_1_Action` within Form1 (setting up an appropriate `event` argument for the handler).

If you are working with non-PowerJ components (for example, a JavaBeans component from another vendor), there is no guarantee that the component will let you simulate an event.

General guidelines for programming in Java

The following list offers some general tips for writing Java classes:

- The Javadoc utility was created by Sun as a way of extracting documentation from Java source code. The key is to create comments in an HTML-like format, with special tags to mark important pieces of information. For example, the `@param` tag marks function arguments, `@return` marks a description of the return value, and `@exception` marks any exceptions generated. For information on Javadoc, see the references in the Bibliography.
- Think about serialization. Most PowerJ classes for JDK 1.1 are serializable, and this includes all the AWT classes. For more on serialization, see Persistence and serialization.
- Think about subclassing. When you write a class, ask yourself whether it is possible or likely that a user will want to base a new class on the class you are writing. If so, you should probably make your data members `protected` instead of `private`, unless you provide other methods of access to those members. Otherwise, users creating subclasses may run into problems that you could have avoided with a little thought. Note that there is a trade-off: declaring data members `private` allows the compiler to do better optimization, so perhaps the best approach is to use `protected` access functions with `private` data members.
- Are you going to provide users with the source code of your class? If so, you'll want to make sure the code is clean. If you're not providing source and you don't want the code to be easily reverse-engineered, you might consider including an "obfuscation" step in your release process.
- Wherever and whenever possible, write to the JavaBeans standard. This boils down to a few simple rules: have a no-argument constructor, use the **getXXXX** and **setXXXX** forms for properties (or **isXXXX** and **setXXXX** for boolean properties), and any events must follow the JDK 1.1 event model. Putting the beans into a JAR file with a manifest and with the associated BeanInfo classes is optional and really depends on how the class is to be used. You wouldn't bother doing this for a set of utility classes, for example.

  For further information on creating beans, see Creating JavaBeans Components.

Chapter 10. Programming standard objects

This chapter examines the standard components that can be placed on PowerJ forms. This includes the components that appear on the **Standard** and **Utilities** pages of the component palette, except for the grid and menu bar components (which are described in the next chapters) and data-bound versions of components (which are described in Using bound controls). The chapter assumes you have read Standard types and Using JDK and PowerJ components.

AWT components and PowerJ

Many of the components on the component palette come from the AWT package. For example, a PowerJ text field is usually just a standard AWT TextField object.

On the other hand, the properties you set at design time may change the class that PowerJ uses to represent an object. For example, if you set any of the database properties for a text field, PowerJ does *not* represent the text field with a standard AWT TextField. Instead, PowerJ creates a DBTextField object (`powersoft.powerj.ui.DBTextField`). The DBTextField class is derived from the AWT TextField, but contains additional properties and methods to support interactions between the database and the text field.

Simple labels

A simple label object is typically used to label some part of a form. For example, you might place a label immediately above a text box to describe its contents or to suggest how to use the text box.

Note: Simple labels only allow a single line of text. If you want a label to contain more than one line of text, use a multiline label. For more information, see Multiline labels.

Simple labels are represented by Label objects (`java.awt.Label`). PowerJ gives simple labels names of the form *label_N*. On the **Standard** page of the component palette, simple labels are represented by the following button:

A simple label may be a bound control for a database. For more information, see Bound controls.

Simple label properties

The following list discusses commonly used simple label properties. Each list item tells the page of the property sheet where the property appears:

Text [**General** page]
The text that is displayed in the label.

Alignment [**General** page]
Specifies how the text is aligned within the label object.

Simple labels at run time

You will almost never write an event handler routine for a label; labels are simply passive descriptions. However, there are still some operations you may want to perform on a label during program execution. For example,

```
label_1.setText( "New label" );
```

changes the label to the given string. As another example, you might use

```
label_1.show( false );        // Java 1.02
label_1.setVisible( false );  // Java 1.1
```

to make a label disappear at times when the label is unwanted.

If you want to change the alignment of a label's text, you can use one of the following:

```
label_1.setAlignment( Label.LEFT );
label_1.setAlignment( Label.RIGHT );
label_1.setAlignment( Label.CENTER );
```

Multiline labels

A multiline label is similar to a simple label, but may contain more than one line of text. (For more information on simple labels, see Simple labels.)
Multiline labels are represented by MultiLineLabel objects (`powersoft.powerj.ui.MultiLineLabel`). PowerJ gives multiline labels names of the form *mlabel_N*. On the **Standard** page of the component palette, multiline labels are represented by the following button:

A multiline label may be a bound control for a database. For more information, see Bound controls.

Multiline label properties

Multiline label properties

Multiline labels support all the properties of simple labels. In addition, they support the following properties:

WordWrap (**General** page)
: If this property is turned on, the multiline label automatically splits long lines of text into several smaller lines in order to fit inside the current size of the label object.

Insets page
: The **Insets** page lets you control the size of the borders between the edges of the multiline label and the text inside the label. For example, if you set the **Top** inset to 10, the first line of text will begin 10 pixels below the top of the label object.

Command buttons

Command buttons are also called "push buttons". When the user clicks a command button, the image on the screen changes slightly to make it look like the button has been pressed.
Command buttons are represented by Button objects (`java.awt.Button`). PowerJ gives command buttons names of the form *cb_N*. On the **Standard** page of the component palette, command buttons are represented by the following button:

Command buttons support the usual properties of components (for example, **Foreground** and **Background**). The **Label** property specifies the text that appears on the button.
When the user clicks the button, the event depends on the version of Java being used.

- **Java 1.02:** A click triggers the **Action** event (`java.awt.Event.ACTION_EVENT`).
- **Java 1.1:** A click triggers the **actionPerformed** event (`java.awt.event.ActionEvent.ACTION_PERFORMED`).

Therefore, if you are using JDK 1.02, you write an **Action** event handler to deal with the user clicking the button; if you are using JDK 1.1, you write an **actionPerformed** event handler.

A typical command button application

A typical command button application

The following example assumes that `Form1` has a button with the name `colorButton`. When the user clicks repeatedly on this button, the background color of the form cycles through a set of three colors:

```
int count = 0;
public boolean cb_1_Action(java.awt.Event event)
{
    count = (count + 1) % 3;
    switch (count)
    {
        case 1:
            setBackground( Color.red );
            break;
        case 2:
            setBackground( Color.green );
            break;
        default:
            setBackground( Color.blue );
    }
    repaint();
    return false;
}
```

Note that this routine is part of the `Form1` class. Therefore, the instruction

```
setBackground( Color.blue );
```

changes the background color of the current form.

Also notice that the `count` variable is declared outside the event handler routine (typically at the end of the definition of the form class). Therefore it retains its value from one invocation of the routine to the next. The variable is only initialized once, not every time the function is called.

Picture buttons

A picture button serves the same purpose as a command button; however, it combines both a picture and a text caption.

Picture buttons are represented by PictureButton objects (`powersoft.powerj.ui.PictureButton`). PowerJ gives picture buttons names of the form *pictbttn_N*. On the **Standard** page of the component palette, picture buttons are represented by the following button:

The **Click** event (`powersoft.powerj.event.ClickEvent`) is triggered on a picture button when the user clicks the button.

Picture button properties

Picture buttons support all the properties of command buttons. They have the following additional properties:

Label [**General** page]
The text that is used as the button's caption.

LabelPosition [**General** page]
Where the text should appear in relation to the picture on the button. Possible values are:

```
PictureButton.LABEL_TOP     // caption above picture
PictureButton.LABEL_BOTTOM  // caption below picture
```

ShowFocus [**General** page]
If this is marked, the image on the picture button will change slightly when the button has the focus; if this is not marked, the picture button will not change whether it has the focus or does not.

Insets [**Picture** page]
The amount of space between the picture and the edges of the button (expressed in pixels).

URL [**Picture** page]
A URL telling where the program should obtain the picture to place on the button. For example, this might specify a GIF or JPEG file. Use the **Type** property to specify whether it is an absolute or relative URL. Example absolute URLs are `http://myhost/logo.gif` and `file:///c:/sybase.jpg`. An example relative URL is `mypic.gif`.

Type [**Picture** page]
The type of URL. Possibilities are `Absolute`, `CodeBased` and `DocumentBased`. These have the following meanings:

`Absolute`	Based only the URL given (i.e. `http://www.sybase.com/image.jpg` or `file:///c:/pic.gif`). Untrusted applets must use the same host as the applet to avoid applet security violations.
`CodeBased`	Relative to the URL from which the class file was loaded. This only applies to applets, and images of this type will not be displayed at design time.
`DocumentBased`	Relative to the URL of the HTML file that refers to the applet. This only applies to applets, and images of this type will not be displayed at design time.

Note: If you are using a relative type, remember that PowerJ generates class files under `Debug` or `Release` subfolders, but for final deployment (using a collection target for example) you probably will not have a `Debug` or `Release` subfolder. Also the Sun and Microsoft applet viewers behave differently than browsers for relative types. Thus for `CodeBased` or `DocumentBased`, you should test using a collection target and run in a browser.

Picture buttons normally use double-buffering to reduce the amount of screen flicker. This means that drawing actions place in one buffer; when the drawing is finished, the result is copied into a second buffer which is displayed on the screen. The result looks better, but it does take more memory. If you want to avoid this double-buffering, turn off the **DoubleBuffered** property.

The Image property for picture buttons

To change the picture on a picture button at run time, use the **setImage** method. The simplest form is

```
// String filename;
pictbttn_1.setImage( filename );
```

This loads an image from the specified file. The **setImage** method does not return until the image has been loaded from the file.

Note: This form of **setImage** is not useful in applets, since applets cannot access files on the user's system.

Another form of **setImage** loads an existing Image object:

```
// Image img;
pictbttn_1.setImage( img );
```

For more information on Image objects, see The Image class.

The final form of **setImage** loads from a URL:

```
// URL url;
pictbttn_1.setImage( url );
```

Picture button states

A picture button can be in the following states:

Unarmed:
The button is raised (not pushed) and the mouse is not positioned on the button.

Armed:
The button is raised, but the mouse is pointing to the button.

Sunken:
The button is sunken into the form (pushed), but the mouse is not positioned on the button.

Sunken and armed:
The button is sunken into the form and the mouse is pointing to the button.

Disabled:
The button cannot be pressed. It is raised, but grayed-out.

The button may have a different image for each state. These are set with the following methods:

```
// Image img1, img2, img3, img4, img5;
pictbttn_1.setImage( img1 );         //unarmed
pictbttn_1.setArmedImage( img2 );
pictbttn_1.setSunkenImage( img3 );
pictbttn_1.setSunkenArmedImage( img4 );
pictbttn_1.setDisabledImage( img5 );
```

There are corresponding **get** methods to obtain the current image associated with each state.

You can set the state with:

```
pictbttn_1.setState( PictureButton.RAISED );
pictbttn_1.setState( PictureButton.SUNKEN );
```

Whether the state is armed or unarmed depends on where the user points the mouse. The **getState** method returns the current state, using the codes

```
PictureButton.RAISED
PictureButton.SUNKEN
```

The **BorderStyle** property specifies the states in which the picture button is drawn with a three-dimensional border:

`pictbttn_1.setBorderStyle(PictureButton.NO_BORDER);`
Never drawn with a border.

`pictbttn_1.setBorderStyle(PictureButton.NORMAL_BORDER);`
Always drawn with a border.

`pictbttn_1.setBorderStyle(PictureButton.ARMED_BORDER);`
Only drawn with a border when the button is armed.

The **getBorderStyle** returns the current border style.

Check boxes

Check boxes can be checked or unchecked to show whether an option is turned on or off.
Check boxes are represented by Checkbox objects (`java.awt.Checkbox`). PowerJ gives check boxes names of the form *checkbox_N*. On the **Standard** page of the component palette, check boxes are represented by the following button:

A check box may be a bound control for a database. For more information, see Bound controls.

Check box properties

The following list discusses commonly used check box properties:

Label [**General** page]
Specifies the text that appears to label the check box.

State [**General** page]
Determines whether the check box is checked or left blank. If you check **State** at design time, the check box is checked when the form is first displayed; if you leave **State** blank, the check box starts off blank.

Group [**General** page]
Specifies the check box's group, if any. For further information, see Check box groups.

At run time, you can determine whether a check box is currently checked or unchecked using **getState**. For example,

```
boolean s = checkbox_1.getState();
```

returns `true` if the check box is currently checked and `false` if the box is blank. Similarly, **setState** sets the state for the check box.

Using check boxes

When the user clicks a check box, the box automatically toggles itself: if it is turned on, it turns itself off, and vice versa. The action also triggers an event on the check box. The type of event depends on which version of Java you're using:

- **Java 1.02:** Clicking a check box triggers an **Action** event (`java.awt.Event.ACTION_EVENT`).
- **Java 1.1:** Clicking a check box triggers an **itemStateChanged** event (`java.awt.Event.Item.ITEM_STATE_CHANGED`). (In the current release of JDK 1.1, there are actually two identical **itemStateChanged** events. This may change in future updates to AWT.)

Since the box automatically toggles itself, your event handler for this event does not need to set the state of the check box. However, the event handler may need to change the state of other check boxes, especially if the check boxes represent options that are mutually exclusive with the check box that was actually clicked.

When you change the state of another check box with **setState**, it does *not* trigger an event on the check box being changed.

In the event handler for a check box, you can determine whether the box has just been turned on or off by using **getState**.

Check box groups

Check boxes are often used to represent groups of mutually exclusive options. When the user clicks one of these options, you want to mark that check box with a check and clear all the other check boxes in the group.

You can do this automatically using a check box group.

- **To create a check box group:**

1. On the **General** page of a check box's property sheet, click the **New** button (beside the **Group** box).
2. Enter a name for the group, then click **OK**.

Note: The name of a check box group must be a valid Java identifier.

Once you have created a check box group, you can specify the check boxes that should belong to this group. To do this, click the name of the group under **Group** on the **General** page of the property sheet for each check box in the group.

When a check box is part of a check box group, it is displayed as a circle rather than a square:

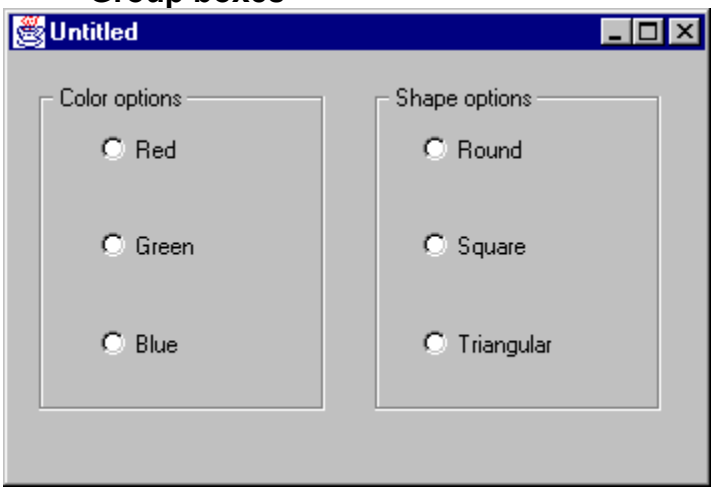
You can create any number of check box groups. When the user clicks any box in a group, your program automatically marks that check box and clears all the other check boxes in the group. The program also triggers an **Action** event (Java 1.02) or an **itemStateChanged** event (Java 1.1) on the check box that was clicked (but not on any of the other check boxes).

At run time, check box groups are controlled by the **CheckboxGroup** property of Checkbox. For example, the following code places `checkbox_2` into the same check box group as `checkbox_1`.

```
CheckboxGroup g = checkbox_1.getCheckboxGroup();
checkbox_2.setCheckboxGroup( g );
```

The **getCheckboxGroup** method returns `null` if the check box currently does not belong to a group.

Group boxes



A group box can surround a number of other objects. For example, you might put a group box around a set of option buttons to show that the buttons are logically connected.

Group boxes are represented by GroupBox objects (`powersoft.powerj.ui.GroupBox`). PowerJ gives group boxes names of the form *groupb_N*.

On the **Standard** page of the component palette, group boxes are represented by the following button:

The most commonly used property of a group box is the **Text** property. This specifies the text that is placed on the border of the group box.

Most applications do not define event handling routines for group boxes. Group boxes are only used to make the form easier to understand.

Picture boxes

A picture box displays a graphic image. Picture boxes are represented by PictureBox objects (`powersoft.powerj.ui.PictureBox`). PowerJ gives picture boxes names of the form *pictb_N*. On the **Standard** page of the component palette, picture boxes are represented by the following button:

Picture box properties

The following properties may be set in a picture box's property sheet:

AutoSize [**General** page]
If this is turned on, the size of the picture box is automatically changed to match the size of the picture.

ScaleImage [**General** page]
If this is turned on, the size of the picture is automatically scaled so that it will fit inside the picture box.

ImageCentered [**General** page]
If this is turned on, the picture is centered within the picture box. Otherwise, it is drawn in the upper left corner of the picture box. This is a design-time property; to change the position in code use the **ImagePosition** property:

```
pictb_1.setImagePosition(
    powersoft.powerj.ui.PictureBox.CENTER );
```

Insets [**Picture** page]
Specifies the distance between the area available to draw the picture and the edges of the picture box (in pixels).

URL [**Picture** page]
A URL telling where the program should obtain the picture to place on the button. For example, this might specify a GIF or JPEG file. Use the **Type** property to specify whether it is an absolute or relative URL. Example absolute URLs are `http://myhost/logo.gif` and `file:///c:/sybase.jpg`. An example relative URL is `mypic.gif`.

Type [**Picture** page]
The type of URL. Possibilities are `Absolute`, `CodeBased` and `DocumentBased`. These have the following meanings:

`Absolute`	Based only the URL given (i.e. `http://www.sybase.com/image.jpg` or `file:///c:/pic.gif`). Untrusted applets must use the same host as the applet to avoid applet security violations.
`CodeBased`	Relative to the URL from which the class file was loaded. This only applies to applets, and images of this type will not be displayed at design time.
`DocumentBased`	Relative to the URL of the HTML file that refers to the applet. This only applies to applets, and images of this type will not be displayed at design time.

Note: If you are using a relative type, remember that PowerJ generates class files under `Debug` or `Release` subfolders, but for final deployment (using a collection target for example) you probably will not have a `Debug` or `Release` subfolder. Also the Sun and Microsoft applet viewers behave differently than browsers for relative types. Thus for `CodeBased` or `DocumentBased`, you should test using a collection target and run in a browser.

The Paint event for picture boxes

The **Paint** event is triggered on a picture box whenever the picture box needs to be redrawn. For example, if some other window is moved over top of the picture box, then moved off again, the picture box must be redrawn to restore its desired appearance.

The default handling for the **Paint** event simply redraws the picture associated with the picture box (if any). However, you may wish to write your own **Paint** event handler to perform special repainting operations. As a simple application, you could use a picture box as a blank "canvas" where users can draw their own pictures. Whenever the **Paint** event occurs, your **Paint** event handler redraws whatever picture the user has previously painted.

For more about writing **Paint** event handlers for picture boxes, see Painting picture boxes and paint canvases.

Paint canvases

A *paint canvas* is similar to a picture box in that it displays a graphic image. The difference is that picture boxes are typically used to display pre-existing images (such as those loaded from GIF or JPEG files) while paint canvases are typically used to display images that the program draws during execution (for example, graphs created from data retrieved from a database).

Paint canvases are represented by PaintCanvas objects (`powersoft.powerj.ui.PaintCanvas`). PowerJ gives paint canvases names of the form *canvas_N*. On the **Standard** page of the component palette, paint canvases are represented by the following button:

The properties of a paint canvas are similar to those of a picture box, except that there is no URL property—paint canvases do not have pre-existing images associated with them.

The most important event associated with a paint canvas is the **Paint** event. This event is triggered by the system when it is necessary to redraw the picture shown on the paint canvas. The **Paint** event may also be triggered explicitly by calling the paint canvas's **paint** method:

```
// java.awt.Graphics g;
canvas_1.paint( g );
```

For more information about painting on paint canvases, see Painting picture boxes and paint canvases.

Panels

A *panel* is an object that can contain other objects. It is similar to a form, but it can be placed on a form.

Panels are represented by Panel objects (`java.awt.Panel`). PowerJ gives panels names of the form *panel_N*. On the **Standard** page of the component palette, panels are represented by the following button:

The property sheets for a panel are similar to the property sheets for a form.

Lists

A list presents the user with a list of choices. If the list has too many items to show in the area provided, scroll bars appear on the side of the list to let the user scroll through the available items. Lists are also called *list boxes*.

Each line in a list has an associated integer index. The top line has an index of 0 (zero), the next line has an index of 1, and so on.

PowerJ supports the following types of lists:

- *Single selection lists*, which only allow the user to select a single item from the list.
- *Multiple selection lists*, which let the user select more than one item (if desired).

A list box may be a bound control for a database. For more information, see Bound controls.

Lists are represented by List objects (`java.awt.List`). PowerJ gives lists names of the form *lb_N*.

On the **Standard** page of the component palette, list boxes are represented by the following button:

Note: AWT lists automatically resize themselves, depending on the data they contain. In particular, they adjust their height so that they do not cut off part of a list item at the top or the bottom of the list. This means that list boxes may change their size at run time; this cannot be controlled.

List properties

The following list properties can be set at design time:

MultipleMode [**General** page]

If this is turned on, the user may select more than one item from the list. If this is turned off, the user may only select one item at a time (selecting one item automatically unselects any item that is already selected).

Items [**Items** page]

This page lets you specify the items that initially appear in the list. Type items into the box one line at a time, pressing ENTER to start a new line. You can also click the **Save** button to save the current list of items in a file, and **Load** to load a list from a file.

Obtaining information about list items

You can obtain the text of a particular item at run time using

```
String str = lb_1.getItem( index );
```

where `index` is the index of the item. The first item in the list has an index of zero.

You can determine the number of items currently in the list by calling a method defined in the List class. The name of the method depends on which version of Java you are using.

Java 1.02: The **countItems** method returns the number of items:

```
int num = lb_1.countItems();
```

Java 1.1: The **getItemCount** method returns the number of items:

```
int num = lb_1.getItemCount();
```

Under Java 1.1, you can also obtain a complete list of all the items in the list box with

```
String arr[] = lb_1.getItems();
```

Adding items to lists

At design time, items can be specified for the list on the **Items** page of the property sheet. During execution, the **addItem** method adds a new entry to the list. For example,

```
lb_1.addItem( "New line" );
```

adds the specified string to the list. The new item is added at the end of the list.

Another form of **addItem** lets you place an item into the list at a specific location. For example,

```
lb_1.addItem( "New line", index );
```

inserts the given line into the list so that the new line has the specified index. The first item in the list has an index of zero. If you specify an `index` argument of `-1`, the item is added to the end of the list.

Changing the contents of a list

There are a variety of method functions for changing the contents of a list. The names of the methods depend on the version of Java you are using.

Java 1.02: To delete a particular item from the list, use

```
lb_1.delItem( index );
```

To delete all the items in the list box, use

```
lb_1.clear();
```

To change the text displayed for an item, use

```
// String newString;
lb_1.replaceItem( newString, index );
```

The **replaceItem** function does nothing if the given index is out of range (less than zero or greater than the index of the last item in the list box).

Java 1.1: To delete a particular item from the list, use

```
lb_1.delItem( index );
```

To delete all the items in the list box, use

```
lb_1.removeAll();
```

To change the text displayed for an item, use

```
// String newString;
lb_1.replaceItem( newString, index );
```

The **replaceItem** function does nothing if the given index is out of range (less than zero or greater than the index of the last item in the list box).

Making an item visible

A list may have too many items to show in the space provided. In this case, the user can't see the whole list at one time because some items are scrolled off the top or bottom of the list. The **makeVisible** method of List adjusts the scroll position of the list so that the user can see a particular item (so that the item appears in the portion of the list that is currently on display in the list box):

```
lb_1.makeVisible( index );
```

The `index` argument states the index of the item that you want to be visible to the user. There is no guarantee of where the item will be displayed in the list box contents—it may be at the top of the visible portion of the list, at the bottom, or anywhere in between.

The **getVisibleIndex** method returns the index of the last item that was made visible using **makeVisible**:

```
int i = lb_1.getVisibleIndex();
```

The **getRows** method returns the number of items that are currently visible in the list:

```
int numRows = lb_1.getRows();
```

Determining single selections

If a list box currently has one and only one item selected, the **getSelectedItem** function returns the text of that item, as in

```
String text = lb_1.getSelectedItem();
```

The function returns `null` if no item is currently selected, or if there have been several lines selected (in a multiple selection list).

Similarly, **getSelectedIndex** gets the numeric index of the item that is currently selected:

```
int i = lb_1.getSelectedIndex();
```

The function returns `-1` if no line has been selected or if there have been several lines selected (in a multiple selection list).

There are also functions to determine if a particular item has been selected.

Java 1.02: The **isSelected** method determines if a specified item has been selected:

```
boolean sel = lb_1.isSelected( index );
```

The result is `true` if the item given by `index` is currently selected; the result is `false` otherwise.

Java 1.1: The **isIndexSelected** method determines if a specified item has been selected:

```
boolean sel = lb_1.isIndexSelected( index );
```

The result is `true` if the item given by `index` is currently selected; the result is `false` otherwise.

Determining multiple selections

The **getSelectedItems** method returns the text strings of all items that are currently selected in the list:

```
String items[] = lb_1.getSelectedItems();
```

If no items are selected, the `items` array will contain no members.

Similarly, the **getSelectedIndexes** method returns the indexes of all items that are currently selected in the list:

```
int indexes[] = lb_1.getSelectedIndexes();
```

If no items are selected, the `indexes` array will contain no members.

You can also determine the set of selected items by calling **isSelected** (**Java 1.02**) or **isIndexSelected** (**Java 1.1**) on each item in the list.

Selecting and unselecting items

The **select** method selects a specified item in the list:

```
lb_1.select( index );
```

If this is a single selection list, selecting one item automatically unselects any other item that might be currently selected. If this is a multiple selection list, selecting an item does not unselect any other items.

The **deselect** method unselects a specified item in the list:

```
lb_1.deselect( index );
```

Selection events for lists

When the user selects or unselects a list item, it triggers an event on the list object. The nature of this event depends on the version of Java you use.

Java 1.02: Selecting an item triggers a **ListSelect** event while unselecting an item triggers a **ListDeselect** event. After one of these events, you can use the standard methods to determine which items are currently selected.

Java 1.1: Selecting or unselecting an item triggers an **ItemStateChanged** event (`java.awt.event.Item.ITEM_STATE_CHANGED`). Your event handler will have a prototype of the form

```
public boolean   lb_1_itemStateChanged(
     java.awt.event.ItemEvent event );
```

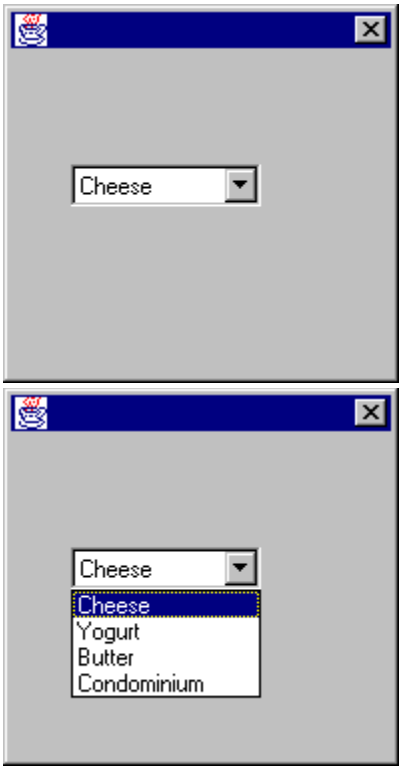
The `event` argument is an object of the ItemEvent class. This class supports the following method:

```
     int newState = event.getStateChange();
```

This returns `ItemEvent.SELECTED` if the item was selected and `ItemEvent.DESELECTED` if the item was unselected.

You can use the standard methods to determine which items are currently selected.

Choices



A choice is a single-selection list, but takes up less space on a form. In its closed state, the choice only shows one list item (the item currently selected). In its open state, the choice shows all of the list (unless the list is so long that it requires scroll bars). A choice may also be called a *combo box*.

Choices are represented by Choice objects (`java.awt.Choice`). PowerJ gives choices names of the form *choice_N*. On the **Standard** page of the component palette, choices are represented by the following button:

Choice properties and methods

Choice events

Choice properties and methods

Choices support many of the properties and methods supported by list boxes.

Java 1.02: Properties and methods supported for choices include:

```
// String itemString;
int count   = choice_1.countItems();
String str1 = choice_1.getItem( index );
String str2 = choice_1.getSelectedItem();
int i       = choice_1.getSelectedIndex();
choice_1.addItem( itemString );
choice_1.select( index );
choice_1.select( itemString );
```

Automatic Resizing: With Java 1.02, a choice list automatically resizes itself to be as wide as the longest entry on the list. For example, if you use **addItem** to add an item to an otherwise empty choice list, the choice box automatically changes its width to match the width of the item added. If you add a longer item later on, the choice box widens to match the new width. This is a built-in feature of AWT, found in Java 1.02 but not Java 1.1.

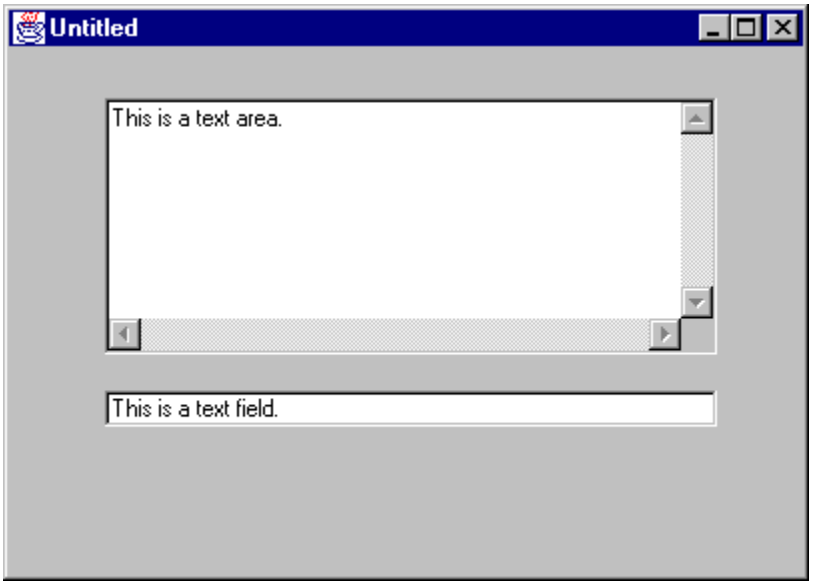
Java 1.1: Properties and methods supported for choices include:

```
// String itemString;
int count   = choice_1.getItemCount();
String str1 = choice_1.getItem( index );
String str2 = choice_1.getSelectedItem();
int i       = choice_1.getSelectedIndex();
choice_1.addItem( itemString );
choice_1.select( index );
choice_1.select( itemString );
```

Choice events

Choices receive the same events as list boxes. For more information, see Selection events for lists.

Text boxes



A text box is an area that can display text and accept text input from the user. There are two types of text boxes:

- *Text areas* are used when there is the potential for displaying many lines of text. Text areas have both horizontal and vertical scroll bars.
- *Text fields* are used for single lines of text (usually short strings conveying a single piece of information, like the user's name or password). Text fields do not have scroll bars.

Text areas and text fields have many features in common. This is why they are treated under the single heading of text boxes.

In a text box, the *caret* marks the position where text will be inserted if the user starts typing. Users may select strings of text in a text box. The methods that act on text boxes let you replace this text, delete it, or perform other simple editing operations.

Text areas are represented by TextArea objects (`java.awt.TextArea`). PowerJ gives text areas names of the form *texta_N*. On the **Standard** page of the component palette, text areas are represented by the following button:

Text fields are represented by TextField objects (`java.awt.TextField`). PowerJ gives text fields names of the form *textf_N*. On the **Standard** page of the component palette, text fields are represented by the following button:

A text box may be a bound control for a database. For more information, see Bound controls.

Text box properties and styles

The following list discusses design-time text box properties:

Text [**General page**]
The text that is originally displayed in the text box.

Editable [**General** page]
Specifies that the user cannot type or edit text directly in this box.

EchoCharacter [**General** page]
Only available for text fields, not text areas. If the user types text into this field, the text itself does not appear. Instead, the user sees a substitute character for each character typed (so that anyone looking over the user's shoulder will not be able to read what has just been typed). To specify an echo character, type the character inside single quotes; for example, `'*'` sets the echo character to an asterisk. You can also enter the decimal ASCII value of the character instead of the character itself.

If you want to specify that there is no echo character, specify an integer value of zero.

Columns [**General** page]
Specifies the number of columns in the text box.

Rows [**General** page]
Specifies the number of rows in a text area. This property is not available for text fields (since text fields only have one row).

ScrollbarVisibility [**General** page]
Only available for JDK 1.1 text areas. This specifies whether the scrollbars are visible all the time, or if they will only be displayed when they are needed. Possible values are:

```
SCROLLBARS_BOTH
SCROLLBARS_HORIZONTAL_ONLY
SCROLLBARS_VERTICAL_ONLY
SCROLLBARS_NONE
```

Determining the text in a text box

The **getText** method returns the current text contained by a text area or field, as in

```
String str = texta_1.getText();
```

Similarly, the **setText** method changes the contents of the text box, as in

```
textf_1.setText( "New text" );
```

When specifying several lines in a text area, use `\n` characters to separate lines, as in

```
texta_1.setText( "First line\nSecond line" );
```

If you want to specify the **Text** at design time, you use the same technique to fill in the **Text** entry of the text area's property sheet. For example, you might type in

```
First line\nSecond line
```

Tip: To erase the contents of a text box, use **setText** to set the contents to an empty string (`""`).

Determining the quantity of text

You can determine the amount of text in a text field using the **length** method of String:

```
String str = textf_1.getText();
int length = str.length();
```

Selecting text in a text box

The **selectAll** method selects all the text in a text box object:

```
texta_1.selectAll();
textf_1.selectAll();
```

The **select** method selects a subset of the text in a text box object:

```
// int start, end;
textf_1.select( start, end );
```

The `start` argument is an integer indicating the position of the first character to be included in the selected text. The first character in the box has an index of zero. The `end` argument is an integer indicating the position of the last character to be included in the selected text.

When the user selects text, you can obtain the selected string using

```
String str1 = texta_1.getSelectedText();
```

You can obtain the beginning and end positions in this string with

```
int begin = textf_1.getSelectionStart();
int end   = textf_1.getSelectionEnd();
```

Editing text areas

A number of methods are available for editing the contents of text areas (but not text fields):

```
// String str;
// int pos, start, end;
texta_1.insertText( str, pos );  // insert at position
texta_1.appendText( str );       // append to end
texta_1.replaceText( str, start, end );
    // replace from start to end with "str"
```

The **replaceText** method can be used to delete a section of text—just replace the unwanted section with an empty string (`""`).

Events on text boxes

Java 1.02: The most important events triggered on AWT 1.02 text boxes are:

GotFocus (`java.awt.event.GOT_FOCUS`)
The text box just received the focus (suggesting that the user is about to edit the contents of the box).

LostFocus (`java.awt.event.LOST_FOCUS`)
The text box just lost the focus. This happens when the user moves to a different object on the form, suggesting that the user has finished editing the contents of the box.

In many programs, the **LostFocus** event handler for a text box checks that the contents of the box are valid. For example, suppose the user is supposed to enter a date in a particular text field. When the text field loses focus, your **LostFocus** event handler can examine the contents of the text field to determine if the user entered a valid date. If not, the program might warn the user that the text field entry is invalid.

KeyPress, **KeyRelease** (`java.awt.event.KEY_PRESS`, `java.awt.event.KEY_RELEASE`)
The **KeyPress** event is triggered when the user presses a key while the text box has the focus. Similarly, the **KeyRelease** event is triggered when the user releases a key while the text box has the focus. You might write **KeyPress** or **KeyRelease** event handlers if you want to respond to text box input on a keystroke-by-keystroke basis.

For many text boxes, you will only write a **LostFocus** event handler. In this way, your program takes no action while users edit the contents of the text box. Your program only reacts after users move focus to some other object, indicating that they have finished working with the text box.

In many situations, you may not write any event handlers for a text box. Instead, you place a button on a form that users can click when they have finished editing the text box. For example, if you have a form with many text fields that the user should fill in, you may not define event handlers for any of the fields. Instead, you place an **OK** button on the form which the user clicks when all the text fields have been filled in. The **Action** event handler for the **OK** button then records all the information that has been entered into the text fields.

Java 1.1: The most important events triggered on AWT 1.1 text boxes are:

focusGained (`java.awt.event.FocusEvent.FOCUS_GAINED`)
The text box just received the focus (suggesting that the user is about to edit the contents of the box).

focusLost (`java.awt.event.FocusEvent.FOCUS_LOST`)
The text box just lost the focus. This happens when the user moves to a different object on the form, suggesting that the user has finished editing the contents of the box.

In many programs, the **focusLost** event handler for a text box checks that the contents of the box are valid. For example, suppose the user is supposed to enter a date in a particular text field. When the text field loses focus, your **focusLost** event handler can examine the contents of the text field to determine if the user entered a valid date. If not, the program might warn the user that the text field entry is invalid.

keyPressed, **keyReleased** (`java.awt.event.KeyEvent.KEY_PRESSED`, `java.awt.event.KeyEvent.KEY_RELEASED`)
The **keyPressed** event is triggered when the user presses a key while the text box has the focus. Similarly, the **keyReleased** event is triggered when the user releases a key while the text box has the focus. You might write **keyPressed** or **keyReleased** event handlers if you want to respond to

text box input on a keystroke-by-keystroke basis.

For many text boxes, you will only write a **focusLost** event handler. In this way, your program takes no action while users edit the contents of the text box. Your program only reacts after users move focus to some other object, indicating that they have finished working with the text box.

In many situations, you may not write any event handlers for a text box. Instead, you place a button on a form that users can click when they have finished editing the text box. For example, if you have a form with many text fields that the user should fill in, you may not define event handlers for any of the fields. Instead, you place an **OK** button on the form which the user clicks when all the text fields have been filled in. The **actionPerformed** event handler for the **OK** button then records all the information that has been entered into the text fields.

Masked text fields

A masked text field is a text box designed to show a date, time, or numeric value in a standard format. The masked text field has two important properties:

- An *input mask*, which prompts the user for input data and restricts the type of input that can be entered. For example, the input mask can specify that the user must enter numeric data in a certain position; if the user tries to enter non-numeric data, it will not be accepted.
- An *output pattern*, indicating how the masked text field displays data once it has been entered.

When the masked text field has the focus, it displays its current value using the input mask. This typically makes it easier for the user to enter new input.

When the masked text field loses the focus, it changes to display its current value using the output pattern. This typically makes it easier for the user to read the value.

Important: Since Java 1.02 does not trigger **gotFocus** and **lostFocus** events in a useful way, Java 1.02 programs have no good way of determining when to switch from input mask display to output pattern display. Therefore, output patterns are only supported for Java 1.1 applications.

For example, suppose a masked text field is intended to display a date. The input mask may make it easier for the user to type the date by specifying the month as a number or a pair of letters. However, the output pattern may make it easier for people to read the date by expanding the month to a full name like "January".

Masked text fields are represented by MaskedTextField objects (`powersoft.powerj.ui.MaskedTextField`). This class is derived from the TextField class; therefore MaskedTextField supports all the methods, properties, and events associated with normal text boxes.

PowerJ gives masked text fields names of the form *mtextf_N*. The **Text** of a masked text field is shown inside the box.

On the **Standard** page of the component palette, masked text fields are represented by the following button:

Masked text field properties and styles

Masked text fields support all the properties of normal text fields. In addition, they have the following properties:

PromptChar [**Formatting** page]
A character that is displayed in the masked text field to show the user where input is required. For example, if you set **PromptChar** to an asterisk, the user sees an asterisk in each position where a character should be entered. By default, the prompt character is an underscore.

InputMask [**Formatting** page]
A string giving the type of input that may be entered. For more information, see Input masks.

OutputPattern [**Formatting** page]
A string giving the pattern for displaying the value in the box. For more information, see Output patterns. Output patterns are only supported in JDK 1.1, not JDK 1.02.

Beep on error [**Formatting** page]
If this is `TRUE`, the computer beeps if the user types an input character that does not fit the current input mask. If this is `FALSE`, the computer does not make a sound (although the masked text field still rejects the character).

Use current date [**Formatting** page]
If you turn on this property for a masked text field that represents a date, the initial value for the field will be the current date.

Flag output error [**Formatting** page]
If you turn on this property, the masked text field will display `ERROR` when a specified value for the field cannot be displayed with the given output pattern.

Input masks

An input mask is a string of characters. Some of these characters may be *literals* which are displayed "as is" when the user enters data. Others are *placeholders* which indicate positions where the user will enter data. For example, the following is a simple input mask representing a North American phone number:

```
(###)###-####
```

The parentheses and the - character are literals; the # characters are placeholders representing decimal digits. If the prompt character is the underscore, the masked text field contains the following when it prompts for input:

```
(___)___-____
```

When the user begins entering data, the first three digits go inside the parentheses, the next three go before the - character and the last four go after the - character. The user only types digits; the parentheses and the - character make it easier for users to understand what they're typing.

The following list shows all the characters that can be used as placeholders:

`#`
: A single decimal digit; the user can only enter characters from `0` through `9`.

`?`
: A single alphanumeric character; the user can only enter characters from `0` through `9`, `a` through `z`, or `A` through `Z`.

`U`
: A single uppercase letter.

`L`
: A single lowercase letter.

`M`
: A single letter (either uppercase or lowercase).

`*`
: Any single character.

All other characters are taken as literals. Several characters have special meanings as literals:

`.` (dot)
: Replaced by the decimal separator character, as defined in the current locale (JDK 1.1 but not with JDK 1.02).

`,` (comma)
: Replaced by the thousands separator character, as defined in the current locale (JDK 1.1 but not with JDK 1.02).

`\`
: Indicates that the next character is to be taken as a literal. For example, `\U` stands for the literal character `U` rather than a placeholder. Also, `\\` stands for a literal backslash character.

Here are some typical input masks:

`###-####`
: A North American format phone number.

`##/##/####`
: A date. Notice that the same input mask may be used, whether the date is treated as `mm/dd/yyyy`

or `dd/mm/yyyy`.

`$#,###.##`
: A dollar value up to $9,999.99.

`**********`
: Any string up to ten characters in length.

Output patterns

Note: Output patterns are only supported in JDK 1.1, not for JDK 1.02.

The output pattern specifies how the masked text field's value should be displayed. The value is automatically converted to this format whenever the masked text field loses focus. Converting to this format does not change the value associated with the box; therefore, no event occurs when the box converts its value into the desired format.

An output pattern string starts with a single character indicating the type of value. Possible characters are:

`D` or `d`
: A date.

`T` or `t`
: A time.

`N` or `n`
: A number.

`S` or `s`
: A text string.

The detailed part of the format string begins with a colon following the initial character. For example, one common output pattern for a date might be specified with

```
D:mmm dd yyyy
```

There can actually be any number of additional characters between the first character and the colon. These additional characters are ignored. Therefore, the following output patterns are equivalent to the one just given:

```
Date:mmm dd yyyy
Dateformat:mmm dd yyyy
```

The extra letters are permitted to make the output pattern string easier to understand.

In the above examples, `mmm`, `dd`, and `yyyy` are formatting codes, representing month, day, and year. Different types of output patterns accept different formatting codes. Codes do not have to be separated by spaces or any other characters. The same code(s) may be used several times in a single output pattern. The case of letters is ignored in codes, so `mmm`, `MMM`, `Mmm` and so on all have the same effect.

If the output pattern does not start with one of the initial characters listed above or does not include a colon, the output pattern is taken to specify a literal text string. For example, if you forget the colon and type `D dd, yyy`, the user will see the string `D dd, yyy` appear in the box, no matter what input is entered.

Any characters that are not part of formatting codes are taken as literals. As with input masks, the `\` character may be used to mark characters that should be taken as literals rather than formatting codes.

Date format codes

Date formats are created using the following codes:

`G`
The era designator (typically `AD` or `BC`).

`YY`
The year number in two digits. The first two digits of the year specification are simply dropped.

`YYYY`
The full specification of the year.

`M`
The number of the month using one or two digits.

`MM`
The number of the month using two digits.

`MMM`
A three-character abbreviation of the English name of the month.

`MMMM`
The full English name of the month.

`d`
The day of the month using one or two digits.

`dd`
The day of the month using two digits, padding to the left with a zero if necessary.

`h`
The hour of the day using one or two digits (`1-12`).

`hh`
The hour of the day using two digits, padding to the left with a zero if necessary (`1-12`).

`H`
The hour of the day using one or two digits (`0-23`).

`HH`
The hour of the day using two digits, padding to the left with a zero if necessary (`0-23`).

`m`
The minute within the hour using one or two digits (`0-59`).

`mm`
The minute within the hour using two digits, padding to the left with a zero if necessary (`0-59`).

`s`
The second within the minute using one or two digits (`0-59`).

`ss`
The second within the minute using two digits, padding to the left with a zero if necessary (`0-59`).

`S`
The millisecond within the second using one, two, or three digits (`0-999`).

`SS`
The millisecond within the second using two or three digits, padding if necessary (`0-999`).

`SSS`
The millisecond within the second using three digits, padding if necessary (`0-999`).

`E, EE, EEE`
Short forms for days of the week (`Sun`, `Mon`, `Tue`, etc.).

`EEEE`
Full names for days of the week (`Sunday`, `Monday`, etc.).

`D, DD, DDD`
Day within the year (`1-366`).

`F`
Number of weekday within month; for example, if a date is the third Monday in the month, the value would be `3`.

`w, ww`
Number of week within the year (`1-53`).

`W`
Number of week within the month (`1-5`).

`a`
AM/PM marker (`AM`, `PM`).

`k`
Number of hour within a day (`1-24`).

`K`
Number of hour within AM or PM (`0-11`).

`z, zz, zzz`
Abbreviation for time zone (e.g. `EST`).

`zzzz`
Full name for time zone (e.g. `Eastern Standard Time`).

`'literal'`
A literal string within the pattern. For example, the output pattern

```
D:'Day='EEEE
```

might yield strings like `Day=Tuesday`.

`''`
Stands for a quote character inside a `'literal'`.

The following examples demonstrate some possible inputs and outputs for date format specifications:

Output pattern	Input	Output
`D:EEEE MMMM d yyyy`	`12/12/12`	`Thursday December 12 1912`
`D:'The month is' MMMM.`	`11`	`The month is November.`
`D:MM/dd/yyyy`	`Jan 5 45`	`01/05/1945`
`D:M/d/yy`	`Jan 5 45`	`1/5/45`

There is one useful feature of masked text fields that is worthy of emphasis. Suppose that you specify an input mask of `"**********"` allowing a long string of arbitrary input characters. In this case, the user can enter a date in any format and the masked text field attempts to convert that date

into a standard format. For example, if the output pattern is

```
D:MMMM d yyyy
```

the user could enter any of the following and the masked text field will display "`January 15, 1997`":

```
15 Jan 97
15 January 1997
1/15/97
15/1/97
1997/1/15
```

If the user enters a date that is ambiguous (as in "`2/3/97`"), the masked text field infers an interpretation for the date based on the **DateOrder** property.

Time format codes

Time formats are created using the same codes as date formats. For example, the following output patterns are equivalent

```
T:MMMM d yyyy HH:mm:ss
D:MMMM d yyyy HH:mm:ss
```

The following examples demonstrate some possible inputs and outputs for time format specifications:

Output pattern	Input	Output
`T:hh:mm`	`4:52:14`	`04:52`
`T:HH:mm`	`4:52:14pm`	`16:52`
`T:sss.SSS`	`4:52:14.6543pm`	`60734.654`

Number format codes

Number format codes produce a value with the general form

```
[prefix]integer[.fraction][suffix]
```

where each item in square brackets may be omitted. The prefix and suffix are arbitrary strings; for example, you could specify `$` as a prefix for numbers that represent a dollar amount.

Two number formats may be specified in the output pattern: one for positive values and a second for negative ones. The two are separated with a semicolon. For example, the following output pattern indicates that negative values should be enclosed in parentheses:

```
N:0.00;(0.00)
```

The following special characters may be used in number format codes:

`0`
: stands for a digit. For example, the code `0.00` asks for at least one digit before the decimal place and exactly two digits after; with this code, .9 would be written as `0.90`.

`#`
: stands for a digit; however, if the corresponding value is zero, no digit is displayed. For example, with the code `#.00`, the value .9 would be written as `.90`.

`.`
: stands for the current decimal separator.

`,`
: stands for the current grouping separator (as in `0,000,000`).

`-`
: stands for the default negative prefix.

`%`
: multiplies the value by 100 and displays it as a percentage. For example, with the output mask

`N:00%`, the value .9 would be written as `90`.

`'`
used to enclose any of the above special characters when used in a prefix or suffix. For example, if you want the value 90 to be written as `90%`, you would use the number format code

```
N:00'%'
```

The following examples demonstrate some possible inputs and outputs for numeric format specifications:

Output pattern	Input	Output
`N:$###.00; ($###.00)`	`123.45`	`$123.45`
`N:$###.00; ($###.00)`	`-123.45`	`($123.45)`

String format codes

String formats are created using the following codes:

`U#` or `u#`
Force up to `#` characters into uppercase.

`U*` or `u*`
Force the remaining characters in the input to uppercase.

`L#` or `l#`
Force up to `#` characters into lowercase.

`L*` or `l*`
Force the remaining characters in the input to lowercase.

`M#` or `m#`
Allow up to `#` characters to be of either case.

`M*` or `m*`
Allow the remaining characters in the input to be either case.

If you want to use one of the special characters above as a literal, put a backslash in front of it. For example, `\U` stands for the character `'U'` (without its special meaning of forcing input into uppercase).

The following examples demonstrate some possible inputs and outputs for string format specifications:

Output pattern	Input	Output
`He\l\lo, \Mr. U1L*`	`FrAnKeNsTeIn`	`Hello, Mr. Frankenstein`
`U*`	`abcdEFGH123`	`ABCDEFGH123`
`L*`	`abcdEFGH123`	`abcdefgh123`

Notice that backslashes were necessary in front of the `l` and `M` characters of the first example so that they were interpreted as literals instead of special format codes.

Text properties

The **Text** property for a masked text field is the raw text that the user typed into the box. For example, suppose the input mask is `(###)###-####`, representing a phone number. The **Text** property is just a string containing the ten digits that the user typed. For example, if the user sees

```
(519)555-1212
```

the **Text** value associated with the masked text field is the String value

```
"5195551212"
```

In other words, the **Text** omits the literal characters that were seen in the input mask.

To get the text including characters from the input mask, use

```
String str = mtextf_1.getMaskedText( );
```

The format method

The **format** method of the Java 1.1 MaskedTextField is a static function that lets you perform conversions in a similar way to a masked text field. Here's a simple use of the function:

```
String str;
str = MaskedTextField.format("12345.678","N:##,###.00"
    true, true);
     // str contains "12,345.67"
```

The first argument is an input value and the second is an output pattern string. The result of **format** is a String value containing the formatted output value. In other words, **format** converts the input into the desired output pattern and returns the result as a String.

The **format** method may take a number of additional arguments. For more information, see the *PowerJ Component Library Reference*.

Note: Since **format** is a static method, you do not have to place a masked text field on the current form in order to use **format**. However, if the form doesn't contain a masked text field, it must import `powersoft.powerj. ui.MaskedTextField`. The **format** method is not available in Java 1.02.

Scroll bars

Scroll bars are used for scrolling through information. For example, you might create a scroll bar whose actions affect a picture box on the form. When the user moves the scroll bar, the picture displayed in the picture box changes to show a different angle or a different portion of the picture. Note that you would have to write your own set of **Scroll** event handlers for the scroll bar—event handlers to change the picture as the scroll position changes.

Note: Many objects automatically create their own scroll bars as needed. For example, a list automatically creates its own vertical scroll bar if there are too many items to show in the space available. You only need to create your own scroll bars if you want to offer scrolling capabilities that aren't offered automatically.

Scroll bars can be used in several ways:

- The user can drag the scroll indicator along the bar to a new position.
- The user can click in the part of the bar on either side of the scroll indicator.
- The user can click the arrows at either end of the scroll bar.

Each of these actions has a different effect, as described in Scroll events.

Scroll bars are represented by Scrollbar objects (`java.awt.Scrollbar`). PowerJ gives scroll bars names of the form *scroll_N*. On the **Standard** page of the component palette, scroll bars are represented by the following button:

Note: By default, scroll bars run vertically. However, if you turn off the **Vertical** property on the **General** page of a scroll bar's property sheet, the scroll bar will run horizontally.

Scroll ranges

A scroll bar represents a range of integer values, from a minimum value to a maximum one. For example, suppose that the scroll bar will be used to scroll through a document. Here are some possible approaches to specifying the range for the scroll bar:

- You could let the range correspond to the number of lines in the document. For example, if the document contains 1000 lines, you could set up the scroll bar to have a range of 1 to 1000.
- If the document is divided into pages, you could let the range correspond to the pages in the document. For example, if the document contains 20 pages, you could set up the scroll bar to have a range from 1 to 20.
- The scroll bar could represent percentages. This would mean that the range of the scroll bar goes from 0 to 100. A value of 50 would correspond to the point 50% of the way through the document.

Other approaches might be reasonable depending on the nature of the document. For example, the range might be taken from the number of sections in the document or some other useful measure.

The position of the scroll bar's slider is represented as a value in the scroll bar's range. For example, if the range runs from 0 to 100, a value of 50 puts the slider in the middle position of the scroll bar.

The following properties are associated with scroll bar position and range:

Minimum
The minimum value represented by the scroll bar.

Maximum
The maximum value represented by the scroll bar.

Value
The current value of the scroll bar.

VisibleAmount
A number indicating the range of values that is currently visible. For example, suppose **Minimum** is 1, **Maximum** is 100, and the scroll bar is associated with an object that can show 20 items at a time. Then **Visible** would have a value of 20.

The size of **VisibleAmount** dictates the size of the scroll indicator in the scroll bar. In the above example, the **VisibleAmount** area indicates one fifth of the total range, so the scroll indicator would be one fifth the size of the total scroll bar.

Note: With JDK 1.02, the **VisibleAmount** property cannot be set at run time. You can determine its value with the **getVisible** method.

With JDK 1.1, there are appropriate **setVisibleAmount** and **getVisibleAmount** methods.

These properties have the usual **get** methods, as in:

```
int max = scroll_1.getMaximum();
int min = scroll_1.getMinimum();
int val = scroll_1.getValue();
```

You can set the current scroll position using

```
// int value;
scroll_1.setValue( value );
```

You can set all the property values discussed in this section using the **setValues** method:

```
scroll_1.setValues( value, visible, minimum, maximum );
```

BlockIncrement and UnitIncrement

Scroll bars may have associated **BlockIncrement** and **UnitIncrement** values:

- The **UnitIncrement** is the amount that the **Value** should change if the user clicks one of the end arrows of the scroll bar.
- The **BlockIncrement** is the amount that the **Value** should change if the user clicks in the blank area above or below the scroll indicator.

Note: UnitIncrement and **BlockIncrement** are JDK 1.1 names. With JDK 1.02, the **UnitIncrement** property is called **LineIncrement**, and the **BlockIncrement** property is called **PageIncrement**. For example, you would use **setLineIncrement** to change the property at run time.

At design time, PowerJ uses the names **UnitIncrement** and **BlockIncrement** for both JDK 1.02 and JDK 1.1.

You can set these values using appropriate methods, as in:

```
scroll_1.setBlockIncrement( 10 );     // JDK 1.1
scroll_1.setUnitIncrement( 1 );       // JDK 1.1
```

If you set the increments for a scroll bar, the default event handlers for the scroll bar automatically change the **Value** and move the slider the specified amount in response to user actions. This is much easier than writing your own event handler to deal with scroll actions.

Scroll events

When the user clicks on a scroll bar, it generates one of the following events:

```
ScrollAbsolute     (java.awt.Event.SCROLL_ABSOLUTE)
ScrollLineUp       (java.awt.Event.SCROLL_LINE_UP)
ScrollLineDown     (java.awt.Event.SCROLL_LINE_DOWN)
ScrollPageUp       (java.awt.Event.SCROLL_PAGE_UP)
ScrollPageDown     (java.awt.Event.SCROLL_PAGE_DOWN)
```

ScrollAbsolute (`java.awt.Event.SCROLL_ABSOLUTE`)
The user dragged the scroll indicator to a new position. There may be multiple **ScrollAbsolute** events for a single drag action.

ScrollLineUp (`java.awt.Event.SCROLL_LINE_UP`)
The user clicked the arrow at the top or left of the scroll bar.

ScrollLineDown (`java.awt.Event.SCROLL_LINE_DOWN`)
The user clicked the arrow at the bottom or right of the scroll bar.

ScrollPageUp (`java.awt.Event.SCROLL_PAGE_UP`)
The user clicked in the blank area above or to the left of the scroll indicator.

ScrollPageDown (`java.awt.Event.SCROLL_PAGE_DOWN`)
The user clicked in the blank area below or to the right of the scroll indicator.

Default Scroll handling

PowerJ supplies default handling for the scroll events. The default handling responds to messages in the following ways:

ScrollAbsolute
Change the **Value** to reflect the new position of the scroll indicator.

ScrollLineUp
Subtract **UnitIncrement** from the current **Value**.

ScrollLineDown
Add **UnitIncrement** to the current **Value**.

ScrollPageUp
Subtract **BlockIncrement** from the current **Value**.

ScrollPageDown
Add **BlockIncrement** to the current **Value**.

The default handling performs appropriate range-checking. For example, if subtracting **BlockIncrement** from **Value** would move the position outside the scroll range, **Value** is set to the minimum value in the valid range.

Setting the ScrollPosition

If you write your own event handler for a scroll event, it can use **getValue** to determine the current scroll position and **setValue** to set a different one. If you do this, the scroll indicator is placed at your specified scroll position rather than moving to the one usually chosen by the default handling.

You might choose to change the user's chosen **Value** if you want to prevent the user from scrolling to a particular region or a particular position value.

Timers

A *timer* lets you set up events based on elapsed time. For example, when your program starts, you may want it to display a window of copyright information for five seconds before calling up a window that lets the user begin working. In this case, you could put a timer on the copyright window, set for five seconds. The timer starts running when the copyright window is created; when the timer runs out, it can trigger an event to get rid of the copyright window and open a new window to begin a work session.

Timers can be set up to go off at repeated intervals. For example, you can create a timer that goes off every ten seconds. You specify timer intervals in thousandths of a second, but timers do not really supply this degree of precision; the precision of any interval is never better than 1/18th of a second. Furthermore, if the system is busy doing other work, the timer may go off considerably later than the exact time specified.

Timers are represented by Timer objects (`powersoft.powerj.util.Timer`).

- **To create a timer at design time:**

1. Click the **Utilities** tab of the component palette.
2. Click the **Timer** button:

3. Click any empty space of the current form.

You will see a Timer icon appear on the form. At run time, this icon is not visible to the user; it is simply a design-time indication that the form has an associated timer. PowerJ gives timers names of the form *timer_N*.

Timer settings

A timer has two properties that control how the timer runs:

Interval

Specifies a length of time in thousandths of a second. The timer will go off after running this length of time.

TickCount

The maximum number of times the timer should go off. For example, if you only want the timer to go off once, set **TickCount** to 1. If you set **TickCount** to zero, the timer will keep going off at the set **Interval** until the timer is explicitly stopped.

The HighPriority property

If the **HighPriority** property is turned on, the execution thread containing the timer is given the maximum allowable priority, in an attempt to ensure that the timer goes off at the time specified. If **HighPriority** is not turned on, the thread containing the timer may be shut out by other (higher priority) threads, with the result that the timer does not go off when desired. Depending on the behavior of the other threads, the timer may go off much later than specified.

Turn on **HighPriority** if it is crucial for the timer to go off at the given moment. Leave **HighPriority** off if the timing is not critical.

For more information about threads, see Using threads.

Starting the timer

If the **Running** property is turned on at design-time, the timer will start running as soon as it is created.

Otherwise, you must explicitly start the timer running with the **start** method. The **start** method can be used with calls of the form:

```
// long interval;
// int tickCount;
timer_1.start( interval );
timer_1.start( interval, tickCount );
```

As an example,

```
boolean success = timer_1.start( 10000, 3 );
```

sets a timer that goes off every ten seconds. After the timer goes off the third time, it will stop. However, you can start the timer again with another **start** call.

Stopping a timer

The **stop** method stops a timer that is currently running. For example, suppose that a timer is set to go off 300 times, every ten seconds. The function call

```
boolean success = timer_1.stop();
```

stops the timer, even if it hasn't gone off 300 times. The timer will not run again unless you explicitly start it with another **start** call.

The Timer event

The **Timer** event (`powersoft.powerj.event.TimerEvent`) is triggered when the timer goes off. The **Timer** event receives a TimerEvent object as its argument. This is based on the usual EventData class, but contains the following additional method:

```
// TimerEvent event;
long t = event.getTime( );
```

This returns the current system time in milliseconds (at the time the timer went off). This makes it easy to determine whether the timer went off later than originally specified.

How Timer events are triggered

A running timer uses a thread to keep track of when the **Timer** event should be triggered. This thread starts running when you **start** the timer. Each time a **Timer** event is triggered, the timer thread makes note of the time and invokes the **Timer** event handler.

When the **Timer** event handler returns, the timer thread calculates how much time is remaining before the next **Timer** event should be triggered. For example, suppose that **Timer** events are supposed to go off every three seconds. If the timer thread invokes a **Timer** event handler which takes one second before returning, the timer thread waits another two seconds, then invokes the **Timer** event handler again so that the **Timer** events are the required three seconds apart. If your **Timer** event handler takes four seconds before returning, the timer thread will immediately trigger another **Timer** event because the event is overdue.

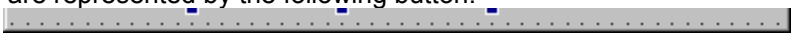
Note that if your **Timer** event handler does not return, the timer thread never regains control. Therefore, there will be no further **Timer** events triggered.

Tab controls

A *tab control* lets the user choose between several pages of data. Clicking a tab control typically displays the page of information associated with that tab. For example, the tab controls in the PowerJ property sheet correspond to different kinds of properties; when you click a tab, you see the page of properties associated with that tab.

Each tab on the tab control is referenced by a numeric *index*. The index of the first tab is zero.

Tab controls are represented by TabControl objects (`powersoft.powerj.ui.TabControl`). PowerJ gives tab controls names of the form *tabctrl_N*. On the **Standard** page of the component palette, tab controls are represented by the following button:



When you place a tab control on a form, set the size of the control so that it is big enough to include both the tabs at the top of the page and the page below the tabs.

Designing tab controls

When you place a tab control on a form, PowerJ automatically creates a new form for each tab on the tab control. These new forms are called *tab forms*. Tab forms are given names of the form *TabFormN*, where *N* is an integer. The area allocated for a tab control contains the tab forms as well as the tabs that are used to switch between forms.

As shown above, a tab control that has just been placed on a form has a single tab, labeled `Default`. This title is just a placeholder. To set up the real tabs for the tab control, use the right mouse button to click the `Default` tab and select **Properties**.

Note: When you want to access the properties or event routines for a tab control, make sure you click directly in the tab area, not on the dotted area under the tab. If you click on the dotted area, you end up on the property sheet for the tab form, not for the tab control itself.

- **To define the tabs of the tab control:**

1. On the **Pages** page of the tab control's property sheet, click on **Default** in the **Tab Text** column.
2. Click **Edit Page** to open the Edit Page dialog box.
3. Under **Tab text**, type the name that you want to appear on the first tab.
4. Click **OK** when you have typed the name. You will see the name of the tab appear in the **Tab Text** column.
5. To add another page, click **Add**. This adds a new row to the **Pages** list, and selects the **Tab Text** field for editing. You can either type the name in directly, or click **Edit Page** to use the Edit Page dialog box.
6. Click **OK**, when you have finished defining the tabs.

When you return to the form design window, the tab control will have the tabs you have defined.

You can change the position of a tab using the tab control's property sheet. On the **Pages** page, click the tab name, then use the arrow buttons below the tab list to move the tab up or down in the list.

You can delete an existing tab page using the **Pages** page of the tab control's property sheet. Click the name of the page you want to delete, then click **Delete**.

Designing the tab forms

Placing objects on a tab form is no different from placing them on any other form. You click the desired type of object on the component palette, then drag across the tab form to place the object. You can also set properties and define event routines for tab form objects in the usual way.

Once you have designed the first tab form, double-click any of the other tabs in the tab control. This displays the tab form associated with that tab. You can then proceed to design the next tab form.

Each tab form is a separate form, with its own properties and methods. For example, you could give each "page" of the tab control a different background color by giving each tab form a different background color.

Tab control properties and styles

The following list discusses commonly used tab control properties:

TabWidth [**General** page]
Is a width for each of the tabs. If you specify a width (in dialog units), each tab has that width. If you specify a width of `-1`, each tab will be as wide as necessary to show the tab's label.

TabHeight [**General** page]
Is a height for each of the tabs. If you specify a height (in dialog units), each tab has that height. If you specify a height of `-1`, each tab will be as high as necessary to show the tab's label.

Note: TabWidth and **TabHeight** are both deferred properties. If you set them at design times, the form design window will *not* show the new values for the properties. However, when you run the application, the tabs in the tab control will have the desired width and height.

The UserData property

Each tab in a tab control may have an associated data object, called its **UserData**. The **UserData** object for a tab has the Object type, allowing any kind of Java object.

The **UserData** object for a tab is often used to store a data structure which summarizes the data of the associated tab form. For example, suppose that a particular tab form has three text fields; then you might define the **UserData** object for that tab to be a structure containing three String objects, each corresponding to one of the text fields.

The following methods manipulate the **UserData** for a tab:

```
// Object userData;
// int index;
tabctrl_1.setUserData( index, userData );
Object ud = tabctrl_1.getUserData( index );
```

The `index` argument is the index of the tab whose user data you want to access.

Referring to the parent form

Each tab form defines a public data member named `__parentForm` (with two underscore characters at the beginning). When the tab form is created, this data member is initialized to refer to the parent form (the form that contains the tab control).

For example, suppose that the Form1 class represents a form that contains a tab control. Then each tab form of the tab control has a data member declared with the following:

```
public Form1 __parentForm;
```

When the tab form is created, `__parentForm` is initialized to refer to the Form1 object that contains the tab control. This makes it possible for tab forms to refer to the main form; for example, they can execute methods on that form and access public data members.

As another, suppose that page one of the tab control is a tab form of the type Form1_Page1 and suppose that this page contains a tab control of its own. Then each tab form in the new tab control has a data member declared with

```
public Form1_Page1 __parentForm;
```

referring to the page form that contains the new tab control. The new tab control can refer to the original form with the expression

```
__parentForm.__parentForm
```

The Select event for tab controls

The **Select** event (`powersoft.powerj.event.SelectEvent`) is generated when the user clicks on one of the tabs in the tab control. The argument for a **Select** event handler is a SelectEvent object. The SelectEvent class is derived from the EventData class. The most important method in SelectEvent is

```
// SelectEvent event;
int i = event.getIndex();
```

The **Index** value is the index of the tab that the user just clicked. For example, if the user clicks the leftmost tab in the tab control, **getIndex** returns 0.

The **Select** event is triggered before the run-time environment displays the tab form associated with the selected tab. If your **Select** routine returns `true`, the run-time environment assumes that the selection has been handled completely; therefore, it does *not* switch to display the correct tab form. In most cases, therefore, your **Select** routine should return `false` so that the run-time environment proceeds to display the desired tab form.

Adding new tabs at run time

Whenever possible, you should set up all the tabs of your tab control at design time; that way, PowerJ automatically does all the work of creating the tab forms, setting up event handlers, and so on. If necessary, however, you can add tabs at run time too.

- **To add a new tab to a tab control at run time:**

1. Use the **add** method of the tab control to specify the text and user data of the new tab.
2. Create a tab form by constructing a form object.
3. Use the **setPanel** method of the tab control to associate the tab form with the new tab.
4. Use the **create** method for the tab form to create and display the tab form.

The following sections examine each of these steps in order.

The add method

The **add** method of the tab control adds a tab at run time:

```
// int index;
// String text;
// Object userData;
tabctrl_1.add( index, text, userData );
```

This adds a new tab with the given `userData`. The `text` is the caption of the tab and the `index` specifies the zero-based position of the tab.

Constructing a new tab form

The general method of creating a new tab form is to construct a form object. For example, suppose the new tab form will be identical to `TabForm2`. Then you could create the tab form with:

```
TabForm2 f2 = new TabForm2();
```

The setPanel method

The **setPanel** method associates an existing form with a tab on a tab control. The simplest form of this method is

```
tabctrl_1.setPanel( index, form );
```

where `index` is the index of the tab and `form` is the new form object.

Creating the new form

The final step is to initialize and display the new tab form using the **create** method of the tab form:

```
f2.create();
```

For example, the following code fully adds a second tab called `NewTab` (assuming you have defined the `TabForm2` class and the following code is in a `try/catch` block):

```
tabctrl_1.add( 1, "NewTab", null );
TabForm2 f2 = new TabForm2();
tabctrl_1.setPanel( 1, f2 );
```

```
f2.create();
```

Other tab control methods

This section examines a number of other methods defined for tab controls. The function

```
int i = tabctrl_1.getCount();
```

returns the number of tabs in the tab control. The function

```
int i = tabctrl_1.getSelected();
```

returns the index of the tab that is currently selected. The function

```
// int index;
tabctrl_1.setSelected( index );
```

selects the specified tab. The function

```
tabctrl_1.delete( index );
```

deletes a tab from the tab control. The function

```
tabctrl_1.deleteAll();
```

deletes all the tabs. The function

```
tabctrl_1.nextPage();
```

selects the next page in the tab control and

```
tabctrl_1.previousPage();
```

selects the previous page. Both **nextPage** and **previousPage** wrap around when they reach the ends of the tab control.

Socket components on the Standard page

The **Standard** page of the component palette offers two types of components related to the Internet: sockets and server sockets. These components come from the `java.net` package.

The **Internet** page of the component palette also offers sockets and server sockets. These are native PowerJ components. They serve similar purposes to the `java.net` versions, but offer enhanced functionality.

For information about all these components, see Sockets.

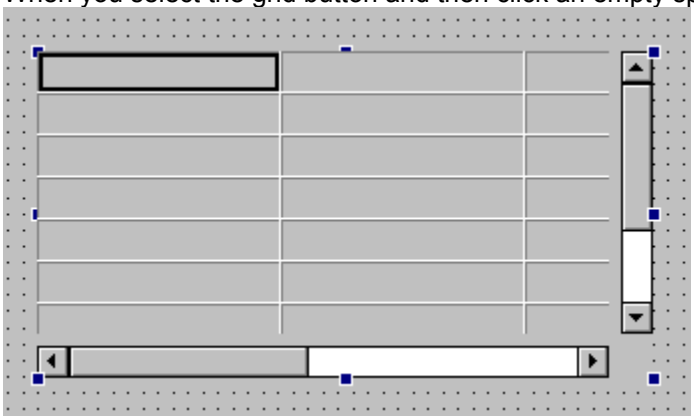
Chapter 11. Using grids

A grid control shows a table of information as a two-dimensional grid. Each entry in the grid is called a *cell*, and can contain text, pictures, or other controls. You can also define horizontal and vertical scroll bars, and headers (labels) for the columns and rows.

Creating a grid

On the **Standard** page of the component palette, grid controls are represented by the following button:

When you select the grid button and then click an empty spot on a form, the default grid appears:



Grid controls are represented by Grid objects (`powersoft.powerj.ui.grid.Grid`). PowerJ gives grid controls names of the form *grid_N*.

When working with grids, bear in mind the following:

- All columns and rows in the grid are numbered, starting with 0 at the upper left corner of the grid.
- To save space, the examples in this chapter assume that you have imported all of the grid classes (`import powersoft.powerj.ui.grid.*`). To be safe, you may prefer to specify complete names (such as `powersoft.powerj.ui.grid.GridEnum` instead of `GridEnum`).
- The GridEnum class specifies static constants that are used by many methods and properties.
- Wherever you must specify a row or column number, you can specify `GridEnum.ALLCELLS` (meaning every cell except labels) or `GridEnum.ALL` (which includes labels).

Note: This chapter covers only the most commonly used features of grid controls. For more information see the online *PowerJ Component Library Reference*.

Grid control properties and styles

Grid control properties and styles

Grid controls support the following design-time properties:

Number of Columns [**Options** page]
The number of columns in the grid control. For more information, see Setting the number of columns and rows.

Number of Rows [**Options** page]
The number of rows in the grid control. For more information, see Setting the number of columns and rows.

ReadOnly [**Options** page]
If this is turned on, the user may not edit any of the entries in the grid control. This property can also be set at run time for individual columns.

LiveEditMode [**Options** page]
If this is turned on, an editing text box appears when the user clicks a cell. (If this is turned off, the user must double-click or start typing for the text box to appear.) If you would rather use a different control than a text box, you can dynamically change the component, as shown in A simple grid example.

FullRowSelection [**Options** page]
If this is turned on, clicking any cell in the grid selects the entire row containing that cell. For more information, see Selection policies.

MultipleMode [**Options** page]
If this is turned on, the user may select more than one cell at a time. For more information, see Selection policies.

ResizableColumns [**Options** page]
If this is turned on, the user can change the size of a column by dragging a column divider line. For more information, see Resizing the grid at run time.

ResizableRows [**Options** page]
If this is turned on, the user can change the size of a row by dragging a row divider line. For more information, see Resizing the grid at run time.

ShowColumnLines [**Options** page]
If this is turned on, the grid control uses lines to separate columns.

ShowRowLines [**Options** page]
If this is turned on, the grid control uses lines to separate rows.

ShowVerticalHeader [**Options** page]
If this is turned on, a vertical column of row labels is shown. For more information, see Adding column and row labels.

ShowHorizontalHeader [**Options** page]
If this is turned on, a horizontal row of column labels is shown. For more information, see Adding column and row labels.

