

Jato Programmer's Guide

VERSION 97.02.05



Copyright © 1997 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. AppModeler, InfoMaker, the Column Design, ComponentPack, DataArchitect, DataExpress, Data Pipeline, DataWindow, MetaWorks, ObjectCycle, Optima++, PowerBuilder Foundation Class Library, PowerScript, PowerTips, Powersoft Portfolio, Powersoft Professional, ProcessAnalyst, SDP, StarDesignor, Sybase SQL Anywhere, Watcom, Watcom SQL, Watcom SQL Server, 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 technology of Sybase, Inc. (U.S. patent pending).

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](#)
- [Ways to use this guide](#)
- [Getting help while you work](#)
- [Document conventions](#)
- [Part I. Fundamentals](#)
- [Part II. Advanced topics](#)
- [Appendices](#)

About this guide

This guide describes the Java development environment which Powersoft is currently referring to using the code name Jato. The guide assumes that you are familiar with the material in Jato Getting Started, supplied as part of your Jato 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 for Windows:

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

If you are a Windows API 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 views in Jato. Then you should read Chapter 1, Basic concepts of Jato. Chapters 2 and 3 describe the mechanical aspects of using Jato (how to save projects, how to edit source code, and so on). The specific details needed for writing Jato code begin with Standard types and events.

The chapters in Part I discuss the most common aspects of writing Jato programs: working with objects on forms, creating your own menus, using predefined dialog boxes, and debugging your code.

The chapters in Part II address advanced topics, including:

- Database and Internet applications.
- JavaBeans components.
- ActiveX controls and ActiveX server components.
- Multithreaded applications.
- Applications that involve graphics and printing.
- Advanced interactions with the Windows operating system.

Since these topics are advanced, most readers will only look at these chapters when they need the specific functionality provided by these features.

Getting help while you work

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

At any point during an Jato 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. Jato provides information about anything you click. When you have finished obtaining information, click off the question mark button.

Document conventions

The printed manuals for Jato 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, C++ classes, methods and events.
<i>Italics</i>	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.
%%%	This symbol is used in this beta documentation to mark areas of Jato that are subject to change before the final version of the product.

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 Jato 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 Jato.

Programs

This guide loosely uses the word “program” to refer to many sorts of applications. This 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.


 Jato Programmer's Guide

Part I. Fundamentals

This part describes fundamental aspects of programming with Jato.


-  Chapter 1. Basic concepts of Jato
-  Chapter 2. Using targets and projects
-  Chapter 3. Using Jato
-  Chapter 4. Standard types and events
-  Chapter 5. Programming standard objects
-  Chapter 6. Using and programming menus
-  Chapter 7. Debugging

 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)


Chapter 1. Basic concepts of Jato


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


 [Java](#)

 [Java and the web](#)


 [Forms](#)


 [Objects](#)

 [Events](#)

 [Jato and AWT](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

Java

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 which 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.

At present, there are two popular implementations of the Java virtual machine: one from Microsoft and one from Sun's JavaSoft. Jato code can run on either version. %%% However, the beta version of the Jato debugger only works with the Microsoft virtual machine.

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`.

[!\[\]\(a3ea015cc5581cad732d1eb81613fe7b_img.jpg\) Java Programmer's Guide](#)

[!\[\]\(c8d96c8885d3000a912c2582004aed63_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(919a2cb85b99741a73c0c31a427236a8_img.jpg\) Chapter 1. Basic concepts of Java](#)

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 aspect of Java makes it 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.

[!\[\]\(e3f8612927870f2e0f9f5989e6dd3064_img.jpg\) Web pages](#)

[!\[\]\(003082e50e3009141f59bd5df831749f_img.jpg\) Java applets](#)

[!\[\]\(17413706fd4997a1a4bdf85c6864eee1_img.jpg\) Sending information to the server](#)

[!\[\]\(faf942dc3e59ce8eb64b4ac481eca7e0_img.jpg\) Other uses of Java](#)

[!\[\]\(cf531ed27e91483460120fcc057b3901_img.jpg\) JavaScript](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 1. Basic concepts of Jato](#)

[Java and the web](#)