HScroll and **VScroll** [**Options** page]
Control whether the grid control contains horizontal and vertical scroll bars. For more information, see Positioning the scroll bars.

Putting data into the cells

A cell can contain any of the following objects:

- A string.
- An image (GIF or JPEG).
- An AWT component (a class that extends `java.awt.Component`).

Any other type of object (for example, an Integer) is displayed using its **toString** method.

Setting the value of a single cell

You can assign a value to a cell using the **setCell** method:

```
// Object value;
grid_1.setCell( rowNumber, columnNumber, value );
```

This sets the value of the cell in the given row and column. For example, to put the number 123 in the top cell (row 0) of the second column (column 1), specify the following:

```
grid_1.setCell( 0, 1, new Integer(123) );
```

Similarly, you can get the value of a single cell by using the **getCell** method:

```
Object obj = grid_1.getCell( rowNumber, columnNumber );
```

Note: If you use **setCell** to change a value in a bound grid, you should notify the query to update the database appropriately. The **flagForUpdate** method for the grid control indicates that a value has changed and the query should update the database.

Using an image or component

To put an image into a cell, use the **getImage** method of `java.awt.Toolkit`. For example:

```
java.awt.Toolkit tk = this.getToolkit();
java.awt.Image im = tk.getImage( "filename.gif" );
grid_1.setCell( 1, 1, im );
```

To put an AWT component into a cell, use the **setComponent** method:

```
grid_1.setComponent( row, col, component );
```

Setting the value of a range of cells

If you want to define several cells at once, you can use the **setCells** method. It is similar to the **setCell** method, but it takes either a two-dimensional array or a Vector of Vectors as its only argument.

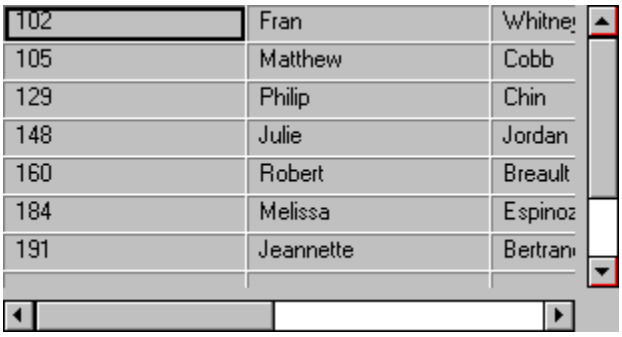
For example, to duplicate the grid that is used for most examples in this chapter, add the following lines to the "data members" section:

```
String cells[][] = {
  {"102", "Fran",      "Whitney",  "100"},
  {"105", "Matthew",   "Cobb",     "100"},
  {"129", "Philip",    "Chin",     "200"},
  {"148", "Julie",     "Jordan",   "300"},
  {"160", "Robert",    "Breault",  "100"},
  {"184", "Melissa",   "Espinoza", "400"},
  {"191", "Jeannette", "Bertrand", "500"}
};
```

Also add the following line to the **objectCreated** event handler:

```
grid_1.setCells( cells );
```

This creates the following grid:



Similarly, you can get the value of all the cells in the grid by using the **getCells** method:

```
GVector cells = grid_1.getCells();
```

Reading cell data from a database

If you want your grid to contain data from a database, you should bind the grid to a database query. For more information, see Bound grids.

Changing the look of a grid

This section describes many ways that you can change the appearance and behavior of the grid control.

Setting the colors of cells

As with other controls, you can specify a foreground color and background color on the **Options** page of the grid's property sheet. You can also use the **setBackground** or **setForeground** method to set the colors of specific cells:

```
grid_1.setBackground ( row, col, color );
grid_1.setForeground ( row, col, color );
```

Repeating (cycling) the colors

You can color either your rows or your columns in an alternating pattern. To do this, you set the **RepeatBackgroundColors** or **RepeatForegroundColors** property to a list of colors. You then specify `REPEAT_ROW` or `REPEAT_COLUMN` for the color argument of the **setBackground** or **setForeground** method.

For example, to make your columns red, white, blue, and red again (and so on if you have more than four columns), add the following lines to the **objectCreated** event handler (assuming you have imported `powersoft.powerj.ui.grid.GridEnum` and `java.awt.Color`):

```
Color[] collist = { Color.red, Color.white, Color.blue };
grid_1.setRepeatBackgroundColors( collist );
grid_1.setBackground(GridEnum.ALLCELLS,
    GridEnum.ALLCELLS, GridEnum.REPEAT_COLUMN);
```

Selected cells

To set the colors for selected cells, use **setSelectedBackground** and **setSelectedForeground**. For example:

```
grid_1.setSelectedBackground(Color.blue);
grid_1.setSelectedForeground(Color.cyan);
```

If **LiveEditMode** is turned on, the selected cell with the input focus is replaced by a text field. If several cells are selected at the same time, the selected foreground and background colors will be used for all of the selected cells except the one with input focus. For more information about live edit mode, see Grid control properties and styles.

Setting the number of columns and rows

On the **Options** page of the grid's property sheet, you can specify the number of columns and number of rows for the grid. (The default is 5 columns and 10 rows.)

In general, you should define the same number of columns and rows as you have in the data. If you specify more columns or rows of data than the grid can contain, the extra values are ignored. If you specify less data, the extra columns or rows are filled with blank cells.

For a bound grid, the number of rows is taken from the database. (Any value that you specify on the **Options** page is ignored.) To control the maximum number of rows that are displayed at a time, use the **DataKeptRows** property on the **Database** page. For more information, see Bound grids.

Resizing columns and rows

You can resize columns and rows with the **setPixelWidth** and **setPixelHeight** methods. For example, to specify that the first column (column 0) is 65 pixels wide, add the following line to the **objectCreated** event handler:

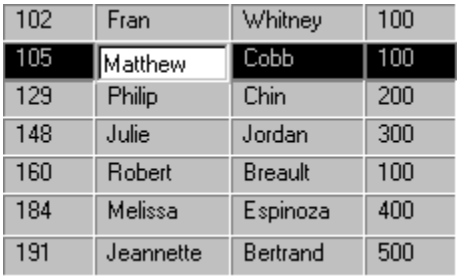
```
grid_1.setPixelWidth(0,65);
```

You can also base the size on the largest cell. For example (assuming you have imported `powersoft.powerj.ui.grid.GridEnum`):

```
grid_1.setPixelWidth(0, GridEnum.VARIABLE);
```

For a bound grid, the default column width is `GridEnum.VARIABLE`.

The following figure shows a grid with variable-width columns:



102	Fran	Whitney	100
105	Matthew	Cobb	100
129	Philip	Chin	200
148	Julie	Jordan	300
160	Robert	Breault	100
184	Melissa	Espinoza	400
191	Jeannette	Bertrand	500

If the column width and row height are not set to `GridEnum.VARIABLE`, some cells may be too small for their contents. In this case, a clipping arrow is shown in the lower right corner of the cell. To turn off clipping arrows, set the **displayClipArrows** property to `false`.

Adding column and row labels

Unless the data in your grid is self-explanatory, you will probably want to label the columns, the rows, or both. Column labels and row labels are like other cells except that they are not editable. `GridEnum.LABEL` is the row or column number for labels.

To add column labels:

1. On the **Options** page of the grid's property sheet, turn on **ShowHorizontalHeader**. This causes a horizontal row of column labels to appear at the top of the grid.
2. Set the label text with the **setColumnLabels** method.

For example, to define a set of column labels with an orange background, add the following lines to the "data members" section:

```
String labels[] = {
  "Employee #", "First name", "Last name", "Department #"
};
```

Add the following lines to the **objectCreated** event handler (assuming you have imported `powersoft.powerj.ui.grid.GridEnum` and `java.awt.Color`):

```
grid_1.setColumnLabels( labels );
grid_1.setBackground( GridEnum.LABEL, GridEnum.ALL,
     Color.orange );
```

The following figure shows the result:

Employee #	First name	Last name	Department #
102	Fran	Whitney	100
105	Matthew	Cobb	100
129	Philip	Chin	200
148	Julie	Jordan	300
160	Robert	Breault	100
184	Melissa	Espinoza	400
191	Jeannette	Bertrand	500

Because the column widths are set to `GridEnum.VARIABLE`, the first and last columns stretch to accommodate the column labels.
Similarly, to add row labels turn on **ShowVerticalHeader** and use the **setRowLabels** method.
To set an individual column or label, use the **setColumnLabel** or **setRowLabel** method. For example:

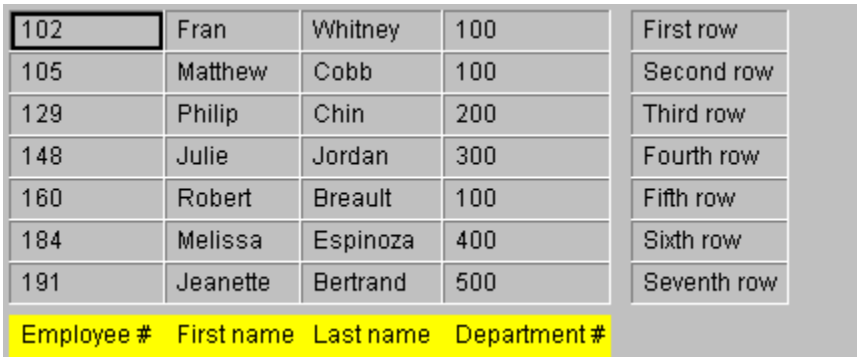
```
grid_1.setRowLabel( 2, "Third row" );
```

Employee #	First name	Last name	Department #	
102	Fran	Whitney	100	First row
105	Matthew	Cobb	100	Second row
129	Philip	Chin	200	Third row
148	Julie	Jordan	300	Fourth row
160	Robert	Breault	100	Fifth row
184	Melissa	Espinoza	400	Sixth row
191	Jeanette	Bertrand	500	Seventh row

Moving labels

By default, labels appear above the columns and to the left of the rows, with no blank space between the labels and the other cells. The following lines move the column labels 5 pixels below the grid and move the row labels 10 pixels to the right of the grid (assuming you have imported `powersoft.powerj.ui.grid.GridEnum`):

```
grid_1.setColumnLabelOffset( 5 );
grid_1.setColumnLabelPlacement( GridEnum.PLACE_BOTTOM );
grid_1.setRowLabelOffset( 10 );
grid_1.setRowLabelPlacement( GridEnum.PLACE_RIGHT );
```



Bound grids

For a bound grid, any column labels that you specify are replaced by the actual names of the columns in the database. You can override these default labels by placing appropriate code in the **dataFillEnd** event handler for the grid. The **dataFillEnd** event is triggered when the program has finished filling the grid with data obtained from a database. At this point, the program has already set the headers to the database column names, so you can use normal **setRowLabel** and **setColumnLabel** calls to change the headers.

Aligning data within cells

You can specify the alignment of a cell as follows:

```
grid_1.setAlignment( row, column, type );
```

The `row` and `column` arguments are integer values giving the row and column of the cell. The `type` argument must be one of the following:

```
GridEnum.TOPLEFT
GridEnum.TOPCENTER
GridEnum.TOPRIGHT
GridEnum.MIDDLELEFT
GridEnum.MIDDLECENTER
GridEnum.MIDDLERIGHT
GridEnum.BOTTOMLEFT
GridEnum.BOTTOMCENTER
GridEnum.BOTTOMRIGHT
```

The next section contains an example.

Spanning multiple cells

Sometimes, you may want a cell to expand to fill the space normally occupied by several cells. For example, in the previous figure you might want to replace the "First name" and "Last name" labels with the single label "Name" centered over both columns.

To do this, you could add the following lines to the **objectCreated** event handler (assuming you have imported `powersoft.powerj.ui.grid.GridEnum` and `powersoft.powerj.ui.grid.CellRange`):

```
java.util.Vector v = new java.util.Vector();
CellRange r = new CellRange( GridEnum.LABEL, 1,
     GridEnum.LABEL, 2);
v.addElement( r );
grid_1.setSpans( v );
grid_1.setAlignment (GridEnum.LABEL, 1,
     GridEnum.MIDDLECENTER);
```

You should also change the label text:

```
String labels[] = {
     "Employee #", "Name", null, "Department #"
};
```

The second label ("Name") appears in the label line and stretches across columns 1 and 2. Any label that is defined for column 2 is ignored. For more information about the **CellRange** method, see Working with cell ranges

The following figure shows the result:

Employee #	Name		Department #
102	Fran	Whitney	100
105	Matthew	Cobb	100
129	Philip	Chin	200
148	Julie	Jordan	300
160	Robert	Breault	100
184	Melissa	Espinoza	400
191	Jeannette	Bertrand	500

Adding borders and margins

You can place a border around your grid. If the grid has labels, separate (but identical) borders surround the row labels, the column labels, and the other cells. Use the **setFrameShadowThickness** method to define the width of the borders, and **setFrameBorderType** to define the type of border (none, etched in, etched out, in, out, or plain). If the shadow thickness is 0 (the default) no border is visible.

You can also place a border around each individual cell. Use the **setShadowThickness** method to define the width of the border, and **setBorderType** to define the type of border (none, etched in, etched out, frame in, frame out, in, out, or plain). If the shadow thickness is 0 (the default) no border is visible.

By default, the cell has a border on all four sides. Use the **setBorderSides** method to override this default. For example, you can specify that the cell in row 1, column 1 has a border only on the top and left as follows:

```
grid_1.setBorderSides( 1, 1, GridEnum.BORDERSIDE_TOP |
   GridEnum.BORDERSIDE_LEFT );
```

By default, there is an invisible margin around the inside of each cell border, where no text can appear. This margin is 2 pixels high and 5 pixels wide. You can override this default with the **setMarginHeight** and **setMarginWidth** methods.

If you want to add vertical lines between columns or horizontal lines between rows, use the **ShowColumnLines** and **ShowRowLines** properties on the **Options** page.

Defining how scroll bars look and behave

This section describes various properties that affect scrolling.

Positioning the scroll bars

By default, horizontal and vertical scroll bars are shown "as needed" (whenever the cells are too big to fit in the defined area). You can override this default by setting the **HScroll** and **VScroll** properties on the **Options** page. You can specify that scroll bars always appear or never appear.

By default, the distance between the grid and the scroll bars is 6 pixels. You can override this default with the **setHorizSBOffset** and **setVertSBOffset** methods.

When the user scrolls, row and column labels always stay in the same position. For this reason, the default scroll bars do not cover the labels. You can override this default by specifying **setHorizSBAttachment** or **setVertSBAttachment** with the value `GridEnum.ATTACH_SIDE`. (The default behavior corresponds to `GridEnum.ATTACH_CELLS`.)

The displayed area taken up by the cells and labels of a grid might be smaller than the size of the grid control that you specify on the **Size** page or by dragging the handles on the design-time grid object. In this case, you might prefer to have the scroll bar line up with the displayed cells or with the side of the underlying grid control. If you line it up with the displayed cells, the scroll bar may move when users vary column widths. If you align it with the side of the grid control, the scroll bar will stay in the same place on the form, but there may be a gap between it and the displayed cells.

You can place the scroll bar relative to the displayed cells or to the underlying grid control. For a non-bound grid, the default position is based on the displayed cells (`GridEnum.SBPOSITION_CELLS`). For a bound grid, the default position is based on the side of the underlying grid control (`GridEnum.SBPOSITION_SIDE`). You can override this default with the **setHorizSBPosition** and **setVertSBPosition** methods. For example, to change the vertical scroll bar in a bound grid to be aligned with the displayed cells, you could add the following code to the **ObjectCreated** event handler:

```
grid_1.setVertSBPosition( GridEnum.SBPOSITION_CELLS );
```

Preventing some columns and rows from scrolling

You can specify that certain rows and columns always remain in the same position when the user scrolls (just like labels). Use the **setFrozenColumns** and **setFrozenRows** methods to specify the number of columns (starting at the left) and rows (starting at the top) that are frozen. For example, the following line freezes the first two columns:

```
grid_1.setFrozenColumns( 2 );
```

By default, frozen rows appear at the top (`GridEnum.PLACE_TOP`) and frozen columns appear at the left (`GridEnum.PLACE_LEFT`). Use the **setFrozenColumnPlacement** and **setFrozenRowPlacement** methods to override this default. For example:

```
grid_1.setFrozenColumnPlacement( GridEnum.PLACE_RIGHT );
```

This statement moves the frozen column or columns from the left to the right of the grid.

If you want to freeze rows that are already at the bottom of the grid, or columns that are already at the right, there is no way to do this directly. Instead, you move the rows to the top (or columns to the right), then freeze them, then specify that frozen rows should be placed on the bottom (or columns on the right). The following example has the effect of freezing the bottom row (assuming that the grid has seven rows and you have imported `powersoft.powerj.ui.grid.GridEnum`):

```
grid_1.moveRows( 6, 1, 0 );
grid_1.setFrozenRows( 1 );
grid_1.setFrozenRowPlacement( GridEnum.PLACE_BOTTOM );
```

For more information about the **moveRows** method, see Moving rows and columns.

Scrolling one column or row at a time

By default, the scroll arrows move through the grid in very small increments. To override this default (for horizontal scrolling, vertical scrolling, or both), use the **setJumpScroll** method. With jump scrolling, the smallest increment is a complete column or row. You can specify the following values: `GridEnum.JUMP_NONE`, `GridEnum.JUMP_HORIZONTAL`, `GridEnum.JUMP_VERTICAL`, and `GridEnum.JUMP_ALL`.

Scroll events

When the user scrolls, a **ScrollBegin** event and a **ScrollEnd** event are triggered.

Adding a caption to the grid

You may want to add a title or caption to the grid. The simplest way to do this is to place a Label object on the form above or below the grid. For more information, see Simple labels.

Programming user interaction

This section describes the events, methods, and properties that let your program do more than passively display data. You can:

- Control which actions the user can perform (in particular, selection and editing).
- Retrieve input from the user (including cell selection and new cell values).
- Based on that input, take appropriate action.

Working with cell ranges

A cell *range* is any set of cells that occupy a rectangular area within the grid. To specify a range, use the CellRange class (`powersoft.powerj.ui.grid.CellRange`). For example, the **setSelectedCells** method can take a CellRange argument indicating the range of cells that you want to select.

To create a cell range, you specify the row and column of the start position, followed by the row and column of the end position:

```
CellRange cr =
  new CellRange( startRow, startCol, endRow, endCol );
```

The start and end cell are considered two corners of a rectangle. The cell range includes all of the cells in this rectangle. The start and end cells can have the same row number, the same column number, or both. (If both the row numbers and the column numbers are the same, the range consists of a single cell.)

To include a complete row or column, specify zero for the start position and `GridEnum.MAXINT` for the end position.

The **reshape** method of CellRange changes the range of cells associated with the CellRange object. The possible forms of **reshape** are:

`cr.reshape( row, col )`
Reshapes the cell range to refer to the single cell at the given row and column.

`cr.reshape( startRow, startCol, endRow, endCol )`
Reshapes the cell range to refer to the specified range.

`cr.reshape( cellRange2 )`
Reshapes the cell range to refer to the range associated with CellRange `cellRange2`.

The CellRange class supports a number of other methods. In the descriptions below, `cr1` and `cr2` are both CellRange objects.

`boolean over = cr1.overlaps( cr2 );`
Is `true` if the two ranges have any cell(s) in common.

`CellRange cr3 = cr1.intersection( cr2 );`
Returns the cell range corresponding to the intersection of `cr1` and `cr2`.

`boolean in = cr1.inside( row, col );`
Is `true` if the cell at the given row and column belongs to the cell range.

`boolean in = cr1.inRowRange( row );`
Is `true` if the given row is in the range of rows for the cell range.

`boolean in = cr1.inColumnRange( col );`
Is `true` if the given column is in the range of columns for the cell range.

Selecting cells

This section describes how to:

- Control which cells can be selected, edited, and traversed.
- Retrieve (get) and change (set) the list of selected cells.
- Create selection events and traversal events.

Selection policies

Cells can be selected by the user or the program. You may or may not want to allow more than one cell to be selected at a time. On the **Options** page of the properties sheet, you can specify any of the following:

- To force an entire row to be selected, turn on **FullRowSelection**. Whenever the user clicks a cell, all of the cells in that row are selected. (Only the cell that was actually clicked gets the input focus.)
- To allow several cells to be selected at the same time, turn on **MultipleMode**. When a cell is selected, the user can select an additional cell by using CTRL+click or select the entire range from the previous click by using SHIFT+click. The user can also hold down the mouse button and drag across multiple cells, or hold down the SHIFT key and press the arrow keys.

If **MultipleMode** does not give you the control you need, you can use the **SelectionPolicy** property. It can have any of the following integer values:

`GridEnum.SELECT_NONE`
: Cells cannot be selected. Clicking a cell does not trigger a **SelectBegin** event. To prevent the user from editing cells, turn on the **ReadOnly** property on the **Options** page.

`GridEnum.SELECT_SINGLE`
: Only one cell can be selected at a time. Clicking a new cell unselects the previous cell. CTRL+click and SHIFT+click are disabled. This value does not override **FullRowSelection**.

`GridEnum.SELECT_RANGE`
: Only one range of cells can be selected at a time. Clicking a new cell unselects the previous range. Both CTRL+click and SHIFT+click select the range from the previously clicked cell to the new cell (and unselect any previously selected cells that are outside the new range).

`GridEnum.SELECT_MULTIRANGE`
: Multiple ranges of cells (which do not need to be contiguous) can be selected at the same time. Clicking a new cell unselects the previous ranges. SHIFT+click select the range from the previously selected cell to the new cell. CTRL+click keeps the previous selections and selects the new cell. This is the same behavior as **MultipleMode**.

For example, you can set the **SelectionPolicy** property as follows:

```
grid_1.setSelectionPolicy( GridEnum.SELECT_SINGLE );
```

Setting **SelectionPolicy** does not change the current selection. For example, if the user has selected one or more cells, the cells are *not* unselected if you set **SelectionPolicy** to `GridEnum.SELECT_NONE`. If you want the cells to be unselected, you must do this explicitly.

Selecting and unselecting cells

The **SelectedCells** property determines which cells are currently selected. It contains either a CellRange or a vector of CellRange objects. The possible forms of **setSelectedCells** are:

```
grid_1.setSelectedCells( cr );
grid_1.setSelectedCells( crVect );
```

The **getSelectedCells** method returns a vector of CellRange objects indicating the ranges that are currently selected:

```
Vector crVect = grid_1.getSelectedCells();
```

To get one CellRange from within the vector, use **getSelectedRange**:

```
grid_1.getSelectedRange ( pos, crVect );
```

The `pos` argument gives the position of the range within the vector. For both **getSelectedCells** and **getSelectedRange**, special values such as `GridEnum.MAXINT` are converted to integers.

To unselect all the cells that are currently selected, use:

```
grid_1.clearSelectedCells();
```

Select events

The **SelectBegin** event is triggered on a grid control when the user begins selecting a range of cells, and the **SelectEnd** event is triggered when the user finishes selecting the range.

More than one **SelectBegin** event can be triggered. An event is triggered for the cell being selected; another for the cell being unselected. Use **getStateChange** to distinguish the events. This method returns `SelectEvent.SELECTED` if the cell is being selected, and `SelectEvent.DESELECTED` if the cell is being unselected.

If there is more than one cell in the range, there will be a **SelectBegin** event each time a new cell is added to the range, but there will still only be one **SelectEnd** event.

Note: If the user selects a range of cells by dragging the mouse, the **SelectEnd** event takes place when the user releases the mouse button. If the user selects a single new cell by tabbing from the currently selected cell, the **SelectBegin** and **SelectEnd** events happen immediately after each other as the new cell is selected.

The `event` argument for both **SelectBegin** and **SelectEnd** has the SelectEvent type (`powersoft.powerj.ui.grid.SelectEvent`). This class accepts a number of methods for determining information about the selection:

`int row = event.getRow();`
Returns the row number of the cell selected.

`int col = event.getColumn();`
Returns the column number of the cell selected. If you are using **FullRowSelection**, the value `-998` (equivalent to `GridEnum.ALLCELLS`) is returned.

`String how = event.getParam();`
Returns a string that indicates how the cell was selected. Possible values include:

`"START"`	Returned if the user selects a new cell by clicking it. When a new cell is selected, *two* **SelectBegin** events with the START parameter are triggered.
`"ADD"`	Returned if the user adds a cell to the selection with CTRL+click.
`"EXTEND"`	Returned if the user selects a range of cells with SHIFT+click, or implicitly selects a range because **FullRowSelection** is turned on. A **SelectBegin** event with the `EXTEND` parameter is triggered for the cell where the action occurred. No event is triggered for the other cells that are added to the selected range.
`"END"`	Returned in the **SelectEnd** event.

`int state = event.getStateChange();`
Returns `SelectEvent.SELECTED` if the cell is being selected, and `SelectEvent.DESELECTED` if the cell is being unselected.

The SelectEvent class also supports a number of methods that your **SelectBegin** and **SelectEnd** event handlers can use to modify the selection. For example, if you want to prevent the user from selecting a cell, you can use **setAllowSelection** as discussed below.

`event.setAllowSelection( bool )`
If the boolean argument is `true` (the default), the selection is allowed; if `false`, the selection action fails and the user is prevented from making the selection.

`event.setRow( row )` and `event.setColumn( col )`
Using these two methods, you can change the row and/or column that will be selected. For example, if the user tries to select a cell in a "forbidden" area of the grid, your **Select** event handler might change the selection to the closest cell in a non-forbidden area.

Traversal events

A **TraverseCell** event is triggered when the user moves to a new cell with the tab and arrow keys or selects a new cell. This event occurs after the value in the previous cell is validated and committed, but before the new cell is highlighted or selected.

This event accepts the following methods:

`int row = event.getRow();`
Returns the number of the previous row.

`int col = event.getColumn();`
Returns the number of the previous column.

`int nextrow = event.getNextRow();`
Returns the number of the new row.

`int nextcol = event.getNextColumn();`
Returns the number of the new column.

`String how = event.getParam();`
Returns a string that indicates how the cell was traversed. If a tab or arrow key was used, the value is `LEFT`, `RIGHT`, `UP`, or `DOWN`. If the mouse was used, the value is `POINTER`. If the program caused the traversal, the value is `null`.

You can also use the following methods to change which cell is traversed:

```
event.setNextColumn( int )
event.setNextRow( int )
```

An **EnterCellBegin** event is triggered before the **TraverseCell** event, and an **EnterCellEnd** event is triggered after the **TraverseCell** event.

Preventing editing or traversal

By default, every cell can be traversed and edited. You can override this default as follows:

```
grid_1.setEditable( row, col, false );
grid_1.setTraversable( row, col, false );
```

Making selected cells visible

By default, when a cell that is only partly visible is selected, the grid scrolls so that the entire cell becomes visible. To override this default, set the **MinCellVisibility** property to the percentage of the cell that should be visible (at minimum). If less than this percentage is visible when the cell is selected, the grid scrolls until this percentage is visible. If this property is set to 0, no scrolling occurs. The default is 100.

Editing cells

Unless a cell is defined as read only, users can edit the contents. This section describes methods and events that you may want to use when you are handling user edits.

Clearing cells

To clear an individual cell, simply set its value to `null`:

```
grid_1.setCell( row, col, null );
```

To clear all of the cells in the grid, use the **clearCells** method:

```
grid_1.clearCells();
```

Committing a cell change

Whenever the user traverses or selects another cell, the currently selected cell is validated and committed. You can also specify that validation and commitment should happen immediately by using the **commitEdit** method:

```
grid_1.commitEdit( hide );
```

The boolean variable `hide` specifies whether the text component should be hidden.

In grid terms, a commit means that the data is saved into the grid. It does not mean (for a bound grid) that the database is updated. For information about updating the database, see Bound grids.

Getting the current cell

To see which cell has the edit focus, use the **getCurrentCell** method:

```
CellPosition cp = grid_1.getCurrentCell();
```

Similarly, to get the current text component, use the **getTextComponent** method. If no cell has the edit focus, both of these methods return `null`.

Value events

Value events (**cellValue**, **rowLabelValue**, or **columnLabelValue**) are triggered when an empty cell is displayed. These events are triggered during the initial grid display, scrolling, and repainting.

For example, you can put a default value of `UNSPECIFIED` in every empty cell as follows:

```
public void grid_1_cellValue( ValueEvent event )
{
   event.setValue("UNSPECIFIED");
   event.setStore(true);
}
public void grid_1_validateCellBegin(ValidateCellEvent event)
{
   grid_1.removeCellValueListener(this);
   event.setValue(event.getValue());
   event.setAllowValueChange(true);
}
public void grid_1_validateCellEnd( ValidateCellEvent event )
```

```
{
   grid_1.addCellValueListener(this);
}
```

Validating the contents of cells

If the user (or the program) changes the value of a cell, you can validate the new value before it is committed.

Setting the validation policy

The **ValidatePolicy** property controls when cell values are validated. It can have any of the following integer values:

`GridEnum.VALIDATE_NEVER`
The cell values are never validated.

`GridEnum.VALIDATE_USER_EDIT`
When the user changes the value in a cell, the grid attempts to validate the new value.

`GridEnum.VALIDATE_SET`
When the program changes the value in a cell, the grid attempts to validate the new value.

`GridEnum.VALIDATE_ALWAYS`
When a cell is changed by either the user or the program, the grid attempts to validate the new value.

Using data types for automatic validation

You can associate a data type with a cell as follows:

```
grid_1.setDatatype( row, column, typeCode );
```

The `row` and `column` arguments are integer values giving the row and column of the cell. The `typeCode` argument must be one of the following:

```
GridEnum.TYPE_STRING
GridEnum.TYPE_DOUBLE
GridEnum.TYPE_INTEGER
GridEnum.TYPE_FLOAT
GridEnum.TYPE_BOOLEAN
```

If the user (or the program) attempts to fill a cell with a value of the wrong type, the grid does not accept the value. For this to work, validation must be enabled. If the **ValidatePolicy** property is set to `GridEnum.VALIDATE_NEVER`, the data type is never checked.

Note: If the grid is bound to a database, the data type from the corresponding column in the database is used. For bound grids, the default **ValidatePolicy** is `GridEnum.VALIDATE_USER_EDIT`. For more information about bound grids, see Bound grids.

If data type validation fails, the exact result depends on whether the validation was triggered by a traversal event (when a tab or arrow key is pressed) or a select event (when the mouse is clicked). For a traversal event, the traversal is rejected and the input focus remains on the current cell. For a select event, the rejected value is replaced by the previous value.

If you want more control over the validation process, use validation events instead of data type validation. (Either do not define data types, or use a data type of String.)

You can also use data type validation and validation events together. The validation events are called only if the data type validation is successful.

Using validation events for manual validation

The grid validates cell values (when necessary) by triggering **ValidateCell** events. The **ValidateCellBegin** event is triggered after a cell has been edited but before the changed value has actually been committed to the grid. The **ValidateCellEnd** event is triggered after the changed value has been committed and displayed. Typically, you would use the **ValidateCellEnd** event to tell the program (or database query) that the data has changed.

The `event` argument for both events has the ValidateCellEvent type (`powersoft.powerj.ui.grid.ValidateCellEvent`). This class has several methods that provide information about the selection, including:

`int row = event.getRow();`
Returns the row number of the cell to be validated.

`int col = event.getColumn();`
Returns the column number of the cell to be validated.

`boolean change = event.getChanged();`
Is `true` if the user or program is attempting to change the cell value and `false` otherwise.

`Object obj = event.getValue();`
Returns the proposed new value for the cell.

`int datatype = event.getDataType();`
Returns the data type of the cell. (The data type of the proposed value has already been validated.)

Based on this information, your application can determine whether to accept the new cell value. It then typically calls one of the following ValidateCellEvent methods:

`event.setAllowValueChange( bool );`
If the boolean argument is `true` (the default), the grid is allowed to change the cell to the specified value. If `false`, the value is considered invalid and the cell returns to its previous value (the commit fails and the value is rolled back).

`event.setValue( obj );`
Changes the value of the cell to the given object. For example, if a cell should contain numbers in the format `n.nn`, and the user enters `1.5`, the **ValidateCellBegin** event may reset the value to `1.50`.

Example

The following simple validation event trims leading and trailing blanks from any value that the user enters:

```
String currentValue = (String)event.getValue();
if (currentValue != null)
{
   event.setValue(currentValue.trim());
}
```

Adding rows and columns

The **addRow** method adds a new row to the grid control:

```
// Object label;
// Vector values;
grid_1.addRow( pos, label, values );
```

The `pos` argument gives the row number for the new row. The `label` argument gives the row heading, and `values` gives a set of values to place in the new row. Both `label` and `values` may be null. Rows below the new one are shifted downward to make space for the new row.

If the `pos` argument is specified as `GridEnum.MAXINT`, the new row is added after the last row currently in the grid control.

The **addColumn** method adds a new column to the grid control:

```
grid_1.addColumn( pos, label, values );
```

The arguments are the same as for **addRow**.

For example, the following statement adds an empty column after the last column (assuming you have imported `powersoft.powerj.ui.grid.GridEnum`):

```
grid_1.addColumn( GridEnum.MAXINT, null, null );
```

Deleting rows and columns

The **deleteRow** method deletes a given row from the grid:

```
grid_1.deleteRow( row );
```

The row and all its values disappear from the grid (including the label, if there is one).

The **deleteRows** method deletes a number of rows from the grid:

```
grid_1.deleteRows( pos, numRows );
```

The `pos` argument gives the number of the first row to delete, and the `numRows` argument gives the total number of rows to delete.

The **deleteAllRows** method deletes all rows from the grid:

```
grid_1.deleteAllRows( );
```

If you want to use the grid again, you must create new rows.

The **deleteColumns** method deletes a number of columns from the grid:

```
grid_1.deleteColumns( pos, numColumns );
```

The `pos` argument gives the number of the first column to delete, and the `numColumns` argument gives the total number of columns to delete.

For example, the following statement deletes the first two rows (0 and 1):

```
grid_1.deleteRows( 0, 2 );
```

Bound grids

Deleting a row or column from the grid does not delete it from the database. If you want to remove the current row from the database, turn on **DataTrackRow** (see Bound grids) and call the following:

```
try
{
     query_1.delete();
}
catch (java.lang.Exception exception )
{
     exception.printStackTrace();
}
```

Moving rows and columns

The **moveRows** method moves one or more entire rows (including labels, if any).

```
grid_1.moveRows( source, numRows, dest );
```

The `source` argument gives the number of the first row to move, and the `numRows` argument gives the total number of rows to move. The rows are moved immediately to the right of the position given by the `dest` argument. Specify zero to move the rows to the top, or `GridEnum.MAXINT` to move the rows to the bottom.

The **moveColumns** method moves one or more columns within the grid:

```
grid_1.moveColumns( source, numCols, dest );
```

The arguments are the same as for **moveRows**.

For example, the following statement moves the fifth column (column 4) to the first (column 0) position:

```
grid_1.moveColumns( 4, 1, 0 );
```

Sorting the grid

The **sortByColumn** method sorts the rows of the grid according to the value of the specified column. The possible forms of **sortByColumn** are:

```
grid_1.sortByColumn( col, interface );
grid_1.sortByColumn( col, interface, direction );
```

The arguments are the column number, the sorting interface (or `null` to use standard numerical or string comparison), and the sorting direction (`Qsort.ASCENDING` or `Qsort.DESCENDING`). For more information about sorting, see the documentation for `powersoft.powerj.ui.grid.SortInterface` in the online *PowerJ Component Library Reference*.

For example, the following statement sorts the grid by the third column:

```
grid_1.sortByColumn( 2, null );
```

If you set the **ColumnLabelSort** property to `true`, a **Sort** event is triggered and the grid is sorted by a particular column whenever the user clicks its label. (In this case, clicking a label does not select the column.) For example:

```
grid_1.setColumnLabelSort(true);
```

To sort the grid by more than one column, you must write your own interface or, for a bound grid, specify sorting in the query. For more information, see Bound grids.

Resizing the grid at run time

The section Resizing columns and rows describes how your program can control the size of the grid. You can also allow the user to control the size at run time by turning on the **ResizableColumns** and **ResizableRows** properties on the **Options** page of the properties sheet.

To resize a column or row, the user drags the row or column divider. This affects only the size of the column to the left of the divider or the row above the divider. All of the other columns or rows remain the same size, which means that the grid as a whole gets smaller or larger.

If the user resizes a column or row whose width or height is set to `GridEnum.VARIABLE`, the size is no longer considered variable. The size will remain the same no matter how large or small the cell contents are.

When the user resizes a cell, a **ResizeCellBegin** event and a **ResizeCellEnd** event are triggered.

Finding out which columns and rows are visible

The **getVisibleColumns** and **getVisibleRows** methods returns the number of columns and rows that are currently visible:

```
int viscols = grid_1.getVisibleColumns();
int visrows = grid_1.getVisibleRows();
```

The **getLeftColumn** method returns number of the column at the left of the visible area (not counting frozen columns). Similarly, the **getTopRow** method returns the top row (not counting frozen rows):

```
int col = grid_1.getLeftColumn();
int row = grid_1.getTopRow();
```

Specifying user data for cells

You can define "user data" (or meta-data) for each cell in your grid. You decide what information to store and what to do with it. PowerJ does not use the data in any way.

The **setUserdata** and **getUserdata** methods have the following form:

```
// Object userdata;
grid_1.setUserdata( row, col, userData );
Object val = grid_1.getUserdata( row, col );
```

User data is tied to a specific cell. If you move or delete the row or column containing the cell, the user data is also moved or deleted.

For a bound grid, PowerJ keeps track of the result set by defining user data for the row labels. Do not define user data for the row labels yourself, because this would interfere with PowerJ's definitions. You are free to define user data for any other cell.

Improving grid performance

This section describes various ways that you can improve performance. This is particularly important for a large grid.

Repainting the screen

By default, the screen is repainted every time a cell value is updated. You can override this default as follows:

```
grid_1.setRepaint( false );
```

If you update many cells, you can improve performance by setting the **Repaint** property to `false`, making the changes, and then resetting it to `true`.

Using double buffering

When the grid is displayed and updated, double buffering is used by default. Double buffering improves performance, but also increases the memory requirements. You can override this default as follows:

```
grid_1.setDoubleBuffer( false );
```

Bound grids

For bound grids, two properties on the **Database** page affect the performance and amount of memory that is used. **DataKeptRows** specifies the number of rows that are displayed at a time. **DataGuardRows** specifies how close you must get to the top or bottom of the list before new rows are added. For more information, see Bound grids.

A simple grid example

This section shows an example of dynamically placing a Choice object in a cell whenever the row containing that cell is selected for editing. The grid itself is implemented as a bound control, with three data columns obtained from a database: the first name, last name, and sex of a person.

This example creates a Choice control in the column corresponding to sex, so that users can only choose `M` or `F`. The Choice control tracks the cursor: when a new row is selected, the Choice is removed from the old row and moved to the appropriate cell in the new row.

```
// add a variable for the Choice control
private java.awt.Choice     choice_1;

// this is the objectCreated event for the form
public boolean
    Form1_objectCreated(powersoft.powerj.event.EventData event)
{
    // create the choice control and fill it
    // with the possible values (M, F)
    choice_1 = new java.awt.Choice();
    choice_1.addItem( "M" );
    choice_1.addItem( "F" );

    grid_1.setVertSBPosition(
        powersoft.powerj.ui.grid.GridEnum.ATTACH_CELLS );

    // initially, set the cell corresponding to the
    // sex column of row 1 to the choice control --
    // the sex column is the 3rd column of the grid
    choice_1.select( (String)grid_1.getCell( 0, 2 ) );

    // set the cell to the choice control
    grid_1.setComponent( 0, 2, choice_1 );

    return false;
}

// this is the traverseCell event on the grid
public void grid_1_traverseCell(
    powersoft.powerj.ui.grid.TraverseCellEvent event )
{
    if( event.getRow() < 0 ) return;

    // remove the choice control from the current row
    grid_1.setComponent( event.getRow(), 2, null );

    // set the value of the choice control to the
    // corresponding value in the cell of the newly select row
    choice_1.select( (String)grid_1.getCell(
        event.getNextRow(), 2 ) );

    // set the cell corresponding to the sex column
    // of the newly selected row to the choice control
    // -- the sex column is the 3rd column of the grid
```

```
    grid_1.setComponent( event.getNextRow(), 2, choice_1 );
}
```

You can use other controls in the same manner, as long as the class extends (directly or indirectly) `java.awt.Component`.

Chapter 12. Using and programming menus

This chapter shows how to add menus to a form and how to design such menus using the PowerJ menu editor.

The MenuBar object

A menu bar is a single object representing all the menus displayed on the menu bar of a form. Menu bar objects belong to the MenuBar class (`java.awt.MenuBar`). PowerJ gives MenuBar objects default names of the form *menu_N*. On the **Standard** page of the component palette, MenuBar objects are represented by the following button:

If you want a form to have menus in its menu bar, you must place a menu bar item somewhere on the form. The position of the object doesn't matter—the menu bar always appears below the title bar of the form.

> **Note:** The icon for the menu bar object you place on the form is visible at design time, but not at run time. The menu bar itself is visible at both design time and run time (provided that it contains at least one menu).

Menus and menu items
Menu bars can only be placed on frames
The default menu bar for a form

Menus and menu items

A menu bar object may contain any number of menus. Each of these menus appears as a separate heading on the menu bar.

A menu may contain any number of menu items. The menu items appear when the user clicks the menu's heading on the menu bar. The menu that contains the menu items is called the *parent menu* of those menu items. Each menu item is called the *child* of its parent menu.

Some menu items may be menus themselves. Such items are called *submenus* of the parent menu. When the user clicks on a submenu, the program displays the menu items belonging to the submenu. Submenus may have submenus of their own, down to any level of nesting.

Every menu and menu item has the following characteristics:

- A *caption*. This is the text that the user sees when the menu or menu item is displayed.
- A *variable name suffix*. This is combined with the name of the menu bar object to create an identifier for the menu or menu item. For example, suppose that a menu bar object named `menu_1` contains an item with the suffix `Item1`. Then the full identifier for the menu item is `menu_1_Item1`, combining the name of the menu bar object with the suffix of the item itself.

All menu items are MenuItem objects. If a menu item is a menu or submenu, it belongs to the Menu class, which is derived from MenuItem. Therefore, everything belonging to a MenuBar object is a MenuItem (although some objects may also be Menu objects if they contain children).

Note: The MenuItem, MenuBar, and Menu objects are all based on a common class called MenuComponent. Therefore, the properties and methods of MenuComponent are available for all menu-related objects.

Menu bars can only be placed on frames

A menu bar object can only be placed on a *frame*, not on a dialog or applet form—this is a limitation of AWT. For example, you cannot place a menu bar on an Applet form, since it is not a frame. In general, you must create a Frame with the Form Wizard in order to have an object where you can place a menu bar.

If you are running an applet and want to make use of menus, open a frame from the original applet. This frame can have menus on it. For example, if MyFrame is a form based on the Frame class, an applet could open the form with

```
MyFrame mf = new MyFrame();
mf.create();
```

For more information on frames and other types of forms, see Defining forms.

The default menu bar for a form

You can associate more than one menu bar object with a form. For example, you might do this if you wanted to have several different sets of menus, depending on the current context. When the context changes, you can change from one menu bar object to another to obtain a completely different set of menus.

When you have more than one menu bar object on a form, you should specify which is the default menu bar object. This is the object that will be used to create the menu bar when the form is first displayed.

- **To specify the default menu bar for a form:**

1. Open the form's property sheet by double-clicking a blank area of the form.
2. On the **Menu Bars** page of the property sheet, open the **Menu** combo box and click which menu bar object will serve as the default.
3. Click **OK**.

At run time, you can change which menu bar is displayed by using the **setMenuBar** method of the form. For example,

```
setMenuBar( menu_2 );
```

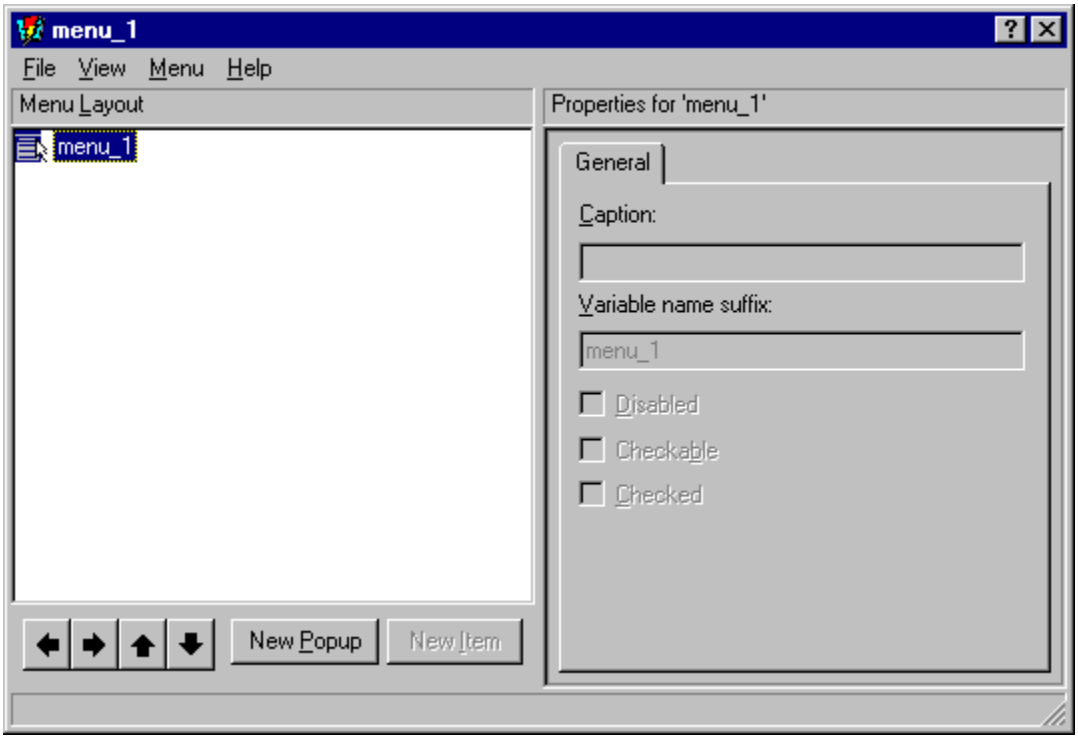
specifies that the form's menu bar should take its menus from the `menu_2` object.

Editing menus

Once you have placed a menu bar object on a form, you can specify menus for the menu bar using the menu editor.

- **To open the menu editor:**

1. Use the right mouse button to click the menu bar object, then click **Edit Menu**. This displays the menu editor.



You can use this editor to specify the menus and menu items of the menu bar.

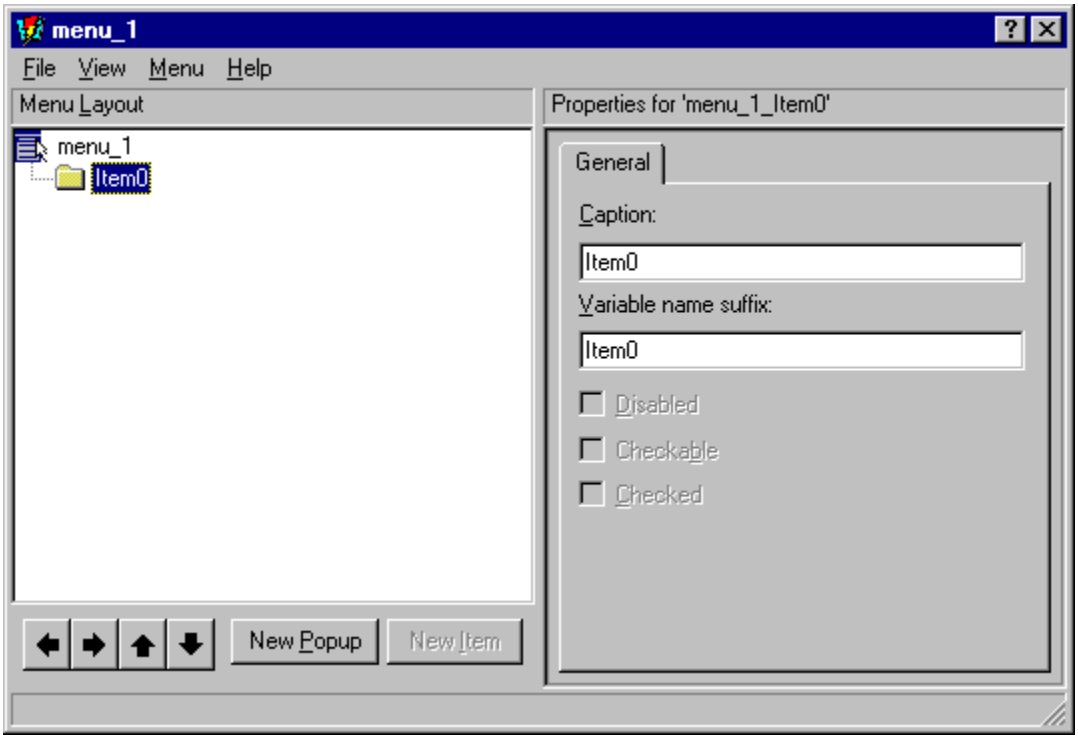
Creating a menu or menu item

You create menus and menu items with the menu editor.

- **To create a new popup menu for the menu bar:**

1. Open the menu editor.
2. Click the **New Popup** button.

When the menu editor creates a new item, it assigns a default caption to that item. Default captions take the form *ItemN*, beginning with `Item0`. The new item is displayed in the menu editor:



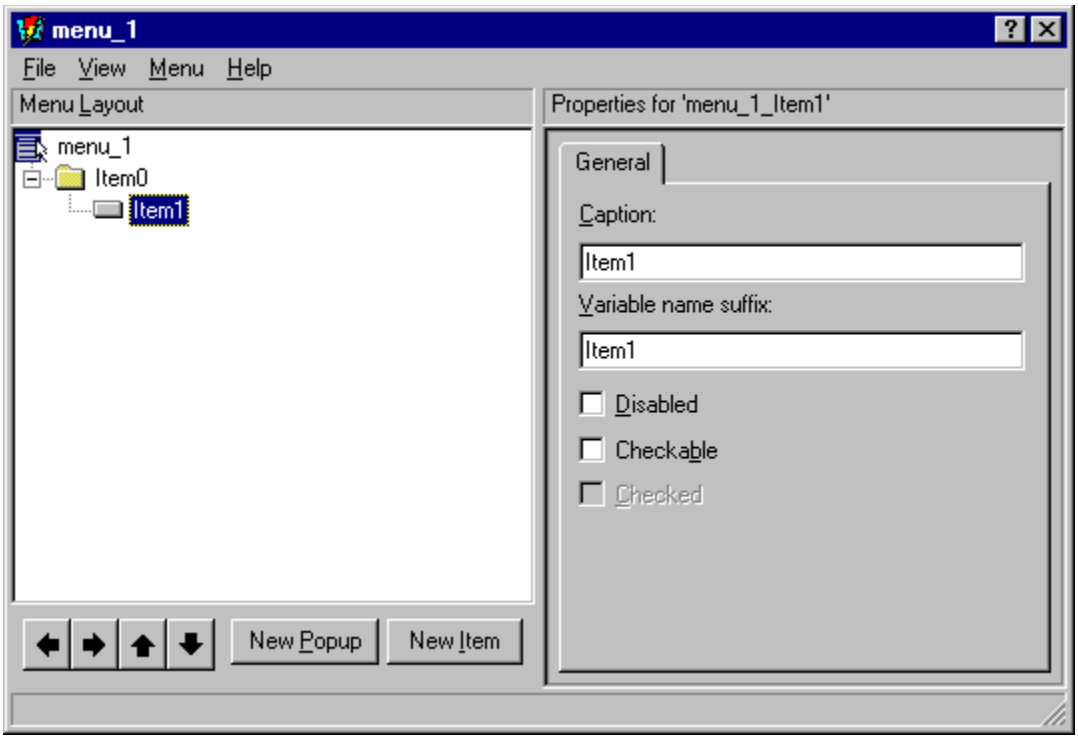
As shown in the property sheet on the right side of the menu editor, the default variable name suffix is the same as the caption.

The process of adding a new menu item to a popup menu is similar to the popup:

- **To add a new menu item:**

1. In the menu layout of the menu editor, click the popup menu that should contain the new item.
2. In the menu called **Menu**, click **New Child Item**. This adds a new menu item to the popup you selected.

The new item appears in the menu layout as shown:



Note that the **New Item** button is now available. You can use it to add additional menu items at the same level as the selected item; it is equivalent to **New Menu Item** under **Menu**. To summarize:

- Under **Menu, New Child Item** adds a menu item that is a child of the current selection.
- The **New Item** button and **New Menu Item** (under **Menu)** add a menu item that is a peer of the current selection.

Once you have added a menu or menu item, you can set its caption and the suffix that is used for the associated variable.

◆ **To change the caption and suffix for a menu or menu item:**

1. In the menu layout on the left side of the menu editor, click the item whose caption and/or suffix you want to change.
2. Under **Caption**, type a new caption for the item. The suffix automatically changes to match the caption.
3. If you want to change the suffix, type a new suffix under **Variable name suffix**.

When you change the caption on the right side of the menu editor, the caption also changes on the left side of the editor.

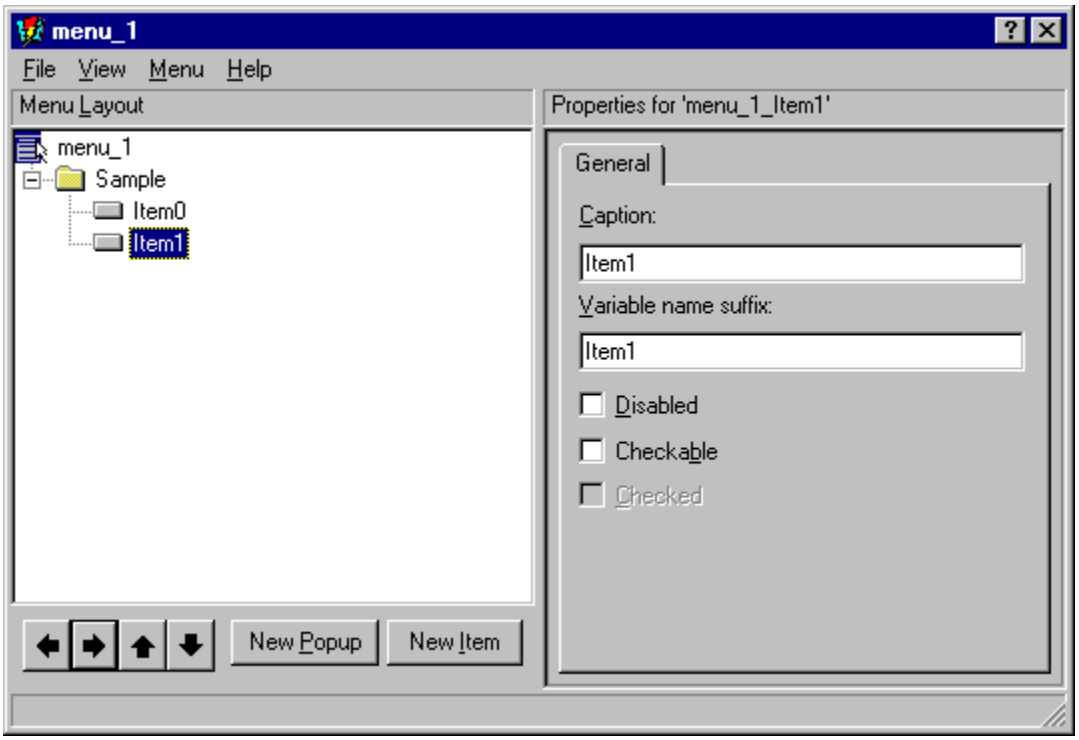
◆ **To delete a menu or menu item:**

1. Use the right mouse button to click the item you want to delete, then click **Delete**.

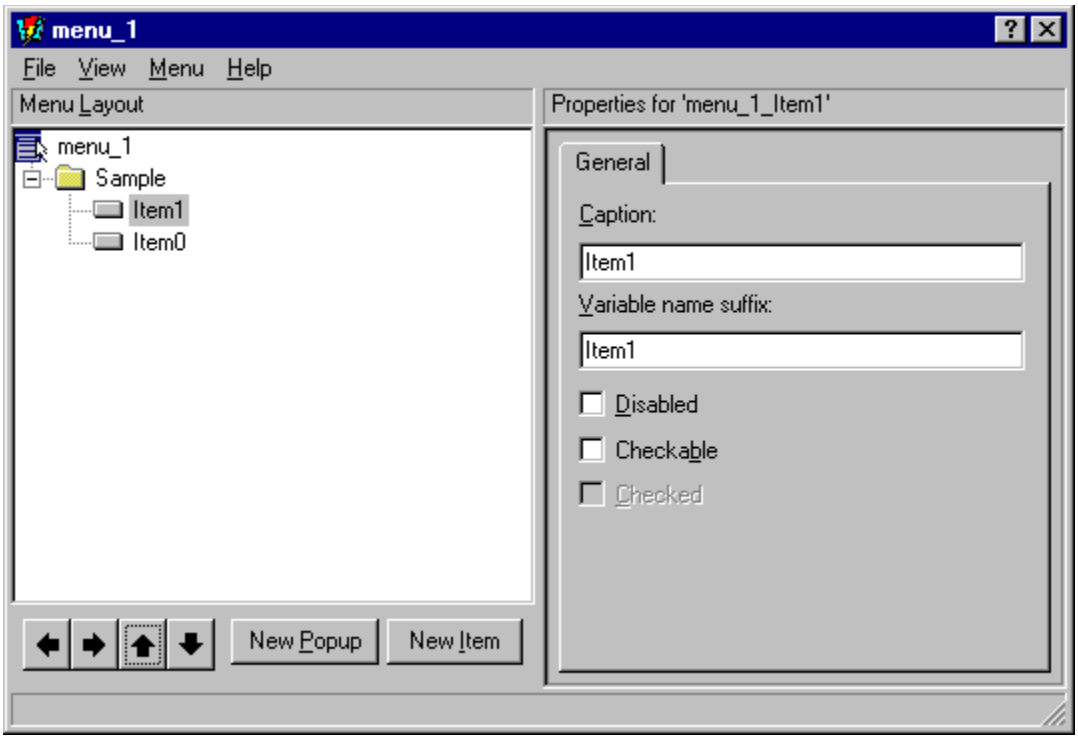
Specifying the menu structure

The arrow buttons at the bottom left of the menu editor let you rearrange the order of items and menus.

For example, suppose you have a popup menu containing two items: `Item0` and `Item1`:



If you click `Item1` in the menu layout, then click the button with the arrow pointing up, the item moves up in the layout:



As the diagram shows, the chosen item moved up in the menu layout. By clicking an item and then using the arrow buttons, you can move the item around the menu structure.

- **To move an item from one menu to another:**

1. In the menu layout of the menu editor, click on the item you want to move, then drag and drop it on the popup menu where you want to place the item. (There is no effect if you drop the item on another item–you must drop it on a popup menu.)

As well as menu items, you can also drag and drop menus in the left pane of the menu editor.

Menu and menu item properties

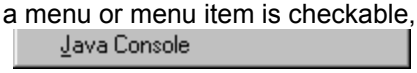
Menus and menu items may have the following properties (specified by clicking the appropriate check boxes in the menu editor):

Disabled

If a menu or menu item is disabled, it is visible to the user but cannot be used. Typically, it is shown "grayed out" rather than in its normal color.

Checkable

If a menu or menu item is checkable, it is possible to display a check mark beside the item, as in:



Checked

This property is only enabled when **Checkable** is enabled. If a menu or menu item is checked, it is marked with a check mark when the menu is first displayed. If the item is not checked, it is not marked with a check mark to begin with; however, you can change the mark during execution using the **setState** method.

Separators

A separator is a bar across a menu, used to separate menu items into groups. For example, the following menu has two separators:

- **To create a separator in a menu:**

1. In the menu layout of the menu editor, click the item that comes immediately before the separator.
2. Under **Menu** in the menu editor, click **New Separator**.

You can change the position of separators using the arrow buttons at the bottom of the menu editor window.

Separators do not have captions. They are assigned default suffixes which cannot be changed.

Menu events

There is only one event commonly defined for menu and menu items:

Action (JDK 1.02) or **ActionPerformed** (JDK 1.1)
This event takes place when the user clicks the menu or menu item.

- **To create an Action event handler for a menu item:**

1. Create your menus and menu items using the menu editor.
2. In the menu hierarchy of the menu editor, use the right mouse button to click on a menu item. Then click **Events** and choose the **Action** event from the list.

This opens a code editor window where you can specify the code for your **Action** event handler.

Menu item methods

This section examines various methods that can be executed on MenuItem objects. Since the Menu class is based on MenuItem, these methods can also be applied to menus and submenus, as well as simple menu items.

To use menu and menu item methods, you will probably want to include the following import statements in your code instead of using fully qualified names:

```
import java.awt.Menu;
import java.awt.MenuItem;
```

Checking an item

If a menu item is declared checkable at design time, the **State** property determines whether a check mark is currently displayed:

```
menu_1_Item1.setState( true );       // check mark
menu_1_Item1.setState( false );      // no check mark
```

There is a corresponding **getState** method to determine if an item is currently checked.

Enabling an item

When a menu item is enabled, the item can be clicked. When an item is disabled, it is "grayed out" and cannot be clicked. You can control this state in code.

Java 1.02: Use the **enable** method to specify whether an item is enabled:

```
menu_1_Item1.enable( true );          // enabled
menu_1_Item1.enable( false );         // disabled
```

Java 1.1: The **Enabled** property determines whether an item is enabled:

```
menu_1_Item1.setEnabled( true );     // enabled
menu_1_Item1.setEnabled( false );    // disabled
```

There is a corresponding **getEnabled** method to determine if an item is currently enabled.

Changing the caption

The **Label** property specifies the caption for an item. For example:

```
menu_1_Item1.setLabel( "New caption" );
```

This changes the caption of the item.

There is a corresponding **getLabel** method to obtain the current caption of the item.

The font of a menu item

The **Font** property can be set at run time to specify the font of the menu item's caption. For example:

```
// Font f;
menu_1_Item1.setFont( f );
```

This sets the font of the given menu item.

The parent of a menu item

The **Parent** property specifies the parent of a menu item. For example:

```
MenuContainer mc = menu_1_Item1.getParent();
```

This determines the parent of the given item.

Menu container methods

This section examines methods that can be performed on menu bar, menu and submenu objects (objects that contain menu item objects).

To use menu container methods, you will probably want to include the following import statements in your code instead of using fully qualified names:

```
import java.awt.Menu;
import java.awt.MenuItem;
```

Counting children

You can count the number of children directly contained by the menu object. The following code determines the number of menus that appear on the menu bar.

Java 1.02: Use the **countMenus** method:

```
int num = menu_1.countMenus();
```

Java 1.1: Use the **MenuCount** property:

```
int num = menu_1.getMenuCount();
```

This does not count the number of items that are included in those menus.

The follow code counts the number of items in the given submenu, but does not include items in submenus of the submenu.

Java 1.02: Use the **countItems** method:

```
int num = menu_1_Submenu.countItems();
```

Java 1.1: Use the **ItemCount** property:

```
int num = menu_1_Submenu.getItemCount();
```

Obtaining children by index

The **Menu** property of the MenuBar class obtains a Menu object corresponding to one of the menus on the menu bar. Menus are numbered from left to right, beginning at zero. For example:

```
Menu m = menu_1.getMenu( 0 );
```

This returns a Menu object representing the first menu in the menu bar.

Similarly, the **MenuItem** property of a Menu object obtains a MenuItem object corresponding to one of the menu items in the menu. For example:

```
MenuItem mi = menu_1_Submenu.getItem( 0 );
```

This returns a MenuItem object representing the first item in the submenu.

Removing an item from a menu

The remove method removes a menu or menu item from a Menu or MenuBar object. You can specify the item to be removed either by an integer index or by specifying the object itself. For example:

```
menu_1.remove( 0 );
```

This removes the first menu from the menu bar.

```
menu_1_Submenu.remove( menu_1_Item1 );
```

This removes the specified component.

Adding an item to a menu

The add method adds a menu or menu item to a Menu or MenuBar object. For example, suppose you have previously removed the object `menu_1_Item1` from a submenu; then

```
MenuItem mi = menu_1_Submenu.add( menu_1_Item1 );
```

puts the item back on the menu again.

If you want to change the contents of a menu during run time by adding or removing items, it is best to specify all possible items at design time. If you don't want some of these to appear when the program begins executing, you can remove the unwanted items in the **objectCreated** event handler for the form. You can then add these items back again if they are needed during execution.

If you do not specify a menu item at design time, it is more complicated to "build the item from scratch" at run time. For example, it is not enough to add the newly created item to a menu; you must also specify an **Action** or **ActionPerformed** event handler for the item. This is why it is easier to set everything up at design time (including the event handler) and then remove them temporarily than to try building items from scratch.

Chapter 13. Using forms

This chapter discusses a number of topics related to PowerJ forms: opening and closing forms, form-related events, and properties which are unique to forms. In particular, it discusses the FDX facilities for exchanging information between forms.

Defining forms

- **To define a new form in a target:**

1. In the **File** menu of the main PowerJ menu bar, click **New**, then **Form**. This opens the Form Wizard.
2. Under **What type of form do you want?**, click the type of form you wish to create. Possible form types are discussed later in this section.
3. Click **Next**.
4. Under **What do you want to call the new form?**, type a name for the form.

   The Form Wizard uses the name you typed for the form as a name for the file, adding the extension `.WXF`. The file name appears in a non-editable field in the Form Wizard and will be `<Form name>.WXF`.
5. Click **Finish**.

If you don't specify a name for the form, PowerJ uses a default name of *FormN* (where *N* is a number).

The standard form types are:

Panel
: The form is based on the Panel class. This is the simplest type of form.

Frame
: The form is based on the Frame class. This is the only type of form that can have a menu bar.

Modal Dialog
: The form is based on the Dialog class. While a modal dialog form is open, the user cannot switch back to any parent forms. Therefore, modal dialogs are useful when you need to obtain information from the user and cannot continue processing until the information is supplied.

Modeless Dialog
: The form is based on the Dialog class. Modeless dialogs do not prevent the user from switching to parent forms. Therefore, modeless dialogs are useful when you want to let the user switch between this form and other forms.

Database forms
: These forms are intended for database applications. There is a database form corresponding to most of the other form types (Database Frame, Database Modal Dialog, and Database Modeless Dialog). The Form Wizard goes through the steps required to create the basic form type, then goes on to gather information about the database connection that this form will use, the SQL query the form will make, and the way in which the query results will be displayed. For further information, see Database dialog forms.

Customizer
: This is a template for creating customizers for JavaBeans components. For more information on customizers, see Customizers.

Property Editor Panel
: This is a template for creating property editors for JavaBeans components. For more information on property editors, see Property editors.

If you have defined any form templates, they will also appear in the Form Wizard's list. For more information, see Form templates.

Important: Frames are the only type of form that can have a menu bar. PowerJ issues a diagnostic message if you try to place a MenuBar object on any other type of form.

Opening a new form

The usual way of opening a new form is to construct an object of the form type, and then apply the **create** method to it.

Constructing a form object

JDK 1.02 and JDK 1.1 use different techniques for constructing a form object.

JDK 1.02: You construct any form object without arguments, as in

```
Form2 f2 = new Form2();
```

JDK 1.1: Under JDK 1.1, you construct dialog forms differently than applets and frames. With applets and frames, you do not have to specify any arguments:

```
Form2 f2 = new Form2();  // applet or frame
```

When you construct a dialog object, however, you must specify a frame that will be considered the *parent* of the dialog. The parent must be a frame. The dialog is considered to be the *child* of the frame.

For example, suppose that Form1 is constructing a Form2 object, and that Form1 is a frame that will be the parent of the new form. You would code this as follows:

```
Form2 f2 = new Form2( this );
```

The argument specifies the parent for the dialog. Since Form1 will be the parent, it refers to itself as `this`.

Creating a form

Once you have constructed a form object, you can open the form by applying the **create** method to the object. For example, an object of the type Form1 could open one of the type Form2 with:

```
Form2 f2 = new Form2();
try {
    f2.create();
} catch (java.lang.Exception e) {
    // do something if open operation fails
};
```

As shown above, the **create** method may throw a `java.lang.Exception` if the new form cannot be opened. Therefore, calls to **create** must be enclosed in a `try/catch` construct.

If you need to initialize the new form before displaying it, you can add initialization code between the instruction that constructs the form object and the call to **create**. For example, suppose that Form2 contains a public String object named `str`. The following code initializes this string before displaying the form:

```
Form2 f2 = new Form2();
f2.str = "Start string";
try {
    f2.create();
    // and so on
```

In this way, Form1 can pass data to Form2 using public data members. However, it is usually better to use FDX to exchange data between forms. For more information on FDX, see FDX.

The objectCreated event for a form

When a form is created, it receives an **objectCreated** event. You can write your own **objectCreated** event handler to perform initialization operations on the form or on the objects that the form contains. For more information, see Object initialization.

Closing a form

The typical sequence for closing a form is to turn the form invisible and then call the **destroy** method (which deallocates the form and all the objects it contains). Under JDK 1.02, you do this with:

```
hide();
destroy();
```

Under JDK 1.1, you use the **setVisible** method instead of **hide**:

```
setVisible( false );
destroy();
```

The objectDestroyed event for a form

Just before a form is destroyed, it receives an **objectDestroyed** event. You can write your own **objectDestroyed** event handler to perform clean-up operations on the form or on the objects that the form contains.

Layout managers

A *layout manager* is a Java class that controls the appearance of a PowerJ form, both at run time and at design time. PowerJ offers the following layout managers:

`powersoft.powerj.ui.ResizePercentLayout`
: The layout manager designed to support **ResizePercentages** property. For further information, see Resize percentages property.

`java.awt.BorderLayout`
: A layout manager that lays out objects on the form using the directions "North", "South", "East", "West", and "Center". For example, if you add an object to the "South" of the form, the object is positioned on the current bottom border of the form.

`java.awt.CardLayout`
: A layout manager for a form that contains one or more "cards". Each component added to a form using this layout is treated as a separate card; only the topmost card is visible at design time. In other words, a card layout form is similar to a tab control, and each component on the form is like a page in the tab control.

`java.awt.FlowLayout`
: A layout manager designed to lay out rows of buttons. It arranges buttons from left to right until reaching the right edge of the form; then it begins a new row of buttons.

`java.awt.GridBagLayout`
: A layout manager that lays out objects using a grid of available cells. Each object may cover one or more adjacent cells.

  If you convert to GridBagLayout from some other layout manager, PowerJ generates code that attempts to approximate the form's previous design-time layout using GridBagLayout. To do this, PowerJ divides the form into grid cells based on the minimum width and height of all components, and places the components in the appropriate grid cell(s). PowerJ tries to use the `ipadx`, `ipady` and `insets` members of the `GridBagConstraints` to do fine-tuning adjustments.

  This approach creates a rough approximation of the original layout, with components in the same relative positions. However, the layout does not match the original exactly. Some components do not work well with `GridBagLayout`, giving preferred sizes that bear little resemblance to their original size. Text areas and grid controls have the most noticeable sizing problems.

`java.awt.GridLayout`
: Another layout manager that lays out objects in a grid. When you set up the grid, you specify the number of rows and columns. If you add more components than there are positions in the grid, the number of rows stays the same but the number of columns increases to accommodate the extra components.

For more information on the `java.awt` layout managers, see the standard references on the AWT kit. For more information on using card layout with PowerJ, see Using card layout with PowerJ. For more information on the `ResizePercentLayout` manager, see the *PowerJ Component Library Reference* and Resize percentages property.

Note: If you choose `powersoft.powerj.ui.ResizePercentLayout`, your applet will require classes from the `powersoft.powerj.ui` package.

Specifying layout manager properties
Using card layout with PowerJ

Specifying layout manager properties

You specify the layout manager for a form using the **Layout Manager** page of the form's property sheet. At the top of the page is a choice box where you can choose one of the supported layout managers. You can also choose `null`, in which case you get no layout manager.

Once you have chosen the layout manager at the top of the **Layout Manager** page, the bottom part of the page may display additional properties specific to the layout manager you have chosen.

The layout manager that you choose for a form also affects the properties of the objects on the form. Each object's property sheet has a **Layout Constraints** page where you can control properties which are related to the current layout manager. For example, if the form is currently using the `powersoft.powerj.ui.ResizePercentLayout` layout manager, the **Layout Constraints** page for each object will let you specify the resize percentage values for that object.

Note: Different layout managers have different properties associated with the form and with objects on the form. If you are using one of the standard AWT layout managers, see the standard AWT references for information on the corresponding properties. If you are using the `ResizePercentLayout` layout manager, see Resize percentages property.

Using card layout with PowerJ

At design time, a form that uses card layout (`java.awt.CardLayout`) only shows the topmost component on the form. This makes it difficult to manipulate other components. You can make your work easier by following these steps:

1. By using the context menu of the Objects window, you can place new components into containers with the `CardLayout` manager.
2. Clicking on a component in the Object Inspector or the Objects window brings the component to the top if the component's container has a card layout manager.

With card layout, you will find that you use the Objects window more often than the usual form design window.

If the cards that you use in a card layout are components, you need to explicitly hide them before the cards are flipped. You can avoid this requirement by using panels for cards, since a panel will be hidden by default. If you use TabControl components as cards, you will need to explicitly call the `hide` method for them.

Card layouts and panels

For some applications, you may wish to use Panel forms as the "cards" in a card layout. It is much easier if the cards are Panel components (from the component palette). In the case of cards that are top-level Panel forms, you must explicitly set the parent of each Panel to be the card container (for example, a panel containing the other panels). The parent must be set before you call **create** to initialize the panels.