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.powersoft.com/products/psnews.html
```

This URL specifies the name of a computer system on the Internet (`powersoft.com`) and the name of a file on that system (`products/psnews.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.

For more information about URLs, see [Writing Internet applications](#).

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.

One of the most important functions of Jato is to help you create dynamic web pages. In particular, Jato makes it easy to create web pages that obtain information from a database, which is then transmitted to the user in an appropriate format. For more information on how to do this, see [NetImpact Dynamo server applications](#).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

 [Java and the web](#)

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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

 [Java and the web](#)

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 [Writing Internet applications](#).


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 [Writing Internet applications](#).

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.

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Java](#)

 [Java and the web](#)

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 Optima++ 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 [Writing Internet applications](#).

This type of servlet is also known as a web server extension.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

 [Java and the web](#)


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 (.exe), a JavaScript application is similar to a DOS batch file (.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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)


Forms


A *form* is a window that can be displayed on the user's system. When you use Jato 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 box).

Note: Do not confuse Jato forms with HTML forms. A Jato 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 form can open another 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](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

 [Forms](#)

Forms in Java

In Jato, 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 will also be called `Form1` and its source file is named `Form1.java`.

Note: Jato 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. When you create a new form, we suggest that you create a *modeless dialog* unless you have a reason for choosing some other form type. A modeless dialog supports more operations than the plain "form" type.


 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)


 [Chapter 1. Basic concepts of Jato](#)

Objects


The first stage of creating a program with Jato is placing objects on a form. To do this, you select a component type from the Java 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](#)


 [Object methods](#)

 [Object properties](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

 [Objects](#)

Object names

Jato associates a name with each object on the form. For example, Jato 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, Jato might create the following declaration in the form class's source code:


```
CommandButton cb_1 = new CommandButton( );
```


This indicates that `cb_1` refers to a `CommandButton` object. Jato also generates code in the form class to initialize the `CommandButton` 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 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

 [Objects](#)

Object methods

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

```
lb_1.delete( 3 );
```

uses the **delete** method of a list box object to delete item 3 from `lb_1`. (The “.” is the standard Java operator for invoking a method of an object.)

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 1. Basic concepts of Jato](#)

[Objects](#)

Object properties

The properties of an object control the object's appearance and behavior. Jato 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.


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


```
cb_1->setText( "New text" );
```

changes the text 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.

The Jato Component Library Reference documents the **get** and **set** functions for a property in the entry for the property. For example, **getText** and **setText** are explained in the entry for the **Text** property.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

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, a **Click** 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 a **Create** event for the `Form2` object.

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


Writing Jato programs is mainly a matter of writing Java routines to handle events. For example, you do *not* have to write a mainline for the program; that is handled automatically by Jato. 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 box, 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 **Click** event handler for `cb_1` can change the contents of a list box elsewhere on the form.


A typical event handler has a name like