For example, suppose you have an applet form named MyApp and you want this to contain an area that has five cards associated it, with each card implemented as a panel object. To do this, you can follow these steps:

1. Place a Panel object on MyApp in the region where you want the multiple "cards" to be stacked. Name this panel `cards`.
2. Create five new panel forms named `panel1`, `panel2`, `panel3`, and so on.
3. In each panel constructor, comment out the code that calls **create**. An example is shown below:

```
public panel1()
{
    super();

  /* Commented out from here ****
    try {
        create();
    }
    catch( java.lang.Exception __e) {
        System.err.println( __e.toString() + " " +
             __e.getMessage() );
    }

  **** down to here */
}
```

4. In the form class, define the following data members:

```
private java.awt.CardLayout cardlayout;
private panel1 panel_1 = new panel1();
private panel2 panel_2 = new panel2();
private panel3 panel_3 = new panel3();
private panel4 panel_4 = new panel4();
private panel5 panel_5 = new panel5();
```

5. Define an **objectCreated** event for the `cards` panel containing code of the following form:

```
cards.add( "panel_1", panel_1 );
cards.add( "panel_2", panel_2 );
cards.add( "panel_3", panel_3 );
cards.add( "panel_4", panel_4 );
cards.add( "panel_5", panel_5 );

try {
    panel_1.create();
    panel_1.hide();
} catch (Exception __e) {};

// similar code for other panels

cardlayout = (java.awt.CardLayout) cards.getLayout();
cardlayout.show( cards, "panel_1" );
```

Note that the above example changes the constructor for each panel so that the constructor does *not* call **create**. This is crucial, because you must set the parent of each panel before the panels are created. Step 5 above sets the parent for each panel by using **add** to add them to the `cards` panel. Once this has been done, you can call **create** on each panel. To begin with, you hide each panel; at the end of Step 5, you use **show** on the card layout to show the first panel. You can display a different panel with code like

```
cardlayout.show( cards, "panel_2" );
```

Resize percentages property

The **ResizePercentages** design-time property controls how objects on a form behave when the size of the form changes. This property is specified on the **Layout Constraints** page of each object's property sheet and as the **ResizePercentages** item in the Object Inspector, provided that the form uses the `ResizePercentLayout` layout manager.

Note: The **ResizePercentages** property requires you to use the `ResizePercentLayout` layout manager. For more information on layout managers, see Layout managers.

The **ResizePercentages** property is of the type `java.awt.Rectangle`. You can specify the four parts of the rectangle on the **Layout Constraints** property sheet under **Resize Percentages**:

Left
Controls how far the left edge of the object moves when the form's size changes.

Top
Controls how far the top of the object moves when the form's size changes.

Width
Controls how much the width of the object changes when the form's size changes.

Height
Controls how much the height of the object changes when the form's size changes.

Each of these is expressed as a percentage. To see what effect these percentages have, suppose that the **Width** percentage of a text box is 30 and that the user increases the width of the form by 100 pixels. Then the width of the text box automatically increases by 30% of 100 pixels, or 30 pixels. Similarly, if the user decreases the width of the form by 50 pixels, the width of the text box decreases by 30% of 50 pixels, or 15 pixels. In mathematical terms:

```
newWidth = oldWidth + widthPercentage*formWidthChange;
```

The same relationship holds for all the other **ResizePercent** values.

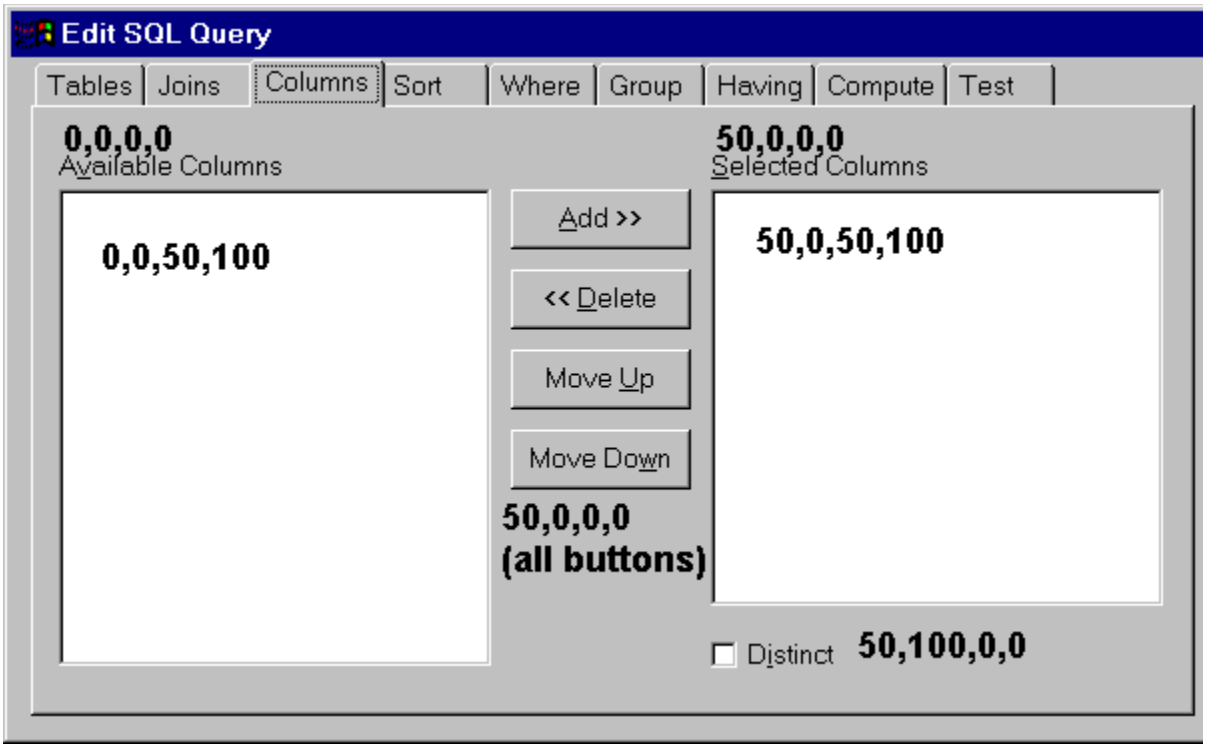
At run time, you can use **getResizePercent** to determine the current resize percentages and **setResizePercent** to change the values. **ResizePercent** is expressed as a Rectangle value, as in:

```
Rectangle r = ((ResizePercentLayout)
    getLayout()).getResizePercent( lb_1 );
```

The `x` and `y` values of this rectangle correspond to the left and top resize percentages, and the `width` and `height` values of this rectangle correspond to the width and height resize percentages.

An example of resize percentages

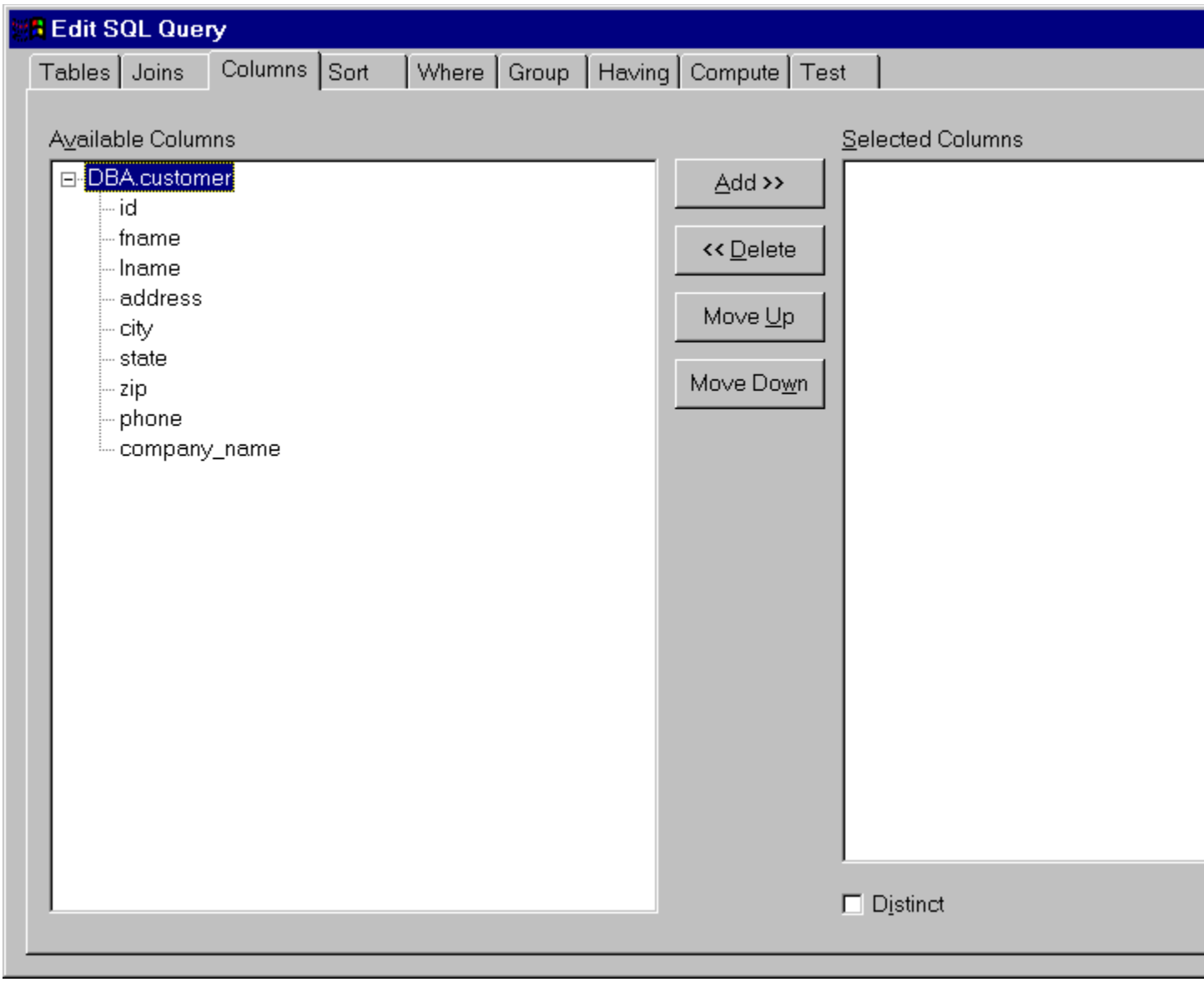
As an example of how this works in practice, the following picture shows a sample form with resize percentages displayed in bold face (in the order, **Left**, **Top**, **Width**, **Height**):



Notice, for example, that the label **Available Columns** has percentages `(0,0,0,0)`; this means that the size of the label does not change, and the position of the label does not change (so that it remains close to the left side of the form). On the other hand, the label **Selected Columns** has percentages `(50,0,0,0)`: the value of `50` for the **Left** percentage means that if the form width increases or decreases, the label moves left or right by half the amount of the increase (so that the label remains approximately in the middle of the form).

The main area of the sample form is occupied by a custom-designed tree view on the left and a list on the right. The tree view under the **Available Columns** label has percentages `(0,0,50,100)`: if, for example, the height and width of the form both double, the width of this tree view increases by half (50%) of the total width increase, while the height of the tree view increases by all (100%) of the total height increase. Similarly, the list under the **Selected Columns** label also has its width increase by 50% of the total width increase, while the height of the list increases by 100% of the total height increase. As a result, any change in width is split half-and-half by the tree view and the list, and any change in height makes the same change in the height of the tree view and the list. The other objects on the form do not change size.

The following picture shows the result of an increase in size for the original form:



Notice that the size of the labels, the command buttons, and the checkbox have not changed. Only the tree view and the list increase in size.

Proportional resizing

If you set any of the **ResizePercent** properties to `-1`, the corresponding dimension changes in strict proportion to the size of the form. For example, suppose you set the **Width** percentage of a list to `-1`. If the width of the form doubles, the width of the list will also double.

Obviously, the easiest way to deal with resizing is to set every object's **ResizePercent** properties to `-1`. However, this may not serve the user as well as setting different percentages for different objects. As mentioned earlier, changing the size of command buttons, labels, and similar objects usually doesn't give the user any benefit; it is often better to keep such items the same size, no matter how the form changes.

Resize percentages at run time

At run time, you can set the resize percentage values for an object using the **setResizePercent** method of the form that contains the object. For example, suppose you want to change the resize percentages of `lb_1` on the current form. You can do this with the following code:

```
((ResizePercentLayout)getLayout()).setResizePercent( lb_1,
     new Rectangle( 0, 0, 100, 100 ) );
```

The **getLayout** method gets the layout manager for the current form and this is cast into a ResizePercentLayout object. The **setResizePercent** method for this object changes the resize percentages for `lb_1` into the values specified by the Rectangle argument. The rectangle argument gives resize percentages in the following order:

```
Rectangle( left, top, width, height )
```

If your code performs an operation of this form, the code should include the following `import` statement:

```
import powersoft.powerj.ui.ResizePercentLayout;
```

FDX

The Form Data Exchange, or FDX, facilities of PowerJ make it possible for a form object to generate a block of information that can be shared with other forms. FDX works by defining a data structure which summarizes all the information that a form wishes to share with other forms.

For example, suppose that FormMain opens FormInfo to obtain information from the user. Also suppose that the information FormInfo gathers should be passed back to FormMain when the user clicks a button named **Close** on FormInfo.

- **To pass information from FormMain to FormInfo and back, using FDX:**

1. During the design phase, define an FDX class type for FormInfo. This describes the information that FormInfo will make available to FormMain.
2. During execution, FormMain creates an object of the FDX class type.
3. FormMain initializes the data members of the object with any values it wants to pass to FormInfo.
4. FormMain constructs a FormInfo object.
5. FormMain assigns the FDX object to the **FDX** property of the new FormInfo object.
6. FormMain executes `FormInfo.readFDX()` which initializes objects on FormInfo from the FDX object.
7. FormMain opens the initialized FormInfo form.
8. Before FormInfo terminates, it updates the FDX object received from FormMain, filling in any new information received by the user.
9. When control passes back to FormMain, it can extract the desired information from the FDX object.

The rest of this section explains how PowerJ makes this process easy for you.

Note: FDX exchange objects can only be defined for frames, and modal or modeless dialog forms. They cannot be defined for applet forms.

Naming the FDX class

- **To begin using FDX:**

1. Finish adding components to your form and specify their properties.
2. On the **FDX** page of the form's property sheet, click **Enable Data Exchange** (so that it is checked).

PowerJ suggests a name for the class that will summarize FDX data from this form. You would usually change this to a more meaningful name.

In this release of PowerJ, you must define your own FDX class; PowerJ cannot generate the class for you. For more information, see Designing an FDX class.

Values in the FDX class

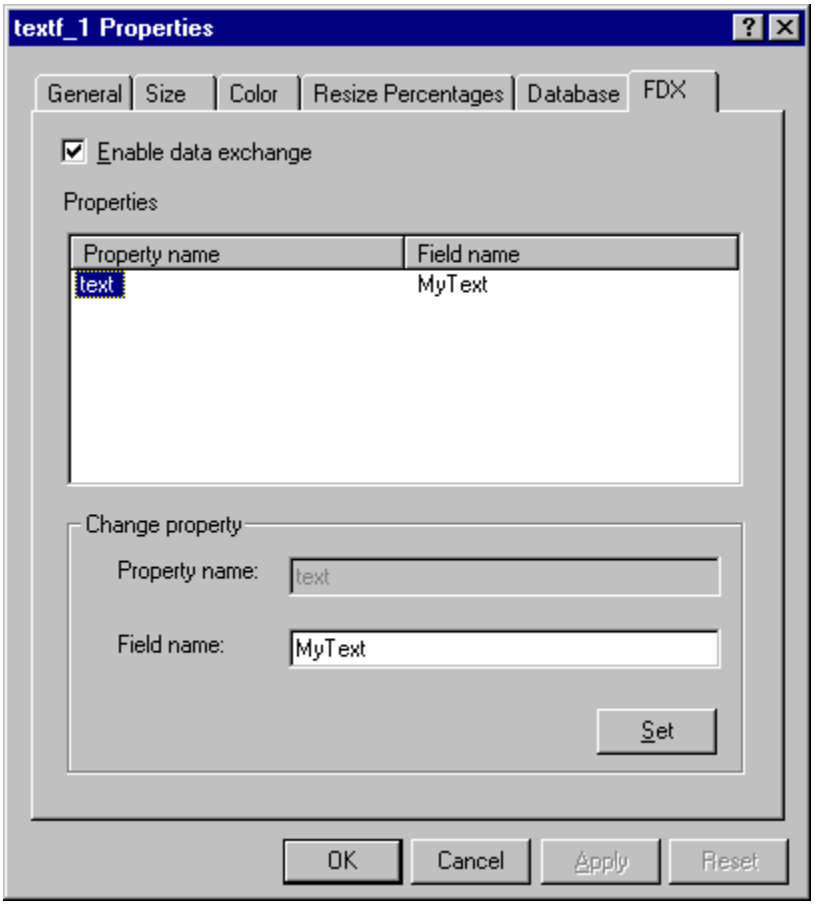
The FDX class contains data members which correspond to specific property values of objects on a form. For example, suppose FormInfo contains a text field, and you want the **Text** of this text field to be available to other forms. The FDX class should contain a String data member where the **Text** of the text field can be stored. You might call this data member `MyText` or some name that describes the meaning of this string.

Using the property sheet for an object, you specify which properties of the object correspond to which data members in the FDX class.

- **To specify that an object's property value is stored in a data member of the FDX class:**

1. On the **FDX** page of the object's property sheet, click **Enable Data Exchange** (so that it is checked).
2. Under **Properties**, click the name of property whose value you want to make available to other forms.
3. Under **Field name** at the bottom of the property sheet, type the name of the data member that will hold this property value.
4. Click **Set**, then **OK**.

For example, here is the **FDX** page in the property sheet for a text field:



This says that the **Text** property value of `textf_1` corresponds to a data member named `MyText` in the FDX class. Since the **Text** property has a String value, the `MyText` data member will also be a String. If you were defining your own FDX class, you would add the data member

```
public String MyText;
```

to the FDX class definition.

Properties that can be shared with FDX

The **FDX** page of an object's property sheet shows which property values can be shared using FDX.

- With many objects, there is only one property which can be shared. For example, a check box can only share its state (checked or unchecked).
- Some objects, however, can share more than one property. For example, a list can share the text of one or more strings; it can also share the index number of the string that is currently selected.

The descriptions below list what property values may be shared in different types of objects.

Check boxes
: A Checkbox object may share its **State** property as a boolean value: `true` means the check box is checked and `false` means the check box is not checked.

Scroll bars
: A Scrollbar may share its **Value** property as an int value.

Text boxes
: TextField, TextArea and MaskedTextField objects may share their **Text** property as a String value.

Designing an FDX class

This section describes how to design a class for sharing FDX data between forms.

- **To define an FDX class:**

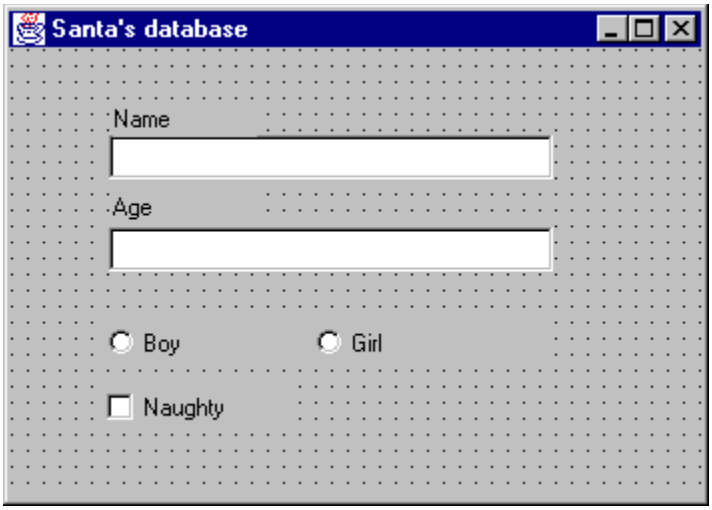
1. In the **File** menu of the main PowerJ menu bar, click **New** and then click **Class**. This opens the Class wizard.
2. Under **What type of class do you want?** click **Standard Java**, then click **Next**.
3. Under **Class name**, type the class name that you specified on the **FDX** page of the form's property sheet.
4. Click **Finish**.

PowerJ opens a code editor window where you can enter code for the new class. After the comment `//add your data members here`, type in declarations for `public` data members, corresponding to the values that the corresponding form will share with other forms.

For example, suppose that FormInfo will share data with other forms using an FDX class called FormInfoFDX. Furthermore, suppose that `textf_1` on FormInfo should share its **Text** property using a data member named `myText`. Then the definition for FormInfoFDX should include the declaration

```
public String myText;
```

As an example, the following picture shows a `Form2` window that can be involved in FDX exchanges with other forms.



You might create the following class to represent information on this form:

```
class Form2FDXClass {
    public String  name;        // Name text field
    public String  age;         // Age text field
    public boolean boy;         // Boy check box
    public boolean girl;        // Girl check box
    public boolean naughty;     // Naughty check box
};
```

Associating an FDX class object with a form

The **FDX** property of a form associates an FDX class object with a form object. As a simple example, suppose FormMain is going to open a FormInfo object, and suppose that FormInfo's FDX class is named FormInfoFDX. FormMain could do this with the following code:

```
FormInfo fi       = new FormInfo();
FormInfoFDX fiFDX = new FormInfoFDX();
// code here to initialize members of fiFDX
fi.setFDX( fiFDX );
try {
    fi.create();
} catch (java.lang.Exception e) {
    // do something
};
```

This code constructs new FormInfo and FormInfoFDX objects. It initializes the data members of the FormInfoFDX object, and then uses **setFDX** to associate the FormInfoFDX object with the FormInfo object. The final step is to call **create** to open the new form.

Since the above code appears in FormMain, FormMain may need the statements

```
import FormInfo;
import FormInfoFDX;
```

to import the definitions of the two classes used.

There is a second way to associate an FDX class object with a form. This makes use of a form of **create** that takes an FDX class object as its argument. We could rewrite the above example as follows:

```
FormInfo fi       = new FormInfo();
FormInfoFDX fiFDX = new FormInfoFDX();
// code here to initialize members of fiFDX
try {
    fi.create( fiFDX );
} catch (java.lang.Exception e) {
    // do something
};
```

In this code, the call to **create** passes the FDX class object. Therefore, there is no need to call **setFDX** explicitly.

Initializing control information

The **readFDX** method is defined for any form class where FDX is enabled. It reads information from a form's associated FDX object and initializes objects on the form using that information.

For example, suppose that the text of a text field corresponds to a data member named `myText` in the FDX object associated with the form. Then **readFDX** copies the current value of `myText` from the FDX object to the text field. Here's an artificial example:

```
FormInfo fi       = new FormInfo();
FormInfoFDX fiFDX = new FormInfoFDX();
// code here to initialize members of fiFDX
fi.setFDX( fiFDX );
fi.readFDX();
try {
    fi.create();
    // and so on
```

This example is similar to the one in the previous section, with the addition of the call to **readFDX**. The **readFDX** method initializes objects on the FormInfo form with values taken from the corresponding data members in the FDX object.

The above was called an artificial example because the **create** method automatically calls **readFDX** when the form is created. Therefore, you do not need to call **readFDX** explicitly, unless you want to "refresh" property values by reading them from the original FDX class object.

Updating control information

The **writeFDX** method is defined for any form class where FDX is enabled:

```
writeFDX();
```

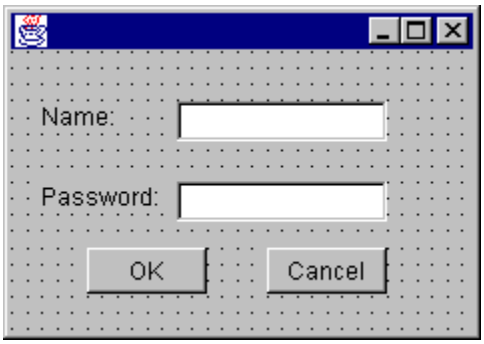
The function copies information from the objects on a form into the form's FDX class object. This means that **writeFDX** is the opposite of **readFDX**.

Typically, you would call **writeFDX** in the process of closing a form. This copies information from the current form back to the caller, so that the caller has access to the information.

There may be times when you call **writeFDX** before closing a form. As an example, suppose a form opens a tab control for use as a property sheet. Typically, you would call **writeFDX** in the **Action** event handler for the **Apply** button, on the assumption that the user wants to "record" the current information specified by the tab control.

A data exchange example

Assume that FormInfo looks like this:



This contains two text fields: `nameField` and `passwordField` (where the user can enter an identifying name and password). This form has an associated FDX class named FormInfoFDX with the following definitions:

```
public String name;
public String password;
```

FormInfo also has two buttons:

- `okButton` that users click to indicate that they have entered the desired information on the form.
- `cancelButton` that users click when they decide not to enter any information.

FormInfo will be opened from FormMain when the user clicks a button named `cb_1` on FormMain. The following shows code in the definition of FormMain related to FormInfo:

```
import MyPackage.FormInfo;
import MyPackage.FormInfoFDX;

// FormMain data members
FormInfo    fi;
FormInfoFDX fiFDX;

boolean cb_1_Action( java.awt.Event event )
{
    fi    = new FormInfo();
    fiFDX = new FormInfoFDX();
    fiFDX.name = "guest";   // default name
    fiFDX.password = "";    // default password
    try {
        fi.create( fiFDX );
    } catch (java.lang.Exception e) {
        // ignore the exception
    };
    return false;
}
```

The code for FormInfo is quite simple. It consists of two event handlers: one for the **OK** button and one for the **Cancel** button.

```
boolean okButton_Action( java.awt.Event event )
{
    hide();
```

```
    writeFDX();
    destroy();
    return false;
}

boolean cancelButton_Action( java.awt.Event event )
{
    hide();
    destroy();
    return false;
}
```

When the user clicks **OK**, the user is saying that the information on the form is complete. Therefore, the event handler calls **writeFDX** to copy information back to the FDX structure. In this way, the FDX structure passes information from FormInfo back to the first form.

The **Action** handler for the **Cancel** button does not call **writeFDX**. Since the user has canceled the operation, the user has decided not to make any changes; therefore, any (changed) information on this form should not be passed back to the caller.

Note: For a more complete example of FDX in action, see the PowerJ samples.

FDX and tab controls

As explained in Tab controls, a tab control object has a number of associated tab forms: one for each tab in the tab control. All of the tab forms can share the same FDX structure as the parent form (the form that contains the tab control object).

- **To set up tab forms to share an FDX structure with the parent form:**

1. Create a managed class which defines the FDX structure you want to use. (For more information about creating managed classes, see Adding classes to a target.)
2. For all tab forms and the parent form: enable FDX, type the name of your FDX structure in **Class name**, and click **Use a class you have defined** so that it is checked.

Once you have done this, the tab forms and the parent form will all share the FDX structure you defined. For example, if another form opens the parent form and passes an FDX structure, the tab forms are automatically initialized from that structure at the same time that the parent form is initialized. This greatly simplifies the process of initializing the tab forms.

Similarly, when the parent form closes, the contents of the FDX structure can be passed back to the invoking form. The tab forms can fill in the appropriate entries of the structure before terminating. The parent form can do the same thing, thereby passing back values from the tab forms and the parent form in a single block.

Implementing interfaces in forms

If you want to use certain types of components, your form must declare itself to implement a particular interface. In particular, if you want to use the JCOutliner component, your form class must declare itself to implement the JCOutlinerListener interface.

- **To specify that a form implements a particular interface:**

1. Open the Classes window.
2. Use the right mouse button to click on the name of the form, then click **Properties**. This opens a window describing the form class.
3. Under **Implements**, type the name of the interface that your form class implements. If there are several interfaces, they should be separated with commas, as in

   ```
   interface1, interface2, interface3
   ```

4. Click **OK**.

The above steps change the declaration of the form to say that it implements the given interface. Typically, interfaces require that you define a set of methods within the class. Therefore, to make your form into a proper implementation of the interface, you must define the desired methods within the form class. For a description of how to add a function to a form class, see Adding new member functions.

Chapter 14. Using graphics

This chapter discusses the use of graphics in a program: how to create graphics and display them on a form.

The Graphics class

The AWT Graphics class (`java.awt.Graphics`) is a device-independent abstract class representing an area where a graphical image may be drawn.

Note: A Graphics object is said to define a *graphic context*. Therefore, drawing on a Graphics object is sometimes described as drawing in a graphic context. The graphic context may describe all or part of a window on the screen, an area on a printer page, etc.

Typically, you use

```
import java.awt.Graphics;
```

in code that references the Graphics class.

Obtaining a graphic context

The usual way to obtain a graphic context is to use the **getGraphics** method for a component. For example, suppose that you want to be able to draw on a picture box:

```
Graphics g = pictb_1.getGraphics();
```

obtains a graphic context for the picture box using the box's **getGraphics** method.

The **create** method of Graphics creates a new Graphics object associated with a particular rectangle in the original Graphics object:

```
// Graphics g1;
Graphics g2 = g1.create( x, y, width, height );
```

For example, suppose you want to draw a picture in a selected portion of a form. You could use **getGraphics** on the form to get a graphic context for the entire form, then **create** on the first graphic context to get a new (smaller) context.

Note that you cannot construct a valid graphic context directly. You either use **getGraphics**, or derive a new graphic context from an existing one.

Common Graphics properties

A Graphics object has the following properties:

ClipRect

The rectangle that marks the boundary of the object's *clipping rectangle*. The clipping rectangle is the area that you can actually draw in. For example, if the Graphics object represents an entire printer page, the clipping rectangle is typically initialized to be the part of the page where you can actually draw.

You can use the **clipRect** method to reduce the current size of the object's clipping rectangle. For example, suppose you have already drawn an image on one side of the graphic context and want to avoid drawing over that image. You can reduce the clipping rectangle to exclude the existing image, so that anything you draw in future cannot touch the image. For more about **clipRect**, see the *PowerJ Component Library Reference*.

Color

The color currently used for drawing on the object. For example, if you draw a line on the object, the line will have the color specified by the **Color** property.

Font

The font currently used when drawing (placing) text on the object.

Disposing of a graphic context

When you finish using a graphic context, you should free up the memory that is used to maintain information about the context. You do this with the **dispose** method of Graphics:

```
// Graphics g;
g.dispose( );
```

Drawing on a graphic context

The Graphics class defines a number of methods for drawing on a graphic context. For example, **drawLine** draws a line, **drawOval** draws an ellipse, **drawString** writes out the text from a String object, and so on. These methods are discussed in detail in the sections that follow.

Note: All the methods for drawing on a graphic context measure size and position in pixels. For an explanation of the difference between pixels and dialog units, see Pixels vs. dialog units.

Drawing lines

The **drawLine** method of Graphics draws a line between two points:

```
// Graphics g;
g.drawLine( x1, y1, x2, y2 );
```

The above function draws a line between (`x1`, `y1`) and (`x2`, `y2`). This line has the color specified by the **Color** property of the Graphics object.

Drawing rectangles

The **drawRect** method of Graphics draws a rectangle:

```
// Graphics g;
g.drawRect( x, y, width, height );
```

The above function draws a rectangle of the given width and height, with its upper left corner at (`x`, `y`). The edges of the rectangle have the color specified by the **Color** property of the Graphics object.

The **fillRect** method fills a rectangle with the current color (as specified by the **Color** property):

```
g.fillRect( x, y, width, height );
```

The **clearRect** method clears a rectangle by filling it with the current background color:

```
g.clearRect( x, y, width, height );
```

Using **clearRect** therefore "reverses" the effects of **fillRect**.

Alternate rectangle types

The Graphics class supports several alternatives to the simple rectangle. For example, a *3D Rectangle* is a rectangle that appears to be raised above the level of other objects, or recessed into its container. The following methods draw such rectangles:

```
// boolean raised;
g.draw3DRect( x, y, width, height, raised );
g.fill3DRect( x, y, width, height, raised );
```

In the above methods, the `raised` argument is `true` if you want the rectangle raised above its surroundings and `false` if you want it recessed into its surroundings. Otherwise, the above methods are similar to the methods for normal rectangles.

A *round rectangle* is a rectangle with rounded corners instead of sharp points. The amount of rounding is specified by two arguments:

`int arcWidth;`
: Specifies a horizontal distance from a corner of the rectangle. For example, suppose that `arcWidth` is 10. Then the top and bottom edges of the rectangle begin rounding themselves off when they approach 10 pixels from the "true" corners of the rectangle.

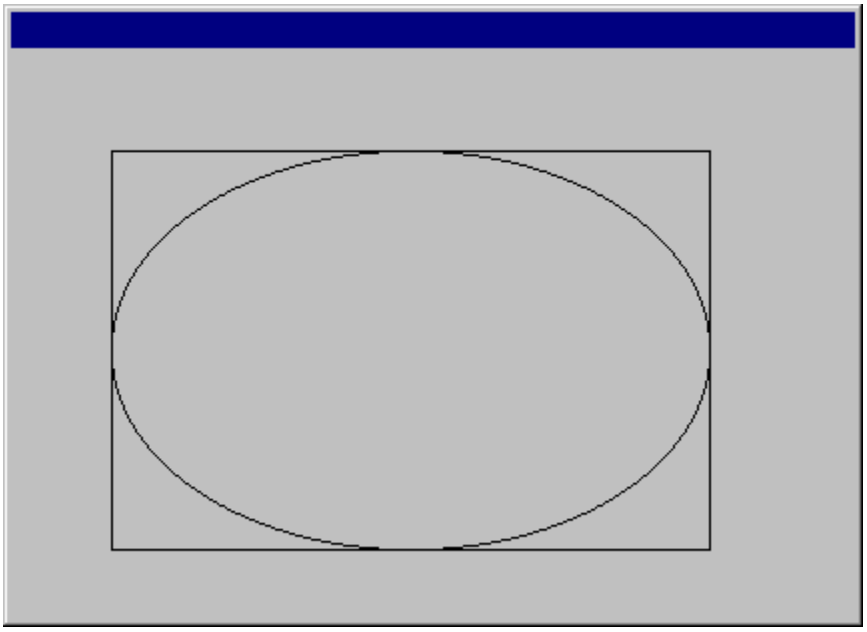
`int arcHeight;`
: Specifies a vertical distance from a corner of the rectangle. For example, suppose that `arcHeight` is 5. Then the left and right edges of the rectangle begin rounding themselves off when they approach 5 pixels from the "true" corners of the rectangle.

The following methods work with rounded rectangles:

```
g.drawRoundRect( x, y, width, height, arcWidth, arcHeight );
g.fillRoundRect( x, y, width, height, arcWidth, arcHeight );
```

Drawing ellipses (ovals)

The **drawOval** method of Graphics draws an ellipse. In order to draw an ellipse, you specify the rectangle that is tangent to the ellipse. In other words, you specify a rectangle and **drawOval** draws an ellipse whose edge just touches the midpoints of each of the rectangle's sides.



The following methods draw ellipses:

```
// Graphics g;
g.drawOval( x, y, width, height );
g.fillOval( x, y, width, height );
```

The **fillOval** method fills the ellipse with the current color specified by the **Color** property for the Graphics object.

Note: A circle is an ellipse whose bounding rectangle is a square.

Drawing arcs and pie shapes

The **drawArc** method of Graphics draws a portion of an ellipse. The ellipse itself is specified by the rectangle that bounds it (as with **drawOval**).

The starting point of the arc is specified by an integer giving angles in terms of degrees. The zero degree mark is at the three o'clock position. Positive angles are measured counterclockwise, so that a value of 90 indicates the twelve o'clock position. Negative angles are measured clockwise, so that a value of -90 indicates the six o'clock position.

The length of the arc is specified in terms of degrees, with positive values indicating counterclockwise rotations and negative values indicating clockwise rotations.

The **drawArc** method is used as follows:

```
// Graphics g;
g.drawArc( x, y, width, height, startAngle, arcAngle );
```

The `arcAngle` argument specifies how much arc is swept out. For example,

```
g.drawArc( x, y, width, height, 0, 90 );
```

draws the upper right quarter of the ellipse, while

```
g.drawArc( x, y, width, height, 0, -90 );
```

draws the lower right quarter.

The **fillArc** method of Graphics draws a pie shape or wedge. The edges of the pie consist of an arc, plus the two radius lines from the endpoints of the arc to the midpoint of the ellipse. The interior of the pie is filled with the color specified by the **Color** property of the Graphics object. The **fillArc** method is used as follows:

```
g.fillArc( x, y, width, height, startAngle, arcAngle );
```

For example,

```
g.fillArc( x, y, width, height, 0, 90 );
```

fills the upper right quadrant of the ellipse, while

```
g.drawArc( x, y, width, height, 0, -90 );
```

fills the lower right quadrant.

Drawing polygons

For the purposes of drawing shapes, a *polygon* is any figure whose sides are straight lines. The point where two lines meet is called a *vertex* of the polygon. In order to draw a polygon, you specify the vertex points in the order that they should be joined by lines. The lines for the polygon are drawn in the order given, using the color specified by the **Color** property for the Graphics object.

The **drawPolygon** method of Graphics draws a polygon. It has the format:

```
// Graphics g;
// int xPoints[];
// int yPoints[];
g.drawPolygon( xPoints, yPoints, N );
```

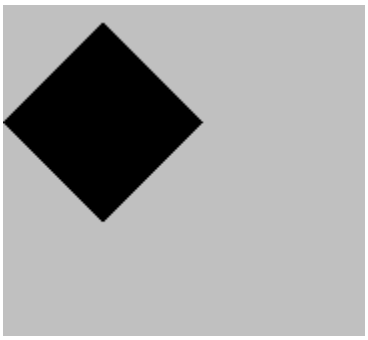
where `N` is the number of vertex points for the polygon. The first vertex of the polygon is (`xPoints[0]`, `yPoints[0]`), the next is (`xPoints[1]`, `yPoints[1]`), and so on. Similarly, the **fillPolygon** method fills the interior of a polygon with the color given by the **Color** property of the Graphics object:

```
// Graphics g;
// int xPoints[];
// int yPoints[];
g.fillPolygon( xPoints, yPoints, N );
```

For example, the following fills a diamond-shaped quadrilateral:

```
int xPoints[] = { 50, 100,  50,  0 };
int yPoints[] = {  0,  50, 100, 50 };
g.fillPolygon( xPoints, yPoints, 4 );
```

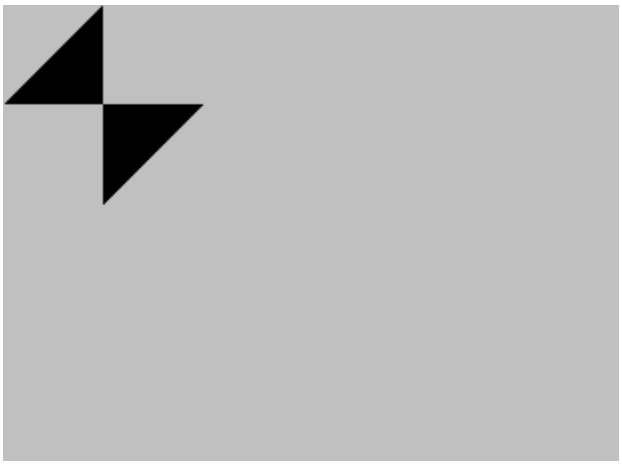
The result is the following.



The order of points in the two arrays is important. The following code uses the same vertex points as the previous example, but in a different order:

```
int xPoints[] = { 50,  50, 100,  0 };
int yPoints[] = {  0, 100,  50, 50 };
g.fillPolygon( xPoints, yPoints, 4 );
```

The result is a much different shape.



Drawing text

There are several methods for "drawing" (placing) text on a graphic context. The simplest is **drawString**:

```
// Graphics g;
// String str;
g.drawString( str, x, y );
```

This places the specified string, using (`x`, `y`) as the starting point for the *baseline* of the string. The baseline is the line on which the characters are drawn; characters may extend above and below this line.

The **drawChars** method is similar to **drawString** but takes its character text from a char array instead of a String:

```
// char text[];
// int offset, length;
g.drawChars( text, offset, length, x, y );
```

The `offset` argument specifies an offset within the character array `text` and the `length` argument specifies the number of characters you want to output. For example,

```
g.drawChars( text, 5, 10, x, y );
```

outputs 10 characters, beginning with the character at `text[5]`.

The **drawBytes** method is similar to **drawChars** but takes its text from a byte array:

```
// byte text[];
g.drawBytes( text, offset, length, x, y );
```

Paint events

The **Paint** event is triggered whenever a graphic context needs to be redrawn. For example, suppose you move a window over top of another window, then remove the first window again. The window that was on the bottom must redraw (repaint) the part of itself that was covered by the other window. It does this by triggering a **Paint** event on each object that was covered up. It is then up to the **Paint** event handler of each object to repaint itself.

The argument for a **Paint** event handler is a PaintEvent object. The most important method of PaintEvent is

```
// PaintEvent event;
Graphics g = event.getGraphics();
```

This returns the graphics object that needs to be repainted.

The default **Paint** event handler for each object repaints the object according to its current properties. For example, the default **Paint** event handler for a command button places the proper **Label** text on the button and depicts the button as either pressed or unpressed.

The default **Paint** event handler does *not* repaint anything that you have drawn explicitly on a Graphics context. For example, suppose that some other event handler uses

```
Graphics g = pictb_1.getGraphics();
g.setColor( Color.red );
g.fillRect( 50, 50, 30, 30 );
```

to draw a filled red rectangle on picture box `pictb_1`. The rectangle will appear in the picture box. However, if you cover the picture box with another window, then remove the covering window, the rectangle will disappear. The reason is that the default **Paint** event handler only redraws the things it knows about. It has no way to know that you drew something extra on the picture box, so it doesn't draw the rectangle.

If you want the red rectangle to be redrawn every time the picture box is repainted, you must define your own **Paint** event handler on the picture box. This event handler might explicitly redraw the rectangle every time by calling **fillRect**. Alternatively, you might make a copy of the Graphics object after drawing the rectangle the first time. Then, each time your **Paint** event handler is called, it can restore the original rectangle by copying the saved Graphics object onto the picture box.

Tip: A number of reference books about AWT provide good discussions on repainting objects in response to various operations (for example, repainting when the user changes the size of the object).

The Image class

The AWT Image class is an abstract class representing graphical images. These can be used as "pictures" on Java forms.

Loading images from files

The easiest way to create an Image object is to load an image from a file or URL. AWT supports GIF and JPEG images.

The **getImage** method creates an image object based on a GIF or JPEG file. This method is defined in the AWT Applet class and as a static method in the Toolkit class. For more information about these classes, see Applet class and Toolkit class.

Here are some sample uses of the method:

```
// URL img2URL;
Toolkit tk = Toolkit.getDefaultToolkit();
Image img1 = tk.getImage( "picture.jpeg" );
Image img2 = tk.getImage( img2URL );
```

Notice that you can specify where to find the image using either a file name or a URL.

The **getImage** method simply associates the name of the file or URL with the Image object. The method does *not* load the image, nor does it check whether the file or URL can actually be opened. The image will not be loaded from its source until the program actually tries to draw the image. At that time, the run-time environment will attempt to access the file or URL, then load the image.

This delayed-loading approach is simple and efficient, since the time-consuming process of loading only takes place when the image is actually needed. On the other hand, some programs need to have more detailed knowledge of the image-loading process. For example, one thread in a multi-threaded program may need to know whether another thread has finished loading a particular image. In this case, you can use

```
// Component comp;
// Image img;
powersoft.powerj.ui.UiUtil.waitForImage( comp, img );
```

to wait for a specified image to be loaded. The `comp` argument specifies the component with which the image is associated. The **waitForImage** method does not return until the image has finished loading.

Image size

You can determine the size of an image using

```
// Image img;
// ImageObserver imgOb;
int h = img.getHeight( imgObj );
int w = img.getWidth( imgObj );
```

Both of these dimensions are given in pixels.

Note: The above methods will not work until the image has actually finished loading.

Drawing images

The **drawImage** method of Graphics draws an image on a graphic context:

```
// Graphics g;
// Image img;
g.drawImage( img, x, y, this );
```

The `x` and `y` arguments specify the upper left corner position for the image. For example,

```
g.drawImage( img, 0, 0, this );
```

draws the image in the upper left hand corner of the graphic context.

The last argument for this form of **drawImage** specifies the component associated with this graphic context. This argument is almost always `this`, standing for the current form.

Note: The last argument for this form of **drawImage** is an ImageObserver value. The ImageObserver class represents objects that can keep track of the progress of loading an image. This includes PowerJ form classes. For more information on ImageObserver objects, see the standard references on AWT.

Drawing scaled images

Another form of **drawImage** lets you scale an image to fit a particular rectangle:

```
g.drawImage( img, x, y, width, height, this );
```

The `width` and `height` arguments specify the width and height to which the image should be scaled when it is drawn.

Setting background color

Two final forms of **drawImage** let you specify a background for the image:

```
// Color bgCol;
g.drawImage( img, x, y, bgCol, this );
g.drawImage( img, x, y, width, height, bgCol, this );
```

The first form draws the image at the given (`x`, `y`) position. The second form also scales the image to the given `width` and `height`.

Painting picture boxes and paint canvases

The **Paint** event is triggered on a picture box or a paint canvas when the contents of the object need to be repainted. This can happen under a variety of circumstances:

- If the user moved some other window over some or all of the object, then removed the covering window again.
- If the user minimized the form containing the object, then brought the form back again.
- If you explicitly invoke the **paint** method for the picture box or paint canvas. The most important action of the **paint** method is to trigger a **Paint** event.

The default handling for the **Paint** event simply redraws the image associated with the object (if any). However, you can write your own **Paint** event handler to perform different processing. In particular, you might use a picture box or paint canvas to display graphs, user-drawn pictures, and other objects that are created through explicit drawing actions rather than loading a pre-existing graphic.

Underlying classes of PictureBox and PaintCanvas

The PictureBox class is derived from the PaintCanvas class (`powersoft.powerj.ui.PaintCanvas`). The PaintCanvas class in turn is derived from the AWT Canvas class (`java.awt.Canvas`). A canvas is just a normal component with a **paint** method that the object can use to repaint itself. In the case of PaintCanvas and PictureBox, the **paint** method triggers a **Paint** event on the object. In this way, you can write your own code to repaint the object.

The EraseBackground property

The PaintCanvas and PictureBox classes support a boolean property called **EraseBackground**. If **EraseBackground** is `true`, the entire visible area of the object will be erased before triggering the **Paint** event. The erasing process is performed by filling the visible area of the object with the current background color.

If **EraseBackground** is `false`, there is no preliminary erasing before the **Paint** event is triggered. This can sometimes reduce flickering of an image if you repaint frequently.

You can set the **EraseBackground** property with **setEraseBackground**, as in:

```
pictb_1.setEraseBackground( true );
```

The PaintEvent class

The **Paint** event for an object receives an object of the PaintEvent class (`powersoft.powerj.event.PaintEvent`). This class is derived from the standard EventData class, but includes the following additional method:

```
// PaintEvent event;
Graphics g = event.getGraphics();
```

The Graphics object returned by **getGraphics** provides a graphics context for the object that needs to be repainted. By drawing on this graphics context, your **Paint** event handler can repaint the object.

Avoiding flicker

A slow drawing speed may sometimes cause "flicker" on the user's screen. You can usually avoid this by using a double-buffering technique. This technique involves drawing to an Image buffer, then copying the graphics area to your actual on-screen Graphics object in a single operation. Various reference books on Java discuss ways of doing this, but here is some simple code to illustrate the principles:

```
// create an image buffer for double-buffering
java.awt.Image imageBuffer = pictb_1.createImage(200, 200);
java.awt.Graphics bufferG = imageBuffer.getGraphics();

// clear the buffer
bufferG.setColor(java.awt.Color.black);
bufferG.fillRect(0, 0, 200, 200);

// draw everything in the buffer
bufferG.setColor(java.awt.Color.white);
bufferG.drawString("some text", 10, 10);

// copy the buffer to the real canvas
java.awt.Graphics visibleG = pictb_1.getGraphics();
visibleG.drawImage(imageBuffer, 0, 0, pictb_1);

// clean up
imageBuffer=null;
bufferG.dispose();
visibleG.dispose();
System.gc();
```

Printing output

There are no facilities for printing output in JDK 1.02. JDK 1.1 introduced PrintJob objects to control printing.

Note: Untrusted applets cannot initiate their own print jobs. Instead, you must use facilities inside the browser or applet viewer to print output. It is also possible to implement printing facilities on a web server using a server extension. For a discussion of printing in Java, see

`http://www.ibm.com/java/education/printing/index.html`

The first stage in a print job is to create such a PrintJob object:

```
java.awt.PrintJob pjob = getToolkit().getPrintJob( frame,
    title, props );
```

The arguments are:

`java.awt.Frame frame;`
Typically, this argument will be `this`, standing for the current form.

`String title;`
A title for the print job.

`java.util.Properties props;`
If you specify a non-null value for this argument, **getPrintJob** will store the user's printing preferences in the object.

Executing **getPrintJob** opens a dialog box which obtains information from the user. On a Windows system, the dialog is typically a normal print dialog box; this allows the user to choose a printer, the number of copies, and other printer information.

Calling **getPrintJob** initiates the print job by obtaining information from the user. Before you can begin the actual printing, you must obtain a Graphics context for the PrintJob object:

```
// PrintJob pjob;
java.awt.Graphics g = pjob.getGraphics();
```

This Graphics object represents a printer page. If **getGraphics** cannot create an appropriate graphics context, it returns null.

Once you have the Graphics object, you may use normal methods for drawing on the graphics context (for example, **drawString**, **drawImage**, and so on).

When you have finished drawing the page, you send the page to the printer with:

```
// Graphics g;
g.dispose();
```

This prints the first page. To print another page, you must get a new Graphics object from the PrintJob object, as in:

```
java.awt.Graphics g = pjob.getGraphics();
// draw next page
g.dispose();
```

When you have printed the last page of the print job, you end the print job itself with:

```
pjob.end();
```

To print a particular object on a graphics context, you can use the object's **print** method. For example, the following code prints the contents of a text field:

```
java.awt.Graphics g = pjob.getGraphics();
textf_1.print( g );
g.dispose();
```

By default, the **print** method simply paints the component onto the printer page (by calling **paint**), in much the same way that the component is displayed on the screen. You may want to create your own **print** method for an object if this isn't satisfactory.

The **printAll** method for an object calls **print** on all the components contained by the object. For example, if `this` refers to the current form

```
java.awt.Graphics g = pjob.getGraphics();
this.printAll(g);
g.dispose();
```

prints the current form, including all the objects on the form, all the objects contained by such objects, and so on.

Chapter 15. Using threads

This chapter examines the use of threads to perform multiple tasks concurrently. It describes what threads are, and why you might want to use them. Finally, it discusses how to create and synchronize multiple threads of execution within a process.

Note: The `Threads` sample project demonstrates the use of threads.

Processes, threads and multitasking

A *process* is an independently executing program. It has its own virtual address space; therefore it has its own code and data, separate from the code and data of other processes. When one process starts another, the new process executes with its own address space, including copies of its own code and data. Each process is started as a single thread of execution, but other threads can be created in the process.

A *thread* is a single line of execution within a program. Threads execute independently. All threads in a process share the virtual address space, static memory, dynamically allocated memory, and system resources of the process. Each thread maintains a private copy of context information, including copies of machine registers, the execution stack, a thread environment block, and a user stack.

In a simple program, there is a single line or thread of execution. When one routine calls another, the first routine waits for the second to finish before resuming its own execution. This kind of program is called *single threaded*; execution proceeds in a linear fashion, with only one routine executing at a time.

A *multithreaded* program has more than one line of execution running concurrently in a single process. For example, one routine may call another, then immediately resume its own execution, without waiting for the called routine to finish. In other words, the calling routine starts a new thread of execution which runs concurrently with the first thread.

Note: PowerJ uses the standard thread-handling facilities of the Java language. The rest of this chapter provides an introduction to those facilities. For more information, see any standard Java documentation.

Multitasking and multiprocessing

Multitasking operating systems allow a single processor to run multiple threads concurrently. The system runs an active thread for a short time period, known as a *time slice*, and then switches to the next active thread. The switch from one thread to another is called a *context switch*. A context switch can also occur if the thread yields control before its time slice is completed. After another time slice, the operating system switches to the next thread. The operating system continuously repeats this action to cycle through all threads for all active processes, with threads of higher priority getting more frequent time slices.

Only one thread can be executing on a single processor at any instant, but multitasking allows the execution of multiple threads to be interleaved. Because the time slice is small, multiple threads appear to run simultaneously, even though only one thread is being executed at a time. Actually, because of system overhead, a system slows down if it has to coordinate too many threads.

Some operating systems, such as Windows NT but not Windows 95, support *multiprocessing*. This means that if your computer has more than one processor, the operating system can allocate different threads to different processors. In a multiprocessing system, more than one thread can be executing at any instant.

Systems that cannot multitask

Some systems (for example, Windows 3.1x) do not support multitasking. In such cases, the Java run-time environment can still provide a form of multithreaded execution, but general program behavior may be different. For more information, see Thread priority.

Differences between processes and threads

It is important to understand the differences between processes and threads. When your program starts a new thread, the thread executes within the process; it shares the same address space, and therefore has direct access to data and code. When your program starts a new process, the process executes as a separate entity; it has a separate address space, and has no direct access to any data or code of the original process.

Unlike processes, threads share code and the memory *heap*, but each thread has its own *stack*. The heap stores global, static variables and dynamically allocated memory items. The stack stores execution information; for example, when one function calls another, the stack holds the address where execution should return when the called function terminates. The stack also provides space for automatic variables. Thus static variables and allocated memory are shared between threads while local automatic variables are not.

Threads incur less system overhead than processes. Since threads do not require their own address space and code, the operating system can create threads and switch between threads more quickly than between processes. The operating system also uses less memory to keep context information for an individual thread than for an entire process.

When to use multiple threads

There are many reasons why you might want to use multithreaded processes. These include wanting a more responsive user interface or wanting to make more efficient use of the hardware.

- If your application performs lengthy tasks that interfere with the responsiveness of its user interface, you could perform the lengthy tasks with a separate thread. The user interface thread would then receive regular time slices and be more responsive, especially if it is given higher priority than the other thread. For instance, it might be appropriate to start a separate thread for tasks where a single threaded version would make the user wait for completion while displaying an hourglass cursor.
- If your program is accessing slow hardware, such as reading a file from disk, you could move that part of the program to a separate thread. That way, other threads can execute during the time that the one thread is waiting for the hardware.
- If your application is computationally intensive, you can divide the work among multiple threads to exploit multiprocessing when it is available.

Keep in mind that using multiple threads consumes more processor time and memory, and that there is added programming complexity for you to avoid conflicts between the threads.

Designing for multiple threads

A fundamental difference between single threaded and multithreaded programs is that the sequence of execution for a multithreaded program is much less deterministic. A multithreaded program cannot be expected to execute in the same order twice, since the operating system is unlikely to allocate time slices to its threads in the same order. This can make a multithreaded program much harder to debug, since you cannot completely reproduce runs when trying to track down a problem.

Because threads share the same address space and context switches between threads can occur at any time, your program must avoid problems that can arise from interactions between threads. You must write a multithreaded program so that while any thread is using a resource, other threads are prevented from using it. For example, consider a data structure read by thread A and updated by thread B. If a context switch from A to B occurs while A is in the midst of reading the data, and B updates the data in its time slice, A can end up with corrupt data. You can prevent this by synchronizing the execution of threads so that shared resources are not accessed by more than one thread at a time.

When designing a multithreaded program, you must also be careful to prevent problems like *deadlock* and *race conditions*. Deadlock occurs when all threads are waiting or are otherwise blocked from executing in an interdependent way, so the process is indefinitely suspended. A race condition arises when one thread relies on the completion of a task in another thread without explicitly waiting for it, so it only works if the second thread "wins the race" and completes its action before the first needs it.

For an example of deadlock, consider three threads A, B and C. The main thread is A. It invokes B then waits for thread C to complete. B invokes C and then waits for A to complete. C waits for B to complete. The sequence of events is:

1. A invokes B.
2. A then waits for C to complete.
3. B invokes C.
4. B then waits for A to complete.
5. C then waits for B to complete.

Each thread is left waiting for another, so none of the threads can resume; the threads are deadlocked.

Synchronization
Master-worker model
Daemon threads
Accessing resources

Synchronization

In order to prevent deadlock or race conditions, to protect resources, or to ensure that execution occurs in the proper order, you need to synchronize the execution of different threads. This can be accomplished in several ways, including:

- Using Java monitors. Monitors are facilities intended specifically for synchronization.
- Using shared data. Since global and static data is shared between threads, it can be used to exchange synchronization information.

These techniques are discussed in more detail in Synchronizing threads.

Master-worker model

To reduce the chance of deadlock and keep multithreading simple, most multithreaded programs operate on the *master-worker model*. In this model, there is a primary thread called the *master thread*. From time to time, the master thread may start up *worker threads* to perform specific tasks. When a worker is finished, it indicates to the master that it has finished, and then it closes (or waits for more work).

For example, suppose you want to display a "Please wait" dialog box while a lengthy operation takes place. The master could start a worker thread to perform the operation, then display the "Please wait" dialog box until the thread was finished.

In the master-worker model, workers do not interact with each other—they only communicate with the master. The master supervises worker efforts to keep things running smoothly. This simple division of labor usually leads to code that is easier to develop and debug.

Daemon threads

Another approach to coordinating threads is the use of *daemon threads*. A daemon thread can be regarded as a worker thread whose services are available to all other threads in the program. For example, you might create a thread whose purpose is to open URLs and load graphic images. Whenever another thread wants to perform this operation, it queues up a request for the "graphics loader" thread. Meanwhile, the loader simply loops until it receives a request, whereupon it does all the work required to fetch the desired image.

Programs typically start daemon threads during initialization or the first time such a thread is needed. From this point on, the daemon threads remain in existence until the program terminates. This ensures that the services provided by daemons are available to other threads whenever needed.

Daemon threads often just loop until their services are needed. One way to do this is for the thread to "go to sleep" for an appropriate period of time, then wake up to see if a request for services has been received. This sleep/wake pattern prevents a thread from taking up too much processor time when the thread's services are not needed. It is also possible for a daemon thread to go into a wait state until another thread notifies the daemon that its services are required.

Accessing resources

A frequent cause of deadlock is having different threads access the same resources in different order. For instance, consider two threads in the following situation:

- Thread A holds resource X and waits for resource Y.
- Thread B holds resource Y and waits for resource X.

Each thread waits for the other, resulting in deadlock.

You can avoid this by minimizing the sharing of resources, by having threads access shared resources in the same order where possible, and by having each thread release a shared resource whenever possible, even if the thread later has to wait to get the same resource back.

Another good strategy is to order the use of resources by how precious the resource is. The least precious resources should be accessed first. Resources should be released in the opposite order if more than one resource is no longer needed at the same time. This way, the most precious resources will be held for the shortest time.

Specifying threads

There are two ways to specify a new thread in your program:

- Create an object of a class that is derived from the Thread class.
- Create an object that implements Runnable.

These two alternatives are discussed in the sections that follow.

The Thread class

The Thread class is a standard Java class (in `java.lang`). One way to create a new thread in your program begins with creating a class based on Thread. Here is a simple example of such a class:

```
public class MyThread extends Thread
{
    public MyThread(String threadName)
    {
        super( threadName );
    }

    public void run( )
    {
        System.out.println( "MyThread in execution now" );
    }
}
```

This class has two methods: a constructor and a **run** method.

- The constructor takes a single argument `threadName`; this is just a string providing a name for the thread. The constructor simply uses **super** to invoke the constructor for the parent class (Thread itself). Therefore, the constructor for MyThread just uses the Thread constructor to create a thread of the given name.
- The **run** method specifies the code that will run when the thread is put into execution. Therefore, **run** can be considered the "mainline" routine of the thread. In the sample above, the thread just prints out a message.

In order to put this kind of thread into execution, you create an object of the given class and then execute the **start** method on it:

```
MyThread t = new MyThread( "DoIt" );
t.start( );
```

The **start** method (defined in Thread and inherited by MyThread) initializes the thread for execution and then invokes your **run** routine. The **run** routine then performs the desired "work" of the thread.

The Runnable interface

The Runnable interface is available when you want the "mainline" of your thread to be a member function in a class that isn't based on Thread. For example, suppose that you want the mainline of your thread to be a member function in a form. Form classes are based on the PowerJ Form class; since Java does not support multiple inheritance, you can't have a form that is based on both Form and Thread. Therefore, you can use the following format to declare your form class:

```
// import java.awt.Frame;
public class MyForm extends Frame implements Runnable {
    // define data members and member functions

    public void run( ) {
        // mainline for thread
    }
}
```

As shown, you specify that the class implements Runnable. You then define a **run** method within the class, in addition to other data members and member functions required by the class. This **run** method serves as the mainline for the thread.

In order to start the thread executing, create an object of the specified type, create a thread associated with that object, and execute the **start** method on the thread:

```
MyForm f = new MyForm( );
Thread t = new Thread( f );
t.start( );
```

The **start** method initializes the thread for execution, and then invokes your **run** routine. The **run** routine then performs the desired "work" of the thread.

The current thread

During execution, a Java program may obtain a reference to the currently executing thread with

```
Thread t = Thread.currentThread( );
```

This executes a static method named **currentThread** defined in the Thread class. The Thread object returned by **currentThread** always refers to the thread that is executing the **currentThread** function call.

Once you have obtained this Thread object, you can use it to invoke Thread methods on the current thread.

Terminating a thread

A thread terminates execution when the thread's **run** method returns to its caller. This is generally the "cleanest" way to terminate a thread. After **run** finishes executing, the run-time environment cleans up after the thread.

Another way to terminate thread execution is to invoke the **stop** method on the thread object. For example, if you start the thread executing with

```
MyThread t = new MyThread( "DoIt" );
t.start( );
```

you can terminate execution with

```
t.stop( );
```

Similarly, if you started the thread using an object that implements Runnable, you can execute the **stop** method on thread object you used to start the thread.

When you execute **stop** on a thread, the run-time environment cleans up after the thread, but the thread doesn't get the chance to do any of its own wrap-up. For example, if you **stop** a thread that is in the middle of writing data to a file, the thread doesn't get a chance to finish writing the rest of the data. This may leave the file's data corrupted.

Because of such difficulties, you may want to avoid using **stop**. Instead, you could set a global variable to a value that indicates you want the thread to terminate. The **run** method for that thread can check this variable periodically. When the variable indicates a termination request, **run** can perform any necessary wrap-up, and then return.

Daemon threads and program termination

Daemon threads and program termination

The **Daemon** property of a thread object is used to identify daemon threads.

A program terminates when the last of its non-daemon threads finishes execution. At this time, there may still be active daemon threads—remember that daemon threads typically stay in some kind of waiting loop through all of program execution, in case some other thread needs a daemon's services. Once all the non-daemon threads are gone, it is assumed that the daemons are just looping without any work to do, so the whole program is terminated.

Since the daemons are automatically terminated when all the non-daemons are finished, your last non-daemon should not terminate until all daemons are finished real work. For example, the primary thread is typically the last thread to terminate. This thread should not terminate until it has checked that all daemons have finished any tasks they have been assigned.

Suspending and resuming thread execution

The **suspend** method of Thread temporarily stops the execution of a thread. The thread does nothing until another thread issues a **resume** operation on the suspended thread. For example, this code immediately suspends the current thread:

```
Thread t = Thread.currentThread( );
t.suspend( );
```

Execution does not resume until another thread re-activates this thread. When execution does resume, it will look as if **suspend** just returned normally. Execution continues with the statement after the call to **suspend**.

To suspend another thread, you must have a Thread object referring to that thread. For example, the following code creates a thread, performs a few more actions, then suspends the thread:

```
MyThread t = new MyThread( "Name" );
t.start( );
      // ...other actions.
t.suspend();
```

In the interval between the **start** call and the **suspend** call, both the new thread and the current thread execute simultaneously.

Note: Typically, you would protect the **suspend** operation by enclosing it in a *synchronized* block. For more information on synchronization, see Synchronizing threads.

The resume method

The **resume** method tells a suspended thread to resume execution. This can only be performed on another thread, using a Thread object. For example, the thread suspended by the preceding code could be re-activated with:

```
t.resume();
```

Execution resumes at the point where the thread was suspended. Therefore, the thread cannot tell that it was suspended at all: the **suspend/resume** process is transparent to the thread.

The sleep method

The **sleep** method of Thread is a static method which pauses the *currently* executing thread for a given length of time. While a thread is sleeping, other threads (if any) are allowed processor time in order to execute.

There are two forms for the **sleep** method. The first gives a time in terms of milliseconds:

```
Thread.sleep( milli );
```

For example,

```
Thread.sleep( 1000 );
```

puts the current thread to sleep for 1000 milliseconds (one second).

The second form of **sleep** is

```
Thread.sleep( milli, nano );
```

The first argument gives a number of milliseconds and the second a number of additional nanoseconds. The thread goes to sleep for the total specified time.

Interrupting a sleeping thread

The **interrupt** method of Thread can be executed to wake a thread that is currently sleeping. The **interrupt** method is executed by one thread on another thread:

```
// Thread otherThread;
otherThread.interrupt( );
```

When a sleeping thread wakes up, it can use the static **interrupted** method to determine whether it slept the whole specified time or was interrupted before the full time elapsed:

```
boolean intrupt = Thread.interrupted( );
```

This method is executed by the current thread to see if it was interrupted. It returns `true` if the current thread was interrupted during the most recent **sleep**, and `false` otherwise.

The **isInterrupted** method is similar to **interrupted**, but is executed on another thread to see if it has been interrupted:

```
// Thread otherThread;
boolean intrupt = otherThread.isInterrupted( );
```

Warning: The **interrupt** method works by setting a bit saying that an interrupt has been requested on a particular thread. Both **interrupted** and **isInterrupted** do their work by checking this bit. Some implementations of Java may not do anything in response to **interrupt** except set the bit. This means that threads always sleep for the full specified time; however, when they wake up, they can check the bit to see if some other thread wanted to interrupt them.

Thread properties

This section discusses some of the properties that a thread may have.

Thread name

All threads have a name, set at the time the thread was constructed. For example,

```
MyThread t = new MyThread( "Foo" );
```

creates a new thread named `Foo`. You can determine the name of a thread with **getName**:

```
String threadName = t.getName( );
```

You can change the name of a thread with **setName**:

```
t.setName( "NewName" );
```

Thread priority

Every thread has a **Priority** property, indicating its urgency relative to other threads. The priority is expressed as an integer between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY` (inclusive). This corresponds to a range of 1 through 10, with normal priority (`Thread.NORM_PRIORITY`) set to 5.

The **Priority** property has the usual **get** and **set** methods. For example, the following code increases the priority of the current thread by 1:

```
Thread t = Thread.currentThread( );
int p = t.getPriority( );
if (p < Thread.MAX_PRIORITY) t.setPriority( p+1 );
```

A thread can set its own priority or the priority of another thread. In order to set the priority of another thread, you need a Thread object referring to that thread.

Note: If you attempt to set a priority outside the allowed range (for example, a higher priority than `Thread.MAX_PRIORITY`), **setPriority** throws an `IllegalArgumentException`.

The effect of priority depends on whether the operating system is multitasking.

- If the operating system is multitasking, high priority threads are scheduled time slices for execution more frequently than low priority threads.
- If the operating system is not multitasking, priority is handled entirely by the Java run-time environment. The environment allows the highest priority thread to execute until one of the following occurs:

  1) The thread terminates.

  2) The thread yields control (as explained below).

  3) The thread starts a new thread with a higher priority.

  If there are several threads which all share the same high priority, the run-time environment chooses one of these threads and lets it run until one of the above three conditions applies. The run-time cycles through all the threads of the highest priority in a round-robin fashion, without letting lower priority threads execute.

In a multitasking system, even low priority threads get a chance to execute occasionally. In a non-multitasking system, a high priority thread can effectively block a low priority thread from ever executing. In World Wide Web's user/server applications, it is possible that some of your users will have non-multitasking systems (for example, if you want to support Windows 3.1x users). In such a situation, you should make sure a high priority thread is not waiting for a low priority thread to terminate, since this often leads to deadlock.

The yield method

The **yield** method of Thread makes it possible for a high priority thread to give up control to another thread:

```
// Thread t;
t.yield( );
```

This allows another running thread to obtain some processor time. In particular, the **yield** method is one way for a high priority thread to yield to a lower priority thread. Note, however, that you cannot

specify which thread obtains processor time after the current thread yields. For example, suppose the program is running on a non-multitasking system, with two high priority threads and one low priority one. If one high priority thread yields, the other high priority thread goes into execution. If the new thread yields, the first high priority thread goes back into execution. In this situation, the two high priority threads may keep yielding back and forth to each other, without ever giving the low priority thread a chance to execute.

The Daemon property

The **Daemon** property determines whether a thread is a daemon thread. (For an explanation of daemon threads, see Daemon threads. You can turn a thread into a daemon thread with

```
// Thread t;
t.setDaemon( true );
```

You can turn a daemon thread into a non-daemon thread with

```
t.setDaemon( false );
```

The **isDaemon** method determines whether a given thread is a daemon thread:

```
boolean dm = t.isDaemon( );
```

The result is `true` if the thread is a daemon thread and `false` otherwise.

Note: The difference between daemon and non-daemon threads is only relevant in determining when a program terminates. For more information, see Daemon threads and program termination.

Thread states

Threads may occupy any of the following states:

New
: A new thread has been created using an appropriate Thread or Runnable object, but has not yet been put into execution using **start**.

Runnable
: The thread has been put into execution using **start** and is eligible to be given processor time. A Runnable thread may not actually be running at the moment; for example, it may be a low priority thread that is stuck behind a high priority thread. Nevertheless, the thread is in a state where it could execute if given a chance.

Not Runnable
: The thread is not available for active execution. This may happen for any of the following reasons:

  The thread is blocked while it waits for an I/O operation to complete.

  The thread has been suspended with the **suspend** method.

  The thread has gone to sleep using the **sleep** method.

  The thread has used the **wait** method to wait for a condition to be fulfilled.

Dead
: The thread has been stopped with the **stop** method, or it has finished execution by returning from its **run** method.

A thread is considered "alive" if it is Runnable or Not Runnable. The **isAlive** method of Thread determines whether a thread is alive.

```
// Thread t;
boolean alive = t.isAlive( );
```

If a thread is not alive, it may be New (hasn't started running yet) or Dead (has finished running).

Synchronizing threads

The Java language provides facilities to synchronize threads through the use of *monitors*. A monitor is a mechanism that is associated with a specific data object, and controls access to that data object. The purpose of a monitor is to ensure that when one thread is accessing a data object, other threads are prevented from accessing the object.

Synchronized critical sections

The `synchronized` keyword in Java is used to label sections of executable code within a class definition. Usually, `synchronized` is applied to an entire method function; it can be applied to blocks of code within methods, but this tends to make the code harder to debug and maintain.

The monitor for a specific data object prevents separate threads from executing synchronized code on the same object. To see what this means, consider a simple class definition:

```
class SingleAccess {
    private int data;

    public synchronized int read( ) {
        return data;
    }

    public synchronized void store( int value ) {
        data = value;
    }
}
```

(In practical terms, you would almost never use `synchronized` with a data object this simple. Nevertheless, the example illustrates a number of fundamental principles.)

This class holds a single integer value. It provides two member functions: **read** to obtain the current value of the integer, and **store** to store a new value. Both of these member functions are marked as `synchronized`.

Now suppose you create an object of this class with

```
SingleAccess sa;
```

Creating an object of this class also creates a monitor to keep track of the object `sa`. The monitor does not allow separate threads to execute synchronized code on `sa`. For example, it won't let one thread try to **read** the object at the same time that another thread is trying to **store** a new value; similarly, the monitor won't let two separate threads **store** values at the same time. Only one thread can access `sa` at a time.

Note that the monitor only controls a single object. If, for example, your program has two objects of the `SingleAccess` type, one thread could **store** a value in one of the objects at the same time another thread was storing a value in the other object.

Critical sections

A block of code marked as `synchronized` is called a *critical section*. While one critical section is executing for a data object, the monitor prevents other critical sections from executing on the same data object.

Once a critical section finishes execution, the monitor allows other critical sections to execute on the object. For example, suppose a thread named X begins executing a synchronized method on a data object and then a thread named Y tries to execute a synchronized method on the same object. The monitor suspends Y at this point of execution until X's critical section terminates. At that point, the monitor executes the **resume** method on Y to allow Y to proceed with its critical section.

Re-entrancy

Java monitors allow re-entrance. This means that the same thread can simultaneously be executing more than one synchronized method on the same object.

For example, suppose an object has three synchronized methods: **setForeColor**, **setBackColor**, and **setBothColors**. The **setBothColors** method calls the other two methods to set foreground and background colors together. A thread can call the synchronized **setBothColors** which in turn calls the other two synchronized methods; this is allowed because the monitor allows the same thread to execute more than one synchronized method on the same object. However, suppose another thread attempts to use **setBackColor** to set the object's background color. The monitor will lock out the second thread because the first thread is already executing a synchronized method on the object.

The notify and wait methods

The **notify** and **wait** methods may be used in `synchronized` routines to coordinate the activities of multiple threads.

Note: The **notify** and **wait** methods are defined in the Object class. This means they can be used on any object, since all Java classes inherit from Object.

The easiest way to understand how these methods work is to consider a simple example. Suppose that **store** and **read** are `synchronized` methods for an object X. The **store** method stores a value in a data member of X; the **read** method retrieves the value from that data member. Thread TS uses **store** to store a new value in X and thread TR uses **read** to read that value; therefore, you want TR to wait for TS to store its value.

You can implement this using the following calls:

- In **read** (used by TR), you call **wait** before reading the value.
- In **store** (used by TS), you call **notify** after writing the value.

The call to **wait** suspends TR indefinitely, until some other thread notifies TR that it can resume execution. The call to **notify** chooses one suspended thread for the object and tells that thread it can resume. In effect, TR waits for TS to set the value; when TS notifies TR that the value has been set, TR proceeds to read the value.

Here's a simple class definition that uses this principle:

```
class SingleAccess {
    private int data;
    private bool newData = false;

    public synchronized int read( ) {
        while ( !newData ) {
            try {
                wait( );
            } catch( InterruptedException e ) {
                // instructions here to cope with interrupt,
                // possibly breaking out of loop or continuing
            }
        }
        newData = false;
        return data;
    }

    public synchronized void store( int value ) {
        data = value;
        newData = true;
        notify( );
    }
}
```

This introduces a variable named `newData` which is set `true` after **store** stores a new value in `data`.

- The **read** method checks `newData` to see if a data value has been stored; if not, it waits for another thread to store that value. If a new data value has already been stored, **read** does not wait but goes straight to obtain the new value.

- The **write** method sets `newData` to indicate that a new data value has been stored. It also calls **notify** to notify any waiting threads that the new value has been stored.

When a `synchronized` routine starts to **wait**, the monitor associated with the object will allow other threads to use `synchronized` routines on the object. If one thread is waiting in **read**, the monitor lets another thread execute **store**, even though both **read** and **store** are `synchronized`.

As a more complicated example, the following definition is designed to let two threads store and read in alternating fashion. Once a value has been stored, **store** waits for it to be read with **read** before making itself available for storing new values.

```
class SingleAccess {
    private int data;
    private bool newData = false;

    public synchronized int read( ) {
        while ( !newData ) {
            try {
                wait( );
            } catch( InterruptedException e ) {
                // deal with interruption
            }
        }
        newData = false;
        notify( );
        return data;
    }

    public synchronized void store( int value ) {
        while ( newData ) {
            try {
                wait( );
            } catch( InterruptedException e ) {
                // deal with interruption
            }
        }
        data = value;
        newData = true;
        notify( );
    }
}
```

In this new version, **read** notifies **store** when it has read the current value, just as **store** notifies **read** when it has stored a new value.

The notifyAll method

The **notify** method notifies a single waiting thread. If there are several threads waiting to use the same object, **notify** chooses one and notifies that thread.

In some cases, you may want to notify *all* of the threads that are waiting to use an object. In this case,

```
notifyAll( );
```

notifies all the threads. You often need **notifyAll** in situations where you may have several threads waiting, with each one waiting for a different condition to come into effect. By invoking **notifyAll**, you can have each thread wake up, check to see if its own condition has been fulfilled, and go back into **wait** if the notification is intended for some other thread.

Volatile data

The `volatile` qualifier is used to mark a data object whose value may be changed by different threads but which is not protected by a monitor. For example, suppose that different threads may change the value of an integer variable. You could protect this variable by making a special class and by protecting it with a monitor, but that imposes a considerable amount of overhead. Another approach is simply to mark the variable as `volatile`.

When an object is `volatile`, the program makes sure to access the object directly every time it is used. This differs from the usual handling of data objects. For example, if you use a variable in one part of an expression, then re-use the same variable in another part of the expression, the compiler may try to optimize performance by storing the variable's value in a hardware register; since it is faster to access a register than normal memory, storing frequently used values in registers can speed execution. However, `volatile` prevents this kind of optimization—it forces the program to obtain the value directly from the data object every time it is used, on the theory that the value may be changed by another thread without warning.

Note: Your program should be careful how it uses `volatile` data. For example, consider the situation discussed in the previous paragraph, where a `volatile` value is used more than once in the same expression. There is a possibility that the value will be changed by another thread between one use and the next, with the result that the same symbol has two different values within a single expression. Obviously, this may lead to confusion and hard-to-find bugs. Therefore, you should avoid the overuse of `volatile` data.

Debugging threads

During program execution, the *Threads* window provides more information about the threads of your program. The Threads window is available during a debugging session, whenever execution has been suspended (for example, at a breakpoint). While you are debugging and execution is suspended, all threads are suspended until execution resumes.

- **To see the Threads window:**

1. On the **Debug** menu of the code editor, click **Threads**.

The Threads window contains an entry for each thread. Each entry gives the name and ID number of the thread, and the current thread has the *Current* state.

Operations on threads

Operations on threads

To perform actions with the Threads window, you first have to click the name of a thread in the list of threads. Then the **Thread** menu offers the following items:

Freeze
Freezes the selected thread. If you resume executing the program, this thread will *not* begin executing. In other words, **Freeze** manually suspends the selected thread.

Thaw
Reverses a **Freeze** operation.

Make Current
Lets you start debugging the selected thread. The display of the execution point, variables, or other debugging information switches to the selected thread.

Part III. Adding and creating components

The chapters in this part describe how to add Java and ActiveX components, including the third-party components included with PowerJ, and how to create JavaBeans components.

Chapter 16. Using other Java components

This chapter explains how to use JavaBeans components or other Java components. For a step-by-step tutorial that shows you how to install and use Java components that come with PowerJ, see Adding and using Java components.

Note: In this chapter, as in general usage, the term *Bean* refers to a JavaBeans component.

Using an existing JavaBeans component

JavaBeans components are standardized, reusable software components that can be integrated and used with the PowerJ design-time environment to assemble powerful applets and applications. A JavaBeans component is basically a Java class definition (a class file) which conforms to the JavaBeans component standard for specifying properties, methods, and so on. They are usually stored in a Java Archive, or JAR, with many supporting classes and icons.

JavaBeans components serve the same purpose as Microsoft Windows ActiveX components: custom-written components which are designed to be reusable in a number of contexts. However, since JavaBeans components are written in Java, they can be used on any system that supports Java, not just on Windows systems.

This guide does not explain the design principles underlying JavaBeans components. For more information, see the most recent release of the JDK. The "official" web site for the JavaBeans component specifications is:

`http://java.sun.com/beans`

This site provides the technical specifications for JavaBeans component, as well as an overview of the JavaBeans component concept and tips for using JavaBeans components.

The standard PowerJ components are all JavaBeans components.

To use a JavaBeans component in PowerJ, you need to add the component to the PowerJ palette, then start using it as you would use any other component. You can use the same design-time facilities, such as drag-and-drop programming and the Object Inspector, for JavaBeans components as you do for standard PowerJ components. For more information on using standard components, see Adding objects to a form.

Note: The JavaBeans component standard was defined for version 1.1 of the JDK, and as such only 1.1 components can be Beans. However, a 1.02 component can be a *transitional Bean* if it adheres to the standard. You can program to any Java class file in PowerJ, but non-standard Java components will not expose all of their functionality in the design environment.

Adding a JavaBeans component to PowerJ

Java component wizard details

Adding a JavaBeans component to PowerJ

PowerJ makes it easy to incorporate existing JavaBeans components into the PowerJ environment. For example, suppose that someone in your company has written a Bean designed to work with your company's database. You can attach this Bean to your PowerJ session by adding the Bean to the **Database** page of the component palette. Once you have done this, you can use the Bean just like any of the standard PowerJ components: you can place it on forms, set properties by property sheets, and so on.

Before you can attach a JavaBeans component to PowerJ, you must have the following information:

- The name of the Bean's class, JAR, or ZIP file.
- The page on the component palette where you want to place the Bean. By default, PowerJ creates a new page named **Classes** and places the Bean there.
- An icon to represent the Bean on the component palette. PowerJ can produce a default icon, let you browse for icons available on your system, or invoke the image editor so you can create a new icon. You must have a large icon (24x24 pixels) and a small one (16x16 pixels).

The process of adding a JavaBeans component to PowerJ is controlled by the Java class component wizard.

◆ **To add a JavaBeans component to PowerJ:**

1. From the **Components** menu, click **Add Java Component**. PowerJ displays the Java Class Component Wizard.
2. In the **Class/Zip/Jar file** field, enter the full path of the class, JAR, or ZIP file where the new Java component is stored. To locate the file on disk, click **Browse**.
3. Select the **Component palette** version that you want, either 1.02 or 1.1, according to the version of Java that you are using. Some Java components, including JavaBeans components, only work with version 1.1, because they use classes that are not included in JDK version 1.02.
4. Click **Finish**.

After you click **Finish**, PowerJ attempts to build the Bean from the file and add it to the component palette. If this is successful, the icon for the Bean will appear on the component palette page that you specified.

Java component wizard details

The default wizard options are sufficient in most cases, but this section provides an explanation of each of the options in case you need to change them.

Options on page 1

Component palette

Allows you to select the version of the JDK for which this component will be used. Normally you should add components to the component palette that corresponds to the version of the JDK that they use. 1.02 components will usually work in 1.1, but not all 1.1 components will work in 1.02.

Page on the component palette

Specifies where the component will be placed on the component palette. This option allows you to group related components together. You can select an existing page from the list, or create a new one by typing a new name.

Class/Zip/Jar file

Specifies the name of the file to be added to the palette. If you select a class file, an icon will be added to the palette for that class. Note that the class must be marked as *public*.

JAR files are Java Archives, the standard storage types for JavaBeans components. They generally contain supporting files in addition to the Bean's main class files. These files can be images, property editors, customizers, serialization files, BeanInfo classes, and anything else that the Bean designer chooses. One special file, `manifest.mf`, is often included. It allows PowerJ to determine which of the classes will be placed on the component palette. If that file is not included, all class files except property editors, customizers, and BeanInfo classes are added to the component palette.

ZIP files contain only class files and no file to determine what classes go on the palette. When you add a ZIP file, PowerJ adds all class files except property editors, customizers, and BeanInfo classes to the component palette.

Display non-visual classes on the tool palette

Places an icon on the palette even for classes that do not have a visual representation. This option is only used for ZIP files because they do not have manifest files to specify what classes will go on the palette. Turn this option off if your ZIP file contains many non-visual classes that are not used at design time.

Use Fast Java class scanning

Uses PowerJ's own scanning facilities, rather than using introspection. Introspection can be very slow, and leaving this option turned on can decrease the time it takes to add a component without affecting the result.

Options on page 2

Append to ClassPath

This option is used only while PowerJ is adding the component to the palette, and not at design time or run time. You will rarely need to add anything to this field.

It allows you to specify the location of a ZIP file or other folders containing class files used by this component, but not accessible in its tree.

Copy to UserClasses

Copies the class files associated with the Bean to the UserClasses folder so that they will be accessible at design and run time. If you are adding a JAR file, this option will unzip the JAR and place its contents under UserClasses.

It is best to leave this option off, since leaving it off decreases the amount of disk space required for this component. However, you must quit and restart PowerJ before the class will be usable, because the classpath cannot be changed while the Java VM is loaded by PowerJ.

Note: If you turn off **Copy to UserClasses**, PowerJ automatically adds the JAR, ZIP, or class files your PowerJ class path, and to the runtime classpath for the applets that use the Beans. You must restart PowerJ before you use the Bean.

Options on page 3

Page three allows you to set the icon that will be used for the component palette, object inspector, and objects window. Most commercial Beans have their own icons; PowerJ will use them if they are provided regardless of the settings on this page. However, for classes that don't provide icons, you may provide one. In the case of ZIP files, the same icon will be used for all components.

You must have a large icon (24x24 pixels) and a small one (16x16 pixels). The large and the small icon should be stored together in a single `ICO` file. If you invoke the Image Editor to edit the icons, you see the 16x16 version of the icon first. Use Add Icon Image to a 24x24 icon image to the icon file. To switch between the different sized images, use the **Select Icon Image** entry of the **Edit** menu in the Image Editor.

JavaBeans components design-time properties

Most of the aspects of using JavaBeans components in PowerJ are exactly the same as using standard PowerJ components, which themselves are also Beans (though with extra PowerJ design-time information). For example, you can access the events and program methods for JavaBeans components in the same way as for standard PowerJ components.

However, to set the design-time properties of JavaBeans components that you add to PowerJ, you must follow different procedures than you use for PowerJ components. Many properties can be edited directly in the Object Inspector, as explained in Object Inspector. You can access the Object Inspector by selecting the Bean on the design-time form and pressing F4.

Note: The Object Inspector includes an entry for **SerializationData**. Do not modify that entry!

Many Beans have *customizers* and *property editors* to assist you in modifying design-time properties that cannot easily be modified by typing text into the Object Inspector. Customizers are full-blown property sheets, and property editors are simple, one-property, editor windows, useful for editing complex properties such as an object's color.

Customizers
Property editors

Customizers

Customizers are analogous to property sheets in that they both allow you to set properties for an entire Bean in a single dialog box. They often allow you to change properties that cannot be easily changed with the Object Inspector. The results of your changes are immediately reflected by the design-time component.

The JClass JCChart that is pre-installed with PowerJ provides a good example of a customizer. To access the customizer for the component, you must place the component on a 1.1 applet, then follow the steps below.

- **To access an object's customizer:**

1. On the design-time form, right-click the component. PowerJ displays the components popup menu.
2. On the popup menu, click **Customize**. PowerJ will display the Bean's customizer.

Note: Customizers are Java programs and as such may take some time to be displayed.

You may manipulate the Beans properties on the customizer and the results will be displayed by the Bean on the design-time form.

Property editors

Property editors allow you to change properties that cannot easily be changed with the Object Inspector. For example, it is convenient to have a visual color well from which you can select the color you want; the color well is an example of a property editor.

Property editors are defined by the Bean itself, and will modify properties in the design-environment. You can tell if a property has a property editor in the Object Inspector; properties with editors have three dots (…) at the right side of their entry in the inspector. You must select a property in the inspector to see the three dots.

- **To use a property's property editor:**

1. Click the JavaBeans component on the design-time form and press F4 to open the Object Inspector.
2. Select the **Properties** page of the Object Inspector, and select the property with an editor.
3. Click the button with three dots at the right side of the property's entry. If the property does not have these dots, then it does not have a special property editor and you can enter its value directly into the object inspector.

If a property has a property editor, PowerJ will use the editor in one of three ways:

- PowerJ will present a text field where you can enter a String value for the property.
- PowerJ will present a drop-down list where you can select a value from a list of possibilities.
- PowerJ will open a dialog window defined by the property editor.

Adding database capabilities to Beans

You may extend an existing Bean to allow it to be bound to a database. For more information on this process, see Data-aware Beans.

Chapter 17. Using ActiveX components and servers

This chapter explains how to:

- Use ActiveX controls (also called OLE controls or OCXs) as components that interact with Java applets on your web pages.
- Use ActiveX server components as programmable extensions of your servlets.

ActiveX controls are based on OLE and ultimately on the COM architecture. The specification for ActiveX controls is a simpler subset of the OLE custom control specification, so that ActiveX components may be smaller and faster than controls that meet the full OLE control specification. As a result, they can be downloaded more quickly and used more easily over the Internet. Both OLE controls and ActiveX controls use the OCX extension.

Note: ActiveX controls cannot be hosted by every browser. Users of your applications need Microsoft Internet Explorer version 3 or newer, or Netscape Navigator 3 or newer with an ActiveX plug-in.

OLE and ActiveX

Object Linking and Embedding (OLE) is a standard that provides a way to share functionality between applications. An ActiveX control (also called an OCX) is a standardized component type which can be used by your Java applets on a web page. ActiveX controls are provided by a large number of vendors and cover a wide range of functionality.

ActiveX controls are similar to native PowerJ components in that the interface to them is implemented as properties, methods, and events. Unlike native components however, with the current implementation ActiveX controls do not appear within the applet's window. ActiveX controls are embedded in a web page along with a Java applet. You can manipulate the ActiveX control in your Java applet by invoking the control's methods and setting its properties.

When the ActiveX control triggers an event, the event is relayed to your Java applet using Visual Basic Script. PowerJ automatically manages the relaying of events so that you can invoke methods and respond to events in the same way that you do with native Java components.

Some ActiveX controls, such as timers, are not visible on the web page, but are instantiated directly by the Java applet. Once you add these controls to your palette, you use them as you would any native control.

ActiveX server components are non-visual libraries that conform to the ActiveX standard. Server components are used by a web server to perform server-side operations. By using ActiveX server components on a web server you can access databases, perform business logic, and so on. Because server components have a standardized interface, they can be used by PowerJ and any development environment that supports ActiveX.

This chapter shows you how to use visual and non-visual ActiveX controls in Java applets and server components in servlets.

Note: ActiveX server components can be created with Power++.

Getting started with ActiveX controls

ActiveX controls are used in much the same way as other components in PowerJ. An ActiveX control is not an application that can be run on its own; it is a control that can be used in other applications by invoking its methods, setting its properties, and responding to its events.

When you add an ActiveX control to the component palette, PowerJ creates a native Java interface to it. The methods and properties of that control are added to the PowerJ Reference Card and are available for drag and drop programming. If the documentation is provided as online help it will be directly accessible from the Reference Card.

After the ActiveX control has been added to the component palette, you use it as you would any native PowerJ component except that ActiveX controls do not appear on Java forms. ActiveX controls are either visual or non-visual. Visual controls are displayed on a web page along with a controlling Java applet.

The control displayed on the web page can be manipulated through its methods and properties by the Java applet. PowerJ automatically gives the applet access to the control, as described in Adding a component to a Java form.

Non-visual controls, such as timers or business logic components, are created by the Java applet and used in code, but not displayed.

ActiveX controls are embedded in a web page and controlled by a Java applet on the same page.

Registering ActiveX controls

Before you can use an ActiveX control, it must be registered with the system. When you register a control, its description is stored in the system registry. This information allows all programs (including those you build with PowerJ) to use the control. Some controls provide an installation program that registers them for you. If this is the case, or if the control has already been registered by some other program, skip the rest of this section.

- **To register an ActiveX control:**

1. In the **Tools** menu of the main PowerJ menu bar, click **Register ActiveX controls**. This displays a list of all ActiveX controls currently registered.
2. If the control is already in the list then it is already registered so you should skip the rest of these steps. Otherwise, click **Register**.
3. Locate and select the control's library (for example, the OCX file or DLL file containing the control). When you have found the library, click **Open**. This opens the file and registers the control.
4. Close the dialog.

Note: Registration is a system-wide operation; registered controls become available to all programs, not just PowerJ. Similarly, PowerJ can use controls that have been registered by other applications.

If you move an ActiveX control's file, you must unregister it, then re-register it in its new location.

Adding an ActiveX component to the palette

Adding an ActiveX control to the PowerJ palette integrates the control with the Reference Card and makes it possible for you to use objects of that type with your Java applet or servlet. When you add an ActiveX control to the component palette, PowerJ generates the class files needed to access the associated methods and properties.

The rest of this section describes the basic steps for adding an ActiveX control to the component palette. The same procedure applies to adding ActiveX server components.

- **To add an ActiveX control to the component palette:**

1. 

mponent(s). You

2. Choose the control that you want to add to the component palette.
3. Select the component palette from the list. You can select JDK version 1.02 or 1.1. If you want to use the control with applets in both versions of the JDK, you must add the component twice.
4. Choose the page on the component palette where you want the new control(s) to reside. You may select a page from the list or create a new page by typing the name of the new page.
5. Click the item that you want to add. Click **Finish**.

At the bottom of the first page of the Component Wizard are the options: **Show type libraries** and **Show automation servers**. These options specify which type of ActiveX information should be displayed. The ActiveX component or server specifies whether it is registered as a type library or server. Most ActiveX controls will be registered as type libraries. Some automation servers are registered as type libraries, some as automation servers. If an item is shown in both lists, you may choose either with the same result.

If the control you wish to register is not in either list, make sure it is registered. For information on registering controls, see Registering ActiveX controls.

Some automation servers provide a type library that is not registered with the system. If the type library you seek is not in the list, click the **Browse** button to search for it on disk. A type library is usually stored as a file with the `TLB` extension in the same folder as the automation server.

When PowerJ has finished, the controls defined by the type library appear on the component palette. There may be a number of controls for each type library. Once an ActiveX control resides on the component palette, you can use it in your projects.

Adding a component to a Java form

An ActiveX component resides on a web page rather than residing on a Java form. To use an ActiveX component in your Java applet, you place a component proxy on your Java form and use the proxy to invoke the component's methods. When the applet is being used in a browser, the control's methods are invoked by your applet through the proxy object.

Place the component that you want to use on your Java form. An icon representing the component will appear on the form. For Java applets, you should open the component's property sheet and turn on the **Externally Created** property. This property causes PowerJ to automatically attach the component to your Java applet.

Attaching components to Java forms

When you put an ActiveX control on a Java form and turn on its **Externally Created** property, PowerJ automatically adds the control to the applet's Web page, and creates a proxy object in your applet. To manipulate the ActiveX control, you invoke methods of the proxy object, which in turn invokes the corresponding methods of the ActiveX control.

The proxy object must be *attached* to the ActiveX control before the control can be used. To accomplish this, PowerJ generates VBScript in the Web page that is executed when the page is loaded. This VBScript calls a method on the applet, also created by PowerJ, which accepts a reference to the ActiveX control on the Web page. In other words, when the Web page is loaded, the VBScript attaches the ActiveX control to the applet so that the control can be used.

For example, if you place a Formula One worksheet component on your form and turn on the **Externally Created** property, PowerJ will declare a private data member as part of your Java form:

```
    private _DVCF1        _DVFC_102_1;
```

Depending on whether you are targeting Java 1.02 or Java 1.1, the data member will be named `_DVFC_102_1` or `_DVFC_11_1`. The rest of this chapter refers to Java 1.02 (replace _102_ with _11_ if you are using Java 1.1).

PowerJ will also create a read-only public method called **Set_DVFC_102_1**:

```
public void Set_DVFC_102_1( Object o )
    {
        _DVFC_102_1 = (_DVCF1)o;
    }
```

Since the component resides on the web page, PowerJ automatically generates an `OBJECT` tag in the default web page for your applet:

```
<OBJECT ID="HTML_DVFC_102_1" WIDTH=200 HEIGHT=100
 CLASSID="CLSID:042BADC5-5E58-11CE-B610-524153480001">
</OBJECT>
```

This tag will be replaced with a Formula One worksheet when it is viewed with an ActiveX-enabled web browser. When the web page is loaded by the browser, the **Set_DVFC_102_1** method must be invoked to provide your applet with the component. PowerJ generates the following Visual Basic script which invokes the **Set_DVFC_102_1** method when the web page is loaded:

```
<SCRIPT LANGUAGE=VBScript>
Sub window_onLoad
    document.applet.Set_DVFC_102_1( HTML_DVFC_102_1 )
End sub
```

```
</SCRIPT>
```

When the web page is loaded, your applet is started, the ActiveX component is created, and the Visual Basic script is executed to attach the ActiveX component to your applet's form.

Effects of being externally created

The following list summarizes the effects of turning on the **Externally Created** property:

- The ActiveX component is embedded as a separate object in an HTML page.
- VBScript provides the ActiveX's reference to the applet.
- The component works for Java applets only.
- The property works for Visual and non-visual ActiveXs.
- The application requires Microsoft Internet Explorer or Netscape with ActiveX plug-in.

The following list summarizes the effects of turning off the **Externally Created** property:

- The ActiveX is loaded using native calls.
- The component works for Java applications only or trusted applets.
- The property works only for ActiveX Servers (non visual ActiveXs).
- The application requires the Microsoft Virtual Machine.

Coding ActiveX controls and servers

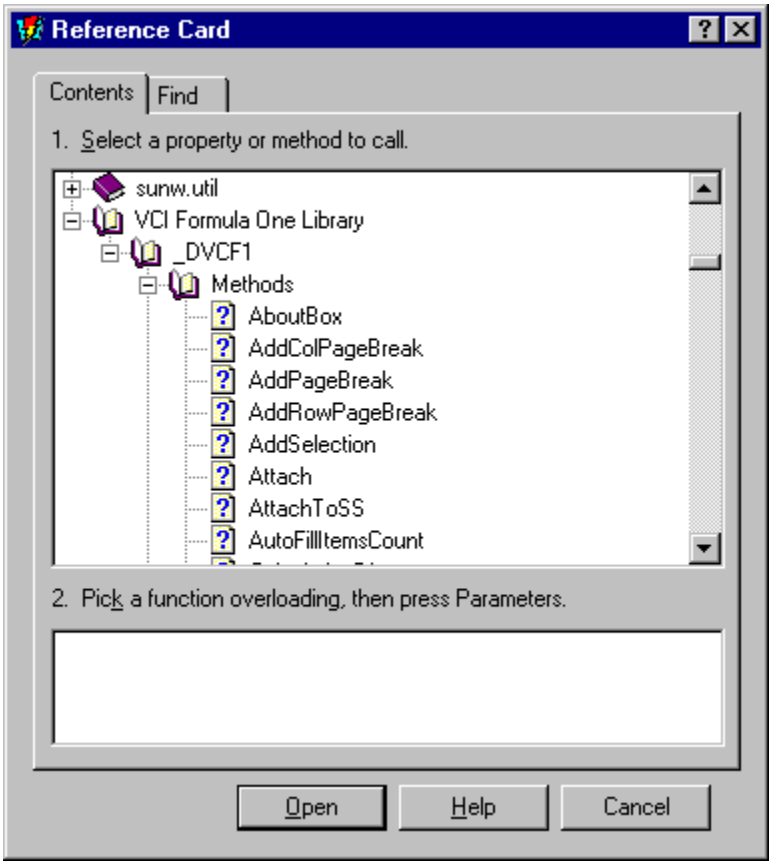
Once you have added an ActiveX control to your Java form and turned on its **Externally Created** property, you are ready to start integrating the ActiveX component with your applet. The ActiveX control is just like any native Java component; it has methods and properties and can trigger events. This section describes how to invoke methods, set and retrieve properties, and handle events.

Methods and properties of ActiveX controls

An ActiveX control's methods constitute the programming interface that allows you to program the control in your code. You can invoke these methods in much the same way you invoke the methods of any control in PowerJ. You can also use the Reference Card and drag and drop programming to simplify writing the code.

Using the Reference Card

Once you have added an ActiveX control to the component palette, its methods are available in the Reference Card. You access these methods and properties in the usual way. The following figure shows the Reference Card displaying the methods and properties of the ctSlide ActiveX component from Gamesman:



You can access online documentation for the methods of an ActiveX control (as long as the control vendor provides this documentation). To access information about a method, select the method and click **Help**.

You can set properties of an ActiveX control in the same way as you invoke methods. Properties are placed in the Properties group in the Reference Card. As with standard Java components, you invoke the property's **set** method to change the value, or the property's **get** method to retrieve the value.

For example, the **setTextRC** method of the Formula One worksheet changes the text in a specified cell:

```
// Set the text in Row 1 Column 2 to "New Text":
_DVFC_102_1.setTextRC( 1, 2, "New Text" );
```

Handling ActiveX events

ActiveX controls trigger events in the same way as native Java components. For example, an event may be triggered when the user clicks the control or when a timer expires. PowerJ allows you to program the event handlers of ActiveX controls within your Java applet.

- **To program the event handler for an ActiveX control:**

1. Right-click the ActiveX control's icon on the design-time Java form.
2. On the popup menu for the control, point to **Events** and click **More**. PowerJ will display the Events page of the Object Inspector for the control that you selected.
3. In the Object Inspector, double-click the event that you want to handle. PowerJ creates the event handler in your code.

The code that you write in the event handler is executed when the control triggers the event. Because the ActiveX control resides on a web page and not on the Java form, the event must be relayed to the Java applet using Visual Basic Script. PowerJ automatically generates the script required to call your event handler when an event is triggered.

For example, the following Visual Basic Script in an HTML document relays the **Click** event from the Formula One worksheet to your Java applet:

```
<SCRIPT LANGUAGE=VBScript>
Sub HTML_DVFC_102_1_Click( nRow, nCol )
  call document.applet.Call_DVFC_102_1_ClickEvent( nRow, nCol )
End sub
</SCRIPT>
```

Running your applet or application

If your applet is using an ActiveX control on a web page, you must run the applet in a web browser that supports ActiveX controls.

- **To change the environment that is used to run your applet:**

1. In the **Run** menu on the main PowerJ menu bar, click **Run options**. If the current project contains more than one target, PowerJ displays a dialog box to determine whose options you want to set. Click the name of the applet target, then click **OK**.
2. On the **General** page of the run options, select the option labeled **Use a web browser**.
3. Click **Configure**. This opens a dialog box that lets you choose from a set of recognized web browsers, or specify your own browser. Click **OK** when you have specified a browser.
4. Click **OK** to close the run options dialog.

Once you have configured the web browser, you can press F5 to run your applet. PowerJ will open your applet's web page using the web browser that you specified.

Because ActiveX controls are executed on the client machine, they have full access to the client computer. Most web browsers inform users when an ActiveX control is instantiated and allow them to decide whether the control will be allowed to execute. When a web page with an embedded ActiveX control is loaded, you will see a dialog box indicating that a potentially dangerous application is running. If you want the ActiveX control to run, you must appropriately answer the dialog box. You may apply for a certificate that will identify and authenticate your controls to users of your applets.

Important: In order for a user to run an applet that contains an ActiveX control, the applet must be considered a *trusted applet* on the user's machine. For more on trusted applets, see Trusted applets.

Chapter 18. Creating JavaBeans Components

This chapter describes how to create your own JavaBeans components. If you would like a quick introduction to this topic, see Creating a Java component.

Note: In this chapter, the term *Bean* refers to a JavaBeans component.

JavaBeans components

JavaBeans components are standardized, reusable software components that can be integrated and used with the PowerJ design-time environment to assemble powerful applets and applications. This section gives a high-level description of the capabilities of JavaBeans components. The rest of this chapter explains how you can create and deploy JavaBeans components, and add data-bound behavior to your Beans.

Beans are Java classes that are designed in a standardized way. As such they can serve the same purposes as classes, and can be easily integrated with PowerJ. For example, you can encapsulate business logic, database transactions, or a commonly used GUI control as a JavaBeans component and reuse it in any of your Java applets or applications. The JScape and KL Group components that are preinstalled with PowerJ are good examples of GUI components. PowerJ also comes with Java database components which provide database access.

Beans are much like native PowerJ components, in that you can put them on your design-time forms, customize them, and use their methods through the Reference Card and drag-and-drop programming. A Bean can have methods that you call at run time, and properties that you set at design time or run time. In addition, a Bean can have events that notify the applet when something of interest has occurred, such as a mouse click or a database update.

Creating a simple JavaBeans component in PowerJ is just a matter of using the class, method, property, and event wizards to create Java classes with methods, properties, and events. More advanced Beans include *customizers*: supporting classes that allow the Bean user to modify the Bean's properties at design time. You can create customizers in PowerJ by writing Java classes that conform to the Beans standard.

Beans can also be bound to a database using the DataHelper class provided with PowerJ.

Normally Beans are packaged in JAR (Java Archive) files, and include images and other files to support the design-time environments of tools such as PowerJ. PowerJ has a JAR target type to help you deploy your Beans.

For more information on JavaBeans components, including a detailed specification, refer to the JavaSoft web site:

`http://java.sun.com/beans/`

The JavaBeans Advisor also has practical tips on creating robust Beans:

`http://java.sun.com/beans/Advisor.html`

Note: When you develop an applet in PowerJ, you begin most of your work from the form design window. When you develop a JavaBeans component, you usually work from the Classes window. In the Classes window you can create new classes, add methods, properties, and events, and quickly find the code for a specific method.

Creating a new Bean

There are two possible approaches to creating a Bean:

- You can create it using the **Bean** target template. This creates a new JDK 1.1 target with a class that forms the basis of a simple Bean. The Bean is based on the `java.awt.Panel` class.
- You can create it using the **Java - Classes** target template. This is a slightly more generic approach to creating the Bean—it does not set up as much of a starting structure. You have to do everything from scratch, rather than beginning with the template provided by the **Bean** target.

Most users will find it easier to start with the **Bean** template. However, if you do not want to base your Bean on `java.awt.Panel`, you must start with the **Java - Classes** target.

Using the Bean target template

Creating a **Bean** target begins in the same way as creating any other type of target:

- **To create a new JavaBeans component using the Bean target template:**

1. From the **File** menu, point to **New** and click **Target**. PowerJ opens the Target Wizard.
2. From the list of target types, click **Bean**. Then click **Next**.
3. Under **Target Name**, type the name of your Bean. The target name must not have any spaces, and must start with a letter.
4. Click **Finish**.

This creates the **Bean** target. The target will have a single form, shown in a design window.

Note: When the form first appears, it contains a number of labels explaining how to proceed with creating the **Bean** class. Typically, you would delete these labels and then place components on the form.

When a **Bean** target is created, it contains a class that will serve as your Bean component. To begin with, the class is named `_RenameMe`. You must change this to the desired name of your Bean.

- **To change the class name:**

1. From the **View** menu, click **Classes**. This opens the Classes window.
2. Expand the entry in the left part of the window, until you see `_RenameMe`.
3. Use the right mouse button to click `_RenameMe`, then click **Rename**.
4. Type the name you want to give your Bean, then press ENTER.

When you rename the Bean, PowerJ changes various elements in the starting structure to match the name you've just given. Therefore it may take a few moments for PowerJ to finish the name-changing process.

Adjusting the Bean's class

At this point, there are several adjustments that you may want to make in the Bean's class. For example, you may want to specify a package name for the Bean. You may also want to specify that the Bean implements one or more interfaces. This can be done through the Classes window.

- **To adjust the Bean's class:**

1. In the Classes window, use the right mouse button to click the name of your class, then click **Properties**. This opens a property sheet for the class.
2. If you want your Bean to belong to a package, type the package name under **Package Name**.
3. If you want your Bean to implement one or more interfaces, type the names of the interfaces under **Implements**.
4. Use the check boxes to specify any other information about the Bean's class.
5. Click **OK**.

Now you are ready to begin defining properties, methods, and functions for your Bean.

Using a Java Classes target

If you do not use the **Bean** target template, you can create a Bean using a Java Classes target.

- **To create a new JavaBeans component using a Java Classes target:**

1. From the **File** menu, point to **New** and click **Target**. PowerJ opens the Target Wizard.
2. From the list of target types, click **Java - Classes**, then click **Next**.
3. Select the version of the JDK that you want to use, 1.1 or 1.02. Classes created with JDK version 1.02 will run under 1.1, but do not have all of the capabilities of 1.1. Classes created with version 1.1 will not necessarily run under 1.02. Click **Next**.
4. Under **Target Name**, type the name of the new Target. The target name must not have any spaces, and must start with a letter.
5. Click **Finish**.

Defining the Bean's class

You have just created a Java Classes target. Now you need to create a new Java class that will implement your Bean component. The Class wizard requires the following information:

- A name for the Bean class.
- An existing class on which the Bean is based. The default is Object, the most general Java class, but you can choose a different class if appropriate. For example, suppose you want to create a Bean which is an enhanced type of list. You can base your Bean on `java.awt.List`, then add new properties, methods, and events to support the enhancements you want to implement.
- A name for the package that will contain the Bean. You can leave this blank if you don't intend the Bean to be part of a package. You should choose a name that describes the group of components in which the Bean resides, and a name that will uniquely describe the group so that it doesn't conflict with other available packages.
- Whether or not the Bean implements an interface. For more information on interfaces, see Classes vs. interfaces.

- **To create a new Java class:**

1. Open the Classes window by pressing SHIFT+F3.
2. In the Classes window's **File** menu, point to **New** and click **Class**. PowerJ starts the Class Wizard.
3. Select the **Standard Java** class type and click **Next**. If you are developing a Bean that will not be visible at run time but uses database components or other non-visual components, you can select **Visual Class**. A visual class acts like a form at design time, in that you can place objects on the class; however a visual class cannot be displayed to the user at run time.
4. Enter a package name if you wish to use one.
5. Enter the class name. This name should be descriptive and unique within its package.
6. In the **Extends** field, type the name of the parent component. Use fully qualified component names such as `java.awt.Canvas`.
7. If the Bean implements an interface, type the name of the interface under **Implements**. In many cases you can leave this field blank.
8. If the Bean will be a public class, make sure **Public** is checked. You should normally turn this

option on.

9. If the Bean will be an abstract class, make sure **Abstract** is checked. In most cases this option should be off.
10. If the Bean will be an interface, make sure **Interface** is checked. In most cases this option should be off.
11. Click **Finish**.

PowerJ creates the new class definition in an editor window. As you can see, there is very little code involved. The code is added as you add methods, properties, and events.

Once you have created the target you should save the project.

Adding properties to your Bean

This section applies to any Bean component, whether you created it with a **Bean** target or **Java - Classes**.

Properties defined

Properties store information about components, such as the text stored in a string, the background color of a button, or the database name of a transaction. Properties are set by the Bean user at design time or at run time. At design time, properties are listed in the Object Inspector, while at run time the Bean user must invoke the *setter* method to change a property. Properties can be retrieved by the Bean user through the *getter* method. The TextField component, for example, has a **Text** property that can be accessed using the getter method **getText** and the setter method **setText**.

The Object Inspector itself regularly uses the setter and getter methods to change the properties of Beans at design time.

Often, the value of a property will be represented by a data member in the JavaBeans component class. For example, if you have a property named **MyValue**, you might create a data member named `_myValue` which holds the property's value. This makes it possible to create very simple inline property methods, as shown below:

```
// String _myValue;
String getMyValue() { return _myValue; };
void setMyValue( String newValue ) { _myValue = newValue; };
```

The above code assumes that **MyValue** has the String data type. Comparable code could be generated for other data types.

Note: If a property has its value stored in a data member as shown above, PowerJ can automatically generate inline property methods using the form of the preceding example. In more complex situations, you must write your own **get** and **set** methods to work with the property. You may also want to create several overloaded versions of a **get** or **set** method, in applications where there may be several useful calling sequences for the property method.

Step-by-step

Before you add a property to a JavaBeans component class, you need to decide the following information:

- A name for the property.
- A data type for the property. For example, if the property contains text information, you might choose the String class.
- Whether you want PowerJ to generate simple inline **get** and **set** methods or you prefer to write your own methods.

Once you have determined this information, you can create the property.

◆ **To create a property:**

1. Open the Classes window by pressing SHIFT+F3.
2. Expand the tree view so that your class is shown and select the class.
3. With the right mouse button, click the class. In the popup menu, point to **Insert**, and click **Property**. PowerJ starts the Property Wizard.

4. In the **Name** field on the Property Wizard, type the name of the property.
5. Select the data type from the list. Enter a data type that is used to store the value of your property.
6. Click **Next**.
7. If you want to write your own property methods, click **Member methods**, then type prototypes for the property functions you want to define.
8. If you want PowerJ to automatically create simple inline **get** and **set** methods, click **Member variable with inline Get-Set** and type in the name of the data member that will hold this property's value.
9. Click **Finish**.

PowerJ will open a code editor window that will let you enter code for the property methods.

Note: In some cases, you may want to define a property before you're ready to write the property methods. You should note that the code will not compile until you give the **get** method a `return` statement that returns a value of the appropriate type.

If a property is not meant to be set at design time, you can have it suppressed by creating your own BeanInfo class. For more information, consult the JavaBeans component specification.

Deleting properties from a Bean

You can delete a property in the same way you delete any other object.

- **To delete a property from a JavaBeans component class:**

1. In the Classes window, use the right mouse button to click on the property, then click **Delete**.

There is one special consideration: if you had the Property Wizard create a data member to hold the value of the property, PowerJ does *not* delete the data member when you delete the property. You must delete the data member manually in the code for the JavaBeans component class.

Adding methods to your Bean

The process of adding a method to a JavaBeans component class is similar to adding a property. Before you add the method, you must decide the following information:

- The name of the method.
- One or more prototypes for invoking the method.

◆ **To add a method to a JavaBeans component class:**

1. In the Classes window, use the right mouse button to click the class where you want to define the property, then click **Insert**, then **Method**. This opens the Method Wizard.
2. Under **Name**, type the name of your method.
3. Under **Prototype(s)**, type one or more prototypes for invoking your method.
4. Click **Finish**.

PowerJ opens a code editor window where you can begin entering code for the method.

Adding events to your Bean

This section explains how to add events to your JavaBeans components. For a step-by-step tutorial on adding an event, see Adding an event.

The Java event model

Events in Java are handled using *event sources* and *event listeners*. Your Bean is an event source that notifies registered listeners whenever an event occurs. When you are using JavaBeans components in PowerJ, all of this happens automatically for you; you only have to write the code for the event handler. When you are adding events to your own Bean, however, you need to understand how these event mechanisms work at a much deeper level.

Parts of an event

When you make your Bean a source for a new event, you must specify which event listener will handle the event when it occurs. You can either use an existing event listener, or you can create a new one. Using an existing listener is the simplest approach because you do not need to define a listener class and an event state object. For example, you could choose to trigger the **PropertyChange** event when an important property changes.

If you want to create a new listener, you must define the following components:

1. An *EventListener* interface. One listener can be used for several related events, just as the MouseListener is used to handle all mouse events.
2. An *event state object*. Event state objects store information about the event. For example, when a mouse is clicked, the event state object relays which button was clicked, where the click occurred, and so on.
3. *Add and remove listener* methods, to allow applets to register and unregister for particular events. If an applet wants to receive notification of mouse click events, for example, it calls **addMouseListener**. To stop receiving notification of events, it calls **removeMouseListener**. When an applet is registered as a listener, it must handle all of the events produced by the EventListener interface. That is, it must implement all of the methods defined by the EventListener interface. In the case of the MouseListener class, that means handling all mouse events.

   PowerJ automatically creates the add and remove registration methods when you make your Bean a source for a new event.
4. Code to notify the registered event listeners whenever an event occurs. This is also known as *firing the event*. PowerJ creates example code that you can copy and quickly modify to notify event listeners when an event occurs.

It is important to note that any number of events can be handled by one EventListener. In other words, your EventListener interface can contain methods for several events for which your Bean may wish to be a source. In practice, events are generally grouped logically so that related events are handled by the same listener. As described in the example above, all mouse events are handled by the MouseListener interface.

Before proceeding further, you may want to run through the tutorial in Creating a Java component, if you have not already done so. It quickly guides you through the process of making your Bean a source for a new event, while the rest of this chapter describes each step in detail.

JDK 1.02 vs. 1.1

The event models in JDK version 1.02 and 1.1 are very different. However, the components that you write with PowerJ will all conform to the 1.1 JavaBeans component model. This ensures that your events will work under either version of the JDK.

Note: If you use code in your event handlers that is specific to one version of the JDK, it will not execute properly under the other version. For more information, see JDK 1.02 vs. JDK 1.1.

Making your Bean an event source

Using an existing listener

In order to make your Bean an event source for a particular event, you must choose an event listener that will handle the event when it is triggered. The simplest route is to use an existing listener class, such as java.awt.event.ActionListener.

- **To make your Bean an event source using an existing listener:**

1. Open the Classes window by pressing SHIFT+F3.
2. With the right mouse button, click the Bean's class.
3. On the popup menu, point to **Insert** and click **Event**. PowerJ displays the Event Wizard.
4. Click the option labeled **Implement an existing listener**.
5. From the list of existing listeners, choose the listener that you want to override. For example, for a JDK 1.1 project you could select java.awt.event.ActionListener.
6. Click **Finish**.

PowerJ generates the following methods; substitute the name of the listener that you chose wherever *Action* appears:

addActionListener
Adds a client applet or application to the list of registered listeners. Any applet that registers as a listener will receive notification of the events defined by the listener for a particular component.

removeActionListener
Removes a client applet or application from the list of registered listeners.

fireActionPerformed
Calls the **actionPerformed** method on all registered listeners whenever an Action event occurs. You can trigger the **Action** event in your code by preparing an event state object and calling the **fireActionPerformed** method.

Note: The **Action** event is a *semantic event*, which means that it corresponds to the most important event of your component. When a button is pushed, for example, the **Action** event should be fired.

Many listener interfaces contain methods for more than one event. For example, the MouseListener interface contains methods to handle all of the mouse events. In cases like this, PowerJ generates one *fire* method for each event that the listener interface includes. You can trigger an individual event by calling its corresponding *fire* method.

Defining a new listener

If no existing event listener is sufficient for your purposes, you may choose to define a new event listener. You would do this, for example, if you needed to provide more information in the event state object of an existing event, or if you create an event that doesn't already exist.

- **To make your Bean an event source using a new listener:**

1. Open the Classes window by pressing SHIFT+F3.
2. With the right mouse button, click the Bean's class.

3. On the popup menu, point to **Insert** and click **Event**. PowerJ displays the Event Wizard.
4. Click the option labeled **Implement a custom listener**.
5. Type the name of your listener in the **Custom Listener** field. The example shown below uses YourEventListener as the name of the listener.
6. Click **Finish**.

PowerJ generates the following methods; substitute the name of the listener that you chose wherever *YourEvent* appears:

addYourEventListener
Adds a client applet or application to the list of registered listeners. Any applet that registers as a listener will receive notification of the events defined by the listener for a particular component.

removeYourEventListener
Removes a client applet or application from the list of registered listeners.

PowerJ also generates a commented block of code, which is an example of the code you will need to write to notify the event handlers each time the event occurs:

```
//  You must fire an event for all events that your listener is
//  listening for. A sample of how an event can be fired is
//  shown below

//    protected void fireCustomEvent(
//                          YourEventStruct __eventData ) {
//        int               count;
//        java.util.Vector  l;

//        synchronized( this ) {
//           l = (java.util.Vector)_YourEventListeners.clone();
//        }

//        count = l.size();
//        for( int i = 0; i < count; i += 1 ) {
//            ( (YourEventListener)l.elementAt( i ) ).
//                        CustomEvent ( __eventData );
//        }
//    }
```

Event listener interfaces

An event listener interface is a Java class which contains one method for each event the listener listens for. In order for an applet to implement the listener interface, it must have code for each of the listener's events. Event listeners such as MouseListener or ActionListener are used by many components, and you may find it useful to reuse them with your own components.

However, if you want to define a new collection of events, you need to define a new event listener interface by following these steps:

1. Create the event listener interface (you only do this once).
2. Add a user function to the listener with the same name as the event. The function should have one parameter: the event state object. You can use an existing class for the event information or you can create a new one.
3. Copy the supplied code sample and modify it to call the user function created in step 2.
4. Call the *fire* method wherever it is necessary to notify the event handlers listening for the event. For example, if you are triggering the **Action** event in response to an alarm, you would call **fireActionPerformed** when the alarm goes off.

Event state objects

Event state objects are classes that contain specific information about an event. For example, when a mouse is clicked, the event state object relays which button was clicked, where the click occurred, and so on. When you send notification of the event's occurrence to the event handlers, you have the choice of using an existing state object from the JDK or the Powersoft library, or creating your own state object.

To make that choice, you need to determine what data will be contained by the state object. The state object should contain whatever information will be useful when the event is handled. For example, a mouse click event contains the coordinates of the event and something to indicate which mouse button was clicked.

If you find a state object that almost fits your needs, you should create a class that inherits from that object and add whatever extra data you require. All JDK 1.1 event state objects are eventually derived from `java.util.EventObject`, so if you find no other useful base class, use the EventObject class.

Before you create an event state object, you need to know the following:

- The name of the class. The name should end in *Event*, as in MouseEvent and ActionEvent.
- What data you need to relay to the event handlers when the event is triggered.
- The parent event state object. This is typically derived from the JDK 1.1 class `java.util.EventObject` or one of its descendants.
- The package name, if any. You can use the same package as the object that fires the event.

◆ **To create an event state object:**

1. Open the Classes window by pressing SHIFT+F3.
2. Under the **File** menu, point to **New** and click **Class**.
3. Select **Standard Java** and click **Next**.
4. Enter the package name.
5. Enter the class name. This name should end in the word `Event`.
6. Enter the object's base object in the **Extends** field. With JDK 1.1., this should be `java.util.EventObject` or a descendant of this class.

Your event state object should have at least one constructor, and should call its parent constructor:

```
public class MyEvent extends java.util.EventObject
{
    //
    // MyEventMethod
    //

    public MyEvent( SourceControlType source )
    {
        super(source);
    }

    // add your data members here
}
```

Once you implement an event state object and provide it to users, you should not change its interface. For this reason, you should provide access methods to any internal data rather than exposing the data

directly. In addition, you should use standard Java naming conventions for setter and getter methods.

Deleting an event

When you delete an event, PowerJ automatically deletes any associated methods that it generated. Extra code that you write, such as when you define a new event listener, is not deleted automatically with the event. Listener classes are also not deleted when you delete events.

- **To delete an event:**

1. Open the Classes window by pressing SHIFT+F3.
2. Select the desired event and press DELETE.

Customizers

A customizer serves the same purpose as a PowerJ property sheet: it lets the Bean user set initial values for some or all of the Bean's properties, typically at design time. When you create a Bean customizer with PowerJ, the customizer will look like a normal form with text fields, labels, check boxes, and so on that help the user set the Bean's properties.

Creating a customizer form

The first step in creating a customizer for a Bean is to create the customizer form. The name of the form class must be the name of the Bean plus the word `Customizer`. For example, if you are creating a customizer for a class named `MyBean`, the name of the customizer form class must be `MyBeanCustomizer`.

- **To create the customizer form:**

1. On the **File** menu of the main PowerJ menu bar, click **New**, then **Form**. This opens the Form Wizard.
2. Under **What type of form do you want?** click **Customizer** then click **Next**. (**Note:** The **Customizer** form is only available for JDK 1.1 targets.)
3. Under **What do you want to call the new form?** type the name of the Bean itself followed by `Customizer` (for example, `MyBeanCustomizer`).
4. Click **Finish**.

You can then design the customizer form by placing components on it in the usual way.

Basic elements of customizer code

The code of the customizer form class must contain the following in the Imports section (at the beginning of the class definition):

```
import java.beans.*;
```

This imports various definitions for creating the customizer.

The default code provided when you create a customizer form contains the following in the Data Members section:

```
protected Object _BeanObject = null;
```

In this declaration, replace `Object` with the class name of your Bean component. The `_BeanObject` variable will be used to hold the Bean object that is being initialized by the customizer.

The customizer form also defines a variable named `_PropertyChangedListeners`. This corresponds to event listeners used in the customization process.

The setObject method

All customizers must have a method named **setObject**. This method is called when the customizer is invoked, in order to specify the Bean object that the customizer will be initializing. The default **setObject** has the following form:

```
public void setObject( Object obj )
{
    _BeanObject = obj;
    try {
        create();
    }
    catch( java.lang.Exception __e) {
        System.err.println( __e.toString() + " " +
            __e.getMessage() );
    }
}
```

You should change the first assignment to

```
    _BeanObject = ( MyBean ) obj;
```

where `MyBean` is the actual class name for your bean. The argument to **setObject** is the object that the customizer will be initializing. The basic action of **setObject** is to store this object in the customizer's `_BeanObject` data member.

The above code for **setObject** also invokes the **create** method for the customizer form. This opens the form and displays it to the user.

It is common for **setObject** methods to initialize various properties for the Bean object and/or for the customizer form.

Note: When you are creating your own customizer, you will use the standard techniques for adding **setObject** to the customizer class. See Adding new member functions for a description.

Property change listener methods

A customizer must implement methods for adding and removing property change event listeners. When PowerJ creates a customizer form, it automatically defines these methods as part of the customizer class.

Event handlers for objects on the customizer form

The final step in creating your customizer is to write event handlers for the objects on the customizer form. For example, suppose that the customizer form will offer a text field `textf_1` where users may type a String value for a property named **Prop** in the associated Bean. This text field would typically have two event handlers:

```
public void textf_1_objectCreated(
    powersoft.powerj.event.EventData event )
{
    textf_1.setText( _BeanObject.getProp() );
}

public void textf_1_textValueChanged(
    java.awt.event.TextEvent event )
{
    _BeanObject.setProp( textf_1.getText() );
    _PropertyChangedListeners.firePropertyChange(null,
        null,null);
}
```

- The **objectCreated** event handler initializes the value of the customizer's text field by obtaining the current value of the property in the Bean.
- The **textValueChanged** event handler calls the Bean's **setProp** routine to set a new value for the property, taken from the text of the text field. Then the event handler fires a **PropertyChange** event to indicate that the value has changed.

The above event handlers should be considered models for all the event handlers that you write for objects on your customizer form.

BeanInfo classes

A *BeanInfo* class provides information about the properties of a JavaBeans component. Typically, you will use PowerJ to make the BeanInfo class after you have finished making the Bean and its customizer.

The name of a BeanInfo class is the name of the Bean followed by `BeanInfo`. For example, if you create a Bean named `MyBean`, the name of the associated BeanInfo class should be `MyBeanBeanInfo`. The BeanInfo class should be in the same package as the Bean itself.

Note: In PowerJ, creating a BeanInfo class makes it possible to set properties for the Bean using the Object Inspector.

Defining a BeanInfo class

The first stage in making a BeanInfo class is to create the class using the Class Wizard.

- **To create the BeanInfo class:**

1. From the **File** menu of the main PowerJ menu bar, click **New**, then **Class**. This opens the Class Wizard.
2. Under **What type of class do you want?** click **BeanInfo** then click **Next**.
3. Under **Package Name**, type the name of the package that contains the Bean component.
4. Under **Class Name**, type the name of the BeanInfo class. This must be the name of the Bean component followed by `BeanInfo` (for example, `MyBeanBeanInfo`).
5. Click **Finish**.

Basic elements of BeanInfo code

Your code should contain declarations of the following form in the Data Members section:

```
Class _bean = null;
Class _customizer = null;
```

The `_bean` variable will refer to a Bean component and the `_customizer` will refer to the Bean's customizer.

The BeanInfo default constructor

Your BeanInfo class must have a default constructor: a public constructor that takes no arguments and initializes the data members of the BeanInfo object. You create this constructor through the Classes window.

- **To create a default constructor for the BeanInfo class:**

1. Open the Classes window (by clicking **Classes** in any **View** menu).
2. Use the right mouse button to click the name of the BeanInfo class, then click **Insert** and **Method**. This opens the Method Wizard.
3. Under **Name**, type the name of the BeanInfo class itself (for example, `MyBeanBeanInfo`).
4. Under **Prototypes**, delete the word `void` in the prototype.
5. Click **Finish**.

This opens a code editor where you can enter the default constructor. The following code shows a typical default constructor:

```
public MyBeanBeanInfo()
{
    try
    {
        _bean = Class.forName( "MyBeanPackage.MyBean" );
        _customizer =
            Class.forName("MyBeanPackage.MyBeanCustomizer");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The arguments of the **forName** function calls should be the fully qualified names of the Bean class and the customizer class.

The getBeanDescriptor function

The BeanInfo class must have a public user function named **getBeanDescriptor**. This function returns a BeanDescriptor object giving the fully qualified names of the Bean component and its customizer.

When PowerJ creates a BeanInfo class, the class contains a version of **getBeanDescriptor** that just returns `null`. Edit the function so that it looks like this:

```
public BeanDescriptor getBeanDescriptor()
{
    return new BeanDescriptor( _bean, _customizer );
}
```

This creates a BeanDescriptor containing the fully qualified names of the Bean component and its customizer.

The getPropertyDescriptors function

The BeanInfo class must have a public user function that returns an array of *property descriptors* for the Bean. Each property descriptor gives the name of a property within the bean. This information can be used by PowerJ for setting property values via the Object Inspector. The information can also be used by other tools that work with the Bean.

When PowerJ creates a BeanInfo class, the class contains a **getPropertyDescriptors** method that just returns `null`. You should edit this function to contain code generating property descriptors. Here is a typical example:

```
public PropertyDescriptor[] getPropertyDescriptors()
{
    try
    {
        PropertyDescriptor[] props = new PropertyDescriptor[2];
        props[0] = new PropertyDescriptor("propName1",_bean);
        props[1] = new PropertyDescriptor("propName2",_bean);
        return props;
    } catch ( IntrospectionException e ) {
        return super.getPropertyDescriptors();
    }
}
```

When creating each element in the PropertyDescriptor array, you specify the name of a property in the Bean. The PropertyDescriptor array should contain entries for each element that you want to be accessible through the Object Inspector.

How the Object Inspector uses property descriptors

The Object Inspector uses the property descriptors to display a list of properties for each Bean object used in a target. When you enter a value for one of these properties, the Object Inspector generates a corresponding **set** instruction to be executed at run time when the object is created. For example, if you use the Object Inspector to specify a value of `"test"` for **prop1** in `bean_1`, the Object Inspector generates code of the following form:

```
bean_1.setProp1( "test" );
```

It is up to you to specify the correct type of value for the property. For example, if you enter a String constant in the Object Inspector, but the corresponding **set** function requires an integer, you will get a compilation error when you try to build the target.

Other methods in the BeanInfo class

When PowerJ created a BeanInfo class, it also defines a number of other standard methods. For example, **getIcon** is supposed to return an icon representing the Bean, **getMethodDescriptors** is suppose to return information on the methods defined in the Bean, and so on. All of these methods are originally defined to return `null`. You should edit these methods so that they return the information that is expected. For further information, see the standard references on JavaBeans.

Property editors

A property editor is a class designed to be used by PowerJ or some other Java development tool. The property editor makes it possible for you to specify a value for some property in a JavaBeans component at design time.

For example, suppose someone is using your JavaBeans component as part of a PowerJ application. The user wants to specify a value for one of the Bean's properties. PowerJ can create an object of the property editor class and use methods of that object in order to set properties for the Bean itself.

Property editors can be created for any kind of property. For example, if a property takes a String value, you can create a property editor that obtains a string from the user (using a text field), then assigns the resulting string to the appropriate property. Property editors are particularly useful in situations where the property does not have a simple type of value; for example, if the value of a property is a complex data structure, providing a property editor that builds such a structure is a real service to the user.

Types of property editors

PowerJ distinguishes three types of property editors, depending on the methods they have implemented. These three types are:

- A type for choosing values that can be represented by a String. For these PowerJ will present a text field where you can enter a String value for the property.
- One for choosing from a set of discrete or enumerated values. For these PowerJ will present a Choice where you can select a value from the list of possibilities.
- A property editor that extends java.awt.Component. These give you full flexibility in how property values can be specified. For these PowerJ will open a dialog window defined by the property editor.

Creating property editors

Creating property editors

PowerJ includes a form template to help you create property editors based on the Panel component, or you can create property editors like any other class, following the usual PowerJ techniques.

To create a property editor you must create a class that implements the java.beans.PropertyEditor interface. This interface has 12 methods. For all types of property editor you should create non-stub implementations for the following methods:

```
void addPropertyChangeListener( PropertyChangeListener
     listener )
void removePropertyChangeListener( PropertyChangeListener
     listener )
Object getValue()
void setValue( Object value )
```

If you want PowerJ to use a text box in for the property in the Object Inspector, your property editor class should also implement also implement the following methods:

```
void setAsText( String text )
String getAsText()
```

It is strongly advisable to have the other two types of components also implement the above two methods, especially if your Bean is not serializable.

If you want PowerJ to use a Choice control to present a drop-down list in the Object Inspector, your property editor class should also implement the following method:

```
String[] getTags()
```

If you want PowerJ to add a browse button to the Object Inspector, your property editor class should also implement the following methods:

```
Component getCustomEditor()
boolean supportsCustomEditor()
```

The other methods in the PropertyEditor interface are:

```
String getJavaInitializationString()
void paintValue( Graphics gfx, Rectangle box )
boolean isPaintable()
```

At design time, PowerJ will use getValue and setValue to store and retrieve the current value of the property being edited. PowerJ will monitor changes in that property value using the PropertyChange event, which your property editor should trigger to indicate that the value of the property has changed.

Associating property editors

Once you have created a property editor, you need to associate with one or more properties. You can do this in two ways:

- Name the property editor class specifically in the BeanInfo, via the **PropertyEditorClass** property of the PropertyDescriptor object returned by your BeanInfo's **getPropertyDescriptor** method.
- Make the property editor available to the PropertyEditorManager for all properties of a given class.

For more information about creating property editors, see the standard JavaBeans references cited in the Bibliography.

Data-aware Beans

Some of the native PowerJ Beans are *data-aware*, which means that they can display fields in a database and will be automatically updated when the current row in the database changes. This behavior is called *data binding*, and Beans that act this way are said to be *data bound*. You can bind your Beans to a database using the classes included with PowerJ, as described in this section.

Note: For more information on databases, see Connecting to databases. For more information on bound controls, see Using bound controls.

The example code shown in this section shows you how to extend a standard text field to a data bound text field. The result will be something very similar to the powersoft.powerj.ui.DBTextField class included with PowerJ. You can open the source code for that class in order to see what goes into creating a bound Bean, as a companion to this section.

Data binding concepts

Two types of objects are used in data binding: a *data source* and a *data target*. A data source provides the data that is displayed or used by the data bound control, and a data target is a bound control. A Query object is the most commonly used example of a data source; it provides data from a database that can be used by a data target. Your bound controls access a data source to get their data, and usually through the DataTargetHelper class.

PowerJ provides a class and three interfaces to make data binding easy:

DataTargetListener
: An interface that receives database events, such as when the cursor position changes, or new data is available. Your Bean must implement this interface.

DataTarget
: An interface that extends DataTargetListener, and defines several properties and methods that allow the data source to manipulate your class. Your component must implement this interface to receive database events and provide property access to the data source.

DataTargetHelper
: A class that handles all of interaction between your components and the data source. Your components will usually defer any interactions with the database to this class.

DataSource
: The interface exposed by the Query object to provide full access to the database.

The difference between the DataSource and DataTargetHelper is this: DataTargetHelper is a simple class that provides the most commonly used methods, and uses the methods of DataSource to get the data. DataSource is an interface that is implemented by the Query object, and provides full access to the data. Most Beans only need to use DataTargetHelper, but more advanced Beans will need to access the methods of DataSource.

The DataTargetListener defines the database events and the DataTarget defines the properties and methods required by the data source. Because your Bean must implement the DataTarget, and the DataTarget extends DataTargetListener, your Bean must implement the events, properties, and methods defined by both of these classes. Once you implement these, your component will integrate well with the PowerJ design environment and will be *data-aware*.

Data bound Bean architecture

Support for data binding is often added in a subclass of the JavaBeans component. For example, the DBTextField is a subclass of the AWT TextField component. In this way the Bean has all of the functionality of the base class, and all of the database support in the subclass. Keeping the database support separated from the base functionality of the Bean provides two main advantages:

1. It minimizes the size of the client side application if the database support is not used. In the case of the DBTextField for example, the standard AWT text field is used unless data binding is required. That eliminates the need for the user to download the database support unless it's required.
2. It makes your classes simpler and easier to maintain. Standard object oriented design calls for class hierarchies to be divided up into small logically related units. With the database support all in one class, and nothing else to clutter the class, development, debugging, and maintenance become much easier.

All of this means that your data bound components should extend the base components.

Adding data binding to a Bean

To add data binding to a Bean, you must do the following:

1. Extend the Bean. It is a good idea to give the same name to the data bound Bean and the base Bean, but add DB to the beginning. For example, DBTextField extends and is the data bound version of the AWT TextField. The data bound Bean must implement the DataTarget so that it will receive database events.
2. Add a DataTargetHelper data member to the class. This class provides a programmable interface for the database, and generally does much of the database work for your component.
3. Add the database event handlers to your Bean, and implement them.
4. Add the other methods of the DataTarget interface to your Bean, and implement them. Often your implementation of the DataTarget methods will simply call a corresponding method on the DataTargetHelper class.

Step 1: Creating the data bound class

First you must create a class that extends the base class and implements the DataTarget class.

- **To create a data bound Bean class:**

1. Open the Classes window by pressing SHIFT+F3.
2. In the Classes window's **File** menu, point to **New** and click **Class**. PowerJ starts the Class Wizard.
3. Select the **Standard Java** class type and click **Next**.
4. Enter a package name if you are using one.
5. Enter the class name. This name should be descriptive and unique within its package. Normally this name will be `DB` followed by the name of the base component.
6. In the **Extends** field, type the name of the parent component.
7. In the **Implements** field, type `DataTarget`.
8. If the Bean will be a public class, make sure **Public** is checked. You should normally turn this option on.
9. Click **Finish**.

Step 2: Adding the data helper member

Next, you must add a data helper member to your class. To do this, just add the following declaration to your data bound class:

```
// add your data members here
    protected DataTargetHelper _dataHelper = new
      DataTargetHelper( this, DataTargetHelper.CELL );
```

In addition, you must import the necessary files:

```
// add your custom import statements here
import powersoft.powerj.db.* ;
```

Step 3: Handling database events

Because your Bean implements the DataTarget, you must write code for all the methods in the interface. Here is a list of the database event handlers that you must add to your class:

```
// Note: DataTarget extends the DataTargetListener interface.
// From DataTargetListener:
public void dataOpen( DataTargetEvent event );
public void dataAvailable( DataTargetEvent event );
public void dataRequest( DataTargetEvent event );
public void dataClose( DataTargetEvent event );
public void dataGuardRow( DataTargetEvent event );
```

In addition, the text box declares a few internal data members:

```
protected boolean          _wasEditable = false;
protected String           _oldText = "";
protected boolean          _ignoreChange = false;
```

These variables are not strictly required, but are generally used to determine the state of the text field as it is being updated. For example, when new data arrives from the database, the text field must be updated. Normally updating the text field causes database events to be fired. To avoid this circular behavior, the `_ignoreChange` member is used; if it is `true`, the database events are not fired. Your Beans may need these or other internal members to control what will happen when database events occur.

The event handlers are treated as methods defined by your Bean, and you can add each method using the same means as was shown in Adding methods to your Bean. Here is a description of each and what you should add to your implementation:

dataOpen

Occurs when a new result set is available. Use your event handler to prepare the target to receive data from the data source.

In this event handler, you should always call the DataTargetHelper's **defaultDataOpen** method, which sets up the data target helper:

```
_dataHelper.defaultDataOpen( event );
```

If the DataTargetHelper class is read-only, you should also make your Bean read-only if possible:

```
// Mark the text box as read-only if
// the bound column is read-only
    if( _dataHelper.getReadOnly() ) {
        _wasEditable = isEditable();
        setEditable( false );
    }
```

dataAvailable

Occurs when new data is available, usually when the cursor moves. Use your event handler to update the targets state according to the event data.

In this event handler, you should check the event's suggested action, and if it is `REFRESH`, you should update the contents of the Bean. Otherwise, call the DataTargetHelper's **defaultDataAvailable** method:

```
boolean old = _ignoreChange;
_ignoreChange = true;
switch( event.getSuggestedAction() ) {
    case DataTargetEvent.ACTION_REFRESHROW:
        setText( _dataHelper.getString() );
        break;
    default:
        _dataHelper.defaultDataAvailable( event );
        break;
    }
```

```
_ignoreChange = old;
```

The `_ignoreChange` variable in this case is used to ensure that **dataChanged** events are not fired while the text box is being updated.

dataRequest

Occurs when data in the target has changed. Use your event handler to invoke **setValue** or **setString** on the data source for any data that has changed. Here is what the text field does:

```
if( _dataHelper.getDataChanged() ) {
    _dataHelper.setString( getText() );
}
```

dataClose

Occurs when the result set is about to close. Use your event handler to clear the target, by calling the **defaultDataClose** method:

```
_dataHelper.defaultDataClose( event );
if( _wasEditable ) {
    setEditable( true );
    _wasEditable = false;
}
```

dataGuardRow

Occurs when a guard row needs to be fetched. Use your event handler to fetch the row from the data source and add it to the target. The text field does not use this method:

```
public void dataGuardRow(DataTargetEvent event)
//*****************************************
{
}
```

Step 4: Implementing methods and properties of DataTarget

Next you must add methods and properties that allow your Bean to be manipulated by the data source. Here is a list of the methods and properties that you must add to your class:

```
// Methods of DataTarget:
public void resetContents();

// Properties of DataTarget:
public DataSource getDataSource();
public void setDataSource( DataSource source );
public String getDataColumns();
public void setDataColumns( String str );
public Object getTargetObject();
```

You can add each method using the same means as was shown in Adding methods to your Bean, and each property following the procedure in Adding properties to your Bean. Most of the properties simply return or set the corresponding property on the data helper member. Here is a description of each and what you should add to your implementation:

resetContents (method)

Called by the data source to instruct your Bean to reset its contents. For the text field, this means clearing the text:

```
boolean old = _ignoreChange;
_ignoreChange = true;
setText( "" );
_ignoreChange = old;
```

Once again, the `_ignoreChange` member is used to suppress database events while the contents are being reset internally.

DataSource (property)

Sets or returns the data source. You should simply use the data target helper's **DataSource** property:

```
public  powersoft.powerj.db.DataSource getDataSource()
{
    return _dataHelper.getDataSource();
}

public void setDataSource(
            powersoft.powerj.db.DataSource  source )
{
    _dataHelper.setDataSource( source );
}
```

DataColumns (property)

Retrieves a string identifying the column or columns bound to the text field:

```
public java.lang.String getDataColumns()
{
    return _dataHelper.getDataColumns();
}

public void setDataColumns( java.lang.String str )
{
    _dataHelper.setDataColumns( str );
}
```

DataTarget (read-only property)

Returns a reference to the data target object, your component:

```
public Object getTargetObject()
{
    return this;
}
```

Further comments on data binding

As you have seen, you can add database support to your components by creating a subclass that implements the DataTarget interface and uses the DataTargetHelper to access the database. This allows your Beans to interact with the PowerJ design-time environment and the native PowerJ database components at run time. The source code for all of the PowerJ bound components, such as DBTextField is provided, allowing you to investigate further data binding techniques.

Cell binding

The text field discussed in this chapter binds to only one column in the current row, called *cell binding*. This is the default and simplest type of data binding, and is used by single-field controls such as the DBTextField, DBTextArea, and DBMaskedTextField. These types of controls are described from the user perspective in Bound controls.

More complex techniques of data binding are used by other components, such as the PowerJ DBChoice (*column binding*), DBLookupChoice (*lookup binding*), and DBGrid (*table binding*). Column and lookup binding are described in Bound lists and choices, and grids are described in Bound grids.

Column and lookups

For column and lookup binding, you must implement the extra properties listed in the section sited above, but aside from some extra functionality the data binding procedure is basically the same as for cell binding. For lookups, the DBLookup class extends the DataTarget to assist you with the extra complexity.

Grids

Grid binding, on the other hand, is much more complex because it requires you to deal with all columns and rows in the database. In addition, because the database can be large and the connection speed slow, the DBGrid implements database optimizations that help minimize the amount of communication between the grid and the database. Describing these techniques is beyond the scope of this guide, but the source for the DBGrid is provided to help you explore this area.

Persistence and serialization

When an applet is loaded, Beans embedded in the applet must also be loaded, and their initial states must be restored. For example, if you put a text box on a form and change its label at design time, the label must also be set when the applet is run. The ability of an object to save its state is called *persistence*.

In simple cases, an object's state can be restored by setting all of its exposed properties to their correct initial values. More advanced Beans, however, will require internal values not exposed through properties to be saved. This is accomplished using *serialization*, a process of writing all an object's internal data to a stream so that it can be recovered.

Beans that you create can persist through their exposed properties (the default) or can be serialized. If you do nothing but create a Bean, it will persist through properties. This approach is sufficient if your Bean can be completely recreated by setting all of its properties. In this case, you need read no further; everything is done automatically.

Note: Other environments may require your Beans to be serialized, and in practice it is a good idea to make all your Beans serializable.

If your Bean stores information not accessible through properties, however, you will need to write special code to have your Bean serialized. A serializable object:

- must implement the java.io.Serializable interface (only defined in JDK 1.1).
- can implement a **writeObject** method to control what information is saved, or to append additional information to the stream.
- can implement a **readObject** method so it can read the information written by the corresponding **writeObject** method, or to update the state of the object after it has been restored.
- must mark fields that are not to be persistent with the `transient` keyword.

◆ **To make your Bean serializable:**

1. Open the Classes window by pressing SHIFT+F3.
2. Select your class, then on the **Class** menu, click **Properties**.
3. In the **Implements** field, type java.io.Serializable. If your class already implements an interface, you may separate the interfaces with commas, as in: java.awt.event.ActionListener, java.io.Serializable.

Saving your object's state
Restoring your object's state
Serialization specification

Saving your object's state

To serialize your object, you use the **writeObject** method.

- **To serialize your object:**

1. Open the Classes window by pressing SHIFT+F3.
2. Select your class, then on the **Class** menu, point to **Insert** and click **Method**.
3. For the name of the method, type `writeObject`
4. Change the prototype to the following:

   ```
   private void writeObject(java.io.ObjectOutputStream out)  throws
   java.io.IOException
   ```

   This is only defined in JDK 1.1.
5. Click **Finish**. PowerJ creates the **writeObject** method for you.

Next, you must write code to write your object to the output stream provided. Generally, you should just call the **writeObject** method for each object that you want to save:

```
private void writeObject(java.io.ObjectOutputStream out)
                                            throws IOException
{
    out.writeObject( data1 );
    out.writeObject( data2 );
    out.writeObject( data3 );
}
```

Your **writeObject** method will be called whenever the Bean needs to be serialized.

There are several other methods that allow you to write data of different types, such as `int`. For more information, consult the JDK documentation for ObjectOutputStream.

Restoring your object's state

Restoring your object's state, you use the **readObject** method and follow precisely the reverse procedure as in saving the state. Restoring an object is often called *deserializing*.

- **To deserialize your object:**

1. Open the Classes window by pressing SHIFT+F3.
2. Select your class, then on the **Class** menu, point to **Insert** and click **Method**.
3. For the name of the method, type `readObject`
4. Change the prototype to the following:

   ```
   private void readObject(java.io.ObjectInputStream in) throws IOException,
   ClassNotFoundException;
   ```

5. Click **Finish**. PowerJ creates the **readObject** method for you.

Next, you must write code to read your object from the input stream provided. Generally, you should just call the **readObject** method for each object that you want to restore:

```
private void readObject(java.io.ObjectInputStream in)
            throws IOException, ClassNotFoundException
{
    Object data3 = in.readObject( );
    Object data2 = in.readObject( );
    Object data1 = in.readObject( );
}
```

Your **readObject** method will be called whenever the Bean needs to be serialized.

There are several other methods that allow you to read data of different types, such as `int`. For more information, consult the JDK documentation for ObjectInputStream.

Serialization specification

For more information on serialization, consult the JDK documentation at:

`http://java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html`

If you installed the JDK 1.1 with PowerJ using the default folder, the JDK documentation on serialization can also be found under your PowerJ folder:

`Java/Sun/JDK11/docs/guide/serialization/index.html`

Deploying Beans

Usually Beans are deployed in a Java Archive (JAR), a platform-independent file format that can collect multiple files into a single file. PowerJ provides a JAR collection target that automatically collects all the files related to your Bean's target. For more information on JAR files, see Using collection targets: CAB, JAR, and ZIP.

Note: JAR files may contain a number of classes. For example, if you create a customizer for your Bean component, you would usually include the customizer class in the JAR file. The same goes for BeanInfo classes.

There are some deployment issues that relate specifically to JavaBeans components. Those will be covered here, while all other deployment issues are covered in Collecting and deploying.

Specifying which classes are Beans

When you create a JavaBeans component target, you should specify what classes are put on the component palette in the *manifest* file. PowerJ creates most of the manifest file for you, but you need to add two lines for each class that you want to be placed on the component palette.

Note: Manifest files are only used with JAR collection targets. The rest of this section assumes that you have created a JAR collection target and added your Beans target to the collection. For more information on JAR files, see Using collection targets: CAB, JAR, and ZIP.

Before you add the file to your collection, create a file called `manifest.mf`. For each Bean in the target, create an entry like this:

```
Name: MyPackage/MyClass.class
Java-Bean: True
```

Name
: Specifies the package and class names, in the format PackageName/ClassName.class

Java-Bean
: Specifies whether the class is a JavaBeans component. You only need to make these entries when this field is true, because false is assumed as the default.

- **To set up the manifest file:**

1. Open the Targets window by pressing SHIFT+F7.
2. Under the **View** menu, ensure that the **Show Property Sheet** menu item is checked.
3. Click the name of the JAR target, then click the **JAR Properties** tab in the corresponding property sheet.
4. In the **Manifest** section of the **JAR Properties** property sheet, select the option labeled **Use this manifest file**.
5. Type the name and location of the manifest file, or click **Browse** to locate the file on disk.
6. Click **Apply**.

When you have completed these steps, PowerJ will include the manifest file in the JAR file. PowerJ adds any other information needed for the JAR, most of which is internal information about the JAR file.

Deploying serializable Beans

When a Bean is serializable, PowerJ generates a `.ser` file providing information about serializing the Bean. This file should be distributed with the Bean along with its usual `.class` file. You can include `.ser` files in CAB, JAR, and ZIP files. Similarly, you can publish `.ser` files using the usual mechanisms of WebApplication targets. For more information on CAB, JAR, and ZIP files, see Using collection targets: CAB, JAR, and ZIP. For more on WebApplication targets, see WebApplication targets.

Finding more information on Beans

This chapter has provided only an introduction to JavaBeans components, with particular emphasis on the facilities that PowerJ provides to create these components. There are more topics that are important when you create JavaBeans components such as the standard design patterns and specific class and interface specifications that are covered in more detail in the JavaBeans component specification. For more information, consult the following web site:

`http://java.sun.com/products/jdk/1.1/docs/guide/beans/index.html`

If you installed the JDK 1.1 with PowerJ using the default folder, the JDK documentation on JavaBeans components can also be found under your PowerJ folder:

`Java/Sun/JDK11/docs/guide/beans/index.html`

Part IV. Accessing databases

The chapters in this part describe database programming with PowerJ.

Chapter 19. Connecting to databases

This chapter introduces the basic elements on connecting to databases. It discusses the JDBC standard for accessing databases, and the JDBC-compatible drivers that are supplied with the PowerJ package. It also describes the two fundamental components for accessing databases: transaction and query objects.

Note: The sample program named `JDBCTransaction` illustrates many of the details discussed in this chapter.

Introduction to JDBC

This section describes how JDBC works when used in Java applets. In particular, it discusses problems in using JDBC with Java 1.02 versus Java 1.1.

Introduction to JDBC

JDBC

JDBC is a standard that describes how to connect to and talk to a database from within a Java application or applet. The formal specification is available from:

`http://java.sun.com/jdbc`

You should read this web page if you plan on doing any serious database work with Java.

One important thing to note is that JDBC is a set of Java interfaces, not actual Java classes. An interface is similar to an abstract class in C++ or other object-oriented languages: it declares a set of methods. Any class that defines those methods is said to *implement* the interface. (The classes are marked as doing so using the `implements` keyword.) You can use the name of an interface in most places that accept a class name (for example, as parameters to or return values from method calls).

A *JDBC driver* is a set of Java classes that implement the JDBC interfaces. JDBC drivers are provided by database vendors or other third parties. When a JDBC driver is installed on your machine (either manually or by downloading across a network), the classes register themselves with a JDBC driver manager. The application or applet then asks the JDBC driver manager for a way to connect to a given database. The JDBC driver manager then scans through its list of registered drivers and returns the first one that is capable of making the connection.

The JDBC API provides Java programmers with a uniform interface to a wide range of relational databases.

Package names and applets

All Java classes and interfaces are packaged together in sets called (appropriately) *packages*. A package is analogous to a C++ namespace—it's a way of grouping related classes or interfaces together without worrying about name conflicts with other classes or interfaces.

The set of core classes and interfaces that ship with Java all have package names that begin with `java`, as in `java.awt` or `java.lang`. The `java` package name is reserved exclusively for core classes.

An *applet* is a special kind of Java application, one that is downloaded from a web server and run inside a web browser. Applets have special security restrictions that plain Java applications do not. Applets cannot read or write to the local file system. Applets can open network connections, but only to the web server they were downloaded from.

A special restriction has to do with the kinds of classes that can be downloaded: classes that have a package name starting with `java` cannot be downloaded across the net. They must be installed locally on your system.

Problems with JDBC and Java 1.02

JDBC is an official part of the Java 1.1 specification. Specifically, the JDBC interfaces and the JDBC driver manager are packaged with Java 1.1. The package name is `java.sql`. When you install Java 1.1, the JDBC interfaces are installed locally and so are available for an applet to use.

The situation is much different with Java 1.02, which is what many browsers are currently using. JDBC appeared after Java 1.02 was released, so the `java.sql` package is not part of Java 1.02.

This causes a problem. The `java.sql` package cannot be downloaded and used by an applet because of the security restrictions mentioned above. For an applet to use JDBC, there are therefore

two options:

1. Force the user to install the JDBC drivers locally on their machine (i.e., in their CLASSPATH); or
2. Rename the `java.sql` package so that the classes in it can be downloaded.

The first option causes trouble if you are building applets you want to deploy across the Internet; it means that only a small number of users will be able to use the applets. At present then, the second option is the most widely used in the industry. Many vendors have been taking the set of JDBC interfaces defined by `java.sql` and redefining them to be in a different package (for example, `jdbc.sql`). They then modify their JDBC drivers to use and return the interfaces defined in `jdbc.sql` as opposed to `java.sql`. An applet developer can then put the applet, plus the `jdbc.sql` interfaces and the driver classes, on the web server where they can be downloaded and run on the user's machine.

Unfortunately, this means that a driver vendor must support two JDBC drivers: one driver that works with the `java.sql` package (so that Java 1.1 users can use the driver) and one that works with `jdbc.sql` (so that Java 1.02 users can use the driver). The driver that uses `java.sql` will only work in Java 1.1 browsers, but will require less download time since the `java.sql` package is already installed on the user's machine. The driver that uses `jdbc.sql` will work on all browsers but will require more time to download (even using Java 1.1, since the driver doesn't make use of the built-in `java.sql` package).

Note: The PowerJ database facilities can use drivers that work with either the `java.sql` or `jdbc.sql` packages. You can specify which JDBC package to use via the **JDBCPackage** property on the transaction and query objects. Note that if you use the option to automatically detect the package, you will have to manually add any required JDBC classes to a WebApplication target or collection target (ZIP, JAR or CAB).

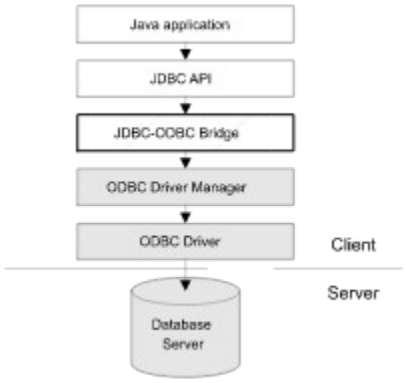
JDBC drivers

The PowerJ package provides support for a number of drivers that can access JDBC databases. This section examines those drivers.

There are four types of JDBC drivers:

Type 1

The *JDBC-ODBC bridge* type uses Java code to call native (non-Java) ODBC binary code and in many cases native client software of the ODBC driver.



In the above diagram, striped shading indicates software that is partially Java and partially native and solid shading indicates native (non-Java) software.
The advantages of this type are that it can make use of existing ODBC drivers.
The disadvantages are due to the use of native code on the client. Since that violates the Java applet security model, it cannot be used in an applet. Also, it requires prior installation of client software on each user's computer.

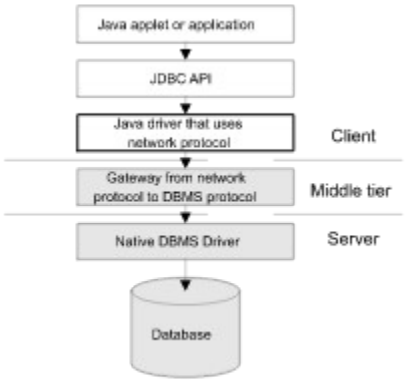
Type 2

The *native-API partly-Java driver* type uses Java code to call native (non-Java) code to a DBMS-specific API, such as the Sybase Open Client API.

In the above diagram, striped shading indicates software that is partially Java and partially native and solid shading indicates native (non-Java) software.
The advantages of this type are that it can make use of existing DBMS-specific drivers.
The disadvantages are due to the use of native code on the client, and the fact that each DBMS requires a different driver. Since the use of native methods violates the Java applet security model, this type of driver cannot be used in an applet. This type also requires prior installation of client software.
If type 2 seems similar to type 1, it is – type 1 is an ODBC-specific subset of type 2.

Type 3

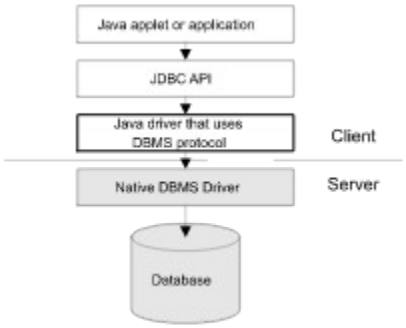
The *net-protocol all-Java driver* type translates JDBC calls into a DBMS-independent network protocol that is then translated by middleware to a specific DBMS protocol.



In the above diagram, solid shading indicates native (non-Java) software.
The advantages of this type are that it uses pure-Java client software, it can thus be downloaded automatically by applets, the client driver is independent of the DBMS, and the middle tier can offer security and performance.
The disadvantage is that the middle tier adds an extra layer of software and perhaps hardware.

Type 4

The *native-protocol all-Java driver* type translates JDBC calls into a network protocol that is used directly by the DBMS.



In the above diagram, solid shading indicates native (non-Java) software.
The advantages of this type are that it uses pure-Java client software, it can thus be downloaded automatically by applets, the driver can be “thin”, and there is no requirement for a middle tier.
The disadvantage is that the driver is vendor specific and, in the absence of a middle tier, more computations are done remotely on the database server.

Listing of JDBC drivers

Sun maintains a listing of JDBC drivers, classified by the above types, at the following Web page:

`http://java.sun.com/jdbc/jdbc.drivers.html`

- The JDBC-ODBC bridge
- The jConnect driver
- Other drivers provided with PowerJ
- Drivers not provided with PowerJ

The JDBC-ODBC bridge

The first JDBC driver to become publicly available was *the JDBC-ODBC Bridge*, developed by INTERSOLV. It is freely available from Sun's web site for Java 1.02 and it is also included as part of Java 1.1.

The bridge is a type 1 driver that lets you access any ODBC datasource from a Java application but not from an applet. The bridge and any ODBC driver it uses must be installed and configured on each user system that runs a particular JDBC application, so it is not suitable for "zero-install" situations.

The bridge has two important restrictions:

- It cannot be used by applets. The bridge is not "100% Pure" Java—it uses native method calls to interface with ODBC, and the ODBC drivers might access the local file system. Browser security restrictions prevent applets from accessing the local file system, so applets can't use the bridge.
- The Microsoft 2.0 VM includes a version of the bridge that works with it (but not with the Sun Java interpreter). The Sun version of the bridge can only be used with Java interpreters that follow the Sun native method calling conventions (the Microsoft VM uses different calling conventions). There is no bridge for earlier versions of the Microsoft VM.

These restrictions mean that the bridge is a useful tool for testing your database applications, but is not useful for general deployment.

The jConnect driver

The Sybase *jConnect* driver is a JDBC driver that is 100% Pure Java and therefore can be used by applets. Furthermore, it can be used with both Sun and Microsoft Java virtual machines.

The jConnect driver provides high performance to most types of relational database. It has direct access to any database that supports native TDS 5.0, including Sybase Adaptive Server Enterprise (formerly SQL Server), Sybase IQ, Replication Server and servers using the Sybase Open Server foundation. Other databases are supported through gateway software that acts as a bridge between jConnect and the server. Gateways include:

- Open Server Gateway for SQL Anywhere (see below).
- OmniCONNECT for more than twenty five enterprise and legacy database servers.
- DirectConnect for direct access to Oracle, AS/400 and others.

jConnect is a type 4 driver for the databases that directly support TDS, and it is a type 3 driver for databases that it accesses through a gateway.

Versions of jConnect

There are different versions of the jConnect driver for Java 1.02 and Java 1.1; these versions offer the same functionality but import different versions of the JDBC classes. The drivers are:

- `jdbc.sybase.jdbc.SybDriver` for 1.02 (uses `jdbc.sql`); this is jConnect version 2.2
- `com.sybase.jdbc.SybDriver` for 1.1 (uses `java.sql`); this is jConnect version 3.0

Applications that use one of these drivers must load the driver by specifying the driver in the **DriverName** property of the Transaction object or by using the code:

```
Class.forName( "jdbc.sybase.jdbc.SybDriver" ); // 1.02
Class.forName( "com.sybase.jdbc.SybDriver" );  // 1.1
```

As soon as a JDBC driver is loaded, it registers itself with the JDBC DriverManager class. To connect to a database, the application simply calls a method in the DriverManager class which scans the list of registered JDBC drivers to make the database connection. When you set the **DriverName** property for a Transaction object, this is done for you automatically. For further information on transaction objects, see Transaction objects.

Limitations

The jConnect driver included with PowerJ requires that you prepare your database for full access to its features. To use jConnect effectively, there are some special steps that you need to take:

- Prepare the database with special tables and stored procedures. For more information, see Using the query editor with jConnect.
- To perform updates with jConnect using the PowerJ query object, you *must* set the following properties for the query object: **PrimaryKeyColumns**, **ColumnTableNames**, and **ReadOnlyColumns** (if there are any such columns). For more information, see Query properties.

The Open Server Gateway

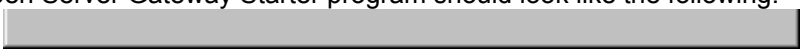
In order to use jConnect with SQL Anywhere, you must have the *Open Server Gateway* running on the system where the database resides. The Open Server Gateway provides an interface between

jConnect and SQL Anywhere. In particular, it provides a TCP/IP port entry that jConnect can use for interactions with the SQL Anywhere driver.

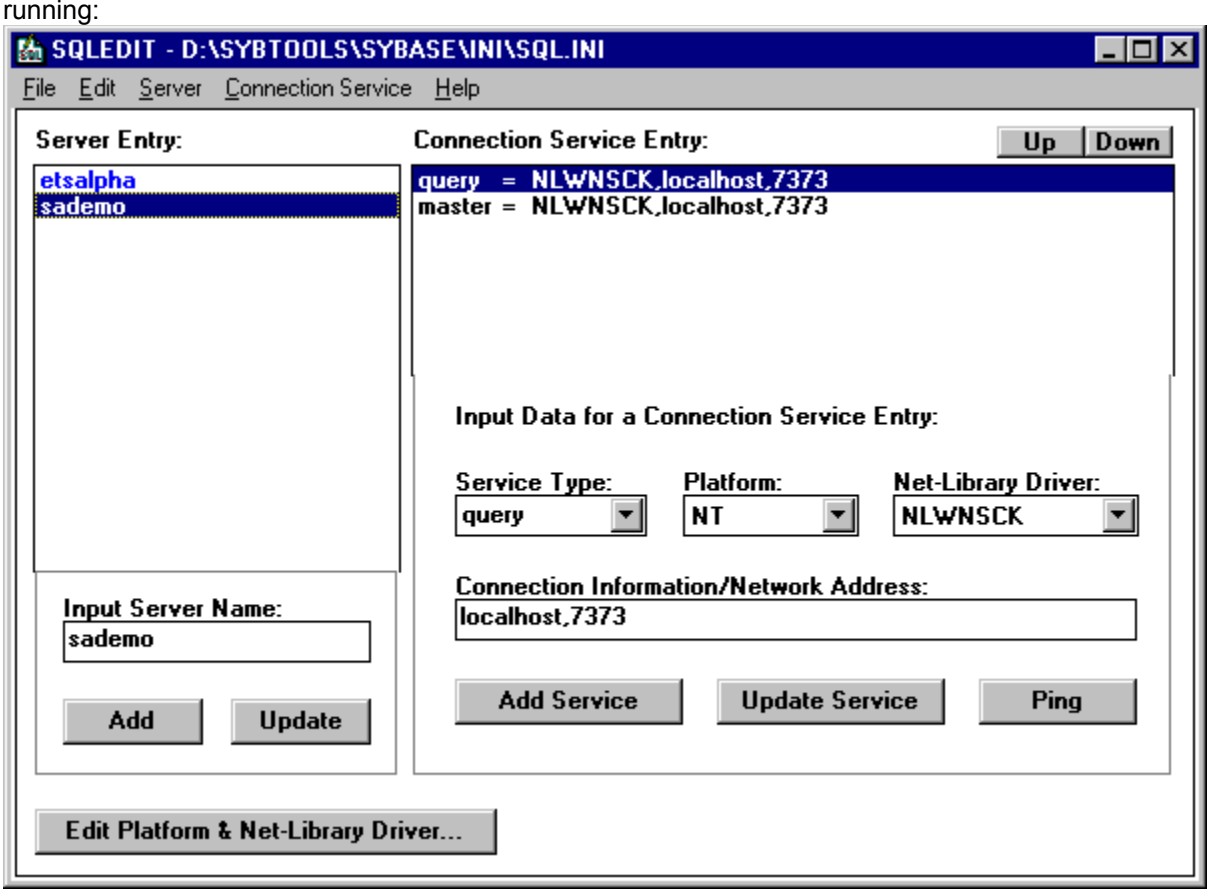
- **To invoke the Open Server Gateway:**

1. From the PowerJ program group, invoke **Open Server Gateway**. This starts the Open Server Gateway Starter program (`OSGStarter.exe`). This program is included with PowerJ to make it easier for you start and configure the Open Server Gateway, but is not part of the Open Server Gateway program itself.

The Open Server Gateway Starter program should look like the following:



This dialog box specifies a default port number for accessing the given database file and database server. In the example above, the port number is 7373. If you wish to change the port number or information about the server or database, click **Run SQLEDIT**. This will start the SQLEDIT program where you can configure the gateway. The diagram below shows what the SQLEDIT program looks like when it is running:



Under **Server Entry**, you can create an entry for each server known to the Open Server Gateway. The above picture shows the entry for `sademo`, the SQL Anywhere 5.0 Sample database which is supplied with the PowerJ package.
For each server, there should be a master and a query service type. Configure each service in the fields under **Input Data for a Connection Service Entry**, then click **Add Service**. The **Connection Information/Network Address** should specify a system name and port number for the connection. The default name is `localhost`, standing for your own computer. However, it is better to type in the actual name of your computer rather than the generic keyword `localhost`; if you leave it at `localhost`, you will only be able to connect to the database from your own system, not from other systems.
You can choose any port number that is not currently being used for some other purpose.

When you have configured the gateway appropriately, save the configuration by click **Save** under the

File menu. You can then exit the SQLEDIT program by clicking **Exit** under the **File** menu.

You should return to the Open Server Gateway Starter program. You can now start the server by clicking the **Start** button. This starts the Open Server Gateway program and the specified SQL Anywhere database server, making it possible to connect with the given database file. Once the Open Server Gateway and SQL Anywhere programs are running, you can close the Open Server Gateway Starter program by clicking the close button.

If you leave the Open Server Gateway Starter program running, you can click its **Shutdown** button when you are finished working with the database. This will shut down the Open Server Gateway program and any database servers that have been invoked through the gateway.

The jConnect test site

To make it easier for you to test applications using jConnect, Sybase has provided a jConnect test site on the World Wide Web. Set the URL of your transaction object to:

```
jdbc:sybase:Tds:jdbc.sybase.com:4444
```

Use a userid of `guest` and a password of `sybase`. This provides jConnect access to a sample database that you can use for testing simple programs.

Note: Before you run your program, you should try "pinging" jdbc.sybase.com, just to make sure that your system has its Internet connection software properly configured to interact with the Sybase site.

Multiple databases with jConnect

If you want to connect to multiple databases with jConnect, you can specify a particular database name as part of the URL. Here is a simple example:

```
jdbc:sybase:Tds:localhost:5000/testdb
```

As shown, the name of the database is given as `/name` at the end of the normal URL.

This approach works well with Sybase System 11 databases, but not with SQL Anywhere databases. The problem is that the Open Server Gateway does not let you select a database in this way. To support multiple SQL Anywhere databases, you must:

- Use SQLEDIT to set up different entries for each database, with different port numbers.
- Run a separate copy of Open Server Gateway for each database.

Encryption through HTTPS tunneling

The jConnect 3.0 driver supports HTTPS tunneling of TDS. This provides support for encryption of passwords and other data transmitted from the user to the server system.

To use HTTPS tunneling, you need a web server that supports the `javax.servlet` APIs and SSL (for example, the Netscape 3.0 server, JavaServer, or IAS). The jConnect 3.0 package comes with a TdsTunnellingServlet (source and class files) which you run on this web server.

As an example of how this works, suppose that your database is running on `dbhost:1234` and the web server listening for HTTPS requests is running on `webhost:433`. You will have to configure this web server to invoke the TdsTunnellingServlet for the `/tds` servlet alias; this allows the web server to pass on information to the database after it has been encrypted. To make the connection with jConnect 3.0 to get SSL encryption, you would use the following URL:

```
jdbc:sybase:Tds:dbhost:1234?proxy=https://webhost:433/tds
```

This consists of the standard prefix for jConnect (`jdbc:sybase:Tds`) plus the address of the database and a proxy connection property. The proxy property directs jConnect to use HTTPS tunneling through the `/tds` servlet alias on `webhost:433`.

Other drivers provided with PowerJ

To use other JDBC drivers with PowerJ, you need to know the class name of the driver and its package name.

PowerJ also includes development versions of the following drivers:

XDB JetConnect
: A simple thin-client JDBC-to-ODBC connection that lets you use PowerJ with any ODBC driver. It can be used as a type 1 or type 3 driver.

Visigenic VisiChannel for JDBC
: This is similar to JetConnect except that it uses IIOP as the wire level protocol, and can provide a very thin client solution when used with version 4 of the Netscape browser (that version includes the Visigenic IIOP protocol drivers). It is a type 3 driver.

For more information on connecting to these drivers, see Connecting to supplied JDBC drivers.

Drivers not provided with PowerJ

There are a variety of other JDBC drivers available on the market, and many can be used with PowerJ programs. The following considerations apply:

- Your JDBC driver must work with either the `java.sql` package, or `jdbc.sql`. This is no difficulty with JDK 1.1 programs, since all compatible drivers are supposed to use the standard `java.sql`. See the discussion in JDBC for more about `java.sql` and `jdbc.sql`.
- Typically, you specify information about the driver using properties of the transaction object (described in Transaction properties). For example, you specify the class name of the driver using the **DriverName** property. You can specify any parameters for controlling the connection using the **ConnectParms** property.
- PowerJ offers facilities for testing a connection at design time. In order for this to work, PowerJ must be able to find the class files for your driver. PowerJ does *not* use your `CLASSPATH` environment variable when trying to find class files at design time, so you cannot use `CLASSPATH` to specify the location of your driver's class files. You can add your JDBC driver to the global classpath (**Tools**/**Options Classpath**) and, when you restart PowerJ, it will be available for testing at design time. Alternatively, you can store a copy of the driver's class files in one of the following folders (under your main PowerJ folder):

  ```
  java\user classes\lib
  java\user classes\jdk11\lib
  ```

  PowerJ searches these folders (amongst others) for classes it needs at design time, so putting your driver class files here makes it possible to access the driver at design time.
- At run time, the class files for your driver should be available through the `CLASSPATH` environment variable or the `CODEBASE` for the applet. For more information, see CLASSPATH and CODEBASE.

Security considerations in database applets

In the interests of security, an applet cannot connect to any computer other than the computer where the applet originated. For example, if you open a web page that downloads an applet from system X, the applet cannot connect to a second system Y. In particular, if you download an applet from one system, the applet cannot connect with a database that resides on another system.

One way to get around this limitation is to use the HTTP gateway for jConnect. This gateway receives HTTP requests and translates them into TDS, the Sybase transfer protocol. The TDS requests may access a database on another system, even if the gateway itself is running on the system where the applet originated. For more information about the HTTP gateway and other jConnect features, see the documentation under

```
java\jconnect\docs
```

under your main PowerJ folder.

Transaction objects

A *transaction object* specifies information for communicating and working with a specific database. The transaction object has associated properties identifying the database, and giving the password and user identification that should be used when accessing the database. The transaction object also manages SQL transactions with the database, either automatically or in explicit commit or rollback operations.

Transaction objects are represented by the Transaction class. This class is serializable. PowerJ gives transaction objects default names of the form *transaction_N*. On the **Database** page of the component palette, transaction objects are represented by the following button:

When you place a transaction object on a form, an icon appears on the form to show that the object is there; however, this icon will not be visible at run time.
If your program only interacts with one database, you typically need only one transaction object in the entire program.

Transaction subclasses

The Transaction class is used as a base class for subclasses java_sql.Transaction (for transactions compiled against the `java.sql` libraries) and jdbc_sql.Transaction (for transactions compiled against the `jdbc.sql` libraries).

Note: For a discussion of the `java.sql` and `jdbc.sql` libraries, see JDBC.

To initialize a transaction object, PowerJ generates code of the following form:

```
Transaction transaction_1 = null;
transaction_1 = Transaction.createFromDriver( "drivername",
     "Form1.transaction_1", true );
```

The **createFromDriver** method looks at the database driver your program is using and determines whether the driver expects transactions to be compiled against the `java.sql` or `jdbc.sql` libraries. Using this information, **createFromDriver** creates an appropriate Transaction subclass object. It also registers the appropriate driver with the JDBC driver manager and sets the **DriverName** property of the Transaction object. This process should be transparent to the code you write, since both Transaction subclasses support the same set of methods and properties.

Note: If you create your own transaction object from scratch (rather than creating it using a design-time transaction object), you must declare the correct type of Transaction object (java_sql.Transaction or jdbc_sql.Transaction).

One good way of figuring out what code you have to write to create a transaction object correctly is to place an equivalent transaction object on a form and see what code PowerJ generates. To do this, follow these steps:

1. From the **Tools** menu of the main PowerJ menu bar, click **Options**.
2. On the **Editor** page of the options dialog, click **Show Generated Code**.
3. Click **OK** to close the options dialog.
4. Create a target that just contains a blank form, and place a transaction object on this form.
5. Use the property sheet for the transaction object to set properties for the object. In particular, set the **JDBCPackage** property to **UseJdbcSql** (JDK 1.02) or **UseJavaSql** (JDK 1.1).
6. Use the right mouse button to click on the form, then click **View Code**.

This opens an editor window that shows all the code PowerJ generates, including the code to construct and initialize the transaction object. Use this code as an example when you are writing your code to create a transaction object from scratch.

- Transaction properties
- JDBCPackage and deployment
- Setting up transaction information at run time
- Connecting to the database
- Managing transactions manually
- A hint for setting up transactions

Transaction properties

After you place a transaction object on a form, you should set the properties for the transaction using the object's property sheet.

Note: For additional information on setting transaction properties for each of the database drivers included with PowerJ, see Connecting to supplied JDBC drivers.

The property sheet for a transaction object includes the following items:

JDBC Driver [**Connection** page]
The JDBC driver used to interact with your database. Click the arrow in the **JDBC Driver** combo box to see which drivers are currently supported.

DataSource URL [**Connection** page]
A URL specifying the location of the database. Parts of the URL are typically separated by colon (`:`) characters. For example, if you are using the JDBC-ODBC bridge with the SQL Anywhere 5.0 Sample database, you would typically specify

```
jdbc:odbc:SQL Anywhere 5.0 Sample
```

If you are using jConnect, you specify a URL of the form:

```
jdbc:sybase:Tds:host:port/dbname
```

Here, `host` is the DNS machine name, `port` is the TCP/IP port number configured for accessing the server (Open Server Gateway for SQL Anywhere), and `dbname` is the optional database name. If you omit the `/dbname` parameter, the default database for the database server will be used. Here are some typical URLs for using jConnect:

```
jdbc:sybase:Tds:jdbc.sybase.com:4444/Pubs2
jdbc:sybase:Tds:localhost:7373
```

For more information about the Open Server Gateway, see The jConnect driver.

UserID [**Connection** page]
The userid that the program should use to connect to the database. If you do not specify a userid, the user may be prompted for a userid when your program attempts to connect with the database.

Password [**Connection** page]
The password that the program should use to connect to the database. For security, PowerJ displays an asterisk (`*`) in place of each character you type. If you do not specify a password, the user may be prompted for a password when your program attempts to connect with the database.

ConnectParams [**Connection** page]
Any extra information needed for connecting to the database. At design time, parameter information is specified in the form:

```
name="value"
```

A line of this form in the **ConnectParams** list leads to code of the form:

```
transaction_1.setProperty( "name", "value" );
```

These properties are passed on to the JDBC driver. Different drivers recognize different connection parameters.

Test Connection [**Connection** page]

Once you have filled in any necessary information on the **Connection** page, clicking **Test Connection** immediately checks whether PowerJ can establish such a database connection. In order for this to succeed, you must already have started any auxiliary software needed for the connection. For example, if you want to use jConnect to access a database on your system, the Open Server Gateway must already be running.

For more information on what to do if **Test Connection** fails, see Debugging database applications.

JDBCPackage [**JDBC** page]
This option lets you specify which version of JDBC you want to use when accessing the database. Possible settings are:

```
UseJavaSqlPackage     // typical for Java 1.1
UseJdbcSqlPackage     // typical for Java 1.02
AutoDetectPackage
```

With `AutoDetectPackage`, the application itself attempts to determine which package is appropriate.

If you use `AutoDetectPackage`, WebApplication targets and collection targets (ZIP, JAR and CAB) may not be able to determine whether JDBC classes should be included in the target. For more information, see JDBCPackage and deployment.

The default is `UseJavaSqlPackage` for Java 1.1 applications and `AutoDetectPackage` for Java 1.02 applications.

AutoConnect [**Options** page]
If this is checked, PowerJ automatically connects to the database when the form is created. If it is unchecked, your code must explicitly issue its own instructions to connect to the database (as explained later in this section).

AutoCommit [**Options** page]
If this is checked, each database operation is committed as it is completed.

ReadOnly [**Options** page]
If this is checked, no updates will be allowed through this transaction object.

TraceToLog [**Options** page]
If `true`, the transaction object automatically records important actions in the program's debug log; for example, it records when the transaction connects to the database. If `false`, the actions are not recorded.

TraceToLog only has an effect in Debug mode; it does nothing in Release mode.

Note that this property controls information written to the PowerJ debug log, not the database's trace log.

TransactionIsolation [**Options** page]
Specifies the isolation mode for the database connection. Possible values are:

`TRANSACTION_DEFAULT`
: Uses the default isolation mode for the database.

`TRANSACTION_NONE`
: Transactions are not supported

`TRANSACTION_READ_UNCOMMITTED`
: Dirty reads, non-repeatable reads, and phantom reads may occur.

`TRANSACTION_READ_COMMITTED`
: Dirty reads are prevented; non-repeatable reads, and phantom reads may occur.

`TRANSACTION_REPEATABLE_READ`
: Dirty reads and non-repeatable reads are prevented; phantom reads may occur.

`TRANSACTION_SERIALIZABLE`
: Dirty reads, non-repeatable reads, and phantom reads are prevented.

Catalog [**Options** page]
: Specifies a catalog name to be used by the database connection to select a subset of the tables in the database.

Note: Not all databases support the concept of catalogs.

LoginTimeout [**Options** page]
: States the number of seconds to wait when attempting to connect to the database. If the database doesn't respond to a login request within this time, your program will give up and assume that the connection request failed. A value of 0 specifies an infinite timeout.

Note: The value that you specify for **LoginTimeout** only applies when you are running your program. When you use the **Test Connection** button at design time, the timeout defaults to one minute. To specify a different timeout for **Test Connection**, add a new registry value called `ConnectionTimeout` (under the `HKEY_LOCAL_MACHINE\Software\Powersoft\Optima\2.3\DesignTimeJDBC\` key) and set its value to the number of seconds you want to wait before timing out.

JDBCPackage and deployment

Various types of PowerJ targets allow you to package programs for deployment. You can package programs using the WebApplication target (see WebApplication targets) or using collection targets (see Using collection targets: CAB, JAR, and ZIP).

Both WebApplications and collection targets are designed to gather together all the class files that will be required to run a program. In most situations, PowerJ automatically decides which class files are needed. However, in some cases related to the **JDBCPackage** property of transaction and query objects, PowerJ cannot figure out all the class files that may be required. In such situations, you must manually specify certain class files that should be included.

If a program uses JDK 1.02 and **JDBCPackage** is `AutoDetectPackage`, the following classes must be manually included:

- Your JDBC driver classes. For jConnect, these are found in the file `syb102.zip` in `Java\jconnect` under your main PowerJ folder. For other drivers, consult the driver documentation.
- The classes `powersoft.powerj.db.*`. You can obtain these classes by using `winzip` to extract the appropriate class files from `powerj.zip` in `Java\Powersoft\JDK102\release\lib` under your main PowerJ folder.
- The classes `powersoft.powerj.db.jdbc_sql.*`. You can obtain these classes by using `winzip` to extract the appropriate class files from `powerj.zip` in `Java\Powersoft\JDK102\release\lib` under your main PowerJ folder.
- The classes `sunw.io.*` and `sunw.util.*`. You can obtain these classes from the `Java\Powersoft\JDK102\release\lib\sunw\io` and `Java\Powersoft\JDK102\release\lib\sunw\util` folders under your main PowerJ folder.

If a program uses JDK 1.02 and **JDBCPackage** is `UseJdbcSqlPackage`, the following classes must be manually included:

- Your JDBC driver classes (as described above).

If a program uses JDK 1.1 and **JDBCPackage** is `UseJavaSqlPackage`, the following classes must be manually included:

- Your JDBC driver classes. For jConnect, these are found in the file `sybjdbc.zip` in `Java\jconnect` under your main PowerJ folder. For other drivers, consult the driver documentation.

If a program uses JDK 1.1 and **JDBCPackage** is not `UseJavaSqlPackage`, follow the rules previously stated for JDK 1.02 programs, except that you obtain `powerj.zip` from `Java\Powersoft\JDK11\release\lib` under your main PowerJ folder. The `sunw.io.*` and `sunw.util.*` classes are not required for JDK 1.1.

You include classes manually by using the **Applet Classes** page of the property sheets for the WebApplication or collection target. For more information, see WebApplication targets and Using collection targets: CAB, JAR, and ZIP.

When you execute this kind of applet, you typically specify the names of necessary ZIP files in an `ARCHIVE` parameter for the `<APPLET>` directive, as in:

```
<APPLET CODE=myapplet.class ...
    ARCHIVE="syb102.zip,powerj.zip" ... >
</APPLET>
```

For further information, see Accessing JAR and ZIP files from a web page.

Setting up transaction information at run time

You can set transaction properties at run time using appropriate methods on the transaction object. For example, suppose your program obtains the connection userid and password from the user rather than hard-coding them at design time. The following code sets the transaction object's properties with this information:

```
// String userid;
// String password;
transaction_1.setUserID( userid );
transaction_1.setPassword( password );
```

Connecting to the database

If you mark **AutoConnect** on the transaction's property sheet, the program automatically connects to the database when the program creates the form that contains the transaction object. Otherwise, your code must explicitly connect to the database at run time, using the **connect** method of Transaction:

```
transaction_1.connect( );
```

This connects to the database using the information associated with the transaction object. If you have not specified a userid and password for the connection, the program prompts the user to enter a password and userid.

The **Connected** property tells you whether the transaction object is currently connected to a database:

```
boolean connected = transaction_1.isConnected();
```

Disconnecting

The **disconnect** method of Transaction discontinues an existing connection:

```
transaction_1.disconnect();
```

Managing transactions manually

If the transaction's **AutoCommit** property is `true`, changes are automatically committed when they are made. This is the default. If you turn off **AutoCommit** using **setAutoCommit**, you must commit changes explicitly.

The **commit** method of Transaction commits changes to the database:

```
transaction_1.commit();
```

The **rollback** method of Transaction cancels any changes made to the database, provided those changes have not yet been committed:

```
transaction_1.rollback();
```

The **commit** and **rollback** methods have no useful effect if **AutoCommit** is `true`.

Note: Some databases close all open cursors when you call **commit**.

A hint for setting up transactions

When you are developing a program that uses databases, it is tempting to specify the database administrator's userid at design time (for example, `DBA`). However, this may not be a good idea if the program will be used by non-administrators. Certain queries and other operations are valid for administrators but not for non-administrators. To develop programs for non-administrators, specify a non-administrator userid for the transaction object at design time. This lets you test the typical behavior of the program, not the special case when the user is an administrator.

Connecting to supplied JDBC drivers

The following sections examine each of the database drivers included with the PowerJ package and give an example of the transaction object properties required to establish an appropriate connection.

For information on troubleshooting database connections, see Debugging database applications.

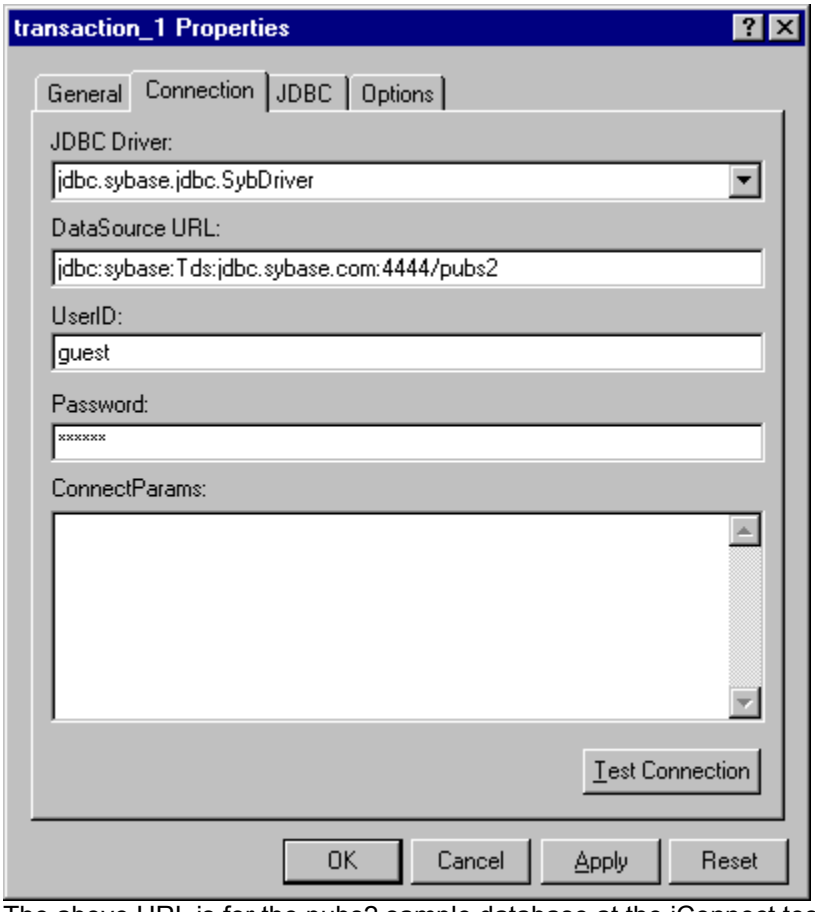
Connecting to jConnect

The jConnect database driver is included as part of the Integrated Components made available when you installed PowerJ. If you did not install it originally, you must install the Integrated Components part of PowerJ before using jConnect.

In order to use jConnect against a SQL Anywhere database, you must have the Open Server Gateway running. For more information about the Open Server Gateway, see The jConnect driver.

Important: In most cases, you will also want to prepare your database with special tables and stored procedures. For example, the Query Editor in PowerJ will not work unless you prepare your database. For more information, see Using the query editor with jConnect.

The following picture shows typical transaction property settings for using jConnect.

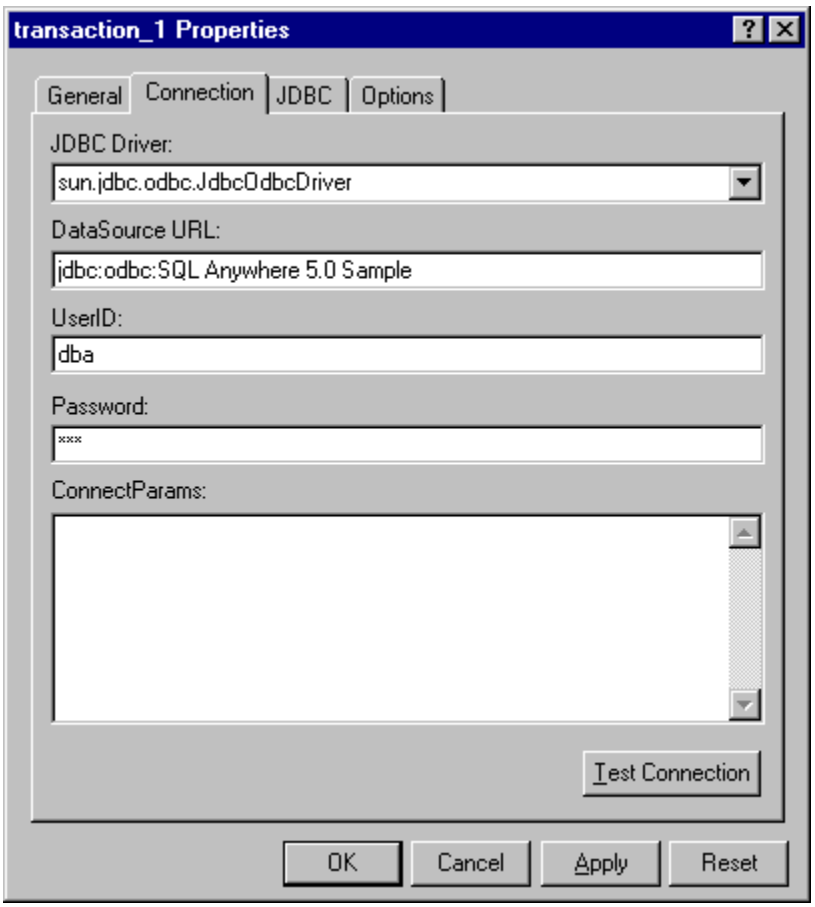


The above URL is for the pubs2 sample database at the jConnect test site. The password is `sybase`. For more information on the jConnect test site, see The jConnect test site.

Connecting to the JDBC-ODBC bridge

The JDBC-ODBC bridge is included as part of the PowerJ package. It allows a Java program to connect with an ODBC database. For information on the restrictions of the JDBC-ODBC bridge, see The JDBC-ODBC bridge.

The following picture shows typical transaction property settings for using the JDBC-ODBC bridge.



Connecting to VisiChannel

To connect to a database using VisiChannel, it must be running on the same machine as the web server, and that machine must have an ODBC connection configured for the database. The PowerJ installation includes a program for installing this server; run this program and follow the instructions. Additional information is provided by HTML files in the same folder.

The Visigenic client comes in two versions: one for JDK 1.02 applet targets (that uses the `jdbc.sql` packages) and one for targets that are JDK 1.02 applications , JDK 1.1 applets, or JDK 1.1 applications.

The driver package names are different: the one for JDK 1.02 has the package name `visigenic.dl.jdbc.sql` and the other uses `visigenic.jdbc.sql`. The "dl" stands for "downloadable". This non-standard naming convention prevents the PowerJ transaction object from automatically selecting the right package, so you must enter the appropriate driver yourself.

Important Notice for Using Applets: In order to work with Java applets that are downloaded from a web server, you must install the WebSupport option with VisiChannel for JDBC. This option is installed by default.

The WebSupport option requires that the JDK be installed on the machine that is running both VisiChannel for JDBC and the web server.

Important: With this release of VisiChannel for JDBC, the WebSupport option must be started with JDK 1.02 or an equivalent Java run-time library which incorporates JDK 1.02. It does *not* work with JDK 1.1 or higher.

Please make sure that you have JDK 1.02 installed on your web server machine (the same machine where the VisiChannel for JDBC server is installed) before you use VisiChannel for JDBC with applets.

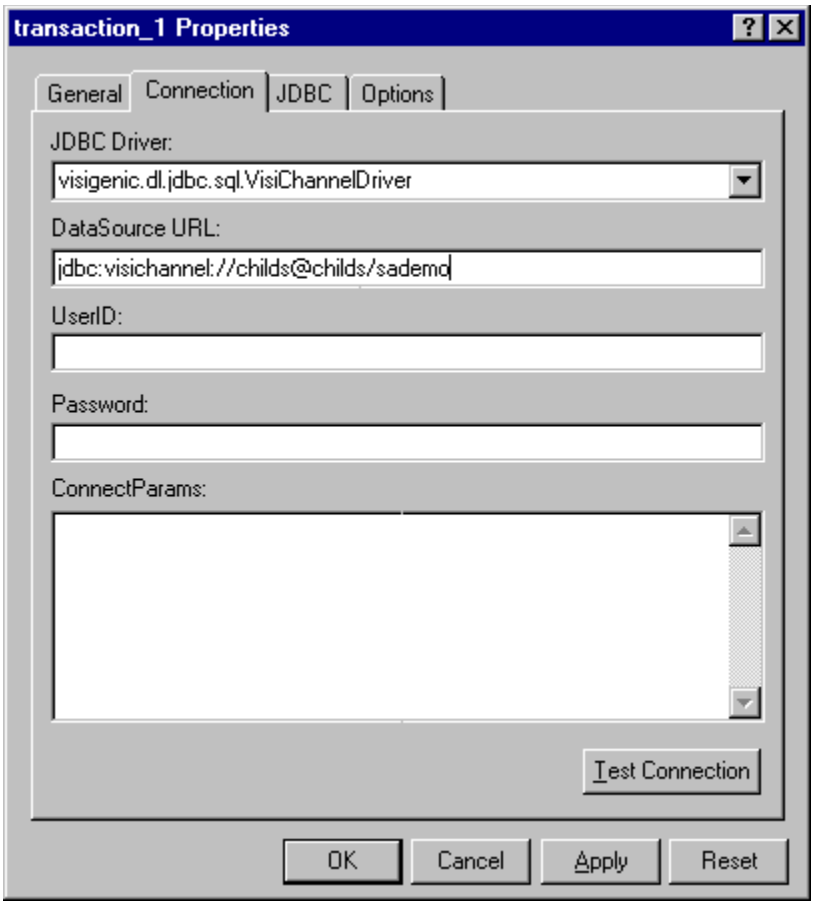
Visigenic client for 1.02 applet targets

When you are using the Visigenic client with the `jdbc.sql` JDBC packages, the URL syntax is typically:

```
jdbc:visichannel://server-name@host-name:port/ODBC Datasource Name
```

The `:port` parameter is optional; if you omit it the default port of 14000 will be used.

The following picture shows typical transaction property settings for using VisiChannel. In this case, the machine name is `childs` and your VisiChannel server is also `childs`. The ODBC data source defined on the server is named `sademo`.



Note that the driver name has `dl` in the package:

```
visigenic.dl.jdbc.sql.VisiChannelDriver
```

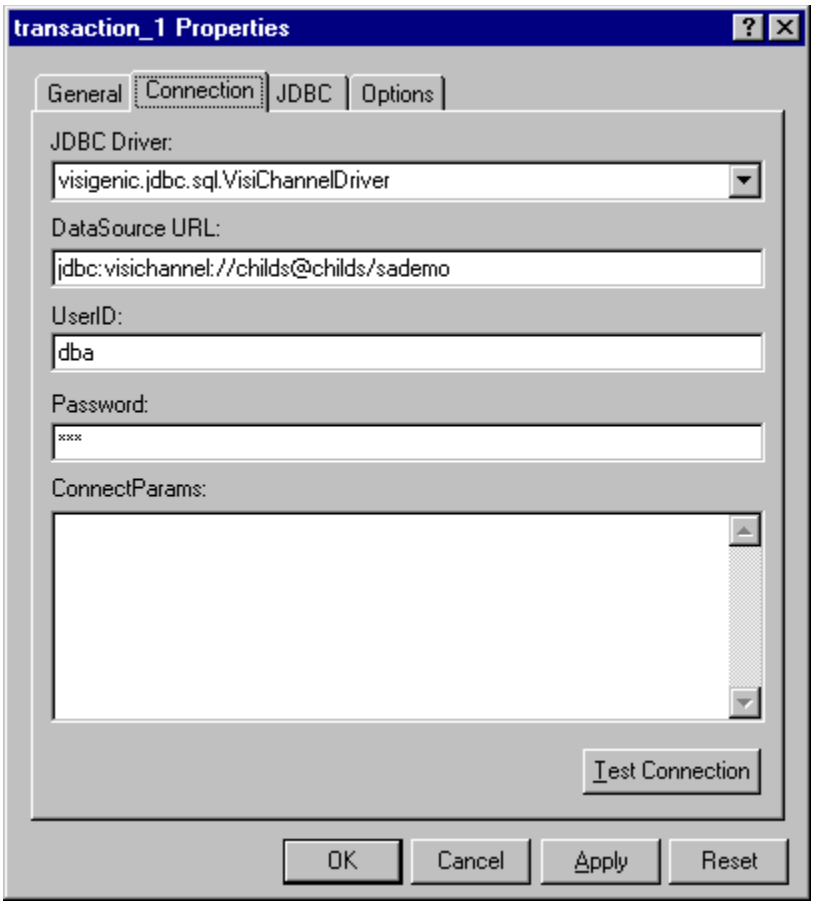
Visigenic client for applications and 1.1 applets

When you are using the Visigenic client with the `java.sql` JDBC packages, the URL syntax is typically

```
jdbc:visichannel://server-name@host-name:port/ODBC Datasource Name
```

The `:port` parameter is optional; if you omit it the default port of 14000 will be used.

The following picture shows typical transaction property settings for using VisiChannel. In this case, the machine name is `childs` and your VisiChannel server is also `childs`. The ODBC data source defined on the server is named `sademo`.



Note that the driver name does not have `dl` in the package:

```
visigenic.jdbc.sql.VisiChannelDriver
```

Connecting to JetConnect

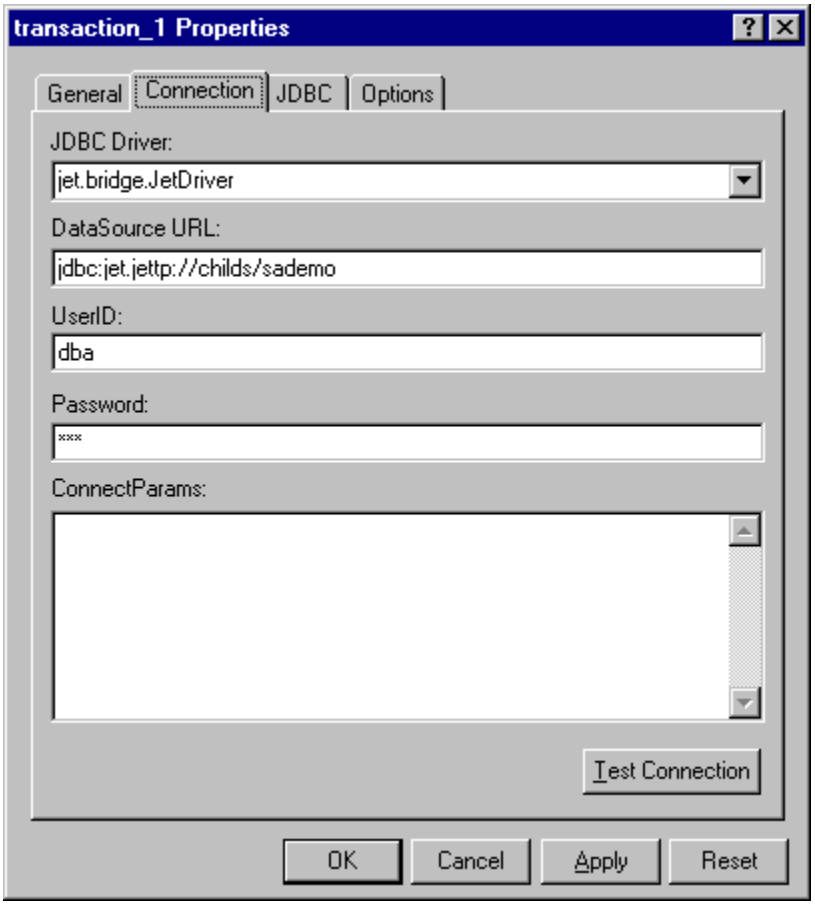
To connect to a database using JetConnect, it must be running on the same machine as the web server, and that machine must have an ODBC connection configured for the database. The PowerJ installation includes a program for installing this server; run this program and follow the instructions. Additional information is provided by HTML files in the same folder.

The JetConnect client does not support JDK 1.02 applets, since it uses the `java.sql` JDBC library.

When you are using the JetConnect client with the `java.sql` JDBC packages, the URL syntax is typically

```
jdbc:jet.jettp://hostname/ODBC Datasource name
```

The following picture shows typical transaction property settings for using JetConnect. In this case, the machine name is `childs` and the ODBC data source defined on the server is named `sademo`.



Query objects

A *query object* represents a query on a specific database and can be used to execute any SQL statement. Most programs that work with databases make extensive use of query objects. Transaction objects just handle the process of connecting to the database; after that, all interactions with the database are done through query objects.

Queries are represented by the Query class. PowerJ gives query objects default names of the form *query_N*. On the **Database** page of the component palette, query objects are represented by the following button:

If a form makes a query on a database, you should place a query object on the form. When you place a query object on a form, an icon appears on the form to show that the object is there; however, this icon will not be visible at run time.
You can specify SQL queries by typing in the query as a normal text string or by constructing the query using the PowerJ Query Editor. For more information, see The Query Editor.
Query objects are serializable. In particular, if you have an open query, you can serialize it and re-read the query later, with all the query's data intact.

Query subclasses

The Query class is used as a base class for subclasses java_sql.Query (for queries compiled against the `java.sql` libraries) and jdbc_sql.Query (for queries compiled against the `jdbc.sql` libraries).

Note: For a discussion of the `java.sql` and `jdbc.sql` libraries, see JDBC.

To initialize a query object, PowerJ generates code of the following form:

```
Query query_1 = null;
query_1 = powersoft.powerj.db.Transaction.createQueryByName(
    "Form1.transaction_1", "Form1.query_1", true )
```

The **createQueryByName** creates a Query object of the appropriate type based on the Transaction subclass used for the specified transaction object.

The implementation of queries

Each query object has several associated data buffers:

- The original buffer: this is a read-only buffer containing data as originally read from the database.
- The primary buffer: this begins as a copy of the original buffer. However, as the user makes changes in the data fetched from the database, the same changes are made in the primary buffer. Therefore, the primary buffer represents the original buffer, modified by any changes, additions, and deletions that the user has made since the data was originally fetched.
- The deletion buffer: as the user deletes records from the primary buffer, the deleted records are moved into the deletion buffer. This makes it possible to "undelete" records by moving them from the deletion buffer back to the primary buffer.
- The filter buffer: it contains rows filtered out of the primary buffer using filter expressions for rows and/or columns.

The values which the user sees are the values that are currently in the primary buffer. The original buffer never changes until changes are made to the database itself.

To update the database, PowerJ generates SQL statements to modify the contents of the database so

that they match the contents of the primary buffer.

- Associating a query object with a transaction object
- Query properties
- Setting up query information at run time
- Run-time only Query properties
- Opening a query
- The results of a query
- The cursor
- The size of the result set
- Positions in the result set
- Closing a query

Associating a query object with a transaction object

Query objects are almost always associated with a transaction object. The transaction object specifies the database on which the query will act.

Usually, you will place the query object on the same form as the transaction object. However, a query may be placed on a different form from its associated transaction object if you wish.

If you do not associate a transaction object with a query object, you must write your own code for obtaining a ResultSet object and assigning it to the query object. For example, if you are using Jaguar CTS to obtain data from a remote database, you might be able to obtain a ResultSet without establishing a connection through a transaction object. For more information on Jaguar CTS, see PowerJ and Jaguar CTS.

If a query object is not linked to a transaction object, you must write event handlers for events that copy results to the query and for events that update the database from information associated with the query. For example, you must write a **queryUpdatingDatabase** event handler to deal with the operation of updating the database from the query, since the program can't perform this update in the normal way (i.e. connecting to the database through a transaction object).

For more information on the events associated with query objects, see Query events.

Query properties

After you place a query object on a form, you should set the properties for the query using the object's property sheet. The property sheet contains the following items:

Transaction [**Query** page]
The transaction object associated with the database that you want to query. Click the arrow at the end of this entry to see a list of transaction objects defined on the current form; click one of the names in this list.

SQL [**Query** page]
Contains the SQL statement you want to execute on the database. Many programs will leave this blank at design time, then fill in an appropriate SQL query at run time. At design time, you can use the query editor to construct the desired statement. For more information, see The Query Editor.

Statement Type [**Query** page]
Provides more information about the given SQL statement. By specifying a statement type, you can improve the efficiency with which your application processes the SQL statement. Possible values are:

General (no parameters or calls) (`Query.GENERAL_STATEMENT`)
The SQL statement contains no placeholders (`?`) or procedure calls.

Prepared (parameters allowed) (`Query.PREPARED_STATEMENT`)
The SQL statement may contain input-only placeholders, represented by `?` characters.

Callable (calls and parameters allowed) (`Query.CALLABLE_STATEMENT`)
The SQL statement may contain placeholders (input or output) and procedure calls.

Automatically detect from SQL
PowerJ examines the SQL statement to determine any special processing that may be necessary. If a statement contains a string of the form `{call ...}`, it is assumed to contain a call to a stored procedure. If a statement contains a question mark (`?`), it is assumed to contain a placeholder. Otherwise, it is taken to be a "general" statement. Using this information, PowerJ chooses one of the other three possible values.

Note: In this document, the terms *placeholder* and *bound parameter* are used interchangeably when referring to SQL statements.

JDBCPackage [**JDBC** page]
This option lets you specify which version of JDBC you want to use when accessing the database. Possible settings are:

```
UseJavaSqlPackage     // typical for Java 1.1
UseJdbcSqlPackage     // typical for Java 1.02
AutoDetectPackage
```

With `AutoDetectPackage`, PowerJ checks the transaction object associated with this query object and uses the same **JDBCPackage** property as the transaction object. Note that this means you cannot use `AutoDetectPackage` if the query object is not associated with a transaction object.

The default is `UseJavaSqlPackage` for Java 1.1 applications and `AutoDetectPackage` for Java 1.02 applications.

If you do not specify `AutoDetectPackage`, the application does not check the **JDBCPackage** setting for the transaction object associated with this query object. This means that the transaction object and query object might have different settings, resulting in numerous errors. If you change the value for the transaction object, you should specify `AutoDetectPackage` for the query object so that the application will match it to the transaction object.

AllowUpdates [**Updates** page]
If this option is checked, PowerJ lets the user make changes (updates) to the database. If this option is blank, PowerJ does not allow updates.

UpdateMode [**Updates** page]
May have one of the following values:

`IMMEDIATE_UPDATES`
PowerJ updates the database with changes whenever you call the **update** or **delete** method by automatically calling batch updates for you.

`BATCH_UPDATES`
PowerJ saves up additions, changes, and deletions, rather than making updates immediately. The database is not updated until you call the query's **batchUpdate** method.

UpdateType [**Updates** page]
Determines how the `WHERE` clause of `UPDATE` and `DELETE` SQL statements are generated to update or delete rows in the database. Possible values are:

`KEYS_ONLY`
Only uses the primary keys (as specified by the **PrimaryKeyColumns** property).

`KEYS_AND_SEARCHABLE`
Uses both the primary keys and the searchable columns, as marked by the JDBC driver.

`KEYS_AND_MODIFIED`
Uses the primary keys and the modified columns.

In all cases, the `WHERE` clause is generated using the original column values as stored in the original buffer.

It is generally best to set **UpdateType** to `KEYS_AND_SEARCHABLE` or `KEYS_AND_MODIFIED`. With these settings, updates will fail if another user has changed the same record that you want to modify; if you set `KEYS_ONLY`, you may not be able to determine that your update has failed. PowerJ does not automatically notify you when an update fails—you must test for the **update** method failing.

If you set **UpdateType** to `KEYS_AND_SEARCHABLE` or `KEYS_AND_MODIFIED`, your database must be set up to ignore trailing blanks.

KeyUpdate [**Updates** page]
Determines whether `UPDATE` statements should be used to update primary keys. Not all databases support updating primary key fields using an `UPDATE` statement; some require that the row be deleted and then reinserted with the new key value. Possible values for **KeyUpdate** are:

`USE_UPDATE`
Uses `UPDATE` statements.

`USE_DELETE_AND_INSERT`
Deletes the original row and inserts a new one with the new primary key value.

BindUpdates [**Updates** page]
Determines whether bound parameters are used when doing updates. Bound parameters can be used for all data types; they are *always* used for binary, stream, or large data values.

If **BindUpdates** is marked with a check, bound parameters are used. Otherwise, data is inserted

directly into the SQL statement.

UpdateConnectionMode [**Updates** page]
Determines whether a database connection should automatically be established when database updates occur (when batch update is called). Possible values are:

`NO_AUTOCONNECT`
If there is no active connection, the query does not try to establish one. This means that update operations will fail.

`CONNECT_AND_LINGER`
If there is no active connection, the query tries to establish one. If the connection attempt succeeds, the query maintains the connection until you explicitly break the connection by calling **close** or **detach**.

`CONNECT_AND_CLOSE`
If there is no active connection, the query tries to establish one. If the connection attempt succeeds, the query makes the required updates, then automatically closes the connection again.

UpdateQuery [**Updates** page]
Specifies a query object to be used when making updates. This second query object is used for executing SQL `UPDATE`, `INSERT`, and `DELETE` statements, to avoid disturbing the result set associated with the original query. If the **UpdateQuery** property is null, PowerJ automatically creates a second query object to do the updates.

In general, you can leave this property null. However, if you want to know why an update failed, you can create your own Query object to use as the **UpdateQuery**, and add an **exceptionThrown** event handler to this object. This event handler will then be invoked if any update operations fail.

The **UpdateQuery** object can be a Query object or an object of a class derived from Query.

Any exceptions that occur during an update operation are thrown by both the **UpdateQuery** object and the original Query object.

PrimaryKeyColumns [**Columns** page]
Specifies the name or number of the primary key column in the SQL statement. Filling in this property is mandatory if you are doing updates. Often, a value of 1 (signifying the first column in the database) will suffice for this property. If there is more than one key, separate their names or numbers with semicolons, as in `2;7`.

ColumnTableNames [**Columns** page]
Sets the list of column-to-table mappings for the result set. The mappings set with this property override any values returned in the metadata for the result set. The primary purpose of this property is to provide extra information that may be missing with certain JDBC drivers.

Table names are necessary for generating `UPDATE`, `INSERT`, and `DELETE` statements. If the JDBC driver does not provide the table names, they must be supplied through this property or by handling the **queryUpdatingDatabase** event.

The value of **ColumnTableNames** are mappings of the form

```
table=column,column,...;table=column,...
```

where `table` is a table name and `column` is a column name or position. The short form

```
table=*
```

can be used to indicate that all columns are from the specified table.

ReadOnlyColumns [**Columns** page]
Sets the list of read-only columns for the result set. This list is used to set values returned in the metadata for the result set. The primary purpose of this property is to provide extra information that may be missing with certain JDBC drivers.

Read-only columns are necessary for generating correct `UPDATE` and `INSERT` statements (for example, so that autoincrement columns are not included in the statement). If you do not specify read-only columns with this property, you should handle the **queryUpdatingDatabase** event to generate the correct SQL statement to update the database.

The value of **ReadOnlyColumns** is a list of the form

```
column;column;...
```

where each `column` entry is a column name or position.

AutoOpen [**Options** page]
Automatically opens the query when the query object is created, executes the associated SQL statement, and moves to the first row retrieved. By default, **AutoOpen** is `true`.

AutoEdit [**Options** page]
If `true`, the query automatically goes into edit mode if the user makes a change in a row. If `false`, your code must explicitly put the query into edit mode if you want to modify an existing row. By default, **AutoEdit** is `false`. For further information, see Modifying existing rows.

TraceToLog [**Options** page]
If `true`, the query object automatically records important actions in the program's debug log; for example, it records when the query is opened. If `false`, the actions are not recorded.

TraceToLog only has an effect in Debug mode; it does nothing in Release mode.

Note that this property controls information written to the PowerJ debug log, not the database's trace log.

QueryTimeout [**Options** page]
States the number of seconds to wait when attempting to execute a SQL statement. If the statement isn't executed within this time, an error is returned for the query. A value of 0 specifies an infinite timeout.

The value that you specify for **QueryTimeout** only applies when you are running your program. When you test the query at design time, it always uses an infinite timeout.

Important: In order to perform updates with any JDBC driver, you *must* set the following properties: **PrimaryKeyColumns** and **ReadOnlyColumns** (if there are any such columns). For some JDBC drivers, such as jConnect, you may also have to specify the **ColumnTableNames** column. This is required because JDBC does not supply sufficient information to determine which columns are primary keys, and the program must know the primary key columns in order to perform updates. Since the program can't get the information from the database, you must supply it.

Setting up query information at run time

You can set query properties at run time using appropriate methods on the query object. For example, suppose your program obtains SQL statements from the user rather than hard-coding them at design-time. The **setSQL** method of Query sets up the query object with this information:

```
String userStatement = "select * from dba.employee";
query_1.setSQL( userStatement );
```

Hint: If you are specifying your own SQL statement, make sure table names are fully qualified. For example, specify `dba.employee` instead of just `employee`.

Run-time only Query properties

A number of Query properties can only be set at run time. The following list discusses some of these:

Opened

isOpened returns `true` if the query is open and `false` otherwise. There is no **setOpened** function.

ReadOnly

isReadOnly returns `true` if the statement is not open or the concurrency level is read-only. It returns `false` otherwise. There is no **setReadOnly** function; however, you can use **setAllowUpdates** to control whether a query is read-only.

Opening a query

The **open** method of Query executes a query on a database. For example,

```
query_1.open();
```

executes the SQL statement in `query_1` on whatever database is associated with the query object.

If you have turned on the **AutoOpen** property for the query object, your program automatically executes **open** after the query object is created.

Once you open a query, you typically work with the result set returned by that query, often using bound controls. However, you can just use **open** to execute a series of SQL statements, as in:

```
query_1.setSQL( "insert ..." );
query_1.open();
query_1.setSQL( "insert ..." );
query_1.open();
```

The above code executes two `INSERT` statements on the database, without worrying about result sets.

The results of a query

Opening a query executes the query's SQL statement. Some types of statements obtain information from the database; for example, the `SELECT` statement obtains data selected according to specified criteria. Other types of statements do not obtain data; for example, `INSERT` and `UPDATE` both place information into the database.

If a statement obtains information from the database, the information is called a *result set*. A result set contains zero or more *rows* of information. Each row contains one or more *columns*, and each column in a row contains a data value. The *current row* is the row whose data is currently available to the Query object.

The first row of the result set is numbered 1 (one), not 0. Similarly, the first column in every row is numbered 1, not 0.

When you use **open** to execute a `SELECT` statement (or some other statement that obtains information from the database), the information is not delivered to your program immediately. To obtain the information, you must use methods which explicitly retrieve the data.

Between the **open** operation and the first data retrieval, you have the opportunity to specify what your program will do with the data that is retrieved. The easiest way to display the retrieved data is to use one or more *bound controls*, as explained in Bound controls.

The cursor

The query object maintains a *cursor* pointing to the current row in the result set. If a query object has bound controls, these controls display values from the current row. The Query class offers several methods which change the cursor from one row to another; for more information, see Moving through the result set.

The size of the result set

The **getColumnCount** method returns the number of columns in the result set:

```
int cols = query_1.getColumnCount();
```

If the return value is zero, there are no columns in the result set; this happens, for example, when the query doesn't select any columns from the database. A result set with no columns is called a *null* result set.

If the return value of **getColumnCount** is greater than zero, the value is the number of columns in a row; however, there is no guarantee that the result set actually contains any rows. There may be no rows which meet the query's criteria. A result set that has columns but has no rows is called an *empty* result set.

Positions in the result set

The Query class offers a number of methods to determine your current position in a result set. The functions

```
boolean first = query_1.isFirst( );
boolean last  = query_1.isLast( );
```

let you test whether the current row is the first or last row of the result set. The functions return `true` if the current row has the specified position.

The functions

```
boolean start = query_1.isBefore( );
boolean end   = query_1.isAfter( );
```

let you test whether you are before the first row in the result set or after the end of the last row in the result set.

Closing a query

The **close** method of Query closes an open query:

```
query_1.close();
```

A number of Query methods automatically close the query if it is currently open. For example, the **open** method automatically closes the query before executing a new query.

Closing a query frees up the memory used to hold the results of the query. It also unbinds any data that is stored in bound controls or bound arrays.

Debugging database applications

The key to debugging database applications is to use the *debug log*.

- **To view the debug log:**

1. On the **View** menu of the main PowerJ menu bar, click **Debug Log**.

This opens the debug log as a separate window.

Note: By default, the debug log is cleared before each run of a target. Therefore, the contents of the debug log represent the results of the most recent run.

You can gather debugging information by turning on the **TraceToLog** properties of the transaction and query objects.

The debug log provides information about every action attempted by the transaction and/or query objects. Errors are marked on the left margin with white X's in red circles.

Troubleshooting connections
Dealing with database exceptions

Troubleshooting connections

You may find that a connection fails when you try testing it via the **Test Connection** button at design time, or when you try running your program.

When you test a connection at design time, the JDBC Connection Monitor will display some diagnostic messages indicating the progress of the test. However this information is very limited compared to the information written to the debug log when you run your program. Therefore, if **Test Connection** fails you may be able to find out more information by running your program.

- **Testing a connection at design time**

1. Use the **Test Connection** button from the **Connection** page of the transaction object property sheet.
2. Read the messages in the JDBC Connection Monitor and note whether the connection succeeds or fails.
3. If the connection fails, and the cause is not obvious from the JDBC Connection Monitor, then try running your program with **TraceToLog** turned on and the debug log displayed.

Note: If you are trying to access a remote computer and the connection fails after a minute or so, you may want to extend the timeout for **Test Connection** from its default of one minute. At design time, the **Test Connection** timeout is independent of the setting for the transaction object's **LoginTimeout** property. To specify a different timeout for **Test Connection**, add a new registry value called `ConnectionTimeout` (under the `HKEY_LOCAL_MACHINE\Software\Powersoft\Optima\2.3\DesignTimeJDBC\` key) and set its value to the number of seconds you want to wait before timing out.

You can find out more about a failed connection if you run your program instead of using **Test Connection**.

- **If a connection fails when you run your program:**

1. Make sure that the debug log is open (via SHIFT+F11 or click **Debug Log** on the **View** menu) and that the **TraceToLog** property is set for the transaction.
2. Run the program.
3. Check the debug log and see if the JDBC driver was even loaded. You might see some kind of exception message from the driver giving a reason for the failure.
4. If that doesn't work, or if the driver couldn't be loaded, make sure that the **JDBCPackage** property is set correctly for the type of driver you are using. Drivers compiled against `java.sql` cannot be used if **JDBCPackage** is set to **UseJdbcSql** and similarly drivers compiled against `jdbc.sql` cannot be used if **JDBCPackage** is set to **UseJavaSql**.

The rest of this section points out some common problems that can lead to failed connections or problems with queries. These points have been covered elsewhere in this chapter, but are repeated here for your convenience.

If your driver does not load, you should check the following:

- Check the URL, paying close attention to case (the case may matter in parts of the URL).
- Have you set up the ClassPath correctly to include the path of your JDBC driver classes?

If you are using the JDBC-ODBC bridge or a type 2 driver, you should also check the following:

- Are there any security restrictions that might prevent the use of sockets? (For example, you cannot

use a type 1 or 2 driver in an applet run in a browser.)

- Can your JDBC driver be used with the Microsoft VM? (The native interfaces of the Sun VM and Microsoft VM are not compatible, so drivers using native interfaces do not work with both VMs.)
- Can Windows locate any needed driver DLLs? You may need to copy them to your Windows system folder.

If you are using the jConnect driver, check the following:

- If you are running against SQL Anywhere, make sure that you have configured and are running the Open Server Gateway.

If you are having trouble with a query, check the following:

- Have you used the **TraceToLog** property in the Transaction and Query objects and turned on display of the debug log?
- If you are trying to perform an update or insertion, have you set the **PrimaryKeyColumns**, **ReadOnlyColumns**, and **ColumnTableNames** properties?
- If you are using jConnect, have you prepared the database? The Query Editor will not work with jConnect unless you have prepared the database.

Dealing with database exceptions

When an exception occurs for a transaction or query object, PowerJ triggers an **exceptionThrown** event for that object. By writing your own **exceptionThrown** event handler, you can deal with selected problems that might occur. For more information, see Exception events.

Performance tips

Once you have debugged your database program, you can take some steps to speed up database access in PowerJ:

- Test the release mode version of your application. It is not sufficient to turn off debugging—you need to tell PowerJ to use the release-mode versions of the database classes. For information on how to run in release mode, see Debug and release versions of a target
- By default the **TraceToLog** property is on for transaction and query objects. While this has no effect in release mode, in debug mode turning **TraceToLog** on slows down database access by a factor of 10 or more. Once things seem to be working, you can turn off this property for faster development. (These properties are on by default, since the tracing is extremely useful for debugging your database applications.)
- Limit the columns in your queries. The more columns in your query, the more data gets fetched. If you are not actually using columns, don't include them in the result set unless they are primary keys (since you will need them if you want to do updates). For example, with the SQL Anywhere sample database a query like `SELECT * FROM EMPLOYEE` causes 20 columns to be fetched each time the cursor moves, which is a waste if all you need are the `emp_id` and `emp_lname` columns.

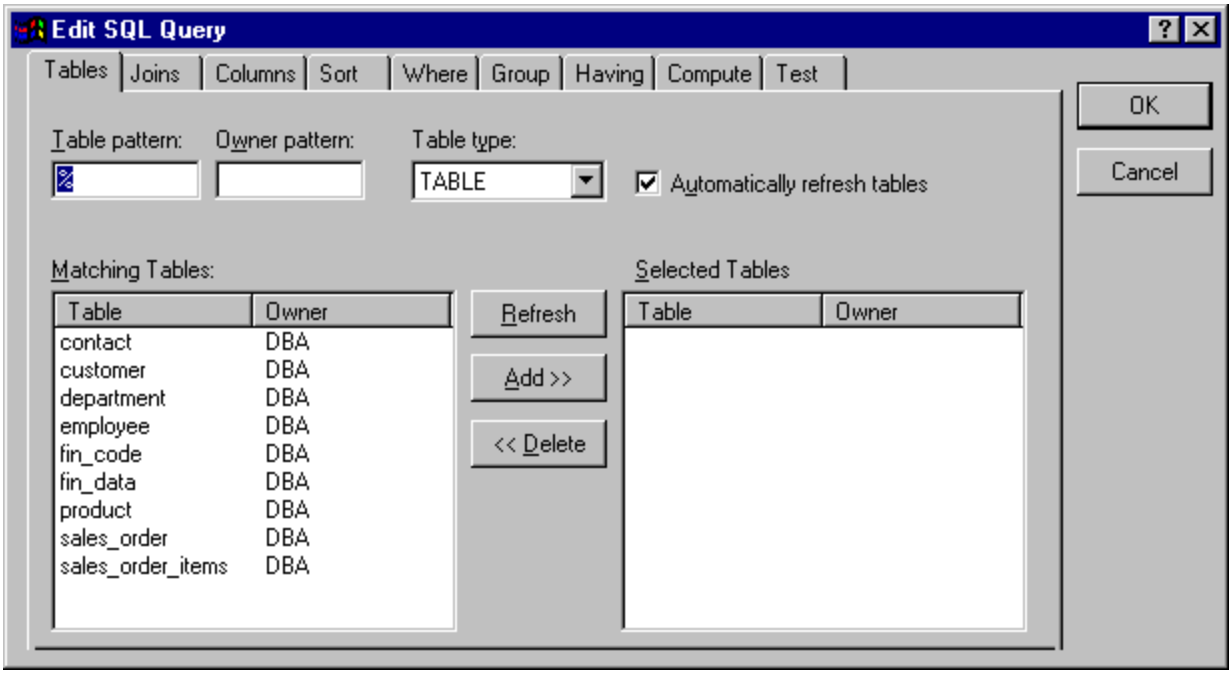
The Query Editor

The *Query Editor* helps you construct the SQL statement that is associated with a Query object. The Query Editor is opened at design time, once you have placed the Query object on a form and have bound the object to a valid transaction object. The Query Editor uses the information specified in the transaction object to connect to the database and examine the database's contents. In this way, the Query Editor can obtain information about the database (for example, the names of database columns) and can perform test queries on the database to make sure that you have constructed your query properly.

- **To open the Query Editor:**

1. On the **Query** page of the Query object's property sheet, click **Edit**.
2. When the Query Editor window opens, click **Refresh**.

The Query Editor displays a window for constructing SQL statements:



If you click **OK**, PowerJ returns to the property sheet for the Query object. You can then test this query by clicking **Test** on the **Query** page of the property sheet, or by using the **Test** page of the Query Editor.

Dates and the query editor

The query editor is written using JDK 1.1. The most important consequence of this is that the query editor analyzes date values in a database using the JDK 1.1 definition of Date, even if you intend to generate a JDK 1.02 target.

As discussed in Date and time classes, the JDK 1.02 implementation of Date typically creates invalid Date values for any date before January 1, 1970; however, the JDK 1.1 implementation of Date handles such dates correctly. This may lead to the confusing situation where the query editor can correctly obtain pre-1970 dates from the database but your program cannot. There is no good workaround for this problem except to convert your program to JDK 1.1—the trouble arises from inadequacies in JDK 1.02 itself.

Using the query editor with jConnect

In order to use the query editor with the current version of jConnect, the database must have some special tables and stored procedures installed.

SQL Anywhere

The required tables and stored procedures are automatically installed for you if you are using the sample SQL Anywhere database that comes with the PowerJ package. If you are using a different database, you must follow the steps given below.

- **To prepare a SQL Anywhere database:**

1. Locate the file `sql_anywhere.sql` stored in the `jconnect/sp` folder under the folder where you installed additional features.
2. Start your database and connect to it using ISQL (available through the **Sybase SQL Anywhere 5.0** window, which can be opened from the PowerJ program group).
3. At the ISQL prompt, use the `read` command to read in the `sql_anywhere.sql` file. For example:

```
read c:\powersoft\powerj20\java\jconnect\sp\
   sql_anywhere.sql
```

4. Press F9.

This installs the tables and stored procedures in the database. You must repeat these steps for every SQL Anywhere database that you intend to access.

Note: The above steps must be taken before you use the query editor in any context. For example, if you use the Form Wizard to create a database form, the wizard may call the query editor to create a query. Therefore, the Open Server Gateway must be running when you are using the Form Wizard in this way, and you must have installed the required tables and stored procedures in the database being accessed.

The instructions in `sql_anywhere.sql` work whether you are using jConnect 2.2 (for JDK 1.02) or jConnect 3.0 (for JDK 1.1). Earlier versions of PowerJ provided a similar script, but the old script was not designed to work with jConnect 3.0. Therefore, you should get the latest version of `sql_anywhere.sql` and run this script against all your databases, even if you ran earlier versions of the script against the same databases.

Sybase Adaptive Server Enterprise

If you are using a Sybase Adaptive Server Enterprise database (formerly SQL Server), you must follow the steps given below to set up the required tables and stored procedures.

This operation requires database administration privileges. If you are not sure how to perform the steps, contact your database administrator.

- **To prepare an Adaptive Server database:**

1. Locate the file `sql_server.sql` stored in the `jconnect/sp` folder under the folder where you installed additional features.
2. Use the `isql` program to connect to your database and read the `sql_server.sql` file.

This installs the tables and stored procedures in the database. You must repeat these steps for every

Sybase Adaptive Server Enterprise database that you intend to access.

Note: The above steps must be taken before you use the query editor in any context. For example, if you use the Form Wizard to create a database form, the wizard may call the query editor to create a query. Therefore, you must have installed the required tables and stored procedures in the database when you are using the Form Wizard in this way.

The Tables page

The **Tables** page specifies the database tables that should be included in the query. This generates a `FROM` clause in the SQL statement being constructed. All tables listed in the Selected Tables list will be included.

- **To select a table:**

1. Click on the name of the table in the Matching Tables list, then click **Add**; or
2. Double-click the name of the table in the Matching Tables list; or
3. Drag the name of the table from the Matching Tables list to the Selected Tables list.

Table type specifies the type of tables listed in the Matching Tables list. By choosing a different table type, you can get a different list of tables.

The **Table pattern** and **Owner pattern** boxes let you restrict the set of tables displayed in the Matching Tables list. A pattern is a string that may contain the character `%` standing for any string of zero or more characters. For example, if you specify `emp%` for **Table pattern**, the Matching Tables list displays all tables whose names begin with the characters `emp`. Similarly, if you specify `%cust%`, the Matching Tables list displays all tables containing `cust` anywhere in their names.

The **Table pattern** entry controls which table names are displayed. The **Owner pattern** text box lets you restrict entries based on the owner name. If you change **Table pattern** or **Owner pattern**, click **Refresh** to get the list of tables which match the given pattern(s).

The Joins page

The **Joins** page specifies the way that selected tables will be joined. For an explanation of all the types of joins available, see your SQL manual.

- **To specify a join:**

1. Click a table from the **Table 1** list.
2. Click a type of join from the **Type** list.
3. Click a table from the **Table 2** list.
4. Click **Add**.

The Columns page

The **Columns** page lets you select the columns that will be included in the result set. This generates a `SELECT` clause in the SQL statement.

The Available Columns list displays columns as a tree view, with the top levels of the tree occupied by the tables selected from the database. Expanding these levels displays the columns defined within each table. All columns listed in the Selected Columns list are retrieved by the query.

- **To select a column:**

1. Click on the name of the column in the Available Columns list, then click **Add**; or
2. Double-click the name of the column in the Available Columns list; or
3. Drag the name of the column from the Available Columns list to the Selected Columns list.

You can change the position of an item in the Selected Columns list by clicking on the item and then clicking **Move Up** or **Move Down**.

The Sort page

The **Sort** page describes how the database should sort the results of the query. This generates an `ORDER BY` clause in the SQL statement.

The **Sort by** column lists the sorting items in order of priority, and whether the sort should be ascending or descending. For example, the following diagram sorts by the employee's last name, then by employee's first name when employees have the same last name.

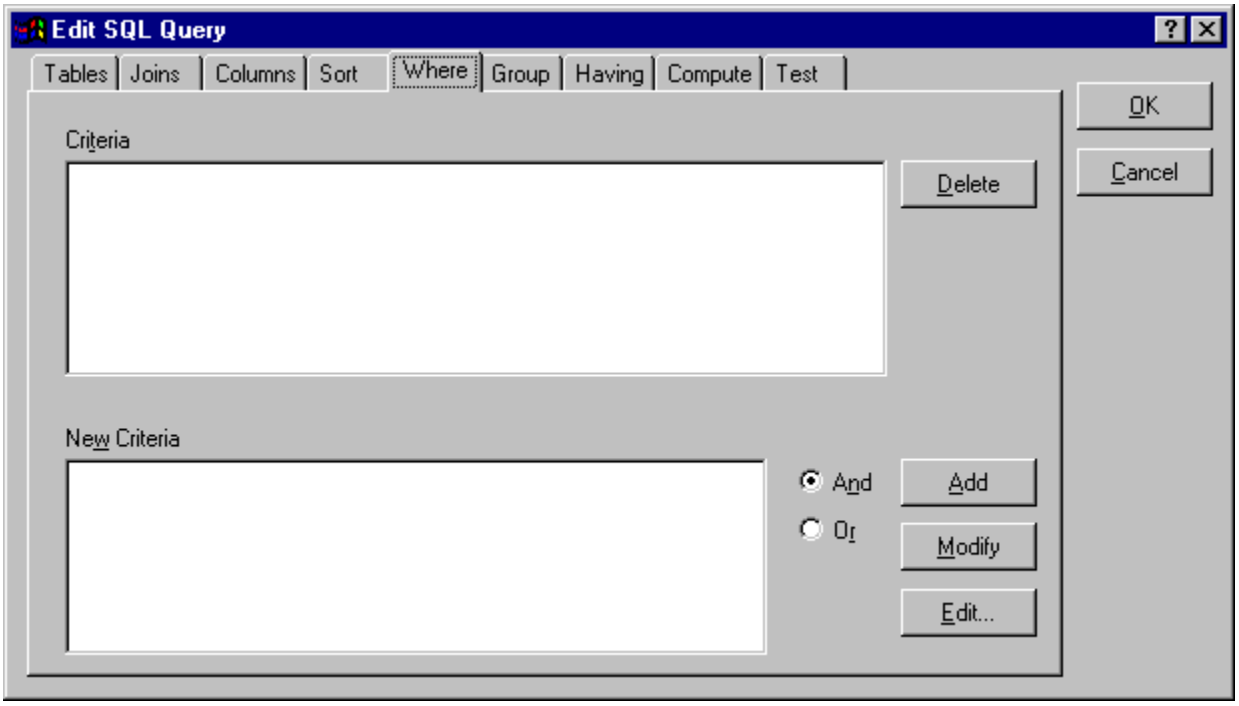
- **To place an item in the Sort by List:**

1. Click on the name of the column in the Available Columns list, then click **Add**; or
2. Double-click the name of the column in the Available Columns list; or
3. Drag the name of the column from the Available Columns list to the Sort by list.

The direction of sorting is shown by the arrow beside the sorting item: an up arrow indicates an ascending sort, and a down arrow indicates a descending sort. You can change the direction of the sort by double-clicking the arrow. You can also specify the direction by clicking the item in the Sort by list, then clicking **Ascending** or **Descending**.

The Where page

The **Where** page lets you specify search criteria for the query. By specifying conditions on this page, you can restrict the rows that are returned by the query. The criteria given on this page will appear in a `WHERE` clause of the final statement.



All the criteria on the **Where** page are connected with `AND` or `OR` operations, as in ((Criterion 1 `AND` Criterion 2) `OR` Criterion 3).

You can specify a criterion in one of two ways. First, you can simply type in the criterion as a string, using normal SQL format:

- **To add a new criterion function as a string:**

1. Type the new criterion into **New Criteria**.
2. Click **And** if the new criterion will be added to existing criteria with an `AND` operation. Click **Or** if the new criterion will be added to existing criteria with an `OR` operation.
3. Click **Add** to add the new criterion to the existing list.

Second, you can specify a criterion using the Criterion Editor.

- **To add a new criterion function using the Criterion Editor:**

1. Click **Edit**. This opens the Criterion Editor.
2. Create your criterion using the editor. For further information, see Using the Criterion Editor. Click **OK** when done.
3. Click **And** if the new criterion will be added to existing criteria with an `AND` operation. Click **Or** if the new criterion will be added to existing criteria with an `OR` operation.
4. Click **Add** to add the new criterion to the existing list.

You can modify an existing criterion by clicking on the criterion in the Criteria list, then clicking **Modify**. This copies the criterion to **New Criteria**, where you can edit it as a string or edit it using the Criterion Editor.

Using the Criterion Editor

The Criterion Editor presents all the information required to construct a criterion expression. The expression is displayed under **Expression**. The **Columns** list shows all the columns that have been selected for the current query, and the **Functions** list shows all the functions that can be used in the criterion expression. The Criterion Editor also supplies buttons for numbers, and various operations that can be performed in the expression.

To add a column name to the expression, double-click the column name in the **Columns** list. Similarly, to add a function call to the expression, double-click the function name in the **Functions** list.

Items are always added to **Expression** at the current location of the cursor. When you have constructed your criterion, click **OK** to return to the **Where** page.

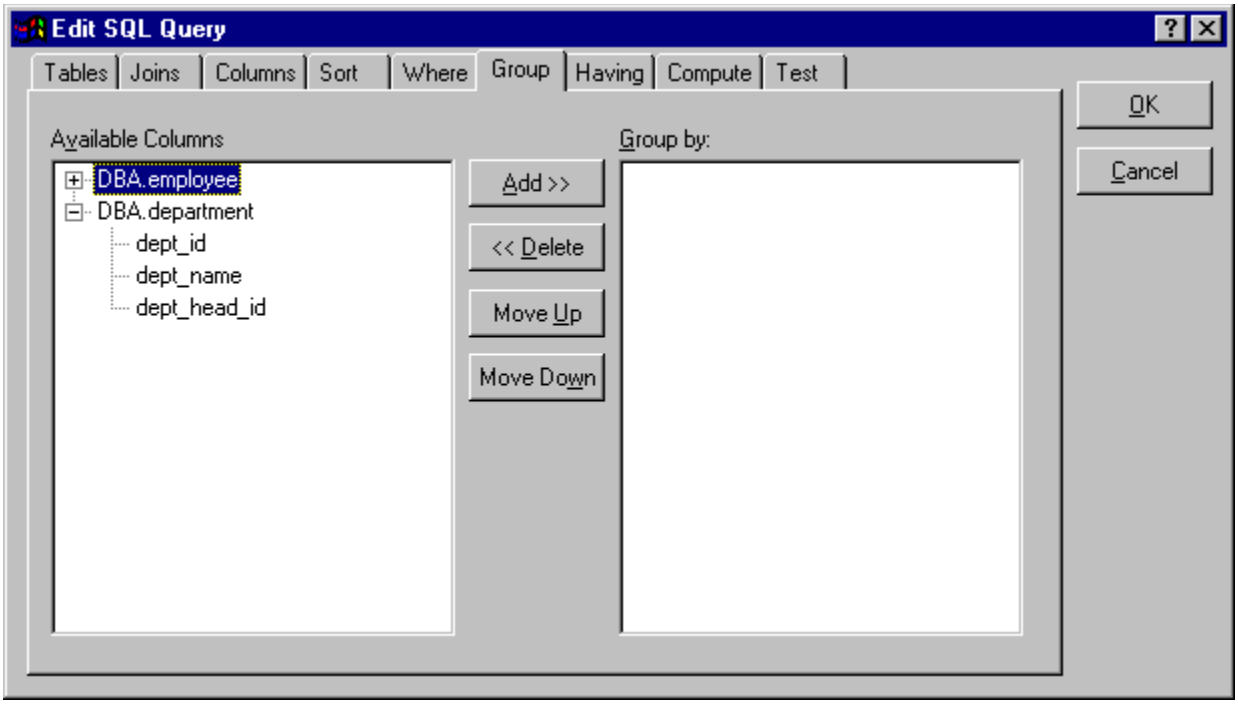
As an example of a simple criterion, suppose you are creating a statement to display employees in a particular department of a company. You might specify

```
DBA.employee.salary > 50000
```

so that the statement only returns information about employees whose salary is greater than $50,000.

The Group page

The **Group** page specifies columns by which rows will be grouped. The columns you select on this page are given in a `GROUP BY` clause in the final query. You must select tables using the **Tables** page before selecting grouped columns. Furthermore, the columns that you select on the **Group** page must appear in the **Selected Columns** list on the **Columns** page.



The Available Columns list shows the columns from the selected tables. To specify a grouped column, you add the column name to the Group by list.

- **To add a column to the Group by list:**

1. Click on the name of the column in the Available Columns list, then click **Add**; or
2. Double-click the name of the column in the Available Columns list; or
3. Drag the name of the column from the Available Columns list to the Group by list.

The Having page

The **Having** page sets up group restrictions. You can restrict which groups will be selected based on the group values and not on the individual row values. The **Having** page creates a `HAVING` clause in the SQL statement.

The **Having** page looks and works like the **Where** page. For example, you can create a new condition by using an editor similar to the Criterion Editor. For more information, see The Where page.

The Compute page

The **Compute** page lets you add new columns to the result set, using calculations on existing columns. For example, suppose you have a database describing company personnel and one column gives each person's salary. You could use the **Compute** page to add an expression similar to

```
salary * 1.05
```

to calculate a new column for the result set. This column would show what the new salaries would be if everyone got a 5% raise.

Columns created through the **Compute** page are incorporated into the result set that your program retrieves. However, the calculated values are not actually added to the database itself.

The **Compute** page looks and works like the **Where** page. For example, you can enter your calculations using an editor similar to the Criterion Editor. For more information, see The Where page.

The Test page

The **Test** page lets you test the results of your query by selecting a limited number of rows from the database. Click the **Test** button on this page to initiate the query.

By default, the Query Editor retrieves 20 rows from the database, to show you sample results of the query. If you want to see more rows to make sure the query worked, click **More rows**. Each time you click **More rows**, the Query Editor retrieves more rows.

You can also test the query by clicking the **Test** button on the **Query** page of the Query object's property sheet.

Chapter 20. Using bound controls

This chapter discusses how to bind objects to a database query so that they automatically update themselves from the results of the query.

Note: For a tutorial example of using databases, see the *Getting Started* guide.

Bound controls

A *bound control* is an object whose value is automatically updated by query operations. For example, you can bind a text box to a query object so that the text box always shows the value of a specified column in the current row. If you move the cursor to point at a different row of data, the text box automatically changes to show the value in the same column of the new row.

The following objects can serve as bound controls:

- Text boxes (including text fields, text areas, and masked text boxes)
- Labels (including simple labels and multiline labels)
- Check boxes
- Lists
- Choices
- Grids
- Picture boxes

If the user changes the value of a bound control, it typically changes the corresponding value in the query object's primary buffer. For example, if a text box displays the value of Column 1, changing the value of the text box typically changes the value of Column 1 in the current row (as stored in the primary buffer). Changes made in the primary buffer can be incorporated in the database itself using the **update** method. For more information, see Making changes in the database.

Setting up bound controls

If you intend to use an object as a bound control, you must click the **Bound Control** checkbox on the **Database** page of the object's property sheet. For example, if you intend to use a text box as a bound control, you must click the text box's **Bound Control** property at design time.

Important: If you do not click the **Bound Control** property at design time, you cannot use the object as a bound control at run time.

Before you use an object as a bound control, you must also set the **DataSource** and **DataColumns** properties for the bound control object.

- At design time, you can set these properties on the **Database** page of the object's property sheet.
- At run time, you can set these properties with appropriate **set** methods, as described later in this section. This is usually done in the **objectCreated** event handler for the form that contains the bound control; however, if you have turned on **AutoOpen** in the query object, you cannot set properties in the **objectCreated** event handler for the form, because the query object is opened before the form's **objectCreated** event is triggered.

The DataSource property

The **DataSource** property specifies the query object to which the control will be bound. For example, suppose that `textf_1` will be a bound control for `query_1`.

- To specify the **DataSource** property at design time, fill in the **DataSource** box on the **Database** page of the bound control's property sheet. This box offers a list of query objects; click the query to which this object will be bound.
- To specify the **DataSource** property at run time, use **setDataSource**, as in

```
textf_1.setDataSource( query_1 );
```

The DataColumns property

The **DataColumns** property determines which column's value should be displayed by the bound control. You can specify the column using the column's name or its ordinal number. For example, suppose that `textf_1` will be bound to the column called `emp_id`.

- To set this up at design time, use the **Database** page of the property sheet for `textf_1`. Set **DataColumns** to `emp_id`.
- To set this up at run time, use the **setDataColumns** of `textf_1`, as in:

```
textf_1.setDataColumns( "emp_id" );
```

If you specify a column by its number, the number is given as a string value:

```
textf_2.setDataColumns( "2" );
```

Some objects can display values from multiple columns, in which case you can specify a list of columns, with entries separated by semicolons:

```
grid_1.setDataColumns( "emp_id;dept_id" );
```

Checked and unchecked values

The **DataValueChecked** property is used with check boxes. It specifies that the check box should be checked when the associated column has the given value. For example,

```
checkbox_1.setDataValueChecked( "Male" );
checkbox_2.setDataValueChecked( "Female" );
```

sets up two check boxes. One will be marked when the associated column value is `"Male"` and the other will be marked when the column value is `"Female"`.

The **DataValueUnchecked** property is also used with check boxes. It specifies the value to be stored in the database when the check box is blank. For example,

```
checkbox_1.setDataValueChecked( "Here" );
checkbox_1.setDataValueUnchecked( "There" );
```

specifies that the check box should be checked if the corresponding column value is `"Here"` and should be unchecked if the value is `"There"`.

Columns with fixed-length strings

If a column in the database contains fixed-length strings, the JDBC specification requires the database driver to pad values with blanks in order to reach to the required length. For example, suppose a column contains strings with a fixed length of six characters. If this column represents a person's gender, the database driver might return the values

```
"Female"
"Male  "
```

Notice that the entry for `Male` is padded with two blanks on the end so that the total string is six characters long.

This has an effect on the use of bound check boxes, since the **DataValueChecked** property must give the *exact* string value returned by the database driver. In the situations just described, you would have to write

```
checkbox_1.setDataValueChecked( "Male  " );
```

so that the **DataValueChecked** string exactly matches the string that is actually returned by the database driver. You must also add an appropriate number of padding blanks if you are setting the property at design time, using the check box's property sheets.

Unfortunately, some database drivers do not match the JDBC specification. For example, the jConnect driver pads with blanks (as required by the JDBC specification) but the JDBC-ODBC bridge does not. Therefore, you would have to use a blank-padded string with jConnect but an unpadded string with the JDBC-ODBC bridge. With other drivers, you may have to experiment to see whether or not they pad strings with blanks.

Note: This situation only arises when the database contains fixed-length strings. It should not apply to columns with variable-length values.

How controls display values

Each type of bound control shows data values in different ways:

- Text boxes and labels show the value of the associated column in the current row.
- A check box is checked if the value of the associated column equals the value of the check box's **DataValueChecked** property. Otherwise, the check box is unchecked.
- Lists and choices can display values in several different ways. For more information, see Bound lists and choices.
- Grids can display several columns at once, giving a list of values from each column. For more information, see Bound grids.
- Picture boxes interpret the bound value as a string giving the URL of a picture (for example, a GIF file). The application opens this URL and displays the resulting picture in the picture box.

Bound control types

When you turn an AWT object into a bound control, PowerJ changes the type of that object. For example, when you turn a label into a bound control, PowerJ uses a DBLabel object (`powersoft.powerj.ui.DBLabel`) to represent that label rather than the usual AWT Label class. This is necessary because the standard AWT classes do not support the methods needed for using an object as a bound control.

Note: The names of bound control types consist of the letters DB plus the standard AWT name (for example, DBList, DBCheckbox, and so on).

All of the DB classes are derived from the corresponding AWT types. For example, the DBChoice class is defined as an extension of the normal Choice class.

Dates in bound masked text fields

This section discusses important considerations for using a bound masked text field to display dates obtained from a database.

When a date is obtained from a database, the value is represented as an object of the java.sql.Date or jdbc.sql.Date class. Both of these classes convert dates to strings of the form `"1997-09-14"`. Unfortunately, the masked text field expects to work with dates in a format recognized by java.util.Date, and the `"1997-09-14"` is not an accepted format.

Therefore, if you want to use a masked text field as a bound control displaying dates, you must do the following:

- Set the masked text field's **InputMask** to

  ```
  ####/##/##
  ```

- Turn off the **BeepOnError** property.

When the program retrieves a `"1997-09-14"` date and attempts to store it in a masked text field that has these property settings, the `"-"` characters are rejected as invalid characters (but there is no beep to signal an error). Therefore, only the numbers of the `"1997-09-14"` string are filled into the masked text field, leaving a string value of the form `"1997/09/14"`. This format is recognized by java.util.Date and therefore the masked text field accepts the date value. The masked text field will display the date in the format specified by the **OutputPattern** property.

For more information about masked text fields, see Masked text fields.

Bound lists and choices

This section discusses the use of lists and choices as bound controls.

List mode and lookup mode

There are basically two ways to use bound lists and choices:

- To show all the values from a column of the current result set. This contrasts with other bound controls which can only display values from the current row, not the whole result set. Using a list in this way is called *list mode*.
- To show all the *possible* values for the bound column. This is called *lookup mode*. To use a list in lookup mode, you can manually set the list of possible values, or obtain them from a second query object. The bound list box indicates the value of the current row by selecting (highlighting) that value in the list of possible values.

The mode is controlled by a property named **DataBindAsLookup**. For further information, see Using lists in list mode, and Using list boxes in lookup mode.

Note: The contents of a list box are cleared as soon as you set a property related to databases (for example, **DataSource** or **DataColumns**).

Using lists in list mode

A bound list in list mode uses the DBList class. In this mode, a list box displays all the values for a specified column in the current result set. Each line in the list box corresponds to a row in the result set.

For example, suppose that a query obtains information about all the employees in a company and that the list box is bound to the column giving the last name of an employee; then the list box displays the last names of all the employees in the company.

To put a list box into list mode, set the **DataBindAsLookup** property to `false`. In this mode, the other **Database** properties of the list box have the following meanings:

DataSource
The name of the query object to which the list box is bound.

DataColumns
The column(s) to which the list box is bound. Columns are specified by name or by number.

DataTrackRow
If **DataTrackRow** is `true`, the selected item in the list box corresponds to the column value for the current row. Selecting a different value moves to a different row in the result set. For example, if the user selects the first item in the list, the first row in the result set becomes the current row.

If **DataTrackRow** is `false`, the selection is not related to the current row. Changing the selection has no effect on the current row.

You may only set **DataTrackRow** to `true` if the list box is in single selection mode (**MultipleMode** is `false`).

DataLookupSource, **DataLookupColumns**
Ignored in list mode.

Using list boxes in lookup mode

A bound list in lookup mode uses the DBLookupList class. In lookup mode, the list box shows a list of all the possible values that might be entered into a particular column in the result set. The highlighted item in the list shows the current value for that column in the current row. In lookup mode, a list box represents a single value (a "cell") in the database.

For example, suppose that a query returns information about employees in a company. You might use a list box in lookup mode to display all the departments in the company. When you display information about a particular employee, the name of the employee's department will be highlighted in the list box. To transfer an employee to a different department, select the new department from the list box.

To put a list box into lookup mode, set the **DataBindAsLookup** property to `true`.

In order to use a list box in lookup mode, you have to set up the contents of the list box to show all the possible values. There are two ways to do this:

- Manually setting the list items. To do this, you explicitly add each item to the list using the **addLookupItem** method of the list box.
- Obtaining the list with a database query. For example, if the list box is supposed to list all the departments in a company, you could obtain the department names from a database. This is often done using a foreign key relationship with the bound column.

Manually setting up lookup lists

To place a lookup value directly into the list box, use the **addLookupItem** method:

```
// String key;
// String value;
lb_1.addLookupItem( key, value );
```

The parameters are:

key
: An actual value that can appear in the column.

value
: The actual string that should appear in the list box item. If this value is `null`, the list box shows the given *key*.

As an example of the relationship between *key* and *value*, suppose that the list box is going to list all the departments in a company. The *key* might be an ID number used to identify a particular department, while the *value* may be the name of the department. The database may refer to departments by ID number, but it is more helpful to show user's the department name.

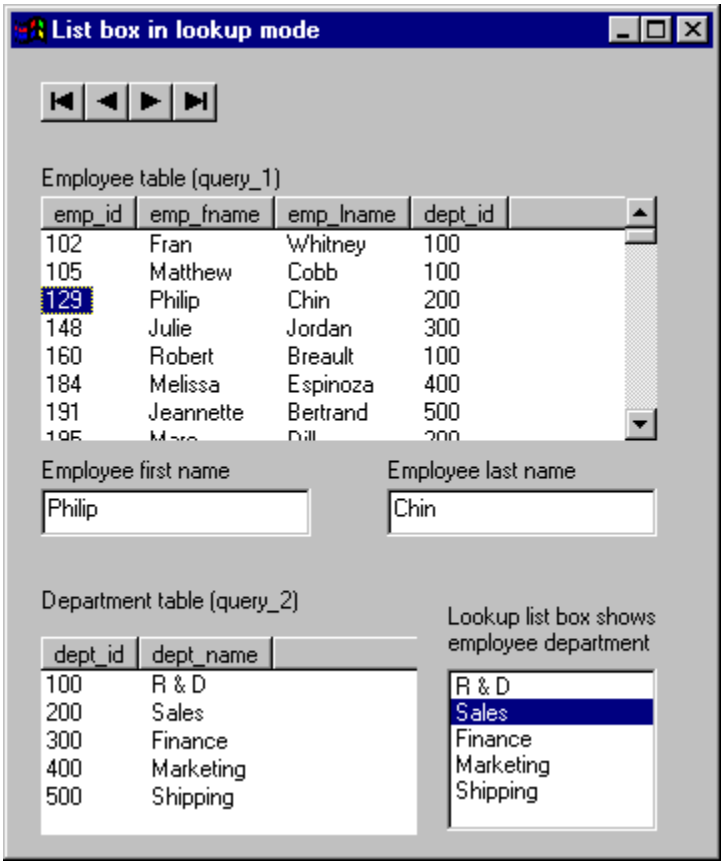
Obtaining lookup lists with a database query

As noted earlier, lookup mode lets you obtain list box entries from a database query. Since this is a complicated situation, it is useful to start with a concrete example. It is based on the following assumptions:

- Suppose that the purpose of a form is to show information about the employees of a company. This information is obtained by `query_1`.
- A list box in lookup mode will specify the department to which each employee belongs. This means that the list box will contain a list of departments. This list is obtained by `query_2`. There is usually

a foreign key database relationship between the values obtained by `query_1` and `query_2`.

- When the form displays information about a particular employee, that employee's department will be highlighted in the complete list of departments that is displayed in the list box.



To set up a list box in this way, you set the **Database** properties of the list box as described below:

DataSource
The query object to which the list box is bound. In our example, this will be `query_1`, the query that obtains information about employees.

DataColumns
The column to which the list box is bound. The column may be specified by name or by number. In our example, this is the ID number for the employee's department.

DataBindAsLookup
Is set `true` to indicate lookup mode.

DataTrackRow
Ignored in lookup mode. If you try to change the value of this property in lookup mode, the operation fails.

DataLookupSource
The query object that obtains the lookup list. In our example, this will be `query_2`, the query that obtains the list of departments.

DataLookupColumns
Specifies one or two columns from `query_2`.

The first column specified by **DataLookupColumns** must have the same type as the column specified by **DataColumns**; this will usually be a foreign key relationship in the database.

If **DataLookupColumns** only specifies one column, the list box displays all the values found in the

given column of the query. If **DataLookupColumns** specifies two columns, the first column gives the values to be matched against **DataColumns** and the second column gives the actual names to be displayed.

To go back to the example, you might set these properties as follows:

DataSource:	`query_1`
DataColumns:	`dept_id`
DataLookupSource:	`query_2`
DataLookupColumns:	`dept_id;dept_name`

The SQL statement of `query_2` obtains the ID number and name of every department in the company. **DataLookupColumns** indicates that the value of each item in the list box will be a department ID number; instead of displaying the ID number, the list box displays the corresponding department name.

The SQL statement of `query_1` obtains information about every employee, including the ID number of the employee's department. Therefore, when the program displays information about an employee, it checks the ID number returned as part of `query_1` against the list of ID numbers returned by `query_2`. The highlighted item in the list box will be the department name associated with the matching ID number.

Derived lookup lists

You can fill up a lookup list from a query without actually binding the list box. To do this, specify values for **DataLookupSource** and **DataLookupColumns** properties without specifying values for **DataSource** and **DataColumns**. In this case, the list box displays all the values obtained from the given source in the given column. The contents of the list box are filled when you open the **DataLookupSource** query.

You can use this technique if you want the user to select values from a lookup list but are not using these values directly to reflect another table in the database.

Bound choices

A bound choice is similar to a bound list box. The text box part of the choice shows the value of a column in the current row, and the list box part shows all the possible values for that column.

As with a list, you can specify whether a choice is in list mode or lookup mode through the **DataBindAsLookup** property. A bound choice in list mode uses the DBChoice class, and a bound choice in lookup mode uses the DBLookupChoice type.

A bound choice in lookup mode has all the properties associated with a list box in lookup mode: **DataSource**, **DataColumns**, **DataLookupSource**, and **DataLookupColumns**. As with list boxes, you can add lookup items to a choice using the **addLookupItem** method of the choice. The method has the same format as for lists. For further information, see Using list boxes in lookup mode.

Similarly, a bound choice in list mode has the properties associated with a list in list mode. For further information, see Using lists in list mode

If you select a new item from the choice, the same change is made in the associated column value of the current row of the database.

Bound grids

A bound grid control is similar to a bound list object. However, a grid can display any number of columns, whereas a list can only display values from one column.

The following properties control how a grid behaves as a bound control.

DataSource [**Database** page]
The name of the query object to which the grid is bound.

DataColumns [**Database** page]
The column(s) to which the grid is bound. Columns are specified by name or by number. You can specify any number of columns. The grid displays the columns in the order specified.

DataTrackRow [**Database** page]
If **DataTrackRow** is `true`, the grid item corresponding to the current row is selected. Selecting a different grid item changes the current cursor position to the selected row. You must set **DataTrackRow** to `true` to perform updates.

If **DataTrackRow** is `false`, the selection is not related to the current row, and changing the selection has no effect on the actual cursor position.

DataKeptRows [**Database** page]
Specifies the maximum number of rows to be displayed in the grid at any one time. By limiting the number of rows, you can reduce the amount of memory used to hold the data and the amount of time needed to fetch the data and store it in the grid. If **DataKeptRows** is zero, the entire result set is displayed.

DataGuardRows [**Database** page]
Specifies an integer threshold for collecting more data from the result set. This is used in connection with **DataKeptRows**. If **DataKeptRows** imposes a limit on the number of rows shown in the grid, the program must add new rows to the grid if you move off the top or bottom of the current contents of the grid. If you move within **DataGuardRows** of the beginning or the end of the current list, the grid adds more rows at this end of the list and discards rows at the other end.

For example, suppose that **DataKeptRows** is 40 and **DataGuardRows** is 10. This means that the grid only contains 40 rows at a time. If you scroll to within 10 rows of the end of the list, the program adds 10 more rows to the end of the list and discards 10 rows from the beginning of the list. Similarly, if you scroll to within 10 rows of the beginning of the list, the program adds 10 more rows to the beginning of the list and discards 10 rows from the end.

Important: DataGuardRows must be less than or equal to one third of **DataKeptRows**. Otherwise, the controls will not behave as expected.

Performing updates on grids

You must set **DataTrackRow** to `true` to perform database updates with a grid control.

Important: Changes to the database are not made until the **update** method is called on the query object. This means, for example, you should call **update** when the form is being closed, so that any changed values in the bound controls are sent back to the database. If you don't invoke **update** explicitly, **update** is called automatically when you use a **move** operation to move to a new row. For more information on **update**, see Adding new rows.

When the program begins filling data into a bound grid, it triggers a **dataFillBegin** event; when the program finishes, it triggers a **dataFillEnd** event. The **dataFillEnd** event is a good time to process the data received, or to change the headers of the grid from the defaults set by the data retrieval operation.

Moving through the result set

The **move** methods of Query move the cursor to point to a new row in the result set. The data shown in bound controls will change to reflect the corresponding values in the new row.

All **move** methods accept boolean arguments named `sendCursorEvents` and `sendDataTargetEvents`. This is explained later in this section.

The **move** methods are:

`moveFirst( sendCursorEvents, sendDataTargetEvents )`
Moves the cursor to the first row of the result set.

`moveLast( sendCursorEvents, sendDataTargetEvents )`
Moves the cursor to the last row of the result set.

`moveNext( sendCursorEvents, sendDataTargetEvents )`
Moves the cursor to the next row of the result set.

`movePrevious( sendCursorEvents, sendDataTargetEvents )`
Moves the cursor to the previous row of the result set.

```
moveAbsolute( row, sendCursorEvents,
    sendDataTargetEvents )
```
Moves the cursor to the row specified by the integer `row`. For example, if `row` is 10, **move** attempts to move to row 10 of the result set.

```
moveRelative( offset, sendCursorEvents,
    sendDataTargetEvents )
```
Moves the cursor forward or backward depending on the integer value of `offset`. For example, if `offset` is +10, **move** moves 10 rows forward from the current row; if `offset` is -10, **move** moves 10 rows backward.

The result of any **move** method is `false` if the given motion moves the cursor to "beginning of file" or "end of file" (before the first row in the result or after the last). For example, this might happen if you attempt to **move** to a row that lies outside the actual number of rows in the result set.

If the `sendCursorEvents` argument of the **move** method is `true`, the **move** method triggers **cursorMoving** and **cursorMoved** events (as described in Query events.

If the `sendDataTargetEvents` argument is `true`, the **move** method triggers **DataTarget** events as appropriate on associated bound controls. This tells all bound controls to update their contents, based on the results of the move operation. For example, when you move from one row to another, you usually want all bound controls to show values from the new row. Therefore, you typically specify `true` for this argument (unless you don't want the bound control values to change).

Because **move** methods can trigger **DataTarget** events, you can use a **move** method to fill a bound control that hasn't filled itself yet. For example,

```
moveFirst( true, true );
```

moves to the first row of a result set and triggers **DataTarget** events to fill in all the bound controls associated with the query.

Caching during moves

Caching during moves

The Query object keeps a cache of the data that's already been read from the database. This means that **move** operations may or may not move a cursor in the database, depending on whether the row has already been read in. The following example illustrates this principle:

```
query.open();
query.moveNext( true, true ); // moves cursor in database
query.moveNext( true, true ); // moves cursor in database
query.movePrevious( true, true ); // only moves local cursor
query.moveNext( true, true ); // only moves local cursor
query.moveNext( true, true ); // moves cursor in database
```

As the comments indicate, rows are only read from the database when they are needed.

Making changes in the database

There are several ways to make changes in a database:

- Deleting existing rows.
- Adding new rows.
- Modifying the contents of existing rows.

In order to make such changes, the query object must be open or detached; you cannot change the database if the query is closed. You must also set the appropriate properties for the transaction and query objects to enable updates. Once these properties have been set appropriately, your program can make changes to the database. The sections that follow describe how this is done.

If **AutoCommit** is `true`, any changes you make are committed as soon as you call **batchUpdate** (see Batch updates). If **AutoCommit** is `false`, the database is not permanently changed until you commit your changes using the **commit** method of the transaction object.

Deleting existing rows

The **delete** method of the query object deletes the current row:

```
try {
    boolean success = query_1.delete( );
} catch ( Exception e ) { };
```

The **delete** method fails if you are at EOF or BOF (past the last row in the result set or before the first row), if the result set is already empty, if you are in the middle of adding or modifying a row, or if you cannot change the database.

Adding new rows

Adding a new row to the database requires the following steps.

- **To add a new row to the database:**

1. Call the **add** method of the query object to begin the process.
2. Set values for the new row in the bound controls or using **setValue**.
3. Call the **update** method of the query object to add the new row to the database.

For example, suppose the current result set has two columns, both bound to text boxes. The following code adds a new row to the database:

```
try {
    query_1.add( false );
    textf_1.setText("Value 1");
    textf_2.setText("Value 2");
    query_1.update();
} catch ( Exception e ) {
    cb_1.setLabel( "error"+e );
}
```

The **add** method has the prototype:

boolean add(boolean *copyValues*);

If *copyValues* is `true`, **add** sets initial values for the new row by copying each column from the current row. If *copyValues* is `false`, the initial values for the new row are blank (undefined).

The **add** method puts the query object into *add mode*. In this mode, you cannot delete rows with **delete**. A call to **update** terminates add mode.

Note: You can terminate add mode without making changes by called **cancelAddOrEdit**. For more information, see Canceling changes.

Adding rows through a grid control

When you add a row using a grid control, you should call

```
grid_1.commitEdit( false );
```

before calling **update**. The **commitEdit** method copies the currently visible information in the grid control into the grid's data buffer. Therefore, the data that the user sees on screen is committed to the buffer, so that **update** uses this data for updating the database.

Modifying existing rows

Modifying an existing row is similar to adding a new row.

- **To modify an existing row:**

1. Use a **move** function to move to the row you want to modify.
2. Call the **edit** method of the query object to begin the process.
3. Modify the values of the row in the bound controls.
4. Call the **update** method of the query object to make the change.

For example, suppose the current result set has two columns, both bound to text boxes. The following code modifies the current row:

```
try {
    query_1.edit();
    textf_1.setText( "Value 1" );
    textf_2.setText( "Value 2" );
    query_1.update();
} catch ( Exception e ) {
    cb_1.setLabel( "error"+e );
}
```

As shown above, **edit** takes no arguments. It simply puts the query object into a mode where the current row may be edited. This is called *edit mode*.

If the **AutoEdit** property is `true`, you do not have to call **edit** explicitly. Your program automatically goes into edit mode if the user changes the value in a bound control or uses **setValue** to change a value. You still have to call **update** after making the changes. If **AutoEdit** is `false`, you must call **edit** explicitly before making changes.

If you call **move** after setting new values, **move** automatically calls **update**. In this situation, you do not have to call **update** explicitly.

For more information about **update**, see Adding new rows.

Note: When you are changing data in a grid control, you should call **commitEdit** before calling **update**. For more details, see Adding new rows.

Canceling changes

The **cancelAddOrEdit** method of a query object can cancel any modifications you have made in the current row. For example, suppose you begin to modify the current row by calling **edit** and then change the value of some bound controls. You can cancel the changes with

```
query_1.cancelAddOrEdit();
```

This resets the bound controls to their previous values.

You must call **cancelAddOrEdit** before calling **update**; once you call **update**, you can't cancel the changes.

The **cancelAddOrEdit** method can also cancel the process of adding a new row. For example, if you call **add** and start setting up values for a new row, **cancelAddOrEdit** cancels the operation.

Failed updates

If **UpdateMode** is set to `IMMEDIATE_UPDATES` and an update fails, the current value of the row is left unchanged. The state of the row does not change either.

In particular, if you call **update** in edit or add mode and the update operation fails, you remain in edit or add mode. Similarly, if a **delete** operation fails, the row will not be deleted from the result set.

Batch updates

If you have set the **UpdateMode** property to `BATCH_UPDATES`, your program does *not* immediately update the database when you call **update** or **delete**. Instead, the program accumulates updates and deletions in a "batch" of pending update actions. This may be useful in situations where you prefer to make all your changes to the database all at once rather than one at a time. For example, if interactions with the database take a long time, users may prefer to delay all updates until the end—one very long wait may be less annoying than a number of shorter waits each time a change is made.

The **batchUpdate** method of Query generates SQL statements to make all the updates, additions, and deletions that have been accumulated since the last **batchUpdate**:

```
boolean success = query_1.batchUpdate();
```

The result is `true` if the operation succeeds and `false` otherwise.

Note that you *must* call **batchUpdate** explicitly if you have set **UpdateMode** to `BATCH_UPDATES`; otherwise, the query does *not* transmit accumulated updates back to the database.

The **batchUpdate** method performs the following actions:

- Turns off **AutoCommit** if it's turned on.
- If the current row is in edit mode, **batchUpdate** calls **update** to keep any changes that have been made.
- Generates a sequence of SQL statements to perform all the changes that should be made in the database (`UPDATE`, `INSERT`, and/or `DELETE` statements).
- Executes the generated statements. If **AutoCommit** was on originally and if all of them succeed, the changes are committed; otherwise, the changes are rolled back. If **AutoCommit** was off originally, then any failures do not cause a rollback, and success does not cause a commit. Rollbacks and commits only occur when **AutoCommit** was on originally (i.e., before it was turned off by **batchUpdate**).
- Resets **AutoCommit** to its previous setting.
- Changes the original buffer associated with the query (a read-only buffer which reflects the actual contents of the database) to show the changes that have just been made.

Canceling batch updates

The **cancelBatchUpdate** method of a query object cancels all changes made since the last call to **batchUpdate**. This sets the contents of the primary buffer back to the data shown in the original buffer. Any changes that you have made since the last **batchUpdate** are permanently lost.

A call to **cancelBatchUpdate** resets all bound controls to values from the original buffer. This is a relatively quick operation, since the required data is already stored in memory.

Note: If **UpdateMode** is set to `IMMEDIATE_UPDATES`, the **batchUpdate** method is implicitly called every time you call **update** or **delete**. Therefore, **cancelBatchUpdate** will have no effect.

Update query objects

Update operations are performed by executing appropriate SQL statements (`DELETE`, `INSERT`, and `UPDATE`). These statements cannot be executed on the original Query object because of the way that JDBC works. JDBC requires that whenever you execute a new SQL statement for a query, the old result set associated with that query must be closed and discarded. Since this is undesirable, PowerJ uses a second Query object for all update operations on a query.

For example, suppose that you execute a command like

```
query_1.delete();
```

to delete the current row. By default, the program creates a new Query object, and executes a SQL `DELETE` statement through this new query. The new query is closed once the `DELETE` operation finishes. By going through a second query object, PowerJ preserves the original result set associated with `query_1`.

If you want more control over update operations, you can use the **UpdateQuery** property of the original query to specify a second Query object to be used in update operations. For example, you could code the following:

```
Query query_update = new Query();
query_1.setUpdateQuery( query_update );
```

This specifies that `query_update` should be used for all update operations performed on `query_1`.

Note: The **UpdateQuery** property may also be set at design time. For example, you might place a Query object named `query_update` on a form, then set the **UpdateQuery** property of `query_1` to specify `query_update`.

By writing event handlers for the update query object, you can get more control over update operations. For example, you might write a **queryOpened** event handler for `query_update` to be executed after the update operation's SQL statement has been executed. This event handler could record information about the success or failure of the update operation.

The update query must always be a different Query object than the original. For example, if you try

```
query_1.setUpdateQuery( query_1 );
```

the operation will fail (and an entry will be written to the debug log).

Update counts

After an **update** operation, the **UpdateCount** property of the update query object is set to the number of rows modified by the last database operation. You can obtain this number with

```
int count = query_update.getUpdateCount();
```

If the last database operation read data from the database rather than writing it, the **UpdateCount** is set to `-1`.

Similarly, after a **batchUpdate** operation, the **BatchUpdateCount** property is set to the total number of rows modified by the **batchUpdate**:

```
int bcount = query_update.getBatchUpdateCount();
```

If no rows were modified, or if you have not yet called **batchUpdate** for the current result set, the **BatchUpdateCount** is zero.

It is important to note that update counts are associated with the update query object, not the original

query. Typically, you would call **getUpdateCount** in the **queryOpened** event handler of the update query object. At this point, the update count will be valid. If you wait until later, the query associated with the update query object will be closed, and **getUpdateCount** will return `-1`.

Empty bound controls

An empty bound control is a bound control that contains no data. For example, you might display a blank text box and ask users to enter information into the text box. The program can then perform an insert operation which inserts a new row into the database, using the value typed into the text box.

A typical application might present the user with a form that contains empty bound controls for first name, last name, address, and so on. You may wish to give users the option of leaving some of these bound controls blank. In this case, there are some special considerations when performing an insert operation:

- An empty bound control has a null value. The insert operation will fail if the corresponding database column has not been set up to accept null values.
- If a database column is not set up to accept null values, you must fill in a suitable default value before performing the insert operation. One way to do this is to define a **queryUpdatingDatabase** event handler for the query object. This event is triggered at the point when the program has generated a SQL statement to perform the insert operation but before it has actually executed this statement. Your event handler therefore has a chance to modify the SQL statement to fill in a suitable default or to modify the null value that is ready to be sent to the database. For more information about **queryUpdatingDatabase**, see Query events.

Database events

This section examines events related to working with databases.

Exception events

The **exceptionThrown** event is a generic event used to indicate exceptions. For example, this event might be triggered when the database triggers an exception. Transactions and queries can both receive **exceptionThrown** events.

The argument to an **exceptionThrown** event handler is an **ExceptionEvent** object (`powersoft.powerj.event.ExceptionEvent`). You can determine the function that caught the exception using

```
String funcName = event.getFunctionName();
```

and you can determine which exception was triggered with

```
Throwable obj = event.getException();
```

For further information, see the *PowerJ Component Library Reference*.

Important: In debug mode, exceptions are also written to the debug log. Therefore, you should consult the debug log whenever you encounter unexpected behavior, to see if the log contains information that will help you analyze the problem.

Transaction events

The following events may be triggered on a transaction object:

transactionConnecting
Triggered just before the transaction attempts a connection to a database.

transactionConnected
Triggered after the transaction has successfully established a connection.

transactionDisconnecting
Triggered just before the transaction attempts to disconnect from the database.

transactionDisconnected
Triggered after the transaction has successfully disconnected.

transactionCommitting
Triggered just before the transaction attempts to commit changes to the database.

transactionCommitted
Triggered after the transaction has successfully committed the changes.

transactionRollingBack
Triggered just before the transaction attempts to roll back changes to the database.

transactionRolledBack
Triggered after the transaction has successfully rolled back the changes.

Event handlers receive a TransactionEvent (`powersoft.powerj.db.TransactionEvent`) object as their argument.

In a **transactionConnecting** event handler, you can use TransactionEvent methods to control the connection. In particular,

```
Object connection = event.getConnectionObject();
```

obtains the connection object that will be used to make the connection. You can modify properties of this object before the connection actually takes place. You can also use

```
// Object obj;
event.setConnectionObject( obj );
```

to specify a different connection object.

The TransactionEvent object supports the following method:

```
// boolean yesNo;
event.setOKToProceed( yesNo );
```

This method may be used in **transactionConnecting**, **transactionDisconnecting**, **transactionCommitting**, and **transactionRollingBack** event handlers to specify whether the program should proceed with the operation. An argument of `false` prevents the operation from happening, while an argument of `true` allows the operation to proceed normally.

Query events

The following events may be triggered on a query object:

queryOpening
Triggered just before attempting to open the query. It is still possible to change the SQL statement at this point (if you need to do any last-minute modifications).

queryOpened
Triggered after the query has been successfully opened.

queryClosing
Triggered just before attempting to close the query.

queryClosed
Triggered after the query has been successfully closed.

queryUpdatingDatabase
Triggered just before making changes in the database.

cursorUpdating
Triggered just before updating the contents of the current row.

cursorUpdated
Triggered after the contents of the current row have been updated.

cursorMoving
Triggered just before moving the cursor to a new row.

cursorMoved
Triggered after the cursor has been moved.

cursorDeleting
Triggered just before deleting the current row.

cursorDeleted
Triggered after successfully performing the deletion.

Event handlers receive either a QueryEvent (`powersoft.powerj.db.QueryEvent`) or **CursorEvent** object (`powersoft.powerj.db.CursorEvent`) as their argument.

The QueryEvent object supports the following method:

```
// boolean yesNo;
event.setOKToProceed( yesNo );
```

This method may be used in events like **queryOpening**, **queryClosing**, and **queryUpdatingDatabase** to specify whether the program should proceed with the operation. An argument of `false` prevents the operation from happening, while an argument of `true` allows the operation to proceed normally.

The QueryEvent object also makes it possible to determine the SQL statement that is about to be executed. For example, in a **queryUpdatingDatabase** event, the query object is about to update the database by executing an SQL statement (such `INSERT` or `DELETE`). You can determine the SQL statement that will be executed with the **getSQL** method:

```
String statement = event.getSQL();
```

You can specify a different SQL statement with

```
// String newStatement;
event.setSQL( newStatement );
```

QueryEvent offers a number of other methods for modifying the effects of a query operation. For example:

- The **setParameterList** method lets you specify bound parameters for a SQL statement that uses such parameters.
- The **getOperation** method lets you determine whether a change to the database is an `INSERT`, `DELETE`, or `UPDATE` operation.
- The **getTableInfo** method lets you determine the name of a table.
- The **getDataRow** method returns the raw data associated with a query.

For more information on all these, see the explanation of QueryEvent in the *PowerJ Component Library Reference*.

Event timing

If a query has the **AutoOpen** property turned on, the program attempts to open the query and fill its bound controls with initial values as soon as the containing form is opened. These operations take place before the form's **objectCreated** event is triggered, since the **objectCreated** event is not triggered until all the objects on the form have been properly created and initialized.

This means that bound controls will receive database-related events before the **objectCreated** event for the form. By the time the form's **objectCreated** event is triggered, the bound controls will display their first values, as obtained from the database.

The data navigator

A *data navigator* provides a simple way for the user to move through a database. When you place a data navigator on a form, it looks like a set of picture buttons; clicking these buttons automatically performs various actions on the related query. The data navigator may contain the following buttons:

- Move to the first row of the result set.
- Move to the previous row.
- Move to the next row.
- Move to the last row of the result set.
- Add a new row (go into Add mode, as with the query's **add** method).
- Delete the current row.
- Edit the current row (go into Edit mode, as with the query's **edit** method).
- Update the current row (by executing **update** on the query).
- Cancel changes to the current row (by executing **cancelAddOrEdit** on the query).

Data navigators are used as bound controls, bound to the query object which retrieves information from the database.

Data navigators are represented by DataNavigator objects. PowerJ gives data navigators default names of the form *dataNavigator_N*. On the **Database** page of the component palette, data navigators are represented by the following button:

If a data navigator is higher than it is wide, the buttons of the navigator are arranged vertically. Otherwise, the buttons are arranged horizontally.

Choosing buttons for the navigator

By default, all of the available buttons appear on a data navigator. However, you can remove selected buttons by turning off properties that appear on the **Database** page of the data navigator's property sheet. For example, if you turn off **ShowUpdate**, the **Update** button will not appear on the data navigator. You can also use methods like

```
dataNavigator_1.setShowUpdate( false );
```

to make the **Update** button disappear at run time, or

```
dataNavigator_1.setShowUpdate( true );
```

to make the button appear.

Data navigator button appearances

Button images in the data navigator can have different colors depending on their state. The possible states are the same as those for normal picture buttons:

Unarmed:
The button is raised (not pushed) and the mouse is not positioned on the button.

Armed:
The button is raised, but the mouse is pointing to the button.

Sunken:
The button is sunken into the form (pushed), but the mouse is not positioned on the button.

Sunken and armed:
The button is sunken into the form and the mouse is pointing to the button.

Disabled:
The button cannot be pressed. For example, when you are already at the end of the result set, the "move to next row" button is disabled.

Colors are given by the **ArmedColor**, **UnarmedColor**, and **SunkenColor** properties of the data navigator. The defaults are:

```
ArmedColor:     blue
UnarmedColor:   black
SunkenColor:    black
```

These can be changed with appropriate **set** methods (for example, **setArmedColor**).

The data navigator also supports a **BorderStyle** property. This dictates the border style for each of the picture buttons in the data navigator. For more information, see Picture button states.

Binding the data navigator to a query

A data navigator must be bound to a query object.

- At design time, you can bind the data navigator by typing the name of a query object in **DataSource** on the **Database** page of the navigator's property sheet.
- At run time, you can bind the data navigator with **setDataSource**:

```
dataNavigator_1.setDataSource( query_1 );
```

Action properties

The **EOFAction** property controls what happens when the user is already at the last row of the result set, then clicks the "move to next row" button. Possible values are:

```
DataNavigator.ACTION_NONE        // take no action
DataNavigator.ACTION_MOVE_FIRST  // move to first row
DataNavigator.ACTION_MOVE_LAST   // move to last row
```

For example, if you choose `ACTION_MOVE_FIRST`, the effect is to wrap around to the beginning of the result set if you move off the end. If you choose `ACTION_MOVE_LAST`, the effect is to keep going back to the last row if the user tries to go past the last row. **EOFAction** may be set at design time on the **Database** page of the data navigator's property sheet, or at run time with **setEOFAction**.

Similarly, the **BOFAction** property controls what happens when the user is at the first row of the result set, then clicks the "move to previous row" button. Possible values are the same as for **EOFAction**. **BOFAction** may be set at design time on the **Database** page of the data navigator's property sheet, or at run time with **setBOFAction**.

In order to handle EOF and BOF actions, the data navigator specifies a **cursorMoved** event handler for the query object. If the cursor is moved off the end of the of result set or before the beginning, the event handler changes the move operation in accordance with the current **EOFAction** or **BOFAction** value.

Events caused by the data navigator

Each button on the data navigator executes a method on the related query. For example, the buttons for executing **move** methods; the **Add** button executes **add**, the **Update** executes **update**, and so on.

Some of these actions will trigger events on the query that is associated with the data navigator. For example, the **Update** button triggers a number of events related to update operations. For information on these events, see Query events.

Some actions do not trigger query events. For example, the **Add** button executes the **add** method on the query. This puts the query into add mode, but does not trigger any events; typically, events will not be triggered until you perform an update operation on the query.

On the other hand, the **Add** button may trigger events on bound controls. For example, if there is a grid control bound to the query, clicking **Add** adds a new blank line to the grid control. This provides a place where the user can type in new information. The action of changing the bound grid control triggers invokes the **dataAvailable** method of the grid control and passes a DataTargetEvent object describing the way in which the grid control should change. The grid control's **dataAvailable** method then performs the action of adding a row to the grid.

If you want to perform special processing when a row is added, you can create a new class derived from the standard DBGrid class and write your own version of **dataAvailable**, as in:

```
public void dataAvailable( DataTargetEvent event )
{
    super.dataAvailable( event );

    if( event.getReason() ==
        DataTargetEvent.REASON_PREPAREFORADD ){
        // do something
    }
}
```

Use advanced class definition to change the grid object to your new class. For further information on advanced class definition, see Changing the class associated with an object.

Data formatters

The DataFormatter class (`powersoft.powerj.db.DataFormatter`) is an abstract class that implements both the DataSource and DataTarget interface. Classes derived from DataFormatter can serve as intermediate objects between queries and bound controls, formatting data that flows from the query to the bound control and vice versa.

For example, the TrimFormatter class (`powersoft.powerj.db.TrimFormatter`) is a concrete class that demonstrates the use of data formatters. TrimFormatter strips trailing blanks from strings sent from a query to a bound text field, and from the bound text field to the query. In order to set up this formatter, you would do the following:

- Create a TrimFormatter object using the appropriate statements in code.
- Bind the TrimFormatter object to a query object by setting the appropriate properties.
- Bind the text field to the TrimFormatter object, as if the TrimFormatter were a Query object.

In this way, the TrimFormatter object appears to be a bound control of the query object. When the query object fetches a string from the database and delivers it to the TrimFormatter, the TrimFormatter strips trailing blanks from the string and passes it on to the bound text field. Similarly, when the user types a new string into the text field, the text field passes the new string back to the TrimFormatter, which strips trailing blanks and passes it back to the query object (for delivery to the database).

The string mapping formatter

The StringMappingFormatter class is a concrete class based on DataFormatter. Its purpose is to map strings from the database into strings for a bound control and vice versa.

The **setMapping** method of StringMappingFormatter establishes what maps into what. The argument of **setMapping** is a string giving one or more mappings. Each mapping has the form

```
dbstring,dbstring,...=bcstring,bcstring,...
```

where the `dbstring` strings are associated with the database and the `bcstring` strings are associated with a bound control. Mappings are separated with semicolons.

Here is a simple example:

```
// StringMappingFormatter smf;
smf.setMapping( "M=male,man,boy;F=female,woman,girl" );
```

Assume that this StringMappingFormatter is used as a data formatter between a query object and a bound text field.

- When the value read from a database is `M`, the StringMappingFormatter maps this into `male` (the first possible string associated with `M`) and sends it on to the bound text field. Similarly, when the value read from a database is `F`, the StringMappingFormatter maps this into `female`.
- When the value entered into the bound text field is `male`, `man` or `boy`, the StringMappingFormatter maps this into `M` and sends it back to the query object. Similarly, when the value entered into the bound text field is `female`, `woman` or `girl`, the StringMappingFormatter maps this into `F` and sends it back to the query object.

If any of the mapping strings contain white space, commas, semicolons or quoted values, you can enclose them in single quotes, as in:

```
smf.setMapping("100='Group 1';200='Group 2';300,400='Others'");
```

You can even match the empty string. `NULL` data values are treated as empty strings.

Comparisons are all done as strings, but the type of the data value doesn't matter. You can specify whether you want the strings trimmed of white space before matching by setting the **TrimStrings** property to `true`. You can perform case-insensitive comparisons by setting the **IgnoreCase** property to `true`.

Custom formatters

The CustomFormatter class is another concrete class based on DataFormatter. You bind a control to a CustomFormatter object, then bind the CustomFormatter object to a query.

The CustomFormatter class can trigger two events:

- **dataFormatterRead** occurs when information is read from the database. The CustomFormatter object should format the database's information and deliver the result to the bound control.
- **dataFormatterWrite** occurs when information has been entered into the bound control. The CustomFormatter object should format the user's information and deliver the result to the query object.

To format the data, you write appropriate **dataFormatterRead** and **dataFormatterWrite** event handlers for the CustomFormatter object. Each of these event handlers receives a DataFormatterEvent object as its *event* argument. An event handler routine can use

```
// CustomFormatter cf;
ConstantDataValue cdv1 = cf.getOriginalValue();
```

to get the (unformatted) value that was originally read or written and

```
// ConstantDataValue cdv2;
cf.setValue( cdv2 );
```

to set a new value to be read or written.

Note: For more information about the ConstantDataValue class, see The DataValue interface.

Database dialog forms

One way to set up a form that uses a database is to select one of the **Database** form types from the Form Wizard. This includes the following types of forms:

- Database Frame
- Database Modal Dialog
- Database Modeless Dialog

Each of these creates a form containing a transaction object, a query object, a data navigator, and bound controls displaying the columns specified by the query.

When setting up a database dialog form, the Form Wizard prompts you for the information like the userid and password for connecting to the database and the SQL statement(s) to be executed for obtaining information from the database. For an example of setting up a form that performs a single query, see the *Getting Started* guide.

The properties of the objects on the form are set up so that the form can be used as soon as it is created. In particular, the transaction object has **AutoConnect** turned on and the query object has **AutoOpen** turned on so that the connection is made and the query opened as soon as the form is created. As a result, the first retrieved values are displayed in the bound text boxes when the form is opened. The user can then use the data navigator to move through the result set.

Master detail views

Master detail views

When you create a database form with the Form Wizard, you are offered a choice between two types of dialogs:

- A *single query* dialog, using only a single query object.
- A *master detail view*.

A master detail view contains two queries: a master query and a detail query. The information displayed in connection with the detail query depends on the information chosen in connection with the master query.

For example, the master query may obtain information about company departments and the detail query may obtain information about employees in each department. Both queries may have bound controls displaying information. For example, there might be several text boxes for displaying the department name and ID number (bound to the master query), and a grid control for displaying employee information (bound to the detail query). You might create a data navigator for moving through the list of departments returned by the master query. Whenever you look at a new department, the grid automatically changes to display information about the employees in that department.

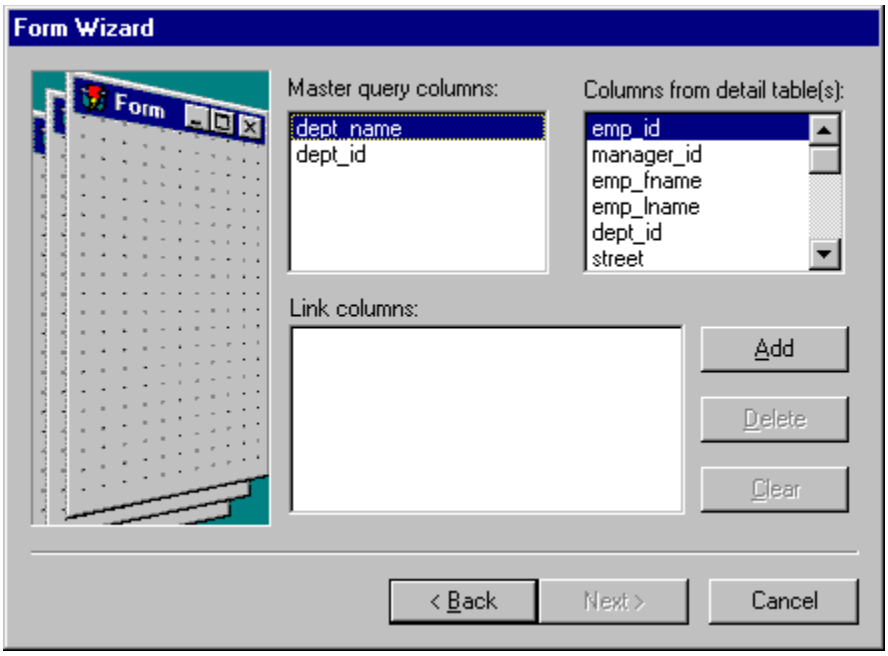
When you create a master detail view with the Form Wizard, you will be asked to enter two SQL statements: one for the master query and one for the detail query. For example, the master query statement can select information describing a department and the detail query statement can select information describing an employee.

Note: If you turn on the **TraceToLog** property for all the controls involved in a master detail view, the program may run very slowly. If you are having trouble with slow performance, turn off **TraceToLog** on as many controls as possible.

Linking master and detail

The key step is to *link* the two queries to show how the detail query depends on the master query. In the example we have been discussing, you might link an employee's department number to the department's ID number. Whenever the form changes to show a department with a new ID number (the master query), the form also changes to show employees whose department number matches the new ID number.

If you use the Form Wizard to create the master detail view, the Form Wizard asks you to specify the links, using a page with the following format:



In this example, you would click `dept_id` under both **Master query columns** and **Columns from detail table(s)**, then click **Add**. This links the detail query to the master query using the value of `dept_id`. When the user changes `dept_id` in the bound controls of the master query, the bound controls of the detail query automatically change to show employees with the same `dept_id`.

The Form Wizard only shows explicitly selected columns from the tables in the master query. However, it shows all the columns from tables in the detail query, whether the columns were selected or not. Therefore, the link columns must be explicitly selected in the master query, but do not have to be selected in the detail query; they will still be available for linking in the detail query, whether they are selected or not.

Note: With many database drivers, the Form Wizard may not be able to obtain the information it needs to determine whether there are conflicts between names in the master query and names in the detail query. Therefore, the Form Wizard may issue one or more warning messages to indicate that these names cannot be determined. In most cases, these conflicts will not cause problems and therefore the warnings can be ignored.

Chapter 21. Advanced client-server

This chapter discusses advanced techniques for interacting with databases using JDBC.

Techniques for dealing with result sets

This section discusses a number of techniques for obtaining information from the current row without using bound controls.

Counting the number of rows

The **getRowCount** method determines the number of rows in a result set. Because of limitations in JDBC and some database drivers, **getRowCount** will *not* return a valid value until your program has fetched the entire result set. The easiest way to ensure that you have fetched the entire result set is to move to the last row.

The following code shows an example of how to make sure **getRowCount** returns the correct value:

```
int count = query_1.getRowCount();
if( count == -1 ) {
    int row = query_1.getRow(); // save current row
    query_1.moveLast( false, false );
    count = query_1.getRowCount();
    query_1.moveAbsolute( row, false, false ); // go back
}
```

Important: Until you reach the last row, **getRowCount** will always return -1. When the last row is reached, it will return the number of rows.

Getting a value from the current row

The **getValue** method of Query gets a value from the current row. This method is a simple way to obtain values, without using bound controls.

The **getValue** method has the form

```
// int column;
ConstantDataValue dv = query_1.getValue( column );
```

where `column` is the number of the column whose value you want. (Columns are numbered beginning with 1, not 0.)

The result of **getValue** is a ConstantDataValue object. For more information, see The DataValue interface.

Finding the column index

The **findColumn** method determines the column index corresponding to a given name. For example:

```
int i = query_1.findColumn( "Employee_ID" );
```

This determines the number of the column that is named `Employee_ID`. This is helpful when used in conjunction with methods that refer to columns by number instead of name.

Column information

The **getColumn** method of Query returns an object that provides information about the column and its contents:

```
DataColumn dc1 = query_1.getColumn( 1 );
```

This determines information about column 1.

The DataColumn class

The result of **getColumn** is a data object of the DataColumn class. This class offers a number of methods that obtain information stored in the object. These include:

```
// DataColumn dc;
int prec    = dc.getPrecision();      // precision
int scale   = dc.getScale();          // scale
String name = dc.getColumnName();     // column name
String tabl = dc.getTableName();      // table name
int dSize   = dc.getColumnDisplaySize(); // display size
String label = dc.getColumnLabel();       // column label
boolean readOnly = dc.isReadOnly();       // read-only?
boolean sign = dc.isSigned();             // signed?
boolean cs   = dc.isCaseSensitive();      // case sensitive?
boolean writ = dc.isWritable();           // writable?
int cType    = dc.getColumnType();        // column type
String tName = dc.getColumnTypeName();    // name of data type
```

For more information on DataColumn, see the *PowerJ Component Library Reference*.

The DataValue interface

The DataValue interface is used to represent raw data read from a database, or destined to be written to the database. DataValue can be considered a sort of "envelope" that encapsulates all the data types that might be seen in the database. It has methods to cast values into other types.

For example, suppose `dv` is an object of a type that implements the DataValue interface. The following lines show a few ways to interpret the value as various PowerJ types.