```
cb_1_Click
```


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

 [Event handler calling sequence](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)


 [Events](#)


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 Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 1. Basic concepts of Jato](#)

Jato and AWT

The *Abstract Windows Toolkit* (AWT) is a standard library of Java class definitions which define the common elements of graphical user interfaces: push buttons, list boxes, 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 box) and events that can happen (for example, the user clicking a command button or entering text in a text box). AWT also defines other useful data classes (such as a String class for character strings and an Image class for “pictures”).

The *Jato component library* is built on top of AWT. For example, one of the data members of a Jato `CommandButton` object is an AWT `Button` object. Similarly, many of the methods that can be performed on a `CommandButton` object correspond to AWT `Button` methods.

Most events that can be received by a `CommandButton` object correspond one-to-one with the events that can be received by an AWT `Button`. On the other hand, some events recognized for Jato objects do not correspond directly to AWT events, and vice versa.

All of this means that the 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 Jato is primarily a matter of learning to use the design environment and the Jato 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 Jato Component Library Reference.

<p>Important: With Jato, you do not have to use sophisticated features of Java. The Jato 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.</p>
--

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

Chapter 2. Using targets and projects

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

[Targets](#)

[Projects](#)

[New projects](#)

[Opening an existing project](#)

[Closing a project](#)

[Running a target](#)

[Target folder contents](#)

[Managing project and targets](#)

[Options for building targets](#)

[Target types](#)


[Build macros](#)


[Using libraries](#)

[Source code control in Jato](#)

[Summary of targets and projects](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

Targets

A *target* is an applet, a web application, a standalone Java application, a Java library, or a DLL that can be used with a web browser (Internet Explorer or Netscape Navigator).


Jato 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` 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](#)

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 2. Using targets and projects](#)

 [_Targets](#)


Target folders


Each Jato 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 files associated with a particular target are kept in the target folder.

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

The default name for a target folder is based on the name of the target itself. For example, if your target is named `targ.exe`, the default name for the target folder is `targ`. Jato gives you the chance to specify a different name for the target folder when you first create the target or when you use **Save Project As** to save the project.

 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)


Projects


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

Some Jato 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](#)

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 2. Using targets and projects](#)

 [_Projects](#)

The project file


Jato 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.


Project files have the file extension `.WXP`. By default, Jato 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; Jato maintains the file for you.

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

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)


New projects

When you begin using Jato, Jato 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, Jato has to create a number of files and store these files on your system. Therefore, Jato stores the untitled project under the *temporary folder* that you specified when you installed the Jato 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), Jato 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 rebuilding some files twice.


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

 [Starting a new project](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [New projects](#)

Starting a new project


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


◆ **To start a new project in the middle of your Jato session:**

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

The new project will automatically be an untitled project.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

Opening an existing project


When you start your Jato 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 Jato menu bar, click **Open Project**. By default, Jato 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 `.WXP`), then click **Open**.

Note: You can only work on one project at a time in a single Jato session. If you are working on one project and then open a different one, Jato 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 Jato session for each project.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

Closing a project


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


◆ **To close a project explicitly:**

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

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

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

Running a target


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

◆ **To run your project:**


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





 [Before your program starts executing](#)

 [Compilation and link errors](#)


 [Running different types of targets](#)


 [Debugging facilities](#)


 [Running with multiple targets](#)

 [Run options](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Running a target](#)

Before your program starts executing

When you run a target, Jato compiles your source code and links the result into an executable program. Compilation can be a lengthy process if you have a lot of code; however, Jato 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.

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

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.

[Jato Programmer's Guide](#)

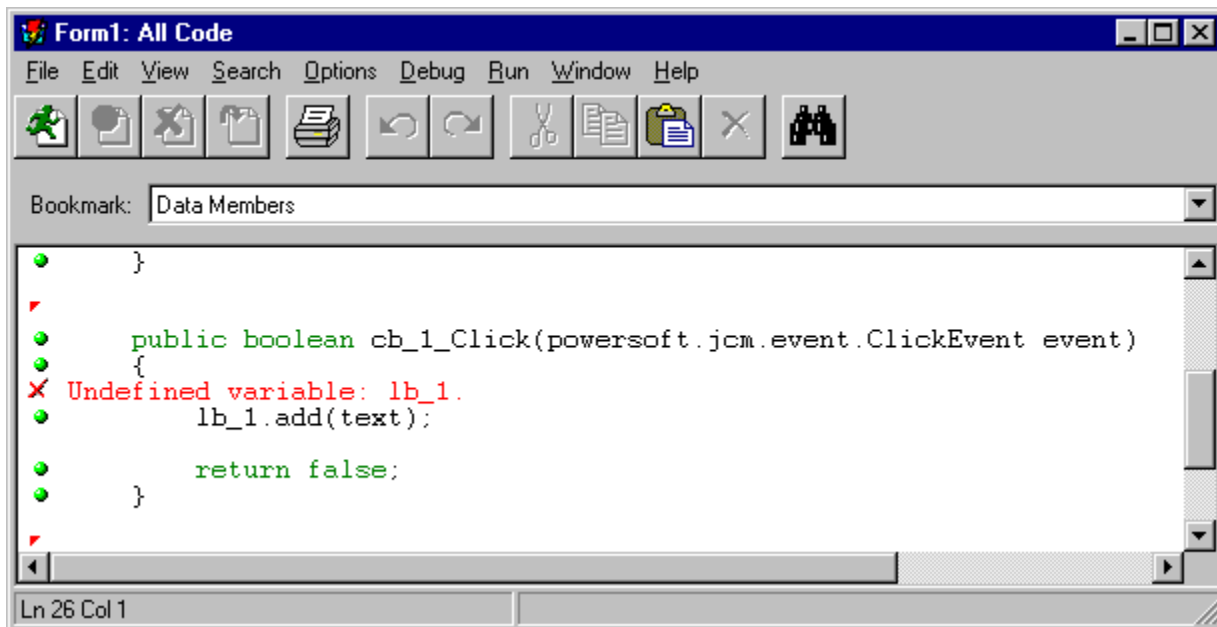
[Part I. Fundamentals](#)

[Chapter 2. Using targets and projects](#)

[Running a target](#)

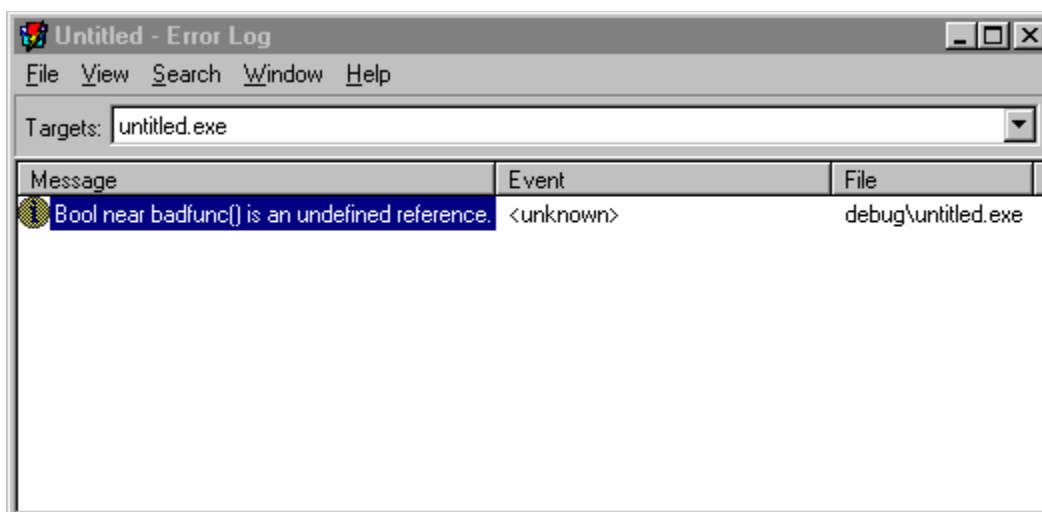
Compilation and link errors

If Jato 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 red X in the left margin.



If the compiler found more errors than can be shown in one code editor window, 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.


Jato 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 a notes that further explains the problem.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)


 [Running a target](#)


Running different types of targets


Jato 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 Applet Viewer program. Applet Viewer can run a compiled applet, without needing 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](#).

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 2. Using targets and projects](#)


 [_Running a target](#)


Debugging facilities


Jato 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 Jato debugging facilities, see [_Debugging](#).

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)


 [_Chapter 2. Using targets and projects](#)

 [_Running a target](#)


Running with multiple targets

Even if your project contains more than one target, Jato only runs one target at a time. If you run a project with several targets that could be run, Jato asks which target you want to run. DLL and library targets can not be run.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Running a target](#)

Run options

The run options for a target control the way that Jato 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. Open the Targets window by clicking **Targets** in any Jato **View** menu.
2. In the Targets window, use the right mouse button to click on the target whose run options you want to set, then click **Run Options**.
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](#).

Target folder contents

You will seldom need to know how Jato organizes your project into files and folders; Jato 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, Jato may compile Java source code files to produce class files (with the extension `.class`).

Jato 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.

%%% The following list is subject to change.

Suppose that you have saved an applet target called `target` in a folder named `targ`. The following list shows the files that you may see in this 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.wxp`

The project file.

`target.wxt`

The target file containing information about the target. This information controls the steps that Jato 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 Jato when you last edited this project (for example, the size of the main Jato window, the view windows you had open, the size of any debugger windows, and so on). This lets Jato restore all those windows to the same positions, the next time you start 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), files containing Java linked targets (`.jlt` 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. **Error! Reference source not found.** 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 Jato 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**.


Jato automatically changes the name of the form file associated with the form.


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.

<p>Important: When you use the right mouse button to click on an object in the <code>Classes</code> window, the resulting menu has a Rename item. Using this item renames the object in the <code>Classes</code> window, but does <i>not</i> make any other changes. In particular, it does not change the name of any files associated with the object or change names used in source code. If you want to change the file name, you must go through Properties, not through Rename.</p>

Backup files

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 2. Using targets and projects](#)


 [_Target folder contents](#)


Backup files