```
boolean bl  = dv.getBoolean();
byte b      = dv.getByte();
byte bvec[] = dv.getBytes();
double d    = dv.getDouble();
int i       = dv.getInt();
float f     = dv.getFloat();
long l      = dv.getLong();
Object obj  = dv.getObject();
short sh    = dv.getShort();
String s    = dv.getString();
```

As an example, if you want to display the data value from column 2 in a text box, you can write:

```
textf_1.setText( query_1.getValue(2).getString() );
```

There are corresponding **set** methods for all of the **get** methods listed above. For example, the following sets the DataValue `dv` to the integer value 2:

```
dv.setInt( 2 );
```

DataValue types

You can obtain the name of a data type using the **typeName** method:

```
// int typeCode;
String typeStr = dv.typeName( typeCode );
```

The strings that might be returned by **typeName** correspond to the standard JDBC types (defined in `java.sql.Types` or `jdbc.sql.Types`). For example:

```
String typeStr = dv.typeName( java.sql.Types.BIT );
```

This returns the string `"BIT"`.

The following table summarizes the standard JDBC types, and the mapping to Java types.

JDBC type	Java type
`BIT`	`boolean`
`TINYINT`	`byte`
`SMALLINT`	`short`
`INTEGER`	`int`
`BIGINT`	`long`
`BINARY`	`byte[]`
`VARBINARY`	`byte[]`
`LONGVARBINARY`	`byte[]`
`CHAR`	`String`
`VARCHAR`	`String`
`LONGVARCHAR`	`String`
`NUMERIC`	`java.math.BigDecimal` or `jdbc.math.BigDecimal`
`DECIMAL`	`java.math.BigDecimal` or `jdbc.math.BigDecimal`
`REAL`	`float`
`FLOAT`	`double`
`DOUBLE`	`double`
`DATE`	`java.sql.Date` or `jdbc.sql.Date`
`TIME`	`java.sql.Time` or `jdbc.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp` or `jdbc.sql.Timestamp`
`OTHER`	Generally `Object`; however, if the column represents a currency value, it is treated as `NUMERIC` above.

In situations where a JDBC class corresponds to more than one Java type, the DataValue conversion method uses the Object type. For example, the **getTime** method returns an Object:

```
// DataValue dv;
Object obj = dv.getTime();
```

Your program can then convert the object to the appropriate type, as in:

```
java.sql.Time time = (java.sql.Time) obj;
```

The following table summarizes the standard mapping from Java types to JDBC types:

Java type	JDBC type
`boolean`	`BIT`
`byte`	`TINYINT`
`short`	`SMALLINT`
`int`	`INTEGER`
`long`	`BIGINT`
`byte[]`	`BINARY`, `VARBINARY` or `LONGVARBINARY`
`String`	`CHAR`, `VARCHAR` or `LONGVARCHAR`
`java.math.BigDecimal` or `jdbc.math.BigDecimal`	`NUMERIC`
`float`	`REAL`
`double`	`DOUBLE`
`java.sql.Date` or `jdbc.sql.Date`	`DATE`
`java.sql.Time` or `jdbc.sql.Time`	`TIME`
`java.sql.Timestamp` or `jdbc.sql.Timestamp`	`TIMESTAMP`
`Object`	`OTHER`

The ConstantDataValue interface

The ConstantDataValue interface only provides **get** conversion methods (not **set**). This means that you can convert the value of such an object to a standard Java data value, but you cannot store a new value in the object. Classes that implement ConstantDataValue are therefore used to hold "read-only" data. Note that the DataValue interface extends the ConstantDataValue interface to include methods for setting a data value.

ConstantDataValue also provides methods to compare data values to each other:

`dv1.compareTo( dv2 )`

For comparing two numeric types. The data values `dv1` and `dv2` are converted to a type that is suitable for comparing the two values; for example, if one is integer and the other is real, the

integer is converted to real before the comparison.

The result of **compareTo** is an integer: zero if `dv1` is equal to `dv2`, negative if `dv1` is less than `dv2` and positive if `dv1` is greater than `dv2`.

`dv1.canCompareTo( dv2 )`
Returns `true` if `dv1` can be validly compared to `dv2` using **compareTo**. Returns `false` otherwise.

`dv1.compareToString( dv2 )`
Converts both `dv1` and `dv2` to a string format and then compares the strings. The result is an integer, similar to the result of **compareTo**.

Note that if you compare the integer `1000` to the real number `1000.0`, the two are different as strings, even though they are equivalent numerically.

`dv.like( patternString )`
The `patternString` argument is a String value giving a pattern suitable for SQL `LIKE` pattern matching. For example,

```
dv.like("%b%_")
```

checks whether the data value (when converted to a string) contains a `"b"` and at least one other character. Only the `"%"` and `"_"` pattern characters are currently supported. The result is `true` if the data value matches the pattern and `false` otherwise.

DataConversion exceptions

The conversion methods of ConstantDataValue and DataValue may throw a DataConversionException if a requested conversion is not valid. For example, the following code attempts to obtain a ConstantDataValue as a boolean value:

```
ConstantDataValue dv = query_1.getValue( 1 );
try {
    boolean b = dv.getBoolean();
    // do something with value
}
catch { DataConversionException e ) {
    // do something
}
```

If you don't want to handle exceptions in this way, all the **get** conversion functions have versions ending in **NoThrow**, as shown in the following example:

```
ConstantDataValue dv = query_1.getValue( 1 );
boolean b = dv.getBooleanNoThrow();
```

The **NoThrow** versions of the **get** functions do not throw exceptions. If the conversion is invalid, they return a "nothing" result (for example, the integer zero, the boolean value `false`, or the `null` string).

Limitations of the JDK 1.02 Date class

The JDBC Date classes are based on the basic Java Date classes. Under JDK 1.02, these classes have several limitations:

- Dates may be expressed as a number of milliseconds since January 1, 1970. In this case, dates before 1970 are considered invalid.
- Dates may also be constructed using strings. In this case, dates before 1900 are considered invalid.

There is no way for the JDBC Date class to create a date earlier than 1900.

With some databases, you may be able to fetch pre-1900 date values as String objects. This depends on whether the driver can automatically convert dates to strings, and whether the database is using a Date object internally. The jConnect driver can obtain dates successfully as strings, while the JDBC-ODBC bridge always fails for dates before 1970, no matter what format is used.

Because of the problems with dates under JDK 1.02, the PowerJ query object for JDK 1.02 always attempts to fetch `DATE` and `TIMESTAMP` values as strings (for example, when filling bound controls). This means that the query object calls **getString** instead of **getDate**. This works better than trying to use **getDate**, but is still unsuccessful with some drivers. If you want to try to treat a date as a Date, you can do so explicitly by converting the DataValue with **getDate**.

SimpleValue and ConstantSimpleValue

DataValue and ConstantDataValue are interfaces, not object classes. If you wish to pass new values to database components, you must use objects of a class that implements DataValue or ConstantDataValue.

For simple data types, you can use the classes SimpleValue and ConstantSimpleValue. A simple data type is one of the "standard" Java types—*not* a type defined in any of the following packages:

```
java.sql.*
java.math.*
jdbc.sql.*
jdbc.math.*
```

The following shows an example of using ConstantSimpleValue:

```
ConstantSimpleValue csv = new ConstantSimpleValue( 100 );
```

You can only set the value of a ConstantSimpleValue object when you create it (since ConstantSimpleValue has no **set** methods). If you do not specify a value in the constructor, you get a ConstantSimpleValue that represents the null data value.

SimpleValue objects are similar, but you can change their value after creating them:

```
SimpleValue sv = new SimpleValue( 100 );
sv.setString( "new value" );
```

If you use a **set** method to change the value of a SimpleValue object, the type of the object changes to match the new value. In the above example, the type of `sv` changes from int to String.

SQLValue and ConstantSQLValue

The SQLValue and ConstantSQLValue classes serve a similar purpose to SimpleValue and ConstantSimpleValue. However, they also support values of SQL-specific data types like `java.sql.Date`.

Because of the conflict between `jdbc.sql` (JDK 1.02) and `java.sql` (JDK 1.1), you must know which package you are using with SQLValue and ConstantSQLValue. For example, if you are using `java.sql`, you might write:

```
powersoft.powerj.db.java_sql.SQLValue sv =
    new powersoft.powerj.db.java_sql.SQLValue();
```

If you are using `jdbc.sql`, you would use

```
powersoft.powerj.db.jdbc_sql.SQLValue sv =
    new powersoft.powerj.db.jdbc_sql.SQLValue();
```

Since this is awkward, PowerJ offers an alternate method to create such values at runtime: use the **createDataValue** method of Query. This method creates an SQLValue of the correct kind for whatever database you are using, and returns the value as a DataValue interface.

SQLValue type conversions

The **convert** method is a static method that converts from one type of DataValue to another. Here is the format for using the method:

```
// ConstantDataValue val;
// int type;
// int scale;
SQLValue newVal = SQLValue.convert( val, type, scale );
```

This says that you want to convert the existing value `val` to a type specified by `type`. Possible values for the `type` argument are given by symbolic constants corresponding to the JDBC types listed in DataValue types: `BIT`, `TINYINT`, `BIGINT`, `NUMERIC`, and so on. The `scale` parameter is only used for conversion to `NUMERIC` and `DECIMAL`; it specifies the number of decimal digits to the right of the decimal point. To obtain definitions of these constants, use:

```
import powersoft.powerj.db.ConstantDataValue;
```

There is also a non-static **convertTo** method. The ConstantSimpleValue and ConstantSQLValue also have **convert** and **convertTo** methods. For more information, see *PowerJ Component Library Reference*.

Updating without bound controls

Making changes in the database describes how to make changes in the database using bound controls. The process of making changes without bound controls is similar; the difference is that you use the **setValue** method instead of changing the value of a bound control.

The following code demonstrates how to do an update:

```
powersoft.powerj.db.DataValue   dv;
query_1.edit();

// Use the createDataValue method on the query object
//   to get a new  DataValue.
dv = query_1.createDataValue();

// first column is an integer
dv.setInt(5);

// The setValue method assigns a data value
//  to a particular column in the current row:
query_1.setValue( 1, dv );
dv = query_1.createDataValue();

// second column is a String (Char on the database)
dv.setString("Insert Me");
query_1.setValue(2,dv);

try {
     query_1.update();
}
catch (java.lang.Exception e)
{
    //handle exception
}
```

The following code shows a simple use of **setValue**:

```
query_1.edit();
query_1.setValue( 1, new SimpleValue( 24 ) );
try {
    query_1.update();
}
catch ( java.lang.Exception e )
{
    // handle exception
};
```

This code uses **edit** to go into edit mode, then uses **setValue** to change the value of column 1 in the current row. Finally, the code calls **update** to perform the actual update in the database.

Similarly, to insert a new row, you call **add**, then **setValue**, then call **update**. In the above examples, simply replace `query_1.edit()` with `query_1.add(false)` to insert a new row.

Note: Instead of calling **update**, you can use **cancelAddOrEdit** to cancel additions or editing changes. This leaves edit mode or add mode, and restores bound controls to their previous values.

If **UpdateMode** is set to `BATCH_UPDATES`, calling **update** doesn't immediately update the database.

Instead, updates are accumulated as they occur. To perform the actual updates, you must call:

```
query_1.batchUpdate();
```

For further information, see Batch updates.

The ConvertOnSet property

If the **ConvertOnSet** property of a query is `true`, a value specified in a **setValue** call is automatically converted to the appropriate type for the column. If the property is `false`, the conversion only occurs when the program actually executes SQL to update the database. By default, the property is on.

Bound parameters

SQL queries may use the question mark (`?`) as a placeholder in statements like:

```
SELECT * FROM employee WHERE manager_id = ?
```

Note: This construction corresponds to SQL statements like:

```
SELECT * FROM employee WHERE manager_id = @a
```

The `?` placeholder is called a *parameter*. Parameters must be filled in with actual values before the statement can be executed.

Important: When a query contains such parameters, you cannot use the **AutoOpen** property for the query (since the parameters must be filled in before the query can be opened). Therefore, you must turn off **AutoOpen** and call the **open** method explicitly in your code, after you have filled in parameter values. After you open the query, the cursor will be positioned before the first row, so you must perform a **moveFirst** operation to reach the first row of data.

Bound parameters cannot be used if you set the query's **StatementType** to `GENERAL_STATEMENT`.

You can use the query editor to create a query that contains a bound parameter. If you try to test the query inside the query editor, the query editor will prompt you to enter a value for the bound parameter.

Working with bound parameters
Bound parameters in master/detail views

Working with bound parameters

To specify values for a parameter at run time, you use the **setParameter** method of the query object. This method assigns a value to a specified parameter in the query. For example:

```
query_1.setParameter( 1,
    new powersoft.powerj.db.jdbc_sql.SQLValue( 200 ) );
```

This sets the first parameter in the query to a value of `200`. The following code gives another way to do the same thing:

```
DataValue dv = query_1.createDataValue();
dv.setInt( 200 );
query_1.setParameter( 1, dv );
query_1.open();
```

The above code demonstrates that you must give the parameter a value before opening the query.

There is a second form of **setParameter** that takes an argument specifying the SQL type of the data value:

```
query_1.setParameter( column, datavalue, sqltype );
```

The `sqltype` argument is typically given with a symbolic constant, using one of the constants defined in the ConstantDataValue interface. For example, you might specify:

```
query_1.setParameter( col, datavalue,
    ConstantDataValue.CHAR );
```

You could also use symbolic constants defined in java.sql.Types, as in

```
query_1.setParameter( col, datavalue,
    java.sql.Types.CHAR );
```

Finally, there are versions of **setParameter** that take simple values rather than DataValue arguments:

```
// String str;
// int ival;
// Object obj;
query_1.setParameter( col, str );
query_1.setParameter( col, ival );
query_1.setParameter( col, obj );
```

These are convenience versions of the **setParameter** method, intended to simplify the most common operations.

The **clearParameters** method clears the value of all bound parameters:

```
query_1.clearParameters();
```

Once you have set a parameter value, the parameter remains valid until you call **clearParameters**, **setSQL**, **setStatementType**, or **destroy**. This means that you can use code of the following form to perform successive queries with different parameter values:

```
query_1.setParameter( 1, value1 );
query_1.open();
query_1.close();
query_1.setParameter( 1, value2 );
query_1.open();
query_1.close();
```

For more information about **setParameter**, see the *PowerJ Component Library Reference*.

Bound parameters in master/detail views

Bound parameters are often used to create master/detail views. Typically, you have one query (the master) which provides a list of items. If the user selects one of these items, the detail query displays more information about the selected item.

For example, the master query could be:

```
SELECT  customer.id, customer.lname
FROM DBA.customer customer
```

The detail query could be:

```
SELECT customer.fname, customer.lname,
    customer.address, customer.city,
    customer.state, customer.zip, customer.phone,
    customer.company_name
FROM DBA.customer customer
WHERE customer.id = ?
```

Note: At design time, if you use the Query Editor to test a query that contains bound parameters, the Query Editor prompts you to enter values for the bound parameters.

Invoking stored procedures

You can call stored procedures in the SQL statement associated with a query. To do this, you enclose the procedure call in braces. Assuming a stored procedure called "foo" that takes two parameters, you can do the following:

```
query_1.setSQL( "{call foo(100, 200)}" );
query_1.open();
```

You could use placeholders for the stored procedure's parameters and use **setParameter** to assign values. The following code shows an example:

```
DataValue dv1 = query_1.createDataValue();
DataValue dv2 = query_1.createDataValue();

dv1.setInt( 100 );
dv2.setInt( 200 );

query_1.setSQL( "{call foo(?,?)}" );
query_1.setParameter( 1, dv1 );
query_1.setParameter( 2, dv2 );
query_1.open();
```

This calls the stored procedure named `foo`, passing integer arguments of 100 and 200. This is equivalent to the previous example.

Note that your stored procedure does not have to return a result set. If it does, then the Query object will detect this and you can then do a **moveNext** as usual.

If your stored procedure returns a scalar value, you must *register* the parameter as an output parameter before opening the query. You then use **getParameter** to obtain the returned value. The following code shows an example:

```
query_1.setSQL( "{?=call foo(100, 200)}" );
query_1.registerOutParameter(1, ConstantDataValue.INTEGER);
if( query_1.open() ){
    ConstantDataValue v = query_1.getParameter( 1 );
```

This code begins by setting an SQL statement containing a stored procedure call which returns a scalar value. The **registerOutParameter** method of Query registers the `"?"` return value as an output parameter. The arguments of **registerOutParameter** are a number indicating which parameter is being registered and the type of value that will be returned. Data types are represented by codes defined in the ConstantDataValue class.

The **getParameter** method of Query can obtain the value of any bound parameter as a ConstantDataValue. The argument of **getParameter** is a number indicating the parameter whose value you want (since the SQL statement can have more than one bound parameter).

Stored procedures with multiple result sets
Output bound parameters

Stored procedures with multiple result sets

Some stored procedures may return multiple result sets. In this case, the first result set is delivered to the query when the query is opened. The next result set is delivered by calling

```
query_1.getMoreResults();
```

You can then work with the information provided in the next result set.

The **getMoreResults** method returns `false` if there are no more result sets. Therefore, you might use it in code of the following form:

```
DataValue dv = query_1.createDataValue();
dv.setInt( 100 );
query_1.setSQL( "{call procedure(?)" );
query_1.setParameter( 1, dv );
if( query_1.open() ){
    while( true ){
        while( query_1.moveNext( true, true ) ){
            ConstantDataValue val = query_1.getValue( 1 );
            // do something
        }
        // process next result set....
        if( !query_1.getMoreResults() ) break;
    }
}
query_1.close();
```

Output bound parameters

When a stored procedure returns a value in an OUT parameter, you must *register* the parameter as an output parameter before opening the query. The following code shows an example:

```
query_1.setSQL( "{call sp_getmessage(100,?)}" );
query_1.registerOutParameter(1, ConstantDataValue.VARCHAR);
if( query_1.open() ){
    ConstantDataValue v = query_1.getParameter( 1 );
    if( v != null ){
        String s = v.getStringNoThrow();
        // and so on
    }
}
```

This code begins by setting an SQL statement containing a stored procedure call which returns an OUT parameter. The **registerOutParameter** method of Query registers the `"?"` parameter as an output parameter. The arguments of **registerOutParameter** are a number indicating which parameter is being registered and the type of value that will be returned. Data types are represented by codes defined in the ConstantDataValue class.

After the parameter has been registered, the query can be opened with **open**. The act of opening the query executes the stored procedure and obtains a value for the bound parameter(s). The **getParameter** method of Query can then obtain the value of any bound parameter as a ConstantDataValue. The argument of **getParameter** is a number indicating the parameter whose value you want (since the SQL statement can have more than one bound parameter).

The above code also shows how the parameter value obtained can be converted to a String by using the **getStringNoThrow** method of DataValue.

The following code shows an example of calling a stored procedure with one input argument and three output ones:

```
query_1.setSQL( "{call proc(?,?,?,?)}" );
query_1.setParameter( 1, 100 );
query_1.registerOutParameter(2, ConstantDataValue.INTEGER);
query_1.registerOutParameter(3, ConstantDataValue.DOUBLE);
query_1.registerOutParameter(4, ConstantDataValue.VARCHAR);
query_1.open();
ConstantDataValue val2 = query_1.getParameter( 2 );
ConstantDataValue val3 = query_1.getParameter( 3 );
ConstantDataValue val4 = query_1.getParameter( 4 );
```

Note: JDBC currently does not support the return of array arguments. To get around this, the stored procedure could create a temporary table and return the table.

Null values

If **getParameter** returns `null`, you can call the **wasNullParameter** method of Query to determine if the parameter actually had a `null` value or if **getParameter** returned `null` to indicate an error:

```
ConstantDataValue v = query_1.getParameter( 1 );
if ( v == null )
{
    if ( query_1.wasNullParameter( 1 ) )
```

```
        {
            // parameter value was null
        } else {
            // getParameter found error
        }
    }
```

Notice that **wasNullParameter** takes an integer argument to specify which bound parameter is being tested.

INOUT parameters

You can also use INOUT parameters in a stored procedure call. To use a parameter as an INOUT parameter:

- Use **registerOutParameter** to register the parameter as an output parameter.
- Use **setParameter** to assign a value to the parameter as an input parameter.

You must do both of the above before opening the query.

Dynamic data targets

Normal bound controls must be established at design time. This gives PowerJ a chance to use the right class for the bound control. For example, if you specify at design time that a particular text field will be used as a bound control, PowerJ uses a DBTextField object for that field rather than a normal TextField.

A *dynamic data target* is an object that is bound to the database query at run time rather than design time. The object may be a visible object (like a text field) or a non-visible one (for example, a Java class).

Note: Dynamic data targets are only available for JDK 1.1 applications.

To bind an object to a database query, you create a DynamicTargetCell object (`powersoft.powerj.db.DynamicTargetCell`). The constructor for the dynamic target takes a single argument—the object that you want to bind to the database, as in:

```
// TextField textf_1;
DynamicTargetCell dt = new DynamicTargetCell( textf_1 );
```

This creates a dynamic target object that is built on top of the existing text field. To bind the dynamic target to a database query, you must specify the following:

- A Query object representing the query.
- The data column to which the dynamic target should be bound.
- The property of the dynamic target that you want to bind to the query.

The following code performs this work:

```
dt.setBoundPropertyName( "Text" );
dt.setDataColumns( "emp_id" );
dt.setDataSource( query_1 );
```

This binds the **Text** property of the original text field `textf_1` to the `emp_id` column of the query. Therefore the value of that column in the current row will be assigned to the **Text** of the text field. From this point on, the text field behaves like a normal bound control.

Using a similar technique, you could bind a Java class to the database query.

Creating transactions and queries at run time

The easiest way to set up transaction and query objects is to define them at design time. However, you can also construct and initialize transaction and query objects at run time if your program requires this ability.

For an example of the code necessary to create transactions and queries "on the fly", see the PowerJ sample program named `JDBCTransaction`. This shows sample code for connecting to each of the following database drivers:

- Sybase jConnect for JDBC
- XDB JetConnect
- Visigenic VisiChannel for JDBC
- Sun JDBC-ODBC bridge

Access to raw JDBC

The PowerJ database classes use JDBC to perform database access. Since the PowerJ classes are wrapped around the JDBC classes, you seldom have to work with JDBC directly. However, there may be occasions when you need to obtain the "raw" JDBC objects that the PowerJ database classes are using, or you may wish to use the PowerJ database classes with a JDBC object you obtained from elsewhere. PowerJ makes both of these operations easy.

Note: The rest of this section assumes that you are already familiar with the class structure that supports JDBC.

A JDBC Connection object can be obtained from a Transaction object by calling **getConnectionObject** and casting the return value to either `java.sql.Connection` or `jdbc.sql.Connection` (as appropriate):

```
java.sql.Connection c =
    (java.sql.Connection) trans_1.getConnectionObject();
```

The **getConnectionObject** method returns a null value if the transaction object is not connected.

In a similar way, you can specify that a transaction should use a different connection object:

```
// java.sql.Connection conn;
trans_1.setConnectionObject( conn, true );
```

The second argument specifies whether to close the old connection or not.

A JDBC DatabaseMetaData object for the current connection can be obtained from a transaction object by calling **getMetaDataObject** and casting the return value appropriately:

```
java.sql.DatabaseMetaData dmd =
  (java.sql.DatabaseMetaData) trans_1.getMetaDataObject();
```

A JDBC Statement object (or one of the subclasses PreparedStatement and CallableStatement) can be obtained from a Query object by calling getStatementObject and casting the return value appropriately:

```
java.sql.Statement s =
  (java.sql.Statement) query_1.getStatementObject();
```

The type of object returned depends on the current setting of the **StatementType** property:

- If **StatementType** is `PREPARED_STATEMENT`, **getStatementObject** returns a PreparedStatement object.
- If **StatementType** is `CALLABLE_STATEMENT`, **getStatementObject** returns a CallableStatement object.
- Otherwise, **getStatementObject** returns a plain Statement object.

The **Statement** object used by a query object can be set using **setStatementObject**. In this case, the **StatementType** for the query is automatically set according to the type of the Statement object.

Result sets and extended result sets

A PowerJ Query object uses a data object of the ExtendedResultSet type. This is similar to a JDBC ResultSet with the following exceptions:

- **The** ExtendedResultSet **associated with a** Query contains the ResultSet for that query as a data member.
- ExtendedResultSet allows forward and backward cursor movement. ResultSet only allows forward

movement.

- ExtendedResultSet contains buffers to support updates to the query's original ResultSet. This includes the primary buffer (containing changes made by the user) and the deletion buffer (containing records recently deleted by the user).

The **getResultSetObject** method of a query returns the ExtendedResultSet for the query. This includes the original ResultSet object, and supports all the usual methods implemented for ResultSet objects. You can also set the result set for the query by calling **setResultSetObject**. This sets the ResultSet object within the query's ExtendedResultSet.

Note: For more information on ResultSet objects, see Processing result sets from other sources.

The **getOriginalResultSetObject** method of a query returns the original ResultSet object, not the ExtendedResultSet. You may need to use this method if you are working with non-PowerJ software that requires a true ResultSet rather than an ExtendedResultSet.

Processing result sets from other sources

Processing result sets from other sources

Typically, your program obtains a JDBC ResultSet object by opening a Query object. However, there are other ways in which you may obtain result sets.

For example, you might call a method on a Sybase Jaguar client object that returns a set of data packaged as a JDBC ResultSet. You can pass this ResultSet object to a query object, then use the query object to drive a set of data-aware controls. In this way, the data values in the ResultSet are displayed in the controls just as if they were the result of a SQL query.

To pass an arbitrary result set to a query object, use the following code:

```
// java.sql.ResultSet rs;
query_1.setResultSetObject( rs );
query_1.moveNext( true, true );
```

When you are finished working with the result set, you can close the query as usual.

Note: Do not close the query before you are finished with the result set—the JDBC standard requires that a ResultSet object must be closed when its associated Statement (or PreparedStatement) is closed. Therefore, as soon as you close the query (or the Statement within the query), the ResultSet becomes useless.

When using an arbitrary result set, updating the data is more complicated. The simplest case is to disallow updates by setting the **AllowUpdates** property to `false`. Otherwise you have two choices:

- You can update one or more database tables directly from the data by associating a valid transaction object with the query, and setting up the query's **PrimaryKeyColumns**, **ColumnTableNames** and other properties as usual. This assumes that the column names of the result set match those of the table(s) you wish to update.
- If the data needs to be transformed or otherwise handled differently, you can create a handler for the **queryUpdatingDatabase** event and handle the update yourself. In this case, you must set the **UpdateHandled** property of the QueryEvent object to indicate that you've handled the update.

Interacting with PowerBuilder applications

PowerJ offers access to PowerBuilder applications via the WEB.PB interface. For further information about this interface, see your PowerBuilder documentation.

The following sections describe how to use PowerJ's WebPB classes to access PowerBuilder.

Note: In this version of PowerJ, the support for PowerBuilder is primarily provided through run-time routines. There is no direct design-time support.

Step 1: Set up a query object

At design time, place a Query object on a form (or visual class). Set the query object's **JDBCPackage** property to `UseJavaSql` or `UseJdbcSql`.

You can attach bound controls to this query object. However, you must not attach the query object to a Transaction object.

Step 2: Create a WebPBCall object

At run time, your code must create an object of the WebPBCall class. If you have set the query's **JDBCPackage** to `UseJavaSql`, you create the WebPBCall object with:

```
powersoft.powerj.webpb.java_sql.WebPBCall webpb =
    new powersoft.powerj.webpb.java_sql.WebPBCall();
```

If you have set the query's **JDBCPackage** to `UseJdbcSql`, you create the object with:

```
powersoft.powerj.webpb.jdbc_sql.WebPBCall webpb =
    new powersoft.powerj.webpb.jdbc_sql.WebPBCall();
```

The WebPBCall object represents the PowerBuilder application with which your PowerJ program will interact.

Step 3: Set up the WebPBCall object

Next you must specify information about the `WEB.PB` object. Here is a typical sequence of instructions for initializing this object:

```
// Invoke the function of_test( first_parm, second_parm )
// on the object "n_cst_java" running on
// the server "niexdpbs"

webpb.setURL(
    "http://www.mycompany.com/scripts/pbcgi60.exe" );
webpb.setServerName( "niexdpbs" );
webpb.setObjectName( "n_cst_java" );
webpb.setFunctionName( "of_test" );
webpb.setArgument( "first_parm", "hello" );
webpb.setArgument( "second_parm", "there" );
```

Step 4: Link the WebPBCall object to the query

You must now attach the WebPBCall object to the query object set up in Step 1 above. This query object will receive result sets passed from the PowerBuilder object. The code for attaching the WebPBCall object has the form:

```
webpb.setResultSetConsumer( query_1 );
```

Step 5: Execute the WebPB function

The **retrieve** method of the WebPBCall object executes the `WEB.PB` function specified by **setFunctionName** in Step 3:

```
webpb.retrieve();
```

The **retrieve** method automatically calls **setResultSetObject** on the query object. This associates the result set with the query. Alternatively, you could execute the **getResultSetObject** on the WebPBCall object; this returns a raw JDBC ResultSet object that your code can manipulate.

Step 6: Fill in the bound controls

The previous step only associated the result set with the query; it did not fill in the bound controls with data from the query. To fill in the bound controls, you must move the query object's cursor:

```
query_1.moveNext( true, true );
```

The two `true` arguments for **moveNext** make sure that all the bound controls get filled with data from the result set.

A shortcut

There is a form of the **retrieve** method that performs Steps 3 and 5 in one move. The arguments specified in this **retrieve** method are the same as the properties set in Step 3. Therefore, you could write:

```
powersoft.powerj.webpb.java_sql.WebPBCall webpb =
    new powersoft.powerj.webpb.java_sql.WebPBCall();
webpb.setResultSetConsumer( query_1 );
webpb.retrieve(
    "http://www.mycompany.com/scripts/pbcgi60.exe",
    "niexdpbs", "n_cst_java", "of_test",
    "first_parm=hello&second_parm=there" );
```

This does the same work as Steps 2-6 above.

Note: The WebPBCall class supports **Name** and **TraceToLog** properties which serve the same purpose as the corresponding properties in the Query class. These are useful for tracing problems while debugging.

On the web.pb side

PowerJ expects that result sets contain metadata (header data). Some PowerBuilder techniques (for example, the **Describe** method) return result sets that do not contain metadata. The WebPBCall class can handle these result sets and will automatically construct some simple metadata (all columns will be marked as `VARCHAR` and given simple names and labels) so that the Query object can use the result sets. You can, however, return result sets with proper metadata.

There is a sample available in the PowerJ download area of the Sybase support site (`http://support.sybase.com`) that includes a PowerBuilder user object that will build the metadata for you.

Row filtering

PowerJ offers facilities for filtering out rows as they are fetched from the database. This adds an extra layer of selection, beyond any criteria that might be expressed in the SQL statement that fetches the data.

The filterRowFetched event

The **filterRowFetched** event is triggered on a Query object each time a row is fetched from the database. By writing your own handler for this event, you can accept or reject rows before they are incorporated into the query's result set.

The argument to the **filterRowFetched** event handler is a FilterEvent object (`powersoft.powerj.db.FilterEvent`). The FilterEvent class is based on the EventData class but has a number of additional methods.

`DataRow getDataRow();`
Returns the data row that has just been fetched from the database.

`void setFiltered( boolean f );`
Lets you specify whether the data row should be accepted or rejected. The statement

```
event.setFiltered( true );
```

specifies that the row should *not* be included in the final result set being fetched. In other words, it should be filtered out. The statement

```
event.setFiltered( false );
```

specifies that the row *should* be included in the final result set.

`boolean isFiltered();`
Returns `true` if the data row is currently filtered out and `false` otherwise.

`void setDataRow( DataRow row );`
Lets you change values in the data row that has been fetched. If you change values and do not filter out the data row, the changed values will appear in the final result set.

The DataRow class is a simple representation of the values in the data row. The data values are given by the public data member:

```
public ConstantDataValue[] columnValues;
```

This is an array giving the column values that have been fetched. Your **filterFetchedRow** event handler can examine these values and decide whether the row should be accepted or rejected for the final result set.

The DataRow class also contains the public data member

```
public Object extraData;
```

which lets you associate your own additional data with each data row.

Per-row filtering expressions

As an alternative to writing your own **filterFetchedRow** event handler, you can set up selection criteria that the query object can use to filter out rows.

Selection criteria are specified using Expression objects (`powersoft.powerj.db.Expression`). Expression objects specify a set of operations to perform, comparing column values to data values. Here is a simple example:

```
DataValue val1 = query_1.createDataValue();
val1.setInt( 100 );
Expression expr = new Expression( Expression.GE, 1, val1 );
query_1.setFilterExpression( expr );
query_1.refilter();
```

The first two statements above create a data value with the integer value 100. The Expression object `expr` is set up to compare whether the value of column 1 in a data row is greater than or equal to 100. This expression is then associated with the query using **setFilterExpression**. Finally, the code calls the **refilter** method to filter the query's result set in accordance with the new criterion. Note that calling **refilter** causes the **filterFetchedRow** events to occur again. The result is that the query filters out all rows with a value less than 100 in column 1.

The Expression class supports a number of operations. For example, the following creates a filter that filters out all values less than 100 and greater than 200:

```
DataValue val1 = query_1.createDataValue();
DataValue val2 = query_1.createDataValue();
val1.setInt( 100 );
val2.setInt( 200 );
Expression expr =
    new Expression( Expression.AND,
        new Expression( Expression.GE, 1, val1 ),
        new Expression( Expression.LE, 1, val2 ) );
query_1.setFilterExpression( expr );
query_1.refilter();
```

For a complete list of operations supported by Expression, see the *PowerJ Component Library Reference*.

Sorting result sets

PowerJ offers facilities for sorting result sets.

Result sets can be sorted in a way similar to how filtering works. There is a Sortable interface that defines methods like **getSort**, **setSort** and **resort**. Here is an example:

```
query_1.setSQL(
    "select emp_id, emp_lname, emp_fname from employee"
    );
query_1.open();
query_1.setSort( "emp_fname A, emp_lname D" );
query_1.resort();
```

The general syntax is : name of column plus a space plus either A or D (ascending or descending). Multiple columns are separated by commas. You can use #column_number if you don't want to use the column name. In the above example, the **setSort** call could be:

```
query_1.setSort( "#3 A, #2 D" );
```

This sorts the result set using the first name (`emp_fname)` column in ascending order. Rows are sorted by last name (`emp_lname)` in descending order within the first names that are equal.

There is a **rowComparisonPerformed** event that allows you to do your own row comparisons for sorting purposes if you so desire.

The rowComparisonPerformed event

The rowComparisonPerformed event

The **rowComparisonPerformed** event is triggered when a comparison is done between two rows. By writing your own handler for this event, you can alter the comparison of the two rows by invoking the **setComparisonResult** method in the event data object.

The argument to the **rowComparisonPerformed** event handler is a RowComparisonEvent object (`powersoft.powerj.db.RowComparisonEvent`). The RowComparisonEvent class is based on the EventData class but has a number of additional methods.

`DataRow getFirstRow();`
Returns the first row to compare.

`DataRow getSecondRow();`
Returns the second row to compare.

`int getComparisonResult();`
Gets the comparison value between the two rows (negative if first < second, 0 if first == second, positive if first > second).

`void setComparisonResult( int compValue );`
Sets the comparison value between the two rows (negative if first < second, 0 if first == second, positive if first > second).

Part V. Internet programming

The chapters in this part cover Internet fundamentals, the Internet components included with PowerJ, how to create server extensions (or servlets) with PowerJ, and how to program for distributed computing.

Chapter 22. Internet fundamentals

Chapter 1 described the fundamentals of the World Wide Web (see Java and the Web).

This chapter has three sections that describe other internet fundamentals:

- The first describes basic internet concepts and protocols that underlie the World Wide Web and other internet applications.
- The second describes traditional internet applications that predate the World Wide Web but are still in widespread use.
- The third describes the fundamentals of web servers and the protocols for extending web severs.

Internet concepts and terminology

The *Internet* is a global collection of interconnected computer networks. It was started over twenty years ago by the U.S. Defense Department as a network to connect far-flung institutions carrying out military research. It has expanded phenomenally to now include millions of computers around the world. Many private networks use internet standards and applications. Such a private network is now usually called an *intranet*, though you may also find one referred to as *an internet* (without "the" or capitalization). When used as an adjective, "internet" is usually not capitalized.

This section describes many aspects of the Internet, including frequently used acronyms. You probably won't need to understand all the layers and underpinnings of internet networking, but you'll want to have some familiarity so you don't get bogged down in acronyms in later chapters.

IP, DNS and TCP/IP

The fundamental network standard which allows computers on the Internet to exchange data is called *IP*, which stands for Internet Protocol. All computers on the Internet use this protocol. It specifies the mechanism for routing packets of data in networks, which is known as the *network layer*. Lower-level layers are the *link layer*, where standards such as ISDN, Ethernet and Token Ring operate, and the *physical layer*, which typically uses fiber optic, coaxial or twisted pair cable.

You may also see "IP" used in other internet terms. Each computer on an IP network has a unique four-byte *IP address*. An IP address is written as four numbers in the range 0 to 255 separated by periods, for example 199.93.177.11. If you are not continuously connected to the Internet, your computer may be assigned a *dynamic IP address* when you connect; the next time you connect, your computer may be assigned a different IP address. *IP spoofing* refers to a security-breaching technique where one computer masquerades as another by using the other computer's IP address, sort of like pretending you're someone else while on the phone.

Since a numeric IP address may not be easy to remember, most internet programs allow the use of a *domain name*, which is an alphabetic internet address like `www.sybase.com`. For this example, `www` is the *host name*, or name of the specific computer, `sybase.com` is the *domain*, and `com` is the top-level domain (TLD) or domain suffix. A domain suffix is usually a two letter code indicating the country (e.g. `uk` for the United Kingdom) or, primarily in the United States, a three letter code indicating the type of organization (e.g. `com` for commercial, `edu` for educational, `net` for network providers, `org` for organizations, `gov` for government and `mil` for military). Case is ignored, so the addresses `www.sybase.com`, `WWW.Sybase.Com` and `WWW.SYBASE.COM` are equivalent.

Assignment of IP numbers

In order for the Internet to function, the assignment of IP numbers and domains must be tightly controlled. InterNIC, under the authority of the Internet Assigned Numbers Authority (IANA), allocates blocks of IP address space. The IANA has given domain name registration authority to several organizations on the basis of top-level domains and geographic regions. For example, InterNIC is the domain name registration authority for the `root`, `edu`, `gov`, `com`, `net`, and `org` domains, and the Asia Pacific Network Information Center (APNIC) is the registration authority for the Asia-Pacific region.

DNS

A vital part of the internet is computers that offer *domain name service*. A *domain name server* resolves host and domain names into the numeric IP addresses that are needed for packet routing, and this is usually the first step in any internet transaction. If you've ever used an internet program and seen a status message like "`Looking up host ...`", that means it has sent a name like `www.sybase.com` to a domain name server and is waiting for the server to reply with a numeric IP address. The acronym *DNS* is used for both domain name service and domain name server.

TCP and UDP

The *transport layer* is the next layer above the network layer. The transport layer protocol specifies how packets are transmitted. Two transport protocols are used with IP: *TCP* (Transmission Control Protocol) and *UDP* (User Datagram Protocol). TCP provides a reliable mechanism for transmitting a stream of data as packets, with flow control, fault detection and error recovery. When one computer sends a packet in TCP, it waits for a signal from the receiver that it has received the packet intact. In UDP a data packet is not part of a stream, and it is sent without any mechanism for the sender to determine whether it was received. UDP is more of a "lean and mean" protocol – it gives up reliability for the sake

of reduced overhead. While most Internet traffic uses TCP, UDP is the protocol of choice for applications like video conferencing or real-time audio where rapid data transmission is more important than data completeness. UDP is also used for DNS requests, where the data transferred is minimal and fast turnaround is required.

TCP/IP

So what is *TCP/IP*? As you might have guessed, it stands for Transmission Control Protocol/Internet Protocol and in a strict sense it means the combination of TCP and IP. However it is usually used to refer to the collection of network, transport and application protocols used on the Internet. These include TCP, UDP and IP, which are usually implemented in the same program.

Sockets and ports

On top of the transport layer is the *session layer*, which supports bi-directional network sessions, and for this layer the primary internet protocol is *sockets*. The sockets standard defines how a session is created and whether the session will send data as a stream (using TCP) or as small, independent datagrams (using UDP). The socket interface standard originated in a version of UNIX from the University of California at Berkeley, hence the standard is sometimes called *Berkeley sockets*. The compatible socket standard defined for Windows by a consortium that includes Microsoft is called Windows Sockets or simply *Winsock*, and the programming interface for it is called the *Winsock API*. Internet applications for Windows are sometimes referred to as *Winsock applications*.

Each socket is associated with a *port* and is either an *active socket* or *passive socket*. The port is a 16-bit integer which defines a connection point to a server or application. An active socket is one that is connected to a remote active socket, while a passive socket listens for a connection request from a remote socket.

Port numbers are associated with specific uses. For example, the port for DNS is 53 and port 80 is used for the World Wide Web. The standardization of port numbers in the "well-known" range (from 0 to 1023) is controlled by the Internet Assigned Numbers Authority (IANA). The IANA also registers, but does not control, the assignment of ports in the range 1024 through 49151. Ports 49152 through 65535 are unassigned: they are intended for private or dynamic uses.

A client-server model is used to establish socket connections. The client creates a socket and sends a connection request to a specific port on a specific target computer. A server program on the target has a passive socket, often called a *server socket*, that waits for incoming connection requests on the port. For instance, a Web browser client will create a socket and send a connection request to port 80 on the Web server computer. A Web server will "listen" on port 80. Once the server program receives the request, it creates a new active socket that the client connects to. The server socket can then return to waiting for a new request on the port.

The process of connecting is analogous to a technical support phone call going through a switchboard operator. The customer (an active socket) calls the phone number (a specific port). The switchboard operator (the server socket) waits for a call (connection request). When a call is received, the operator creates a connection with a technical support person (another active socket) and then returns to waiting for a call.

A port can have a number of active sockets, but only one server socket.

Application protocols

Since sockets only provide a means of communication, the client and server programs must agree on a *application layer* protocol to use across a socket connection. The mechanism of agreement is via the port number.

There are many standard application protocols and associated port numbers. The most popular application protocols are those for file transfers (FTP), electronic mail (SMTP and POP), Usenet news (NNTP) and the World Wide Web (HTTP). These protocols will be discussed in the rest of this chapter.

Traditional Internet applications

The traditional use of the Internet has been for transferring file-based data, in a sense acting as ‘‘plumbing’’ between computers. For example, file transfer, electronic mail and newsgroups are internet applications that have been well established for more than a decade and they are still widely used today. The following sections provide background information for those traditional internet applications.

File transfer

The standard for transferring files on the Internet is *FTP* which is short for *File Transfer Protocol*. FTP is one of the earliest internet application protocols that is still widely used. An FTP client program connects to a remote FTP server program (on port 21) and the client logs in with a userid and password. Many FTP servers allow *anonymous FTP*, where the userid is either `ftp` or `anonymous` and your email address should be given as the password.

Once it is logged in, the client sends text commands to the server. These commands may include ones to instruct the server to change folders, list folder contents, transfer files or set transfer options. Commands and transfers use different TCP sockets; by default file transfers use port 20. In the simplest FTP client programs you must type the commands yourself, but there are now many programs, including World Wide Web browsers, that use graphical interfaces to hide the FTP commands.

Electronic mail

Electronic mail (*email*) is one of the oldest Internet services, and it continues to be widely used. There are many standards for both the transfer of email messages and their format. Currently two protocols predominate for the transfer of email messages: SMTP and POP3.

The *Simple Mail Transfer Protocol* (*SMTP*) is used for transferring electronic mail between two computers that are connected to the internet. In SMTP, a client first connects to an SMTP server using a TCP socket on port 25 and logs in. Then the client sends text commands and the server responds with text prompts to transfer messages. Although a person using a terminal session could type in client commands directly, for all but debugging purposes a client program is used.

Often the email messages are sent via other intermediate computers, using routing tables to determine the route. If the sender (or an intermediate) is unable to deliver messages to the target computer, it will periodically retry to until a timeout threshold is reached. If a message is not delivered within a specified time (usually a few days), the transfer will be abandoned.

SMTP is used for sending and receiving email in cases where the sender and recipient computers are always on and connected to the Internet. However for a computer that is not always connected to the internet, using SMTP is not a good choice for receiving email. This would result in many retries and errors whenever the recipient computer was not connected and another computer was trying to send email to it. For such computers, SMTP is used to send email but a different standard must be used to receive email.

The *Post Office Protocol* (*POP*) is the most common internet standard for receiving email on computers that are not continuously connected. The current version of this protocol is *POP3*. Incoming SMTP email messages are spooled into user mailboxes on a computer running a POP server. This computer should be continuously connected to the Internet. A POP client can then connect to the server (on port 110 for POP3) and retrieve the spooled email messages from the user's mailbox. Note that POP requires user authentication, and that involves sending an unencrypted password over the connection.

As the name implies, POP is similar to post office boxes in a regular Post Office: mail is delivered to the PO box as it arrives (similar to a POP server spooling email) but it does not reach its ultimate destination until the box owner comes in and collects it. Another analogy is that SMTP is like someone making a telephone call who keeps calling back until there is an answer, and POP provides an answering machine. The answering machine is not needed if someone is always there to answer the phone.

POP is also an acronym for another, unrelated internet term: *point of presence*. An Internet Service Provider is said to have a point of presence in an area if one can use a local telephone number in that area to connect to the internet.

The Internet standard format of email messages is to have a header, consisting of ASCII text fields (such as `To:`, `From:`, `Subject:`, and `Date:`), a blank line separating the header from the body, and a body. The transfer standards were designed for message bodies that are human-readable ASCII text. For messages with non-textual data in the body (such as a picture or a spreadsheet) the sending client must encode the non-textual data into a text form which the mail client at the receiving end must decode. The *Multipurpose Internet Mail Extensions* (*MIME*) specification is becoming the standard for encoding different types of binary data, but older standards like UUEncoding are still in common usage. Any type of encoding can be used as long as both email clients understand it.

The `To:` field in an email message specifies the *email address* to send the message to. Email addresses are typically of the form `user@computer.domain` or, if the routing tables support it, the specific computer name can be omitted. For example, the famous email address `billg@microsoft.com` just uses the domain name. This is usually preferable, since domains are less likely to change than the particular computer used by a user for email.

News and Usenet

Electronic mail is good for sending a message to a single person or a small group of people, but what if you want to send a message that potentially all Internet users in the world could read? That's what Internet *news* is for. With news, you can *post* and read *articles*, categorized by topic into *newsgroups*, much as you might place and read classified ads in a newspaper.

(As an aside, some people *do* send unsolicited email to a very large group of people, or post messages to a multitude of inappropriate newsgroups – these practices are often referred to as *spamming*. The name derives from a Monty Python skit where the word "spam" is repeated ad nauseum.)

The set of shared Internet newsgroups and the software and standards for maintaining them is called *Usenet*.

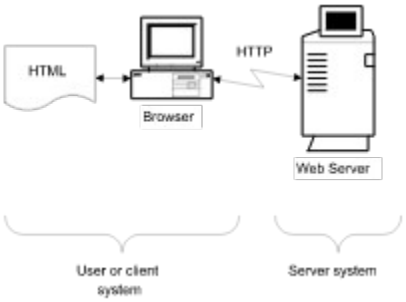
News articles differ from email messages primarily in the addressing and distribution. Instead of sending a message to a user, you post an article to a newsgroup. Instead of the `To:` field in an email message, a news article has a `Newsgroups:` field that specifies which newsgroups the article is posted to. Newsgroups are organized hierarchically by topic, for instance `comp.lang.java.security` is a newsgroup for articles about security aspects of the Java computer programming language. The group names are more specific from left to right, for example all groups under the `sci` hierarchy (for science topics) start with "`sci.`". Other top-level hierarchies are `comp` (computers), `rec` (recreation), `soc` (society), `misc` (miscellaneous) and `alt` (alternate). The rules for creating new groups vary between the hierarchies, and many companies are now creating their own top-level hierarchies.

When you post an article, it is propagated amongst *news servers* throughout the world (unless a restricted distribution was specified). Newsreader programs can connect to a news server to read and post articles. News servers and newsreader clients use the *Network News Transfer Protocol* (*NNTP*). NNTP supports reading newsgroups, posting new articles, and transferring articles between news servers. NNTP uses TCP sockets on port 119.

Web server basics

When you browse a web site, there are two computer systems involved:

- The *User* system (also called the *client* system), where your web browser displays information on your monitor.
- The *Server* system, which supplies the information that your web browser displays.



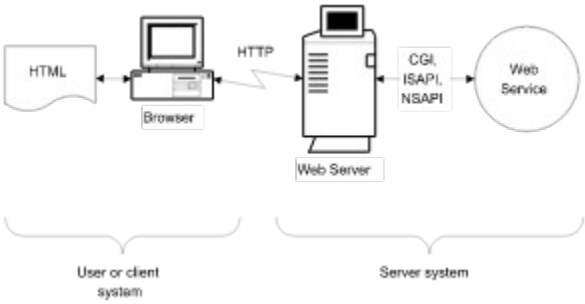
The User system requests information from the Server by specifying a *URL* (Uniform Resource Locator). You are probably familiar with simple URLs like:

`http://www.sybase.com/products/powerj/index.html`

When a URL begins with `http:`, the web browser obtains information from the web server using a protocol named HTTP. If the URL does not explicitly specify a protocol, many web browsers automatically insert `http:` at the beginning of the URL. For example, if you try to connect to `www.sybase.com`, many web browsers will automatically convert this to the URL `http://www.sybase.com`. On the Server side, incoming requests using the HTTP protocol are handled by programs called *web servers*.

URLs usually refer to files on the Server machine. In the URL given above, the host system is `www.sybase.com`. The web server locates the file `products/powerj/index.html` on the Server system and sends it to the web browser on the user's system. Typically, the file contains text data in the form of HTML code. (HTML is a language that makes it easy to break text into paragraphs, specify headings, make links to other URLs, and so on.)

A URL may correspond to a command or service to be executed on the Server machine. A *command* URL may also *post* data to that command. For example, the user may fill in the fields of a form and then the browser sends the filled-in information to the server. Software that is invoked in response to a URL is called a *web service* or *web server extension*. The three most common standards for interfacing a service to a web server are CGI, ISAPI and NSAPI; these are described later in this chapter.



The web server determines how to handle every URL corresponding to a file or a web service. If the URL specifies a service, the web server invokes that service, passing any data from the user's form. When the service produces output, the web server sends that output back to the user's web browser. Usually this output is HTML, although it can be some other type of data instead (for example, a JPEG graphic).

- CGI
- NSAPI and ISAPI
- HTTP Cookies
- Using HTTP

Forms

HTML makes it possible to create *forms* on the user's machine. These are not the same as PowerJ Form objects, but they serve a similar purpose—they obtain information from the user using text boxes, check boxes, and so on.

Every HTML form specifies a web service in its URL. This service is called the form's *action*. When the user clicks the **Submit** button for a form, the web browser sends the form data to the action URL. The web service named by the action URL handles the data that the user has entered on the form.

The HTML for creating a form also specifies a method for submitting data to the web service. There are two methods:

```
METHOD=GET
```

This submits information from the form as part of the URL. The information is given as a sequence of *query variables*, with names and values specified in the URL. The URL uses a `'?'` character to mark the beginning of query variable definitions.

The other method of submitting data uses:

```
METHOD=POST
```

The web browser will deliver information to the web service as a collection of *form variables*, passed as a block of data that accompanies the URL.

It is up to the programmer to pick which method is most suitable. Large forms with lots of data should use `POST`, which places no restriction on the amount of data sent.

CGI

The Common Gateway Interface (CGI) defines a way to pass a URL request to a web service invoked by the web server. A web service invoked via CGI writes data to its standard output and the web server passes that data directly to the web browser.

CGI is simple to use, but it has several drawbacks. In particular, you have to start a separate program each time you receive a URL; there is no direct way to start a program on the Server side and have it interact back and forth with the User side.

One way to work around this limitation is to encode "state information" in the data that the program sends back to the User side. For example, suppose that an interaction with the user involves two forms.

- The user enters data in the first form, then clicks **Submit** to send the data to the web server. The server invokes the web service specified in the URL, and that service processes the data.
- The program transmits a second form back to the user. This form contains `HIDDEN` fields which are not visible to the user, but which contain any necessary information from the first form.
- When the user submits the second form, the web server invokes a second web service which receives all the information from the second form: the hidden fields derived from the first form as well as any new information filled into the second form.

In this way, the second web service program can process all the information from the two forms.

This approach to processing information is slow, since a new program must be invoked for each URL. Performance becomes even slower if a database has to be accessed or if input fields have to be checked for validity, since this usually means more back-and-forth communications between the Server and User sides. Furthermore, programmers may need time to get used to programming in such an environment, since it is significantly different from other types of programming.

NSAPI and ISAPI

To improve efficiency, vendors have introduced two other environments for writing Internet based programs:

- The Netscape Web Server Plug-in environment (NSAPI)
- The Microsoft Internet Information Server environment (ISAPI)

These environments are provided through HTTP web servers which provide special interfaces for executing programs on the Server side. These custom interfaces are proprietary, and are therefore not "universal" like CGI. They are also more complex than CGI and therefore more difficult to learn. On the other hand, they provide better performance and high speed response to user input. Furthermore, they allow a single program to stay in execution for back-and-forth communications with the User side, instead of invoking a new program for every exchange.

Note: This guide makes no attempt to explain technical details of CGI, NSAPI, or ISAPI. This chapter only explains how PowerJ provides access to the three environments. For information on the environments, see the following Web sites:

CGI:
`http://hoohoo.ncsa.uiuc.edu/cgi/`

NSAPI:
`http://www.netscape.com/newsref/std/server_api.html`

ISAPI:
`http://www.microsoft.com/win32dev/apiext/isalegal.htm`

HTTP Cookies

HTTP identifies web sessions using *cookies*. A cookie is just an identifier string that is intended to be unique across the Web. For example, different web browsers on the same system will have different cookies. In this way, a web server can distinguish different users on the same system.

Every cookie value contains an indication of when the cookie was created. A cookie value *expires* when too much time has elapsed after the cookie's creation time. For example, suppose a web server is interacting with a user and a full day elapses between one interaction and the next. Even if the cookie identifier shows that the two interactions come from the same web browser, the web server almost always assumes that the two interactions are independent operations—the second interaction is *not* a direct follow-up to the first. In fact, most cookies are considered to expire after five or ten minutes (although an application on the server side may set a different expiration time if appropriate).

Cookies are used to determine when a series of interactions are all part of the same *session*. This means that the interactions all come from the same web browser and are close enough together in time that each interaction can reasonably be considered as a direct follow-up to the previous interaction.

Using HTTP

When connecting to a remote HTTP server, it is important to observe the HTTP standard rigorously. One common error is to omit the `\r\n` characters required at the end of an HTTP header.

For more information about HTTP headers and HTTP in general, see:

`http://www.ics.uci.edu/pub/ietf/http/`

and:

`http://www.w3.org/pub/WWW/Protocols/Overview.html`

Also try to test with different servers where possible, since some may be more tolerant of HTTP errors than others.

Chapter 23. Internet components

This chapter discusses the components that appear on the **Internet** page of the component palette. To use these components effectively, you should be familiar with the internet fundamentals described in Internet concepts and terminology.

Sockets

As described in Sockets and ports, a socket provides a two-way connection between programs running on different systems on the Internet. For example, suppose a branch of a company wants to communicate with the home office. A program on the branch's computer could establish a socket connection with another program on the home office's computer. When one program outputs data into the socket, the other program receives the data from the other end of the socket. Transmission of the data is handled through the Internet (or some other supported communication channel).

Standard JDK sockets vs. PowerJ sockets

PowerJ offers two different groups of sockets:

- Socket objects defined by the standard JDK specifications, using class definitions in `java.net.*`. These objects are found on the **Standard** page of the component palette.
- Socket objects defined by PowerJ itself, using class definitions in `powersoft.powerj.net.*`. These objects are found on the **Internet** page of the component palette.

Both groups of objects serve similar purposes, but PowerJ sockets offer a wider range of functionality.

For information about JDK sockets, see the standard JDK documentation. This guide only describes native PowerJ sockets.

Socket objects

Normal PowerJ sockets are represented by Socket objects. PowerJ gives sockets default names of the form *socket_N*. On the **Internet** page of the component palette, sockets are represented by the following button:

Socket objects placed on a form are visible at design time, but are not seen at run time.

Blocking vs. non-blocking sockets

A *blocking* socket is used in a synchronous way. The effect of blocking depends on whether data is being sent or received.

- Suppose that program X sends data into a connected socket. The socket then blocks (refuses to take more data) until the data has been buffered for delivery by the I/O subsystem library. This does not mean that the data has been successfully delivered to the remote machine; it just means that the I/O subsystem library has made its own copy of the data and has taken responsibility for transmitting the copied data.
- Suppose that program Y calls a function indicating that it wants to receive data from the socket. In this case, the function call does not return until some data has actually been received from the other end. This means that your program does nothing else until some data actually comes in.

A *non-blocking* socket is used in an asynchronous way. A program may output data into a connected socket, then output more data before the first transmission is acknowledged. Similarly, the program does not have to wait to receive data; instead, a **SocketDataArrival** event is triggered on the socket when there is data ready to read, so the program can be written to respond to such an event rather than calling a function that waits until something comes in.

This shows that non-blocking sockets are used in an event-driven way. Non-blocking sockets are therefore more in keeping with PowerJ programs. However, blocking sockets have their uses too; they are especially appropriate in server applications where multithreading is likely.

Types of sockets

A *connection-based socket* is a socket that must be explicitly connected before you can send data to a remote system.

PowerJ defines the following classes derived from the Socket class:

DatagramSocket
: Represents a non-connection-based *datagram* socket (that uses the UDP protocol). A datagram is essentially a complete packet of data, kept together during transmission. You cannot create a DatagramSocket object directly. Instead, you create a normal Socket object and set the **Type** property to `SockDGRAM`.

  Since this type of socket is not connection-based, you can **send** data down the socket as soon as the socket object is created and you have specified a remote system and port.

StreamSocket
: Represents a connection-based socket, whether it is a true stream socket (using the TCP protocol) or a datagram socket. You cannot create a StreamSocket object directly. However, the base Socket class creates a StreamSocket object if you create a connection-based socket.

  Since this type of socket is connection-based, you must explicitly **connect** the socket to another system before sending data.

Since both of these socket types are based on Socket, they share all the methods and properties of Socket.

Note: The DatagramSocket and StreamSocket classes are not meant to be used directly. Instead, PowerJ generates them indirectly, based on information specified for the **Type** property of a Socket object.

Design-time socket properties

The following list discusses socket properties that can be set at design time:

Type [**General** page]
The type of data that will be transmitted using the socket. The default is `SockSTREAM`, indicating a connection-based TCP socket. `SockDGRAM` indicates a non-connection-based UDP socket and `SockUNKNOWN` represents an unknown type of socket. For a complete list of possible types, see the *PowerJ Component Library Reference*.

RemotePort [**General** page]
The port number on the remote computer. If you specify a port number of zero, the socket uses any free remote port.

RemoteHostName [**General** page]
The name of the remote computer. This is a String giving the address of the remote computer in the standard Internet format.

Note: At design time, you should not enter a numeric IP address for the **RemoteHostName**; only use the domain name form.

Asynchronous [**General** page]
If this property is turned on, the socket is non-blocking; otherwise, it is blocking. By default, **Asynchronous** is turned off, producing a blocking socket.

AutoConnect [**General** page]
If both **AutoConnect** and **Asynchronous** are turned on, the program automatically attempts to connect to the remote system when the socket object is created at run time.

If **AutoConnect** is turned off, the program does not attempt to connect to the remote system and does not attempt to initialize the socket when it is constructed.

Using these properties you can specify a socket that connects to a remote computer for intersystem communications. For example, if you specify a **RemoteHostName** and **AutoConnect**, the program automatically attempts to establish a connection with the remote system when the socket object is created at run time.

Other socket properties

Sockets have a number of properties which can only be set at run time. The following list discusses some of these properties:

RemoteInetAddress
This is a read-only property similar to **RemoteHostName**, but it expresses the Internet address as an array of four bytes. For example, if the numeric form of the address is `172.31.2.50`, **getRemoteInetAddress** returns an array of bytes giving the four values `172`, `31`, `2`, and `50`.

LocalHostName
This is a read-only property giving the name of the system where PowerJ is running. The property is only guaranteed to have a valid value when the socket is connected (since your local computer may have several possible IP addresses).

LocalHostAddress
This is a read-only property giving the IP address of the system where PowerJ is running. The address is expressed as an array of four bytes. **LocalHostAddress** is only guaranteed to have a valid value when the socket is connected (since your local computer may have several possible IP addresses).

LocalPort
Specifies a local port for the socket. This is not a read-only property, but connection-based sockets should not set the local port; instead, the local port is determined by the system at connection time. If the local port is set to zero, the socket can use any free port that is available.

BytesWaiting
When you use a blocking socket, the **BytesWaiting** property tells the number of bytes of data that are waiting to be received by your program. If **BytesWaiting** is non-zero, and you attempt to receive the given number of data bytes from the socket, the **Receive** function will not block, since there is that much data immediately available.

The values of the above properties can be determined with an appropriate **get** method. For example, **getBytesWaiting** determines the number of bytes currently waiting to be read. Similarly, if a property is not read-only, there is a **set** method to set the value of the property. For example, **setRemoteHostName** sets the name of the remote host.

Note: Most of the rest of this section applies to stream mode sockets, as opposed to datagram sockets. With stream mode sockets, data transfer is reliable, sequenced, and unduplicated.

Connecting to a remote system

The **connect** method attempts to connect a socket to the remote system that is specified by the socket's properties. For example,

```
socket_1.setRemoteHostName( "system.domain" );
boolean status = socket_1.connect( );
```

attempts to connect the socket with the given system. The **connect** method returns `true` if the connection attempt succeeds and `false` if it fails.

Writing on a socket

The **send** method writes data into the socket so that the data is transmitted to the remote application:

```
// byte buf[];
boolean status = socket_1.send( buf );
```

The `buf` argument specifies a buffer containing the data to send. The **send** method returns `true` if the operation succeeds and `false` otherwise.

There is a second version of **send**:

```
// int length;
boolean status = socket_1.send( buf, length );
```

This sends the given number of bytes from the buffer.

Reading data from a socket

The **receive** method reads data from the socket:

```
// byte buf[];
int result = socket_1.receive( buf );
```

The `buf` argument specifies a buffer where **receive** can store the data read from the socket. The maximum number of bytes that can be read by this version of **receive** is the number of bytes in the `buf` array. The result of **receive** is:

- `-1` if an error occurs.
- `0` if the remote system closes the connection.
- Otherwise, the result is the number of bytes received. For datagram sockets, the result is always the number of bytes received.

There is a second form of **receive**:

```
// int length;
int result = socket_1.receive( buf, length );
```

The `length` argument specifies the maximum number of bytes you want to receive. The result of **receive** is the same as in the previous form.

Closing a socket

When your program is finished using a socket, you can use the **close** method to close the socket:

```
boolean success = socket_1.close( );
```

The result of **close** is `true` if the operation succeeds and `false` if an error occurs.

Handling socket errors

Most socket methods return `false` if an error occurs during an operation. In this case, you can use **getLastError** to determine what went wrong. The result of **getLastError** is an integer indicating the source of the error.

For example, suppose that you attempt to **connect** to a remote system, but the **connect** method returns `false`. You can then use

```
int code = socket_1.getLastError( );
```

to obtain an error code indicating the nature of the problem. For example, **getLastError** might return the value `SockEIsConnected` to indicate that the socket is already connected to another system.

Note: Possible error code values are defined in the SocketExceptionCode interface. For more information, see the *PowerJ Component Library Reference*.

The **getLastError** method returns the most recent error to occur on the socket. This means that you should check the last error after any socket operation that fails. Suppose, for example, that you attempt to **connect** to a socket, but the operation fails; then suppose that you try to **send** data down the socket. The **send** operation will receive an error, since the socket isn't connected on the other end. This error from **send** overwrites the error information recorded for **connect**. As a result, your program will no longer be able to tell why the **connect** failed.

Resetting after errors

The **resetLastError** method cleans up after the last error detected. For example, suppose your program receives an error after a **send** operation. You use **getLastError** to determine what went wrong and then correct that problem. Once the problem has been corrected,

```
socket_1.resetLastError( );
```

turns off the error flag associated with the socket and resets the socket into a workable state. Whenever you recover from an error, you should use **resetLastError** to mark the socket as "clean".

Socket events

Sockets which have the **Asynchronous** property turned on may receive events. Possible events are:

SocketConnect
A connection request has been accepted by the remote system.

SocketHostResolved
The socket support software has determined the true Internet address of the remote system. This event takes place after you have specified a new remote system with **setRemoteHostName**. The event tells you that the name has been successfully converted into a raw Internet address.

SocketDataArrival
Data has arrived from the remote system and is ready to be read by **receive**.

SocketError
An error occurred during an operation on the socket.

SocketSendComplete
This event is triggered when all the data from a **send** operation has been delivered to the I/O subsystem for handling.

The event handlers for these events receive an *event* data block providing information about the event. For example, a **SocketDataArrival** event handler receives a pointer to a SocketDataArrivalEvent object; this is derived from the usual EventData, but also includes the method **getBytesReceived** which returns the number of bytes that just arrived.

For more information on any of these events, see the *PowerJ Component Library Reference*.

Datagram sockets

Datagram sockets do not need to be connected before sending data. For example, you can simply use the following sequence

```
// int portNumber;
Socket sendsocket_1 = new Socket( );
sendsocket_1.create( powersoft.powerj.net.Socket.SockDGRAM );
sendsocket_1.setRemoteHostName( "system.domain" );
sendsocket_1.setRemotePort( portNumber );
sendsocket_1.send( buffer );
```

to send to the designated system. Notice there is no call to **connect**.

On the receiving end, you can prepare the receiving socket with:

```
// int portNumber;
Socket rcvsocket_1 = new Socket( );
rcvsocket_1.create( powersoft.powerj.net.Socket.SockDGRAM );
rcvsocket_1.setLocalPort( portNumber );
rcvsocket_1.receive( buffer );
```

This receives the datagram from the specified port. After the datagram has been received, the **RemoteHostName** property for the socket will contain the host name of the sender (as obtained from the datagram). Similarly, the **RemotePort** property will contain the sender's port number.

These two operations establish both ends of the socket. Once this has occurred, either system may **send** or **receive** on the socket.

Note: Datagram sockets do not guarantee reliable, sequenced, or unduplicated data transfer.

Server sockets

A *server socket* is used by network server applications to listen for incoming requests from client applications. Server sockets can be compared to telephone switchboard operators—they listen for incoming calls and connect each call to its intended recipient. This means that a server socket waits for incoming socket connection requests, then services each request by creating an appropriate socket and making the new socket available to the program that will actually use it.

Server sockets are represented by ServerSocket objects. PowerJ gives server sockets default names of the form *srvsocket_N*. On the **Internet** page of the component palette, server sockets are represented by the following button:

Server socket objects placed on a form are visible at design time, but are not seen at run time. Server sockets support many of the same properties and methods as normal Socket objects. These include the properties:

```
Asynchronous
RemoteHostName
RemoteInetAddress
RemotePort
LocalHostName
LocalHostAddress
LastError
```

and the methods

```
close
resetLastError
```

Server sockets do not support **send**, **receive**, or **connect**. Server sockets may not have the datagram type (`SockDGRAM`).

Note: Server sockets are not intended to be used for reading or writing. They are merely used for making connections.

Server socket events

Server socket methods

Server socket events

Server sockets support the following events (in asynchronous mode):

SocketError
An error occurred during an operation on the server socket.

SocketConnectionRequest
The server socket has received a connection request from a client.

For further information on these events, see the *PowerJ Component Library Reference*.

Server socket methods

A server socket's job is to wait for clients to attempt a connection, and to accept connections when they are received. Before you can listen for incoming connection requests, you must set the **LocalPort** property so that the server socket knows where to listen.

The **accept** method waits for a connection request to be received and then accepts the request:

```
Socket socket = srvsocket_1.accept( );
```

The result of **accept** is a Socket object which can then be used to communicate with the program that submitted the connection request. For example, you can execute the **send** method on this socket to send data down the socket to the remote system. Here is typical code for using a server socket:

```
// int portNumber;
socket.setLocalPort( portNumber );
Socket client = socket.accept( );
if ( client != null )
{
    String msg;
    int received = client.receive( msg );
    if ( received > 0 )
        System.out.println( msg );
}
```

Note: The client socket returned by **accept** is an asynchronous socket. This means, for example, that when you call **send** on this socket, the actual data transmission takes place on a separate execution thread. You should not close the socket until you have received a **SocketSendComplete** event indicating that the send operation is finished.

Internet components

PowerJ supports several components that can be used to establish various types of Internet connections:

- The Internet class (`powersoft.powerj.net.Internet`)—this can be used to open a connection to a remote Server system using a variety of protocols.
- The HTTP class (`powersoft.powerj.net.HTTP`)—this is based on the Internet class but offers additional methods for submitting HTTP requests to the remote web server.
- The FTP class (`powersoft.powerj.net.FTP`)—this is based on the Internet class but offers additional methods for submitting FTP requests to a remote FTP server.

All of these classes are intended for use on the User side of an Internet connection; they are not appropriate for use on the Server side.

The Internet component class

An Internet object makes it easy to establish a connection with remote servers on the Internet. PowerJ gives Internet objects default names of the form *internet_N*. On the **Internet** page of the component palette, Internet objects are represented by the following button:

Internet objects placed on a form are visible at design time, but are not seen at run time.

Internet object properties

The **URL** property of an Internet object associates a URL with the object. The **URL** is typically constructed from separate properties specifying the components of the URL as String objects:

- **ProtocolName**: the name of the protocol (e.g. `http` or `ftp`).
- **ServerName**: the name of the remote server (e.g. `www.sybase.com`).
- **ServerPort**: the port number on the remote system (this is an int value, not a String). A value of -1 means any available port.
- **File**: the file part of the URL.
- **Ref**: any additional reference information for the URL.

For example, the following sequence of calls initialize the URL associated with an Internet object:

```
internet_1.setProtocolName( "http" );
internet_1.setServerName( "www.sybase.com" );
internet_1.setFile( "products/powerj/index.html" );
```

In a similar way, you could construct URLs to use services like Gopher, FTP, and so on. If a property is not initialized explicitly, it is set to a null string (except for **ServerPort**, which is set to -1).

Internet objects also support String properties named **UserName** and **UserPassword**. These properties are often useful when interacting with remote servers that require login procedures of some kind (for example, FTP).

Note: The parts of a URL can also be set at design time using the property sheet of the Internet object.

Establishing an Internet session

The **open** method of the Internet class attempts to establish a connection with the URL specified by the Internet object's **URL** property. The following code sets up the parts of a URL, then attempts to open a connection:

```
internet_1.setProtocolName( "http" );
internet_1.setServerName( "www.sybase.com" );
internet_1.setFile( "products/powerj/index.html" );
internet_1.open( );
```

The process of opening a URL establishes an input stream for your program. This input stream receives data from the remote server as a stream of bytes. For example, if you open a URL that specifies an HTML file, the HTML data is delivered to your program using this input stream.

Reading the input stream

You can read the input stream using the **readFile** method of the Internet class. The simplest version of this is

```
// byte buffer[];
int byteCount = internet_1.readFile( buffer );
```

The **readFile** method reads bytes into the specified buffer, up to the maximum number of bytes that the buffer can hold. The return value of **readFile** specifies the actual number of bytes read. The data that is read comes from the remote server, sent in response to your URL.

The **readFile** method may also take a second form:

```
// byte buffer[];
// InputStream stream;
int byteCount = internet_1.readFile( buffer, stream );
```

This form reads the specified input stream rather than reading the input sent by the remote server.

Once you have an input stream, you can read it using the standard **readLine** method of Java's DataInOutStream class.

Determining how much data is available

The **queryDataAvailable** method of Internet lets you determine how much data is available on the input stream associated with the URL connection:

```
int byteCount = internet_1.queryDataAvailable( );
```

The result is the number of bytes of data waiting to be read. This data can be read by calling **readFile**.

Closing a connection

The **close** method of Internet closes a connection that has previously been opened with **open**:

```
internet_1.close( );
```

The HTTP component class

The HTTP class is derived from the Internet class. Therefore, it has all of the "generic" Internet connection services defined in Internet, but has additional methods which are specific to HTTP connections. In particular, HTTP objects make it easy to construct HTTP requests and submit them to a web server on the remote system.

Note: This guide makes no attempt to provide technical details about HTTP requests and other interactions with web servers. For information, see RFC 1945, available at either of:

`ftp://ds.internic.net/rfc`
`ftp://nic.merit.edu/documents/rfc`

PowerJ gives HTTP objects default names of the form *http_N*. On the **Internet** page of the component palette, HTTP objects are represented by the following button:

HTTP objects placed on a form are visible at design time, but are not seen at run time.
Since the HTTP class is derived from the Internet class, it supports all the methods and properties of Internet. In particular, every HTTP object may have an associated URL which is established by setting the properties **ServerName**, **ServerPort**, **File**, and **Ref**. You do not have to set **ProtocolName**, since that is assumed to be `HTTP`. For more information about these properties, see Internet object properties.

Establishing an HTTP connection

The **connect** method of HTTP connects to the URL associated with the HTTP:

```
boolean success = http_1.connect( );
```

The result is `true` if the connection succeeded and `false` otherwise.

The **connect** method attempts to establish two I/O streams with the remote web server:

- An input stream that your application can read using the standard **readLine** method of Java's DataInoutStream class. This input stream delivers data from the remote web server to your application. For example, this stream might contain HTML code or a JPEG graphic.
- An output stream that your application can use to send HTTP requests to the remote web server.

HTTP requests

An HTTP request consists of a number of *headers*. These are text lines of the form

```
headername: value
```

Submitting an HTTP request follows these steps:

1. Begin constructing the request with the **openRequest** method.
2. Create a sequence of headers using **addRequestHeader** to create each header.
3. Transmit the request to the remote web server using **sendRequest**.
4. Free up the memory occupied by the request with the **closeRequest** method.

Each of these steps is discussed in the sections that follow.

Constructing the request

The **openRequest** method of HTTP constructs the opening line for the HTTP request. It has the format:

```
// String object;
// String verb;
// String version;
boolean success = http_1.openRequest( object, verb, version );
```

The result is `true` if the request can be opened and false otherwise. Here's an example of a typical **openRequest** call:

```
boolean success =
    http_1.openRequest( "http://www.sybase.com/file.html",
        "GET",
        "HTTP/1.0" );
```

Adding request headers

The **addRequestHeader** method of HTTP adds a header line to the request being constructed:

```
// String headerField;
// String headerFieldValue;
// int headerFieldFlag;
boolean success =
    http_1.addRequestHeader( headerField,
        headerFieldValue,
        headerFieldFlag );
```

For example, the following constructs a typical header:

```
boolean success =
    http_1.addRequestHeader( "Content-Encoding", "x-gzip", 0 );
```

The `headerFieldFlag` argument is one of the following values:

`HTTP_ADDREQ_FLAG_ADD_IF_NEW`
: Only adds this request header if it is new. If there is already a request header of this type, the specified header is not added and **addRequestHeader** returns `false`.

`HTTP_ADDREQ_FLAG_REPLACE`

Adds this request header and deletes any previous header of this type if one already exists.

If you specify a value of zero for `headerFieldFlag`, the default is `HTTP_ADDREQ_FLAG_ADD_IF_NEW`.

You may call **addRequestHeader** any number of times to add request headers to the request (or to replace existing headers with new values, provided you specify the flag `HTTP_ADDREQ_FLAG_REPLACE`).

Sending the request

The **sendRequest** method of HTTP transmits the current request and its headers to the remote web server:

```
boolean success = http_1.sendRequest( );
```

The result is `true` if the request is sent successfully and `false` otherwise.

Closing the request

The **closeRequest** method of HTTP performs clean-up after a request has been sent:

```
http_1.closeRequest( );
```

You can also use **closeRequest** to delete a request that you have been constructing. For example, if you begin building a request, then decide not to send it after all, you can delete the partly-constructed requested using **closeRequest**.

Receiving the server's response

Once you have sent an HTTP request, you can receive the web server's response using the **readFile** method:

```
// byte buffer[];
int byteCount = http_1.readFile( buffer );
```

The `byteCount` gives the number of bytes in the server's response. The maximum number of bytes read by **readFile** is the maximum number of bytes that can be stored in `buffer`.

Closing an HTTP connection

The **closeConnection** method of HTTP closes a connection that has previously been opened with **connect**:

```
http_1.closeConnection( );
```

The FTP component class

The FTP class is derived from the Internet class. Therefore, it has all of the "generic" Internet connection services defined in Internet, but has additional methods which are specific to FTP connections. In particular, FTP objects make it easy to perform common FTP operations (for example, changing the current directory, and getting or putting a file) in cooperation with the remote FTP server.

Many FTP operations are only allowed if you have appropriate permissions. For example, you may be allowed to read files from the remote system but not to write files to that system.

A brief introduction to FTP is given in File transfer. For detailed information about the FTP protocol, see RFC 959, available at:

```
ftp://ds.internic.net/rfc
ftp://nic.merit.edu/documents/rfc
```

PowerJ gives FTP objects default names of the form *ftp_N*. On the **Internet** page of the component palette, FTP objects are represented by the following button:

FTP objects placed on a form are visible at design time, but are not seen at run time.

> **Note:** This section does not examine all the methods available in the FTP class; it only deals with the most commonly used. For a complete description of all methods, see the *PowerJ Component Library Reference*.

Establishing an FTP connection

The **connect** method of FTP connects to the URL associated with the FTP:

```
boolean success = ftp_1.connect( );
```

The result is `true` if the connection succeeded and `false` otherwise.

The **connect** method attempts to establish two I/O streams with the remote web server:

- An input stream that your application can read using various FTP methods. This input stream delivers data from the remote FTP server to your application.
- An output stream that your application can use to send FTP requests to the remote FTP server.

Note: For many FTP connections, you must set the **UserName** and the **Password** properties of the FTP object before calling **connect**. For more information about these properties, see Internet object properties.

Obtaining a directory listing

The **retrieveDirectoryListing** method of FTP retrieves the contents of a specified directory on the remote system:

```
// String dirName;
Vector files = ftp_1.retrieveDirectoryListing( dirName );
```

The result is a vector of file names expressed as String objects. For example,

```
Vector files = ftp_1.retrieveDirectoryListing( "pub" );
```

retrieves the contents of the `pub` directory in the current directory. The file names are given by `files[0]`, `files[1]`, and so on.

Changing the current directory

The **changeCurrentDirectory** method of FTP changes the current directory on the remote system:

```
// String dirName;
boolean success = ftp_1.changeCurrentDirectory( dirName );
```

The return value is `true` if the operation succeeds and `false` otherwise (for example, if the specified directory doesn't exist).

Retrieving a file from the remote system

The **retrieveFile** method of FTP retrieves a file from the remote system and copies it to a local file:

```
// String remoteFile;
// String localFile;
// int transferType;
// boolean failIfFileExists;
boolean success = ftp_1.retrieveFile( remoteFile,
    localFile, transferType, failIfFileExists );
```

If `failIfFileExists` is `true`, **retrieveFile** terminates if the specified local file already exists. If `failIfFileExists` is `false`, **retrieveFile** overwrites the local file if it exists.

The `transferType` argument may have one of the following values:

```
powersoft.powerj.net.FTPInterface.TYPE_ASCII
powersoft.powerj.net.FTPInterface.TYPE_BINARY
powersoft.powerj.net.FTPInterface.TYPE_EBCDIC
powersoft.powerj.net.FTPInterface.TYPE_IMAGE
powersoft.powerj.net.FTPInterface.TYPE_LOCAL
```

These indicate the type of data file being retrieved.

The result of **retrieveFile** is `true` if the file is successfully retrieved, and `false` otherwise.

The following shows a typical example of using **retrieveFile**:

```
boolean success = ftp_1.retrieveFile( "rfc959.txt",
    "mycopy.txt",
    powersoft.powerj.net.FTPInterface.TYPE_ASCII,
    false );
```

Sending a file to a remote system

The **putFile** method of FTP sends a file from your local computer system to the remote system:

```
// String localFile;
// String remoteFile;
// int transferType;
// boolean overwriteFile;
boolean success = ftp_1.putFile( localFile,
    remoteFile, transferType, overwriteFile );
```

If `overwriteFile` is `false`, **putFile** will not try to send your file if there is already a file of the given name on the remote system. If `overwriteFile` is `true`, **putFile** will overwrite the remote file, if it exists.

The `transferType` argument may have one of the following values:

```
TYPE_ASCII
TYPE_BINARY
TYPE_EBCDIC
TYPE_IMAGE
TYPE_LOCAL
```

These indicate the type of data file being sent.

The result of **putFile** is `true` if the file is successfully delivered, and `false` otherwise. For example, if you do not have permissions to write files to the remote system, **putFile** returns `false`.

The following shows a typical example of using **putFile**:

```
boolean success = ftp_1.putFile( "myprog.zip",
    "yourprog.zip", TYPE_BINARY, false );
```

Closing an FTP connection

The **closeConnection** method of FTP closes a connection that has previously been opened with **connect**:

```
ftp_1.closeConnection( );
```

Chapter 24. Server extensions

This chapter explains how to use PowerJ to extend web servers, using CGI, NSAPI, ISAPI or Dynamo servlet interfaces.

Support for interface environments

PowerJ supports the CGI, NSAPI, ISAPI and NetImpact Dynamo interface environments. This section describes the underlying architecture which provides this support.

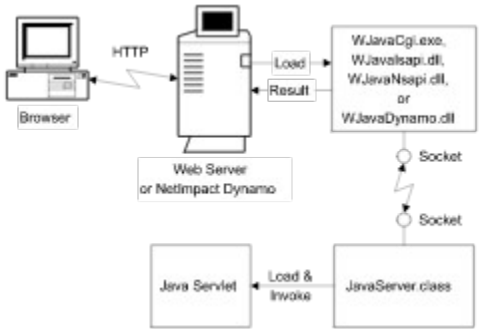
For an introduction to CGI, NSAPI, and ISAPI, see Web server basics.

The following web servers are supported:

- Microsoft Internet Information Server (IIS) using the ISAPI interface.
- Netscape Commerce and Fasttrack servers, using the NSAPI interface.
- O'Reilly WebSite using the ISAPI emulation.
- Any web server that supports CGI.

Basic server support architecture

The following diagram illustrates the basic architecture on which server support is modeled:



As shown, the user's web browser communicates with a web server (or with NetImpact Dynamo) on the Server system. This communication takes the form of a URL, sent using the HTTP protocol. In response to the URL, the web server loads one of the following to access your servlet:

- `WJavaCgi.exe`, a program for responding to CGI requests.
- `WJavaIsapi.dll`, a DLL for responding to ISAPI requests.
- `WJavaNsapi.dll`, a DLL for responding to NSAPI requests.
- `WJavaDynamo.dll`, a DLL for responding to NetImpact Dynamo requests.

The web server invokes `WJavaCgi.exe` every time CGI a URL containing `WJavaCgi.exe` is received from a user. For the DLLs, the web server only has to load the DLLs once, so with those types of web server extensions the server can minimize the overhead of reloading the extension code.

The program or DLL loaded by the server opens a socket to communicate with your servlet application. Your servlet receives and sends information on this socket using methods in a group of classes called the Java Server classes.

In general, your program does not have to be aware that data passes from the web server through a socket—the process is transparent, handled by the support library and the DLL or EXE file loaded by the server.

Restrictions

The basic architecture is governed by the following restrictions:

- Your web server must be running on a Windows 95 or Windows NT system. However your Java servlet can be on any networked machine with a Java VM, since it connects via sockets to the DLLs or EXE running on the server.
- If you are using CGI, each request from the user starts a new application—you cannot retain information from one request to the next.

Creating a servlet target

The process of creating a servlet is similar to creating any other type of target.

- **To create a servlet application:**

1. From the **File** menu on the main PowerJ menu bar, click **New**, then **Target**. This opens the Target Wizard.
2. Under **What type of target do you want?**, click **Java WWW Server Application** (for a JDK 1.02 target) or **Java11 WWW Server Application** (for a JDK 1.1 target), then click **Next**.
3. Select the folder where you want to store the servlet target, then click **Finish**.

A servlet does not have a user interface. Therefore, the servlet does not have a form associated with it. In many cases, however, you will find it useful to create a visual class for the servlet, where you can place objects like transaction and query objects. For more information, see Visual classes for web services.

JavaServer for servlets

Before you can run a servlet target, you must have the JavaServer program running. This program is written in Java and implemented through `JavaServer.class`. Source code for this class is given in the file `JavaServer.java` which is provided for you whenever you create a servlet target (check the Files window).

JavaServer must be invoked as a separate process before you run the servlet. Since it is written in Java and obtains service requests through a socket, it can be run anywhere in your network. You can run the JavaServer program with either of the following commands (for the Sun and Microsoft VMs respectively):

```
java JavaServer
jview JavaServer
```

When you execute the program, you should make sure that your `CLASSPATH` environment variable makes it possible to find the classes of the `powersoft.powerj.server` package. For more information on `CLASSPATH`, see How PowerJ uses the CLASSPATH environment variable. Also, your servlet classes should be published to a directory that JavaServer will search when it loads classes.

Once the JavaServer program is executing, you can invoke your servlet by specifying an appropriate URL to the web server. As part of the URL, you typically use HTTP requests to set the following variables:

```
jserverhost
jserverport
```

You can do this by using a URL of the form

```
http://site/service?class="classname"&jserverhost=host
    &jserverport=port
```

where `classname` is the name of the initial class in the servlet. Alternatively, you can use the HTTP `POST` methods to send the required variables from a form (using HIDDEN fields).

If you omit settings for `jserverhost` and/or `jserverport`, the default is:

```
localhost:1776
```

This specifies port number 1776 on the current system.

Web server Java servlets

When you create a servlet target, PowerJ creates a number of files, class definitions, and so on which are used to build the target. Most of these files are maintained by PowerJ; you shouldn't try to edit them yourself.

To see the items created as part of the target, open the Classes window and expand the entry for the servlet target. In this list, you will see a class which has the same name as the target itself. For example, if the target is called `MyServ`, there will be a class called `MyServ` in the expansion for the target. This is called the *primary class* of the servlet target.

The invoke method

Click the primary class in the left half of the Classes window. In the right half of the Classes window, you see that the primary class contains a number of methods including one called **invoke**. The **invoke** method is the heart of your servlet: this is where you write the code that does the main work of the servlet.

The **invoke** method has the prototype:

```
public void invoke( WebConnection server, String args[] )
     throws IOException;
```

The arguments are:

`WebConnection server`
: An object providing an interface to the web server. For more information, see The WebConnection class.

`String args[]`
: The command line arguments received by your application when it was invoked.

Note: When you create a servlet, the **invoke** method contains sample code which demonstrates a number of programming techniques often used in writing servlets. To write your own servlet, delete this sample code and replace it with your own code.

When the invoke method runs

Your servlet will be invoked when the user sends a URL to the web server, specifying that the servlet should run. Eventually, the request will be passed to JavaServer.

JavaServer creates a new thread which takes care of loading your servlet. The servlet class **invoke** method is called to handle the request.

You add your own user code to the **invoke** method. You can also add your own data members to the primary class.

Once **invoke** has returned, the servlet terminates. If the User sends another URL which invokes the servlet, JavaServer creates a new object of the primary class and goes through the same procedure again. Each invocation of a servlet is handled by a new servlet object.

PowerJ provides facilities that let you store data outside the servlet and retrieve it later, based on the connecting browser. You can also store persistent data in static members of your class. For more information, see Preserving data from one servlet invocation to the next.

The WebConnection class

The **invoke** method receives an argument of the WebConnection class:

```
WebConnection server;
```

The WebConnection class offers methods for interacting with the web server. For example, WebConnection has methods that can obtain form variables and query variables from the web server as well as setting up HTTP response headers. The methods of WebConnection are discussed in the sections that follow.

For more information about HTTP, see:

http://www.ics.uci.edu/pub/ietf/http/
http://www.w3.org/pub/WWW/Protocols/Overview.html

WebConnection exceptions

Many of the methods in the WebConnection class may throw IOException exceptions. Such exceptions indicate that there has been a communication error between your servlet and the web server. For example, this exception occurs if there is a failure on the socket that connects your servlet to the web server.

If a WebConnection method throws an IOException, your code may catch the exception and attempt to deal with the problem. Alternatively, you may choose not to try/catch the exception. Since you are executing these methods inside your **invoke** routine, the result will be that **invoke** passes the IOException up to its caller. The library function that calls **invoke** will deal with the IOException by terminating your servlet.

Response headers

When you begin sending data to the User side, the first thing you should send is a HTTP *header* describing the data to be sent. For example, if you are going to transmit an HTML file, the header is:

```
text/html
```

If you are going to transmit a JPEG graphic, you first send the header:

```
image/jpeg
```

This header makes it possible for the web browser to know how to handle the data that follows. The content types specified by response headers are defined in the MIME standard (Multipurpose Internet Mail Extensions). For information about MIME, see:

http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html

Data sent to the user may have multiple headers. Each header has the form:

```
HeaderName: value
```

To set a header that will be sent to the User side, you use the **setResponseHeader** method of WebConnection, as in:

```
// WebConnection server;
server.setResponseHeader( "Content-Type", "text/html" );
```

The first argument is a String giving the text of the header itself and the second argument is a String giving the value to be associated with the header.

Removing headers

Some interface environments may automatically create headers that you don't actually want to transmit to the user. You can prevent such headers from being delivered to the user with:

```
server.removeResponseHeader( "HeaderName" );
```

Default headers

All the headers that you have defined are automatically sent to the user the first time you execute one of the **write** methods of WebConnection. If you call a **write** method without setting headers first, the default headers are:

```
HTTP/1.1 200 OK
Content-type: text/html
```

Note that the `"200 OK"` specifies a result code of 200 and a message of `OK`, indicating success.

Sending data to the user

The two WebConnection methods for sending data to the user are **write** and **writeLn**. These both write a string to the HTML document. The difference is that **writeLn** automatically adds a new-line character at the end of the string while **write** does not. For example, several **write** operations in a row all write to the same line, while several **writeLn** operations in a row each write to a new line.

The following shows a simple example of **writeLn**:

```
// WebConnection server;
boolean success = server.writeLn( "<P>Hello there." );
```

This writes the given string to the HTML document. Notice that this string contains the `<P>` HTML formatting directive. The HTML file that is created through such calls to **write** and **writeLn** will eventually be sent to the User system.

The result of **write** and **writeLn** is `true` if the output operation succeeds and `false` otherwise.

Note: WebConnection also supports methods named **print** and **printLn**. These are identical to **write** and **writeLn** (respectively).

Setting up a print stream

Another way to send output to the User side is to use a PrintStream object associated with the User system:

```
PrintStream ps = server.getPrintStream();
```

Any data written to this PrintStream object is automatically transmitted to the User system, as your response to the URL sent by the user. You can use any of the standard PrintStream methods to write on this stream.

You may also use a PrintStream designed for logging debug information:

```
PrintStream debug = server.getDebugOutput();
```

Form variables

Form variables are created when the HTML form uses `METHOD=POST`. The name of a form variable is the name given in the HTML form, and the value of a form variable is a string giving the data specified by the user when filling out the form.

To obtain the value of a simple form variable, use:

```
// WebConnection server;
String str = server.getFormVariable( "varName" );
```

This returns the value of the variable as a string.

Some form variables may have multiple values. In this case,

```
String str = server.getFormVariable( "varName", N );
```

to get the `N`th value associated with the variable. The first form variable has an index of zero.

You can set a new string value for a form variable with:

```
server.setFormVariable( "varName", "newValue" );
```

This new value can be retrieved later with:

```
str = getFormVariable( "varName", 0 );
```

The **setFormVariable** method does not affect any of the `N` other values associated with the given name.

Determining whether data was POSTed

The **getIsPostMethod** method of WebConnection determines whether this servlet was invoked from a form with `POST`ed variables:

```
boolean posted = server.getIsPostMethod( );
```

returns `true` if the form used `POST`ed variables and `false` otherwise.

Query variables

Query variables are created when the HTML form uses `METHOD=GET`. The WebConnection methods for dealing with query variables are similar to those for form variables:

```
String str = server.getQueryVariable( "varName" );
String str = server.getQueryVariable( "varName", N );
server.setQueryVariable( "varName", "newValue" );
```

Result codes

The **setResultCode** method of WebConnection sets the HTTP 1.0 result code for your application:

```
setResultCode( 200 );
```

The result code is returned to the web browser that invoked your servlet. Set the result code before sending any output to the user, otherwise the default of 200 is used. The result code is returned as part of the headers sent to the user system.

Note: You can change the result code any number of times before the headers are sent to the user. For example, you may wish to set an `OK` result code when you are first invoked, then change the value if you encounter any errors during processing. The result code that the user receives will be the value at the time that headers are sent (typically by the first **write** or **writeLn** call).

Redirecting the browser to another URL

The **redirect** method redirects the user to another URL:

```
server.redirect( "http://www.anothersite.com" );
```

The effect is to force the user's web browser to access another file or site.

Preserving data from one servlet invocation to the next

The Session class makes it possible for you to preserve data from one invocation of your servlet to the next. It does this by providing access to the web server's own facilities for preserving data across invocations by using "cookies".

Obtaining the initial Session object

The first time your servlet is invoked, it should obtain an object of the Session class, using the following WebConnection method:

```
// WebConnection server;
// String name;
Session session = server.getSession( sessionName );
```

The `sessionName` argument can be any string; your servlet uses this name to identify itself and distinguish itself from other servlets that might be running at the same time.

You must call **getSession** before you do any output (with **write** or **writeLn**).

Important: Session names must be unique among all the applications on the web server. This requirement is forced by the way that HTTP identifies sessions.

Obtaining a previous Session object

At the beginning of each invocation, your servlet can use **getSession** to obtain the current Session object. You can tell whether this is the first invocation by using **getSessionPresent**:

```
String id = server.getSessionPresent( sessionName );
```

The **getSessionPresent** method returns `null` if there is no valid identifier ("cookie") for the present session, indicating that there have been no previous invocations in the session. Therefore, you might use:

```
String id = server.getSessionPresent( sessionName );
if ( id == null ) {
    session = server.getSession( sessionName );
}
```

This code attempts to get a pre-existing identifier for this server session. If no such identifier exists (typically because this is the first time the servlet has been invoked in this session), the servlet obtains a new Session object for this session.

If the identifier is valid for the present session, the **getSessionPresent** method will return the value associated with the identifier (the value of the cookie).

The **getSessionPresent** method may return `null` if too long an interval has passed between this invocation and a previous invocation by the same user. In this case, the session is considered to have *expired* because there was too long a gap between interactions. Therefore, your servlet should consider that this is the start of a new session rather than a continuation of a previous one.

Persistent variables

The Session class supports two important methods:

```
boolean success = session.setValue( "varName", "value" );
```

```
Object value = session.getValue( "varName" );
```

The first stores a `name=value` variable in the web server. The second retrieves the value of a specified variable.

Now, suppose you save a number of session variables in one invocation of your servlet. The next time your servlet is invoked, it can again call:

```
session = server.getSession( sessionName );
```

If the servlet was invoked from the same User system and User browser, and if it has been a relatively short time since the last invocation of this servlet (the maximum length of time is typically ten minutes), the new invocation of the servlet receives the same Session data object as the previous invocation. The servlet can then use this Session object to retrieve all the session variable data stored in the last invocation. This allows persistent data to survive from one invocation to the next.

Visual classes for web services

Web services run on the Server system; they interact with programs on User systems but do not interact directly with human users. Therefore, web services cannot have PowerJ forms associated with them. Typically, they use PowerJ visual classes instead.

For example, if you have a web service that interacts with a database, you can begin designing the application by placing a transaction object and query object on a visual class form. In this way, you can use the normal design-time property sheets to set up the transaction and the query.

If you use a form instead of a visual class, you must make the form non-visible. (Turn off the **Visible** style in the form's property sheet.) This prevents the application from trying to display the form. If a web service tries to display a **Visible** form, it will receive an execution error, since there is no place for the service to display the form.

Debugging servlets

The best way to debug a servlet is to invoke the servlet through a WebApplication target. WebApplications are discussed in WebApplication targets.

In order to debug the servlet, you set the WebApplication's run options to run the Java server (typically, the JavaServer program, as discussed in Basic server support architecture). You will probably want to run a browser, so turn on the appropriate run options and pick an initial URL that will invoke the servlet program. For example, if you are trying to debug a CGI servlet, your URL should eventually cause `WJavaCgi.exe` to run. Depending on your web server, you might specify this program directly in the URL or invoke the program with the `POST` method in an HTML form.

Note: Make sure you specify the servlet name correctly. Also ensure that your `CLASSPATH` includes the servlet class files when you start JavaServer.

PowerJ waits for JavaServer to load and run your servlet. This only happens when JavaServer is invoked by one of `WJavaCgi.exe`, `WJavaIsapi.dll`, `WJavaNsapi.dll`, or `WJavaDynamo.dll`. For example, if you set a breakpoint to go off when your servlet begins executing, the breakpoint is only triggered when JavaServer begins running the servlet.

NetImpact Dynamo server applications

The NetImpact Dynamo web server provides access to databases located on the Server system. NetImpact Dynamo servlets are typically used as part of a larger web application, where applets running on the User system interact with servlets running on the Server system.

For an explanation of the basic architecture underlying NetImpact Dynamo server applications, see Basic server support architecture.

- **To create a NetImpact Dynamo server application:**

1. From the **File** menu on the main PowerJ menu bar, click **New**, then **Target**. This opens the Target Wizard.
2. Under **What type of target do you want?**, click **Java Dynamo Server Application** (for a JDK 1.02 target) or **Java11 Dynamo Server Application** (for a JDK 1.1 target), then click **Next**.
3. Select the folder where you want to store the Dynamo target, then click **Finish**.

A Dynamo servlet does not have a user interface. Therefore, the servlet does not have a form associated with it. In many cases, however, you will find it useful to create a visual class for the servlet, where you can place objects like transaction and query objects. For more information, see Visual classes for web services.

For details about the capabilities of NetImpact Dynamo, see A Dynamo WebApplication and the *NetImpact Dynamo User's Guide*.

Running a Dynamo servlet

In order to run a NetImpact Dynamo servlet extension, you have to perform the following setup operations.

1. Configure Dynamo for the servlet

In order to run the servlet as a Dynamo extension, you must configure Dynamo to recognize the servlet. This means that you must add the following line to Dynamo's `autoexec.ssc` (found in the folder `SybTools\site\system`):

```
site.AddExtension( "javaservlet", dllLocation );
```

The `dllLocation` argument is the full file name of the `WJavaDynamo.dll` file in the `system` subfolder under your main PowerJ folder. The line should be added before the last `-->`. Note that you should use forward slashes (`/`) or double the backslashes.

The following shows an example:

```
site.AddExtension( "javaservlet",
   "c:/Program Files/Powersoft/Powerj/System/WJavaDynamo.dll");
```

Once you have added this to `autoexec.ssc`, stop the Dynamo server and restart it again.

2. Create a template that invokes the servlet

Next you must create a Dynamo template that invokes the servlet. The following shows a simple example:

```
<HTML>
<TITLE>MyServletTemplate.stm</TITLE>
<BODY>
<H1>Java Servlet Sample</H1>

<!--javaservlet
Param1
Param2
-->

</BODY>
</HTML>
```

Notice the following:

- The servlet is invoked by the `<!--javaservlet ... -->` directive.
- For information on how pass the class name, host and port number, see JavaServer for servlets.
- Subsequent arguments are parameter values you want to pass to the servlet.

For more information about creating a Dynamo template, see the *NetImpact Dynamo User's Guide*. There is also an example of setting up a Dynamo template in A Dynamo WebApplication.

3. Start the JavaServer

Next you must start the JavaServer program. For the Sun VM you should use the following command:

```
java JavaServer
```

For the Microsoft VM you should use the following command:

```
jview JavaServer
```

For more detailed information on this program, see JavaServer for servlets.

4. Invoke the template from a web browser

The final step in running a Dynamo servlet extension is to invoke the Dynamo template from a browser. Typically, you would do this by opening a URL of the form

```
http://hostname/site/MyServerTemplate.stm
```

where `hostname` and `site` match your Dynamo server configuration and `MyServerTemplate.stm` is the template you created in a previous step.

Tip: One easy way to invoke the template is to create a WebApplication target whose run options are set to open the given URL with a web browser. You can also use a WebApplication target to run and debug servlets under JavaServer. For more information, see WebApplication targets. Be sure to pass in the class, host and port number, as described in JavaServer for servlets.

Dynamo Java servlets

When you create a Dynamo servlet target, PowerJ creates a number of files, class definitions, and so on which are used to build the target. Most of these files are maintained by PowerJ; you shouldn't try to edit them yourself.

To see the items created as part of the target, open the Classes window and expand the entry for the Dynamo target. In this list, you will see an item which has the same name as the target itself. For example, if the target is called `MyDynamo`, there will be an item called `MyDynamo` in the expansion for the target. This is called the *primary class* of the Dynamo target.

The invoke method

Click the primary class in the left half of the Classes window. In the right half of the Classes window, you see that the primary class contains a number of methods including one called **invoke**. The **invoke** method is the heart of your servlet: this is where you write the code that does the main work of the servlet.

The **invoke** method has the prototype:

```
public void invoke( DynamoConnection server, String args[] )
     throws IOException;
```

The arguments are:

`DynamoConnection server`
: An object providing an interface to the Dynamo server. For more information, see The DynamoConnection class.

`String args[]`
: The command line arguments received by your application when it was invoked.

Note: When you create a servlet, the **invoke** method contains sample code which demonstrates a number of programming techniques often used in writing servlets. To write your own servlet, delete this sample code and replace it with your own code.

When the invoke method runs

Your servlet will be invoked when the user sends a URL to the web server, specifying that the servlet should run. Eventually, the request will be passed to JavaServer.

JavaServer creates a new thread which takes care of loading your servlet. The servlet class **invoke** method is called to handle the request.

You add your own user code to the **invoke** method. You can also add your own data members to the primary class.

Once **invoke** has returned, the servlet terminates. If the User sends another URL which invokes the servlet, JavaServer creates a new object of the primary class and goes through the same procedure again. Each invocation of a servlet is handled by a new servlet object.

You cannot preserve data from one invocation to the next inside the servlet itself. However, NetImpact Dynamo provides a feature that will retain data for five minutes. It works like this:

- Your servlet can set any number of `name=value` variables within NetImpact Dynamo. This is done using methods of the DynamoSession class. For further information, see The DynamoSession class.

- If Dynamo receives a request from the same User browser within five minutes of the termination of the previous servlet, Dynamo allows the new invocation of the servlet to obtain the data saved by the previous invocation.

In this way, Dynamo makes it possible for you to define data that is persistent from one invocation of the servlet to the next and specific to a given user access.

Dynamo exceptions

Many of the methods in the Dynamo-related classes may throw IOException exceptions. Such exceptions indicate that there has been a communication error between your servlet and the Dynamo server. For example, this exception occurs if there is a failure on the socket that connects your servlet to the Dynamo server.

If a method throws an IOException, your code may catch the exception and attempt to deal with the problem. Alternatively, you may choose not to try/catch the exception. Since you are executing these methods inside your **invoke** routine, the result will be that **invoke** passes the IOException up to its caller. The library function that calls **invoke** will deal with the IOException by terminating your servlet.

The DynamoConnection class

The **invoke** method receives an argument of the DynamoConnection class:

```
DynamoConnection server;
```

The DynamoConnection class offers methods for interacting with the NetImpact Dynamo server.

Several of the methods in DynamoConnection return data objects which provide specific information about interactions with Dynamo. These methods are:

`DynamoDocument doc = server.getDynamoDocument();`
Gets a new object representing the document object on the Dynamo server. For more information, see The DynamoDocument class.

`DynamoSession ses = server.getDynamoSession();`
Gets a new object representing the Dynamo session. For more information, see The DynamoSession class.

`DBConnection db = server.getDBConnection();`
Gets a new object representing a connection to the Dynamo session's connection to a database. For more information, see The DBConnection class.

`DynamoQuery q = server.createDynamoQuery( SQLstmt );`
Gets a new object that represents the specified query on the Dynamo server. For more information, see The DynamoQuery class.

`DynamoQuery q = server.getDynamoQuery();`
Obtains the current DynamoQuery object associated with this Dynamo session.

Sending output to the user

The **writeOutput** method of DynamoConnection sends a string to the User system (the user's web browser):

```
// String str;
// boolean addNewLine;
server.writeOutput( str, addNewLine );
```

If `addNewLine` is `true`, **writeOutput** automatically appends a new-line character to the end of the string once it has been written. If `addNewLine` is `false`, **writeOutput** does not add a new-line character.

The **writeByte** method is similar, but writes a single byte:

```
// int b;
server.writeByte( b );
```

Typically, the output sent to the user should take the form of HTML code.

The DynamoSession class

The DynamoSession class preserves information from one invocation of a servlet to the next. Each time your servlet is invoked, it receives a new DynamoConnection object. However, each invocation in a connected sequence of interactions receives the *same* DynamoSession object. This means that the DynamoSession object can store data during one invocation of the servlet, then retrieve the stored data during a subsequent invocation.

DynamoSession has two important methods: **setValue** and **getValue**. These methods make it possible for you to store data in the form of `name=value` variables within NetImpact Dynamo itself.

The **setValue** method has the prototype:

```
boolean setValue( String name, String value )
    throws IOException
```

For example,

```
boolean success = session.setValue( "myvar", "abc" );
```

stores a variable named `myvar` with a value of `"abc"`. This data is only retained for five minutes between invocations of the servlet. If there is a longer gap, your servlet may not have access to this information.

The **getValue** method has the prototype:

```
String getValue( String name ) throws IOException
```

For example,

```
String value = session.getValue( "myvar" );
```

obtains the value that was previously assigned to `myvar`.

The DynamoDocument class

The DynamoDocument class provides methods for creating an HTML document that will be sent to the User system in response to the URL that invoked this servlet. Each time **RunApp** is invoked, you get a new DynamoDocument object. This object corresponds to the template that originated this servlet request.

Information requests

The DynamoDocument class supports a number of methods which provide information about the current document (based on the corresponding Dynamo template):

`String name = document.getLocation();`
Gets the full file name of the document (including folder names).

`String name = document.getName();`
Gets the file name of the document, including extensions but not including folder names.

`int id = document.getId();`
Gets the internal ID of the Dynamo document.

`int connId = document.getConnectionId();`
Gets the ID of the associated connection object.

`String connName = document.getConnectionName();`
Gets the name of the associated connection object.

`String desc = document.getDescription();`
Gets any description comment associated with the Dynamo document.

`String arg = document.getArgument( varName );`
Gets the value of `DynamoDocument.value[varName]`. This is usually an argument from an HTML form.

`document.setArgument( varName, value );`
Sets the value of `DynamoDocument.value[varName]`.

`String val = document.getValue( varName );`
Gets the value of `DynamoDocument."varName"`.

`document.setValue( varName, value );`
Sets the value of `DynamoDocument."varName"`. This creates the variable if it doesn't already exist.

For further information on any of these methods, see the *PowerJ Component Library Reference* and the NetImpact Dynamo documentation.

Sending HTML code to the user

The most important methods of DynamoDocument are **write** and **writeLn**. These both write a string to the HTML document. The difference is that **writeLn** automatically adds a new-line character at the end of the string while **write** does not. For example, several **write** operations in a row all write to the same line, while several **writeLn** operations in a row each write to a new line.

The following shows a simple example of **writeln**:

```
// DynamoConnection server;
DynamoDocument document = server.getDynamoDocument();
```

```
document.writeLn( "<P>Hello there." );
```

This writes the given string to the HTML document. Notice that this string contains the <P> HTML formatting directive. The HTML file that is created through such calls to **write** and **writeLn** will eventually be sent to the User system.

For more information about the DynamoDocument class, see the *PowerJ Component Library Reference* and the documentation on NetImpact Dynamo.

The DBConnection class

The DBConnection class provides an interface to the Dynamo connection object, for a single request from the NetImpact Dynamo web server. This allows access to the current connection for the current document. The current document corresponds to the template that originated this request.

Information about the connection

The DBConnection class provides a number of methods that provide information about the database connection:

```
String connName = dbconn.getName();
```
Gets the name of this connection object.

```
String desc = dbconn.getDescription();
```
Gets the description entry for this connection object.

```
String ds = dbconn.getDataSource();
```
Gets the name of the data source for this connection.

```
String user = dbconn.getUserid();
```
Gets the userid being used in this connection.

Creating Dynamo queries

The **createDynamoQuery** method of DBConnection creates a DynamoQuery object with a given SQL statement. For example:

```
// DynamoConnection server;
DBConnection connection = server.getDBConnection();
DynamoQuery dq =
    connection.createDynamoQuery(
        "select id, lname from customer" );
```

This creates a PowerJ DynamoQuery object that has the given SQL statement as its **SQL** property. You can then perform normal methods on the Query object to examine the results obtained from the database.

The DynamoQuery class

The DynamoQuery class provides an interface to a NetImpact Dynamo DynamoQuery object.

Note: The usual way of creating a DynamoQuery object is to use the **createDynamoQuery** method of DBConnection. For more information, see The DBConnection class.

DynamoQuery properties

The properties of a DynamoQuery object are available through **get** methods:

`boolean valid = dynQuery.getValid();`
Determines whether the DynamoQuery object is valid.

`String name = dynQuery.getName();`
Gets the name of the Dynamo query object.

`String name = dynQuery.getColumnLabel( colNumber );`
Gets the name associated with the given column in the result set. Columns are numbered beginning at 1.

`int num = dynQuery.getColumnCount();`
Gets the total number of columns in the result set.

`boolean empty = dynQuery.getEmpty();`
Determines whether the result set is empty (`true` indicates that it is empty).

`int count = dynQuery.getRowCount();`
Gets the number of rows in the result set.

`int code = dynQuery.getErrorCode();`
Gets the current ODBC error code associated with the Dynamo query object.

`String info = dynQuery.getErrorInfo();`
Gets the current ODBC error information for this Dynamo query object.

`int state = dynQuery.getState();`
Gets the current ODBC state of the Dynamo query object.

Obtaining a result set

The following methods are related to obtaining a result set from a query:

`dynQuery.setSQL( sqlStatement );`
Assigns a new SQL statement to this query. The argument is a string giving the SQL statement itself.

`boolean success = dynQuery.execute();`
Executes the SQL statement currently associated with the query. This obtains the result set.

Moving in the result set

The following methods change the current position in the result set:

`dynQuery.move( rowNumber );`
Moves to the row with the specified number.

`dynQuery.moveFirst();`
Moves to the first row in the result set.

`dynQuery.moveLast();`
Moves to the last row in the result set.

`dynQuery.moveNext();`
Moves to the next row in the result set.

`dynQuery.movePrevious();`
Moves to the previous row in the result set.

`dynQuery.moveRelative( numRows );`
Moves the given number of rows. Positive values move forward and negative values move backward.

Obtaining values from the result set

The following methods obtain values from a specified column in the current row of the result set:

`String val = dynQuery.getString( colNumber );`
Gets the value from the specified column of the current row in the result set. The value is obtained as a string.

`boolean val = dynQuery.getBoolean( colNumber );`
Gets the value from the specified column of the current row in the result set. The value is obtained as a boolean value.

`int val = dynQuery.getInt( colNumber );`
Gets the value from the specified column of the current row in the result set. The value is obtained as an integer.

`double val = dynQuery.getDouble( colNumber );`
Gets the value from the specified column of the current row in the result set. The value is obtained as a double value.

`Object val = dynQuery.getObject( colNumber );`
Gets the value from the specified column of the current row in the result set. The value is obtained as an Object object.

Scheduled Dynamo scripts and templates

PowerJ provides support for scheduled Dynamo scripts (`.sss` files) and templates (`.sts` files) by treating them as if they were HTML files. For example, when you open one of these files, PowerJ opens an HTML editor to edit the file's contents.

Chapter 25. Distributed computing

This chapter describes how to use PowerJ to make distributed computing applications using either the Remote Method Invocation (RMI) standard or the Common Object Request Broker Architecture (CORBA) standard.

RMI

The Remote Method Invocation (RMI) standard, developed by Sun for Java, enables you to create distributed applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, which may be on different hosts.

A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. The RMI compiler (`rmic`) builds client and server stubs for the Java classes that you want to use remotely.

For more information on RMI, see:

`http://java.sun.com/products/jdk/1.1/docs/guide/rmi`

To use RMI with PowerJ, you need to create all the class files in a single folder (via the **Output Directory** page of the Java target property sheet) and then, outside of PowerJ, you need to run the `rmic` compiler (supplied with the JDK included with PowerJ) on those class files. Other than needing to set the output folder for your class files, you can follow the tutorial directions given in:

`http://java.sun.com/products/jdk/1.1/docs/guide/rmi/getstart.doc.html`

CORBA

The Common Object Request Broker Architecture (CORBA) was established by the Object Management Group (OMG) to define platform-independent ways for objects to make requests and receive responses. For an index of information about CORBA, see:

`http://www.acl.lanl.gov/CORBA/`

Interface Definition Language (IDL)

The CORBA Interface Definition Language (IDL) provides an implementation-independent way of defining the interface to an object. For example, you might specify the interface to an object in IDL, but implement the object itself in C++ on a particular platform. On another platform, you might implement the same object in Java. An application using the object could access it on either platform by making use of the interface definition written in IDL.

In PowerJ, you may add IDL files to any Java target. You may also open an IDL file to edit it. In this case, PowerJ opens up a code editor window that is adapted to recognize IDL. For example, IDL reserved words are marked in a special color to distinguish them from other syntactic elements.

When you build the Java target, PowerJ can process IDL files by compiling them into Java (`.java` files). In particular, PowerJ supports the Visigenic and Iona compilers. These compilers are not included in the PowerJ package, but PowerJ has been tested for compatibility with them.

If you have a different IDL-to-Java compiler installed on your system, you may use that compiler instead. To do so, you must change the Java target's property sheets.

- **To change IDL compilers:**

1. Open the Targets window.
2. In the **View** menu of the Targets window, click **Show Property Sheet** so that it is marked with a check.
3. In the left part of the targets window, click on the target whose IDL compiler you wish to change.
4. In the property sheet on the right part of the Targets window, click the **IDL Compiler** tab.
5. Set options on the **IDL Compiler** page, then click **Apply**.

The compilation process creates a number of `.java` code files for each original IDL. These `.java` files are considered source files for the Java application being built, and are compiled like any other Java source files.

VisiBroker and OrbixWeb

PowerJ will attempt to detect the presence of VisiBroker by Visigenic and OrbixWeb by Iona on your machine. The detection process is based on environment variables.

- Installing VisiBroker creates an environment variable named `ORBELINE`. The value of this variable is the name of a configuration folder. PowerJ uses the name of this folder to derive the locations of the rest of the installed files.

Note: For VisiBroker for Java version 2.5, you must manually set the environment variable to the `ADM` folder under your VisiBroker folder. PowerJ relies on the `ORBELINE` setting to locate VisiBroker.

- Installing OrbixWeb creates an environment variable named `IT_CONFIG_PATH`. The value of this variable is the name of the root folder where OrbixWeb is installed. This value makes it possible to derive the locations of the rest of the installed files.

Note: The PowerJ sample projects include VisiBroker and OrbixWeb applications, showing a variety of programming techniques. There are VisiBroker samples for two versions (1.2 and 2.5) of VisiBroker for Java.

VisiChannel and VisiBroker

If you have installed the VisiChannel JDBC driver that is an option in the PowerJ installation, you need to insure that your CORBA program uses the VisiBroker class files instead of the VisiChannel class files.

This means that you need to configure your projects which use VisiBroker for Java so that the VisiBroker class files appear in the class path before the VisiChannel class files. (The VisiChannel class files are embedded in the USER_CLASSES system build macro.) For information on setting the PowerJ class path, see How PowerJ uses the CLASSPATH environment variable.

Creating CORBA applications

The typical way to create a CORBA project is to begin by creating an applet, add or create an IDL file, then fill in the rest of the implementation as required.

- For a server object, this means fleshing out the skeleton code that gets produced by the IDL-to-Java compiler.
- For a client object, this means using the stub functions that get produced by the IDL-to-Java compiler.

Each vendor's product requires that the Implementation Repository be started before server objects may be started.

Once the Implementation Repository has been started, you may start the PowerJ project for the server object applet. This applet can be run and debugged from within PowerJ.

Note: With OrbixWeb, you must invoke the Implementation Repository Daemon (`orbixd.exe`) with the `-u` switch to allow for unregistered Implementations to be placed in the Repository.

Once the server object applet is running, you can start another instance of PowerJ and use it to load the client applet project. The client applet can be run and debugged from within this second instance of PowerJ.

Chapter 26. PowerJ and Jaguar CTS

This chapter describes how you can use PowerJ to write programs that work with Powersoft Jaguar CTS. A development version of Powersoft Jaguar CTS for Windows NT is included with PowerJ.

The main source of information about Jaguar CTS is the documentation accompanying Jaguar CTS. This document only discusses the simplest details of accessing Jaguar functionality through PowerJ.

In the rest of this chapter, "Jaguar" may be used as an abbreviation for "Jaguar CTS".

Jaguar fundamentals

Jaguar CTS is a *component transaction server*, aimed at supplying data and business logic to client programs. The computer where Jaguar runs is called the server system; the computer where the client program runs is called the user system.

For example, an Internet user might use a web browser to open a web page obtained from any system (usually the server system, but this is not mandatory). The web page contains a Java applet which runs under the web browser on the user system. In response to user actions, the applet may invoke the Jaguar server on the server system and have it run a server component (again on the server system). The server component uses Jaguar services to access and interact with a database, reading and/or writing to the database in accordance with information received from the applet on the user system. Finally, the server component may transmit data to the applet via the Jaguar server, thereby providing the user with immediate results.

Jaguar itself is a web server, just like the Microsoft and Netscape servers. Typically, companies will use Jaguar in addition to a standard server like Microsoft or Netscape; however, Jaguar does support a subset of HTTP and can serve as a standalone server for some applications.

Jaguar provides a connection manager for JDBC, providing efficient access to databases. A server component can therefore deal with the connection manager and JDBC to perform operations on registered data sources.

At present, Jaguar has been tested with the JDBC-ODBC bridge and with jConnect. Support for other JDBC drivers has not yet been tested.

Note: In order to develop and test applications that use Jaguar CTS, you should have Jaguar installed on your development system. You can do some Jaguar development without having Jaguar installed; for more information, see Developing for Jaguar on Windows 95 You must also have the Jaguar server installed on the computer that you will use as your server system during the testing phase. (This may be the same machine as the your development system.)

JDK versions and Jaguar

The server-side software (the server component) must use JDK 1.1. The user-side software (the applet) may use JDK 1.02 or JDK 1.1.

Jaguar software

There are two main pieces of software associated with Jaguar CTS:

- The Jaguar Server—this receives requests from user applets, then accesses databases (or other data sources) in response to those requests. The server also sends database access results back to the user applet.
- The Jaguar Manager—this lets you specify the data sources that Jaguar can access and the server components that Jaguar can execute in response to user requests.

Connection caches

A *connection cache* is a database or comparable data source that has been registered with the Jaguar server. Registering a connection cache provides Jaguar with the information it needs in order to access the cache. For example, when you register a database as a connection cache, you specify information that lets Jaguar find and access the database.

- **To register a connection cache:**

1. Start the Jaguar Server and Jaguar Manager on the server machine.
2. In the Jaguar Manager, expand the entry for **Servers** and then the entry for **Jaguar** until you see an entry for **Connection Caches**. Click on **Connection Caches**.
3. In the **File** menu, click **Install Connection Cache**. This opens a dialog to register a new connection cache.
4. Click **Create and Install a New Connection Cache**.
5. Type a name for the new cache, then click **OK**. This opens a property sheet for the new cache.
6. Use the property sheet to specify information about the connection cache. For example, if the cache is a database, you would specify information about the driver that you will use to access the database and the URL that represents the database. Click **OK** when you have finished entering this information.

The information specified when defining a cache is similar to the information specified when setting up a Transaction object in PowerJ. For example, if you are connecting to the sample SQL Anywhere database using jConnect, you might specify information of the form:

Server name: `jdbc:sybase:Tds:localhost:7373`
User name: `dba`
Password: `sql`
DLL or Class Name: `com.sybase.jdbc.SybDriver`

As shown, the server name is the URL to access the database and the DLL or Class Name is the Java class that represents the JDBC driver.

Note: The Jaguar Manager is case-sensitive, so you must type alphabetic characters in the desired case. For example, the name that you give to the new cache is case-sensitive; applications that want to access that cache must specify the name using the correct case.

For further information about using the Jaguar Manager, see the documentation for Jaguar CTS.

Developing Jaguar server components

This section discusses details of developing server components for use with the Jaguar server.

Note: Jaguar server components must use JDK 1.1.

How Jaguar server components work

A Jaguar server component is represented by two Java classes:

- A user-side class that allows applets to access methods in the server component. This class is called the *stub*.
- A server-side class whose methods do the real work. This is called the *implementation*.

For example, a Jaguar server component might be represented with a stub class named MyServComp and an implementation class called MyServCompImpl. This is the usual naming convention for Jaguar server components—the Impl on the end of the implementation class name indicates that it is the implementation of the corresponding stub.

When an applet wants to invoke the server component, it connects with the Jaguar server and creates an object of the stub class. For example, the applet might use the following code to connect with Jaguar and to create an object of the stub class.

```
String host = "localhost";
String port = "7878";
String userid = "dba";
String password = "sql";
String stubpackage = "MyServCompPkgStubs";
String serverpackage = "MyServCompPkg";
String component = "MyServComp";

// bring variables together into a single URL string
String url = new String("ejb:jdbc102//" + host + ":"
    + port + ":" + userid + ":" + password + ":"
    + stubpackage + "/" + serverpackage + "/" + component);

// connect and create stub class object
MyServComp js = (MyServComp)
    CommunicationManager.createInstance(url,null);
```

Creating this object automatically submits a request to the Jaguar server to create a corresponding object of the MyServCompImpl class on the server system.

The applet can then execute methods on the stub object, as in:

```
js.generateCustomerList();
```

Executing a method on the stub object submits a request to the Jaguar server. This request typically executes a method of the same name on the corresponding implementation object.

The implementation method may read data from a database driver and deliver that data to the Jaguar server. The Jaguar server then delivers the data to the applet as an in-out parameter or ResultSet object that can be obtained using an appropriate method on the stub. In a similar way, the applet might use its stub object to submit requests that modify the contents of the database.

From the applet's point of view, these methods are being executed on a stub object that exists within the applet itself (on the user system). This object transmits service requests to the server system via the Jaguar server, and a corresponding method is executed on the implementation object. The implementation method typically transmits data back to the user system; this data is returned as the result of the stub method that the applet executed.

Jaguar server and JavaBeans components

If you develop a Jaguar server component with PowerJ, the server component class will have the basic structure of a JavaBeans component (without event handling).

However, there is more to making a Bean than just the underlying structure. For example, the methods that you define in the server component should follow the naming conventions for JavaBeans components. Therefore, if you want your server component to be a true Bean, your code must conform to the conventions. For more information on creating JavaBeans components, see Creating JavaBeans Components.

Note: PowerJ makes it easy to create server components that are JavaBeans components. To do this, however, the programmer must follow a few naming and coding conventions to conform with the JavaBeans component guidelines.

If you have an existing JavaBeans component and want to use it as a Jaguar server component, you typically have to create a *wrapper class* for the Bean.

For example, suppose the Bean you want to use creates a ResultSet object. The Jaguar server component might be an implementation class that invokes the Bean, obtains the ResultSet, and delivers that ResultSet to the Jaguar server in the manner that the server expects. Typical methods in the wrapper class would be very small—just invoking the corresponding Bean method, then delivering the result to the Jaguar server. Of course, the wrapper class could be written to be a Bean itself, so that the whole server component was a Bean.

In summary

If you are writing your own Jaguar server component, you can make sure that the server component is a Bean provided you make sure your code conforms with the JavaBeans component guidelines. If you want to use an existing JavaBeans component as a Jaguar server component, you typically have to create a simple wrapper class to act as an intermediary between Jaguar and the Bean.

Implementing the server component classes

In PowerJ, you typically begin by creating the implementation class as a **Jaguar Server Component** target. By convention, the name of the implementation class should be the name of the stub class plus Impl.

The code for your implementation class will probably access a number of features of the Jaguar server. For more information on techniques for using the Jaguar server, see Interfacing with the Jaguar server.

Once you have created the implementation class, you can get the Jaguar Manager to generate the stub class automatically. For more information, see Generating the stub class.

Registering an implementation class

To make it possible for the Jaguar server to run a particular server component, you must register the implementation class through the Jaguar Manager. Once the class has been registered, the Jaguar server will be able to run methods from the implementation class in response to requests from user-system applets. The Jaguar Manager will also be able to generate a stub class automatically.

Server component classes are associated with Jaguar packages. A Jaguar package is a normal Java package containing classes that have been registered with Jaguar. For example, you might create a Jaguar package named `myservcomp` to contain the MyServComp stub class and the MyServCompImpl implementation class.

Note: For an explanation of how to associate a PowerJ target with a particular Java package, see Specifying package names.

- **To register a package with the Jaguar Manager:**

1. Start the Jaguar Server and the Jaguar Manager.
2. In the Jaguar Manager, expand the entry for **Servers** and the entry for **Jaguar** until you see the entry for **Packages**. Click **Packages**.
3. In the **File** menu, click **Install Package**.
4. Click **Create and Install a New Package**. This opens the New Package dialog box.
5. In the New Package dialog box, type a name for the package you will be installing.
6. Click **OK**.

The name of the new package should appear in the **Packages** list under **Jaguar** in the Jaguar Manager.

Once you have created a package, you can register server component classes under it. Typically, you would create the implementation class first using PowerJ, then *import* the existing class into the Jaguar Manager. When you do this, the Jaguar Manager automatically registers all the methods in the class as well as the class itself.

- **To register server component classes under a package:**

1. In the Jaguar Manager, click the name of the package where the new class will be registered.
2. In the **File** menu, click **Install Component**. This opens a dialog box to let you install a Java class component in the package you have just created.
3. Click **Create and Install a New Component**. This opens a New Component dialog box.
4. Type the name of the stub class, then click **OK**. (Remember that the names of Java classes are case-sensitive.)
5. In the Create Component dialog box, click **Import Java Class** so that it is checked, then click **OK**. This opens a property sheet where you can enter information about the class.
6. Next you need to enter the name of the class, including the package prefix. This Java class must be in the class path that is being used by the Jaguar Server.
7. Use the property sheet to specify the name of the stub and its implementation class. You also specify the name of the Jaguar package that should contain these classes; typically the stub and its implementation should go into the same package. Click **OK** when you have finished entering information about the classes.

The Jaguar Manager will examine an existing class when you register it and automatically register all the public methods defined in that class. This makes it possible for a stub in an applet to execute methods from the implementation.

Finding the server component classes

When you register server component classes with the Jaguar Manager, the manager does *not* record the full pathnames of the `.class` files. Therefore, the pathnames are not "hard-coded" into Jaguar.

The Jaguar server always attempts to find `.class` files relative to the current `CLASSPATH` settings on the server system. These settings are given by an environment variable named `CLASSPATH`. The value of this variable is a list of folder names where the server should search for desired `.class` files.

For example, suppose that the Jaguar server needs to create a MyServCompImpl object. The server obtains the current value of `CLASSPATH`, then looks for a file named `MyServCompImpl.class` under each folder listed in the `CLASSPATH` value. The server uses the class definition from the first such file found.

Important: It is crucial to make sure that the server system's `CLASSPATH` list includes the names of all folders that contain registered server component classes. If the Jaguar server cannot locate the necessary `.class` file in one of the `CLASSPATH` folders, it will be unable to execute the server component. (The server will therefore throw an exception.)

Instead of adding a new folder name to `CLASSPATH` for each server component, we recommend the following:

1. Use a single folder as the root for storing the class files for all your Jaguar server components. To avoid having to change the `CLASSPATH` on for the Jaguar server, you should use the following folder under the folder where you installed Jaguar CTS:

   ```
   HTML\Classes
   ```

2. Make certain that you are using a package for your classes. The package name will determine which subfolders will be used for your classes under `HTML\Classes`.
3. For each Jaguar server component, create a WebApplication target that publishes all the server component's `.class` files to `HTML\Classes`. This is a simple way to copy all the necessary files to the folder. For more about WebApplication targets, see WebApplication targets.
4. In the **Applet Classes** page of the property sheet for your WebApplication target, click **Add classes, excluding the following packages** and add the following package to the list:

   ```
   com.sybase
   ```

   This will exclude the Jaguar server classes from being added to your WebApplication target.

Now you can build the WebApplication target to build your class files and copy them to a location where Jaguar can find them.

Generating the stub class

In the process of registering a server component, you specify the names of the stub class and its implementation. At the time you register the server component, however, the stub does not exist; only the implementation exists. After you register the server component, you can instruct the Jaguar Manager to generate code for the stub class, based on the implementation class.

- **To generate a stub class:**

1. Make sure that the current `CLASSPATH` list includes the folder that contains the implementation `.class` file.
2. In the Jaguar Manager, click the entry for the server component. (This entry is under the package name, under **Packages**.)
3. In the **File** menu, click **Generate Stub/Skeleton**. This opens the Generate Stub & Skeletons dialog box.
4. In the dialog box under **Code Base**, specify the folder where the `.java` file for the stub should be placed. You also need to specify whether the client needs JDK 1.02 or JDK 1.1.

   This dialog also lets you generate skeleton code for implementing the server component, but this is not generally needed with Java development.
5. Click **Generate**.

The Jaguar Manager generates a definition for the stub class and stores that definition in a `.java` file under the specified folder. If you look at the code for this class, you will see that the stub's methods submit requests to the Jaguar server, asking the server to execute corresponding methods from the implementation class.

To use the stub class in a PowerJ target, you would import the class's `.java` file into the target. For information about importing classes, see Chapter 7 of the *PowerJ Programmer's Guide*.

Changing a method's calling sequence

The calling sequence of a function consists of the data types and order of arguments passed to the function, and the type of value that the function returns.

When you register a server component with the Jaguar manager, the manager records the calling sequence for every method defined in the component. If you change the calling sequence for one or more methods (for example, if you add a new argument to a method or change the data type of an existing argument), you must update the Jaguar manager's information about the methods that are affected.

- **To update the Jaguar manager's information about method calling sequences:**

1. Open the Jaguar manager and expand the list of registered components to find the component that has changed.
2. Use the right mouse button to click the name of the component that has changed, then click **Delete**. This deletes the old information about the component.
3. Follow the instructions in Registering an implementation class to register the new version of the component.
4. Follow the instructions in Generating the stub class to generate a new stub class for the component.

Any client applications that use the old stub class must be updated to use the new stub class.

Preserving session data

You may wish to design an application where the user interacts with a Jaguar server component, goes to a different (non-Jaguar) HTML page, then returns to Jaguar again. In this case, it would be useful to preserve data from the first interaction and make it available when the user comes back to Jaguar.

One way to do this is to use the facilities of the Java VM (virtual machine) that is running the application. When you create a Jaguar instance, you should make that instance available globally to the Java VM running in the browser. This is usually done by storing it in a static (or *class*) member variable, and writing static methods to set and get this member variable:

```
public class MyClass
{
   private static JaguarObject _jagobject;
   public static void setJaguarObject( JaguarObject jagobject )
   {
       _jagobject = jagobject;
   }
   public static JaguarObject getJaguarObject( void )
   {
     return _jagobject;
   }
   ...
}
```

Then, you can just retrieve the existing instance when you return to your page, or from any other applet running in the same browser instance.

Interfacing with the Jaguar server

This section discusses techniques that the stub and implementation classes can use in order to make use of Jaguar services.

Jaguar packages

There are several Java packages of interest to developers writing applications for use with Jaguar. These are discussed below:

`com.sybase.jaguar.beans.enterprise`
: Classes and interfaces used to implement Java components and to create stubs for remote communication. These classes are based on an early draft of the JavaSoft Enterprise Java Bean (EJB) specification. Future releases of the JDK will likely provide classes with the same functionality.

`com.sybase.jaguar.jcm`
: Classes and interfaces for managing cached JDBC connections in server-side Java code. "JCM" stands for Jaguar Connection Manager.

`com.sybase.jaguar.server`
: Utility classes used in server-side Java code.

`com.sybase.jaguar.sql`
: Interfaces for objects that construct and send row results from a Java server component to the client.

`com.sybase.jaguar.stub.jdbc102`
: Contains many internal classes used by generated Jaguar stub code, and JJDBCStub, which is the base class for generated stubs. Code that will run in a Java 1.02 virtual machine should import this package.

`com.sybase.jaguar.stub.jdbc11`
: Contains many internal classes used by generated Jaguar stub code, and JJDBCStub, which is the base class for generated stubs. Code that will run in a Java 1.1 (or later) virtual machine should import this package.

`com.sybase.jaguar.util`
: Utility classes that are used in both server-side and client-side Java code.

`com.sybase.jaguar.util.jdbc102`
: Holder classes for use in code that will run in a Java 1.02 virtual machine.

`com.sybase.jaguar.util.jdbc11`
: Holder classes for use in code that will run in a Java 1.1 (or later) virtual machine.

Note: Most Java classes related to Jaguar have names beginning with the letter J (for example, JException).

Working with the Jaguar server

This section describes a typical sequence of instructions that a JDK 1.1 implementation class can use to perform a database query and return the results to an applet. In order to execute this code, the implementation class should contain the following import statements:

```
import com.sybase.jaguar.jcm.*;
import com.sybase.jaguar.server.*;
import com.sybase.jaguar.util.*;
import java.sql.*;
```

The first step in accessing a database is to obtain an object that represents the database. You obtain this object from the Jaguar Connection Manager (JCM) using a statement of the form:

```
// String userName, password, url;
JCMCache cache = JCM.getCache( userName, password, url );
```

The user name, password and URL strings must refer to a connection cache that has already been registered with Jaguar. For example, if you wanted to use jConnect to connect with the sample SQL Anywhere database through the Open Server Gateway, you could write:

```
JCMCache cache = JCM.getCache( "dba", "sql",
     "jdbc:sybase:Tds:localhost:7373" );
```

The result of this statement is a JCMCache object that can be used to refer to the database in future operations.

The next step is to make a connection to the database itself. This uses the **getConnection** method of the JCMCache object, as in:

```
Connection conn = cache.getConnection( JCMCache.JCM_WAIT );
```

For information on possible arguments to **getConnection**, see the Jaguar documentation.

The result of **getConnection** is a JDBC Connection object, representing an active database connection. It serves much the same function as a PowerJ Transaction object.

Once you have established a connection, you can execute an SQL query on the database:

```
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery( "SELECT ..." );
```

The **createStatement** method creates a SQL statement that is associated with the existing database connection. The **executeQuery** method then executes a `SELECT` statement on the JDBC Statement object; this has the result of reading data from the database and returning it as a JDBC ResultSet object.

After obtaining the result set, you must deliver it to the Jaguar server. This makes it possible for the server to transmit the data to the user applet:

```
JContext.forwardResultSet( rs );
```

Once the server component has forwarded the result set to the Jaguar server, the server component has fulfilled its purpose. Therefore, the only work left is to disconnect from the database:

```
cache.releaseConnection( connection );
```

Exceptions

Jaguar throws a JException (`com.sybase.jaguar.util.JException`) to indicate that a problem has occurred. For example, the **getConnection** method of JCMCache throws this exception if a

connection cannot be established. Therefore, the code discussed above should be enclosed in a `try/catch` block that catches JException.

JDBC methods throw an SQLException when an error occurs. For example, the **executeQuery** method of Statement throws this exception if the query cannot be executed. Therefore, the code discussed above should also be able to catch SQLException.

Writing an applet to interact with a Jaguar server component

To begin writing a client applet that connects with a Jaguar server component, you use PowerJ to create a **Jaguar Client Applet** target (JDK 1.1) or a **Jaguar Client Applet 1.02** target (JDK 1.02).

In order for an applet to interact with a Jaguar server component, it must start by connecting with Jaguar. Here is a typical example:

```
// import com.sybase.jaguar.stub.*;
String host = "localhost";
String port = "7878";
String userid = "dba";
String password = "sql";
String stubpackage = "MyServCompPkgStubs";
String serverpackage = "MyServCompPkg";
String component = "MyServComp";

// bring variables together into a single URL string
String url = new String("ejb:jdbc102//" + host + ":"
    + port + ":" + userid + ":" + password + ":"
    + stubpackage + "/" + serverpackage + "/" + component);

// connect and create stub class object
MyServComp js = (MyServComp)
    CommunicationManager.createInstance(url,null);
```

This specifies user name, password, and other information for a server component. This code connects to Jaguar and instantiates the server component as well as creating an object of the stub class.

Next, the applet creates an object of the stub class. Suppose that the JCustomerList class is a server component which has been registered with Jaguar and which obtains customer information from a database. The applet might execute the following code:

```
// import package containing JCustomerList definition;
JCustomerList customer = new JCustomerList();
customer.getCustomerInfo();
```

Executing the **getCustomerInfo** method on the stub objects sends a request to the Jaguar server. The Jaguar server executes a method of the same name on a corresponding implementation object. Presumably, the implementation performs a JDBC database query and forwards the result set to the Jaguar server using **forwardResultSet** (as discussed in the previous section).

The applet can obtain the result set by using **getResultSet** on the stub object:

```
ResultSet rs = customer.getResultSet();
```

The applet can assign the result set to a Query object, then perform all subsequent actions on the Query, as in:

```
query_1.setResultSetObject( rs );
query_1.moveNext( true, true );
```

The process of assigning a result set to a query is discussed in the *PowerJ Programmer's Guide*.

Exceptions in the applet

As with operations in the server component, the Jaguar-related methods used by an applet may raise exceptions. Therefore the code must be enclosed in a `try/catch` block, as shown below:

```
try {
    String url = "ejb:jdbc102//localhost:7878:dba:sql:"
        + "JCustListPkgStubs/JCustListPkg/JCustomerList";
    JCustomerList customer = (JCustomerList)
        CommunicationManager.createInstance(url,null);
    customer.getCustInfo();
    ResultSet rs = customer.getResultSet();
    query_1.setResultSetObject( rs );
    query_1.moveNext( true, true );
} catch (JException je) {
    System.out.println("Jaguar Exception caught!");
    je.printStackTrace();
} catch (Exception e) {
    System.out.println("Exception caught!");
    e.printStackTrace();
}
```

PowerJ-Jaguar sample programs

The PowerJ package provides several sample programs that interact with Jaguar CTS.

JaguarBeanDemo
: This project shows how to create a wrapper around a JavaBeans component. The component itself returns a `double` value.

JaguarDatabaseDemo
: This project uses a Jaguar server component to make a query on a database. The user specifies query information in an applet. The applet forwards this information to a server component via the Jaguar server, and the server component obtains a ResultSet from a JDBC database (the SQL Anywhere Sample database). Finally, the server component returns the ResultSet to the applet, which displays the result in a bound control.

The above projects can be found in the `JaguarSamples` folder under the `PowerJ Samples` folder. Each of the projects has a file named `ReadMe.txt` which explains how to run the associated application (including instructions on how to register server components with the Jaguar Manager).

Developing for Jaguar on Windows 95

You cannot use Jaguar CTS under Windows 95. Even so, you can still use PowerJ on a Windows 95 system when you are developing projects for Jaguar. Here's one simple approach:

1. Develop the server-side implementation class using PowerJ. This can be done under either Windows NT or Windows 95.
2. Use the Jaguar Manager to generate the stub class from the implementation class. This is most easily done on Windows NT; therefore, if you developed the implementation on a Windows 95 system, you must copy the implementation `.class` file and other relevant files to a Windows NT system where the Jaguar Manager is installed.
3. Copy the `.class` files for the stub class and other relevant classes to a Windows 95 system.
4. Develop client applications using PowerJ on the Windows 95 system. These applications can use methods in the stub class to access services on the server side. When you test such applications, they must contact a system that is actually running Jaguar (for example, the original Windows NT system where you developed the implementation class).

When you ask PowerJ to create a Jaguar client applet target, the build options are set up to find Jaguar-related classes in the folder:

```
$(%jaguar)\HTML\Classes
```

The construct `$(%jaguar)` refers to the value of an environment variable named `jaguar`. If your system has Jaguar installed, this environment variable is automatically initialized to point to the folder where you installed Jaguar (for example, `c:\Powersoft\JaguarCTS`). However, if you are creating a client applet on a Windows 95 system (where Jaguar can't be installed), you must do the following:

1. Create a subfolder named `HTML\Classes` under an appropriate folder (for example, `c:\Powersoft`).
2. Create an environment variable named `jaguar` whose value is the name of the folder that contains `HTML\Classes`.
3. Copy the `.class` files for the stub and any other relevant classes to the `HTML\Classes` folder. Be sure to preserve the directory structure from the original machine. For example, if the stub `.class` file is

   ```
   c:\Jaguar\HTML\Classes\MyPkg\MyClass.class
   ```

   on the original machine, copy it to

   ```
   $(%jaguar)\HTML\Classes\MyPkg\MyClass.class
   ```

   on the Windows 95 system.
4. Copy the `.class` files for Jaguar objects to the `HTML\Classes` folder. For example, your client applet will probably need `.class` files from the class:

   ```
   com.sybase.jaguar.stub
   ```

   Copy all necessary files from the original machine to the Windows 95 system. Again, you must preserve the directory structure from the original machine.

Note: The `jaguar` environment variable does not have to specify the name of a folder on your own machine. For example, if Jaguar is installed on another machine in your local network, you can set the `jaguar` variable to point to the Jaguar installation folder on that other machine. In particular, you could

generate the stub's `.class` file on a Windows NT machine, then develop client applets with a Windows 95 machine on the same network. In this case, you do not have to copy files from the original system to the Windows 95 machine; you just have to set the Windows 95 `jaguar` variable to refer to the appropriate folder on the Windows NT machine.

Debugging Jaguar server components

You can use the standard PowerJ debugger to debug Jaguar server components. For more information about PowerJ debugging facilities, see Debugging.

At present, debugging only works if you are using the Sun VM to run Java code on the server system. Idiosyncrasies of the Sun VM make the debugging process more complicated than debugging other types of PowerJ applications. The sections that follow give a step-by-step explanation of what you have to do in a typical debugging situation.

Starting the debugging server

In order to debug a Jaguar server component, you must use the debugging version of the Jaguar server.

Important: The normal Jaguar server does not provide debugging services. If you already have the normal Jaguar server running on your test machine, you should terminate it.

To start the debugging version of the Jaguar server, click the **Jaguar Server (dev)** icon in the Jaguar folder, or open a command window and execute the file

```
devbin\serverstart_dev.bat
```

found under your main Jaguar folder.

When the server begins executing, it starts the Sun VM. The VM displays some copyright information and then a line of the form

```
Agent password=xxxxxx
```

where `xxxxxx` is typically five or six alphanumeric characters. Copy this password onto a piece of paper, or copy it into the Windows system scratch pad. You will need to use the password later.

Note: The agent password is different each time the Sun VM begins execution. Therefore, you will have to copy the new password each time you start the debugging version of the Jaguar server.

Preparing the server component

The next step is to set up the server component so that it runs with the Jaguar server.

Begin by starting PowerJ and opening the project that contains the server component. This component must be a JDK 1.1 **Java Classes** target or a **Jaguar Server Component** target. Once you have loaded the project containing the server component target, you may set breakpoints in the target's code.

Next, open the run options for the server component target (for example, by clicking **Run Options** in the main PowerJ **Run** menu). On the **Remote Debug** page of the run options, type in the name of machine where the Jaguar server is running (for example, `xyz.com`). In the field for the agent password, fill in the password that you copied when the Sun VM started up. Close the run options dialog after you have entered this information.

When you have finished the above steps, build the server component target (for example, by clicking **Build** in the main PowerJ **Run** menu).

Note: Do not quit this PowerJ session when you have finished building the server component target. Later on, this PowerJ session will display debugging information about the server component.

Publishing the class files

The next step is to publish the server component's class files to the server system. The easiest way to do this is to create a WebApplication target that performs the publishing action. For more information about WebApplication targets, see WebApplication targets.

Manually or using a WebApplication target, you should copy the class file for the server component to an appropriate location in the Jaguar server's `CLASSPATH`, such as Jaguar's `HTML\Classes` folder. This is discussed in Finding the server component classes. You should also copy any other class files needed by the server component.

For example, suppose that the server component is named MyServComp. You must obtain `MyServComp.class` from the location where PowerJ built the server component target, and copy the file to an appropriate location in the Jaguar server's `CLASSPATH`. Similarly, if MyServComp makes use of a class named MyOtherClass, you should copy `MyOtherClass.class` to an appropriate location in the Jaguar server's `CLASSPATH`.

If you have not already done so, you should register the server component class with the Jaguar server. For an explanation of how to do this, see Registering an implementation class.

Running the server component

Once you have registered the server component with Jaguar and published all the necessary files, you should run the server component. The easiest way to do this is to use the same WebApplication target that publishes the class files: set up the WebApplication so that it runs the server component after publishing the files. You can also run the server component target directly, by setting appropriate run options and then clicking **Run** in the PowerJ **Run** menu.

Running the client applet

You have now started the debugging version of the Jaguar server, built the server component, and copied the necessary files to a location where the Jaguar server can find them. You are therefore ready to run a client applet.

You can start the client applet in a number of ways:

- Start a second copy of PowerJ, load the project containing the applet, and run the applet.
- Start a web browser and open a URL containing the applet.
- Write yourself a command file (`.bat`) that invokes an applet viewer which opens the applet.

The applet should interact with the server component using stub class methods, as discussed in Interfacing with the Jaguar server.

Debugging the server component

When the applet connects with the Jaguar server, the server will invoke the server component using the `.class` files that you placed in the server's `CLASSPATH`. If you previously set one or more breakpoints in the server component's code, those breakpoints will be triggered in the usual way (when execution reaches a point where a breakpoint has been set).

Debugging information is displayed by the PowerJ session where the server component target is open. You can use most of the standard debugging techniques when debugging the server component. For example, you can step through code one statement at a time, examine data objects, modify data values, and so on. However, there is one important restriction imposed by the Sun VM: the Sun VM will not let you change values that are stored in the call stack. In particular, you cannot use debugger facilities to change the values of arguments that a function has received from its caller.

When you have finished your debugging session, you must explicitly terminate both the client applet and the server component.

- To terminate the client applet, close the browser or applet viewer that you used to open the applet.
- To terminate the server component, click **Terminate** on the PowerJ **Run** menu or click the appropriate button on the PowerJ toolbar (the button with the red X). You must terminate the server component in order to detach the debugging facilities from the Jaguar server.

When you terminate the server component, it does *not* shut down the Jaguar server itself.

Making corrections

In the typical debugging process, you do some testing, make any necessary corrections, then do more testing. This section looks at this process.

To make corrections, use the PowerJ session where you have opened the server component target. Follow these steps:

1. Edit the source code as necessary, and rebuild the server component target.
2. Copy the new `.class` files to the Jaguar server's `CLASSPATH`, as discussed in Publishing the class files.
3. Shut down the Jaguar server and restart it. Remember that there will be a new agent password when the Sun VM starts; you must copy the new password and fill it into the appropriate field in the server component's run options.
4. Run the server component target.
5. Start the client applet again.

When the server begins executing the applet, it will load the new versions of the `.class` files. Therefore, you will be able to test the changes you just made.

Note: If your changes to a server component class change the calling sequence of one or more methods in the class, you must change the information that has been registered about those methods. For a description of how to do this, see Changing a methods calling sequence. You must stop the Jaguar server and start it again in order for these changes to take effect.

Appendices

This part includes the bibliography and index.

Bibliography

The following is a selection of books and online resources on topics related to programming in Java:

Books

Java in a Nutshell (Second Edition)
David Flanagan (O'Reilly); ISBN 1-56592-262-X

The Java Programming Language
Ken Arnold, James Gosling (Addison-Wesley); ISBN 0-20163-455-4

The Java Application Programming Interface, Volume I, II
James Gosling, Frank Yellin (Addison-Wesley); Volume 1 ISBN 0-20163-453-8, Volume 2 ISBN 0-20163-459-7

Java Networking and AWT API SuperBible
Nataraj Nagaratnam, Brian Maso, Arvind Srinivasan (Waite Group); ISBN 1-57169-031-X

Core Java (Second Edition)
Gary Cornell, Cay S. Horstmann (Prentice Hall); ISBN 0-13596-891-7

Graphic Java: Mastering the AWT (Second Edition)
David M. Geary, Alan L. McClellan (Prentice Hall); ISBN 0-13863-077-1

The Java Class Libraries, An Annotated Reference
Patrick Chan, Rosanna Lee (Addison-Wesley); ISBN 0-20163-458-9

On-line magazines

Java Developer's Journal
http://www.sys-con.com/java/

Javology
http://www.javology.com/javology/

Javaworld
http://www.javaworld.com

Web sites

JavaSoft
http://java.sun.com

Java Computing, SUN
http://www.sun.com/java

The Java Developer Connection
http://java.sun.com/jdc/
http://java.developer.com

Microsoft's Java Center
http://www.microsoft.com/java/

Cup o' Joe
http://www.cupojoe.com

Earthweb's Gamelan
http://www.gamelan.com

Java Applet Rating Service(JARS)
http://www.jars.com

Java Financial Objects Exchange
http://www.jfox.com

Java News
http://www.radix.net/~cknudsen/javanews/welcome.html

Java Tools (IDE's at a glance, Available Tools, etc.)
http://www.cybercom.net/~frog/javaide.html

Java Zone
http://www.geocities.com/SiliconValley/Pines/3360/

JavaWorld
http://www.javaworld.com

Javology
http://www.magnastar.com/javology/

Porting C++ to Java
http://www.taligent.com/Technology/WhitePapers/PortingPaper/index.html

Que's Java Resource Center
http://www.mcp.com/372866662749526/que/javarc/

SunSITE UTK Java SITE
http://sunsite.utk.edu/java/

The Java Oasis
http://www.oasis.leo.org/java

The Java Repository
http://java.wiwi.uni-frankfurt.de/

Newsgroups

comp.lang.java
comp.lang.java.api
comp.lang.java.programmer
comp.lang.java.tech
comp.lang.java.announce
comp.lang.java.security
comp.lang.java.misc
comp.lang.java.setup
comp.lang.advocacy

Email lists

Sun's email lists

http://java.sun.com/mail.html

advanced-java email list

To subscribe, send email to majordomo@xcf.berkeley.edu with the body:

```
subscribe advanced-java
end
```