Jato 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. Jato 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`.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)


Managing project and targets


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

Jato lets you add new or existing source files to a target. Source files may be of several types:


- Java source code files.
- Java class files
- Files containing images (for example, .GIF files).
- Library and DLL files.
- Managed class definitions (either new ones or ones created for other targets).
- Forms (either new ones or ones created for other targets).
- Existing targets.


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

 [Adding a target to a project](#)


 [Adding source files to a target](#)


 [Adding classes to a target](#)


 [Adding forms to a target](#)


 [Targets that depend on each other](#)


 [Deleting targets and source files](#)

 [Read only folders](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Managing project 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 Jato project).
- ◆ **To add a new target to a project:**
 1. From the **File** menu of the main Jato 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 Jato can create. This includes several types of libraries and executables. Click the type of target you want to create.
 3. Type in a name for the new target, then click **Next**.
 4. Specify where to store the files associated with the new target. By default, this is a new folder under the Jato `Projects` folder, using the name you just chose for the new target. Accept this default, or specify another location. Then click **Next**.
 5. 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.


When you have finished entering this information, Jato creates the new target and all the necessary source files. When you ask to run this target, Jato will run that target and any others that it depends on.


Note: Whenever you start your Jato session, Jato creates a default target: an executable program named `untitled`. If you create a new target, and the existing target is an unmodified version of the default target, Jato 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. Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Managing project and targets](#)

Adding source files to a target

You can add files using the Files window, the Targets window or the **File** menu on the main Jato 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).

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. Jato prompts you to type in a name for the new file. This should not be the name of an existing file, because Jato will overwrite the file if it already exists. Jato 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.


Jato 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**. Jato 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.


Jato adds the file to the target. For example, if you add a class file, the file will be added to the set of class files used in linking/loading your program. If you add any Jato form file, the form and all its associated code will be added to the current target. Jato 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, Jato adds a reference to the original file in the list of files associated with the current target; in this case, the reference is a pathname relative to the target folder. With other types of files, Jato makes a copy of the original file and stores a copy in the target folder. From this point on, Jato builds your target from the copy, not from the original file; therefore, if you make changes to the original file, they are *not* automatically inherited by your project.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Managing project and targets](#)

Adding classes to a target

You can add a new class to a target by using the **File** menu of the main Jato menu bar, or the Classes window, 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 Jato has records of what function(s) the classes contain. Managed classes are stored in files with the extension `.WXC`.

There are several 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](#).

Standard Java

Any type of class that is not covered by the other choices.

◆ 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. 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 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 you do not want to use the default file name for this class, type a different name under **File name**.
9. If this will be a public class, make sure **Public** is checked.
10. If this will be an abstract class, make sure **Abstract** is checked.
11. If this will be an interface, make sure **Interface** is checked.
12. 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 **Error! Reference source not found.** for instructions on adding member functions.

Jato 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, Jato 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](#).

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 2. Using targets and projects](#)

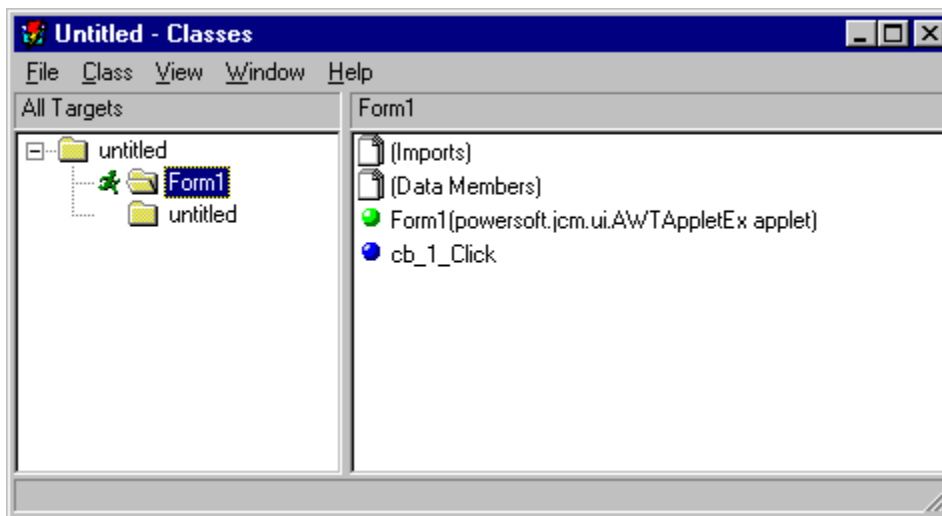
[Managing project and targets](#)

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 **Error! Reference source not found.**

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](#).


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 Class**.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Managing project and targets](#)

Targets that depend on each other

A project may contain one target that depends on another. For example, the project may contain a library target and a program that uses that library. In this case, you have to tell Jato about the dependency, so that Jato will remake the program whenever the library 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, Jato 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 (.WXP 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. This is why the dependency must be set up manually.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Managing project and targets](#)

Deleting targets and source files

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


◆ **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, Jato 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 library 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 Files window.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Managing project and targets](#)

Read only folders

Some source files “never” change. For example, one of the source files for a target may be a standard AWT class library that only changes with new releases of AWT. Similarly, the standard class files that come with Jato itself will only change if you install a new version of Jato.

If you believe that all the source files in a particular folder will not change during the course of developing a project, you can designate that folder as a *read only folder*. When rebuilding the project, Jato will not check the change dates of files in any read only folder; by skipping this date check, Jato can make the rebuilding process faster.

◆ **To designate a folder as a read only folder:**

1. From the **Tools** menu on the main Jato menu bar, click **Options**, then click the **Read Only Folders** tab.
2. Type the name of the folder at the bottom of the current **Folder list**.
3. Click **OK**.


If you change the list of read only folders, all targets are built from scratch the next time you build them.


If any of the files in a read only folder actually do change, you should explicitly build any other files that depend on the changed files.

◆ **To force a file to be built:**

1. Open the Targets window.
2. Use the right mouse button to click on the file you want to rebuild, then click **Properties**.
3. On the **General** page of the resulting property sheet, click **Build now**.

 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)


 [Chapter 2. Using targets and projects](#)


Options for building targets


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

In the Targets window, you may use the right mouse button to click any of the file names displayed and obtain a *property sheet* for the file.


 [Target properties](#)


 [Source file properties](#)


 [Debug and release versions of a target](#)

 [Default options](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Options for building targets](#)

Target properties

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

- Any specific options for building and linking the target
- What folders will be searched for `.CLASS` files needed to link the target

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

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 2. Using targets and projects](#)

[Options for building targets](#)

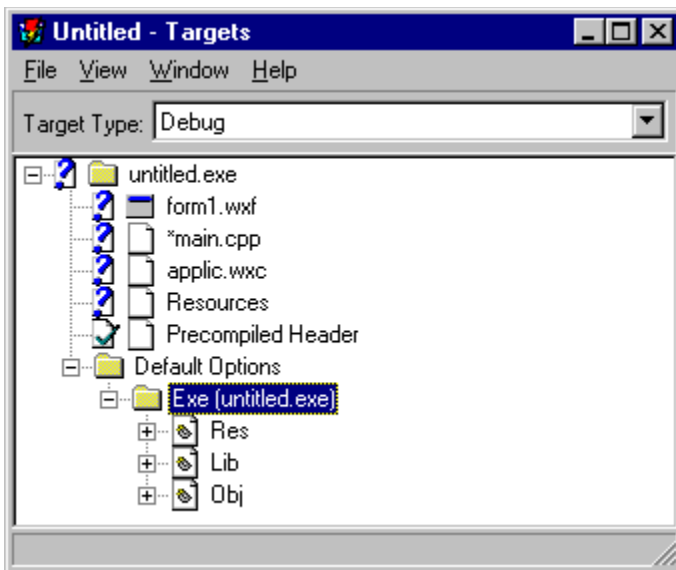
Source file properties

For a source file, the property sheet controls the actions that Jato takes when processing the file. This includes:

- %%% Any specific options for processing the source file


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


If you change the default properties associated with a source file, the file's name is marked with an asterisk in the Targets window. For example, if you define some special macros for compiling `Form1`, the Targets window prefixes `form1.wxf` with an asterisk:





Any file that is not marked with an asterisk is using the default options for source files of that type.

If you want to change a source file back to default option, use the right mouse button to click on it and click **Use Default Build Options**.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Options for building targets](#)

Debug and release versions of a target

When you are debugging a target, Jato 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 Jato menu bar, click **Targets**. This displays the Targets window.
2. Click the **Target Type** drop down list, then click **Release**.

From this point on, Jato 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.


Note: If you need to, you can debug a release target if the Targets window is set to the release target type.
--


For more information on deploying an application, see **Error! Reference source not found.**


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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Options for building targets](#)

Default options

The Targets window shows an entry called **Default Options** under every target. Expanding this entry shows various types of files associated with the target (Java source files, Java class files, and so on). To see the default options for a given type of file, look at the property sheet associated with the appropriate entry under **Default Options**. For example, the property sheet for **Java Target** shows the default options used for JLT files.


If you change the default options for a target, Jato asks what you want to do with source files that are *not* currently using the defaults.

If you have changed the options for a target or source file, then decide to change back to the defaults, you use the Targets window.

◆ **To change a source file or target back to the default build options:**

1. Open the Targets window.
2. Use the right mouse button to click on the target or source file whose options you want to change, then click **Use Default Build Options**.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

Target types


This section looks at types of targets that can be created with Jato. 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 library
- Web application
- Java Web server application (servlet)
- Java Dynamo Server application (servlet)
- User-defined targets (using target templates)


The following sections describes 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](#)

 [Java applications](#)

 [Java libraries](#)

 [Web applications](#)

 [Java web server applications](#)

 [Dynamo server applications](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 2. Using targets and projects](#)

[Target types](#)

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`) defining global information for the applet
- A form file (`.WXF`) for the applet's initial form

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.

Run options for Java applets

The following run options apply to Java applets:

Use Microsoft 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. This is either the name of the EXE file containing the web browser or a `RUN` command to run the browser program.

HTML File [General page]

Specifies the HTML file that the web browser or Applet Viewer should open. By default, the HTML file is the basic HTML file created as one of the source files for the Java applet target.

Don't use the debugger [Debug page]

Prevents the use of the debugger while running the applet. %%% In this beta release, this is necessary if you are using the Sun virtual machine.

Run with the debugger [Debug page]

Runs the program under control of the Jato debugger. In this case, you must specify the class that you want to debug. Typically, you specify 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 Jato debugger. For more information, see [Breakpoints](#).


Messages page


Specifies how you want to display error messages that come from the Java interpreter. This can be a fixed size console window or a resizable window.

Properties for Java applet targets

%% Not yet finalized

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Target types](#)

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) defining global information for the application
- A form file (.WXF) 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 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.

Don't use the debugger [Debug page]

Prevents the use of the debugger while running the applet. %%% In this beta release, this is necessary if you are using the Sun virtual machine.

Run with the debugger [Debug page]

Runs the program under control of the Jato debugger. In this case, you must specify the class that you want to debug. Typically, you specify 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 Jato debugger. For more information, see [Breakpoints](#).


Messages page


Specifies how you want to display error messages that come from the Java interpreter. This can be a fixed size console window or a resizable window.

Properties for Java application targets

%% Not yet finalized

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Target types](#)

Java libraries

%%% Information about this type of target has not yet been finalized


Run options for Java libraries


%%% Not yet finalized

Properties for Java library targets

%%% Not yet finalized

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)


 [Chapter 2. Using targets and projects](#)


 [Target types](#)

Web applications

A web application target brings together a Java applet and a web server extension. For further information, see [Web application targets](#).

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)


 [_Chapter 2. Using targets and projects](#)


 [_Target types](#)

Java web server applications

A Java web server application is sometimes called a servlet. Specifically, this type of target creates an ISAPI servlet; this is implemented as a DLL which attaches to the Microsoft ISAPI web server when invoked by a User application. For further information, see [ISAPI web server applications](#).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Target types](#)

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](#).

Build macros

Jato 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 `Jato 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 Jato 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.

◆ To define a macro for use in path names:

1. From the **Tools** menu of the main Jato 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. As shown above, you use a macro in the form

```
$(macroname)
```

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


◆ To change an existing macro:


1. From the **Tools** menu of the main Jato 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 Jato.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Build macros](#)

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`.

Using libraries

%%% Library control has not yet been implemented in Jato.

Many targets may need to obtain code from libraries. For example, a target may call a routine from a library of utility routines shared between several projects. The property sheets for a target file specify which libraries should be used in preparing that target.

◆ To view the property sheet for a target file:

1. In the Targets window, use the right mouse button to click the target file, then click **Properties**.

The **Libraries** page of the property sheet specifies the libraries that are needed for building the file. You can add one or more libraries to the list using the **Libraries** page:

◆ To add a library to the list:

1. Type the name of the library in **Library Name**.
2. To add the name to the list, click **Add**.

Alternatively, you can drag the name of a library file from the Windows Explorer and drop it into the Files window.

When Jato is looking for a function or variable, it searches through the libraries in the order given in the list. This means that you may have to arrange the libraries in an order that ensures Jato finds the correct versions of each required function.

◆ To change the position of a library in the library list:

1. Click on the library whose position you want to change.
2. Use the arrow buttons underneath the list to move the library up or down the list.

The **Library Path** page of the property sheet tells the folders where Jato should look for the library files specified on the **Library** page. For further information, consult the online help.

[!\[\]\(f4912148590488019602cab6e009e597_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(8af806fb1314382d09bc5ec5b767526c_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(2e897e890e69d81eae4503a8342c36b0_img.jpg\) Chapter 2. Using targets and projects](#)

Source code control in Jato

This section explains the Jato facilities for performing source control in cooperation with various source management systems.

[!\[\]\(74d4806277d7e73349d8e8c0897931e9_img.jpg\) Source control support](#)

[!\[\]\(0aff635c4179ba9e710b00f4b01d3b20_img.jpg\) Supported source control systems](#)

[!\[\]\(830769b31eeeaca920791081939ff8ba_img.jpg\) Configuring Jato for ObjectCycle](#)

[!\[\]\(0b5e7e25e8775f7e7e80906ada4f0021_img.jpg\) Configuring Jato for PVCS](#)

[!\[\]\(8bba887393ca45b761e5cb49e755e762_img.jpg\) Configuring Jato for Visual SourceSafe](#)

[!\[\]\(6bb0e4f14c4133b37d2887cb37e67ddd_img.jpg\) Configuring Jato for RCS](#)

[!\[\]\(47734e4656765d20df4fdbd5b7aff048_img.jpg\) Configuring Jato for Source Integrity](#)

[!\[\]\(bd3b31712ad9bab5a241210fa6925cdd_img.jpg\) Configuring Jato for a generic system](#)

[!\[\]\(0fb13ad0bfa3d86868cdd3883e5665b3_img.jpg\) Local files](#)

[!\[\]\(799877f5c2f906134441300079881630_img.jpg\) Checking in a project for the first time](#)

[!\[\]\(41aea2746216b27a6939d696d8e035da_img.jpg\) Checking in files](#)

[!\[\]\(7bc43b319a082987e20f7bf78f4bab80_img.jpg\) Checking out files](#)


[!\[\]\(e50091943b385fe16d3277389202856f_img.jpg\) Undoing a check out operation](#)

[!\[\]\(4436e6b00b9d5e62c2a161129eb3e4d0_img.jpg\) Getting the latest revision of a file](#)


[!\[\]\(179f167ede0522ebb4ea025b3ad78ca7_img.jpg\) Opening a new source control project](#)

[!\[\]\(4a7b4ce770af8456e11a71f9565c8c2b_img.jpg\) Source control options](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

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 Jato 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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 2. Using targets and projects](#)

[Source code control in Jato](#)

Supported source control systems

Jato 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 1.5 or later).

If you don't have any of these source control systems, you can configure Jato to use a generic source control system in which you specify the commands Jato should execute to check files in or out.

You should be familiar with your source control system before configuring Jato to use it. The rest of this discussion assumes your source control system is already installed.


Registered systems


When you begin configuring Jato to use a source control system, Jato attempts to determine which systems are currently installed on your computer. For example, Jato may check the registry on your machine and see that both Visual SourceSafe and ObjectCycle are currently installed. Jato will therefore ask which of these two packages you want to use to control your Jato projects.


In some situations, Jato 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 Jato 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 Jato to use a source control system.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Configuring Jato for ObjectCycle

ObjectCycle stores files in collections called projects. Files within a project are stored hierarchically. Each Jato target has its own node in the ObjectCycle project. Target files are stored under the target's node.

You are free to store Jato files under any ObjectCycle project you like. You could create one ObjectCycle project for every Jato project, or you could create a single ObjectCycle project to hold all Jato projects in your organization.


Once you have created an ObjectCycle project for Jato, you do not have to worry about creating nodes in ObjectCycle; Jato takes care of that for you.


Note: You will need to know which version of ObjectCycle you are using.


◆ To set up Jato to use ObjectCycle:


1. Using the *ObjectCycle Manager* program, create a new ObjectCycle project to hold your Jato projects.
2. Start Jato.
3. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click either **ObjectCycle 1.0.00 (native)** for ObjectCycle version 1.0, or **ObjectCycle** for ObjectCycle version 1.5.
4. Click **OK**.

The next time you open a project, you will be prompted to log in to your ObjectCycle server.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Configuring Jato for PVCS

◆ To set up Jato to use PVCS:

1. Start Jato.
2. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **PVCS**.
3. Make sure **Show commands as they are executed** is checked.
4. If you want Jato to pause after executing each command, make sure **Pause after each command** is checked.
5. 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 Jato has stopped executing.


By default, PVCS 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 Jato 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.WXP` 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Configuring Jato for Visual SourceSafe

Visual SourceSafe stores files in collections called projects. It is up to you to create Visual SourceSafe projects using the *Visual SourceSafe Explorer* program.


Jato 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 Jato with Visual SourceSafe.


Jato assumes the following Visual SourceSafe settings:


- **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 Jato to use Visual SourceSafe:**
 1. Start Jato.
 2. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **Microsoft Visual SourceSafe**.
 3. Click **OK**.

The next time you open a project, you will be prompted to log on to Visual SourceSafe.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)


 [Source code control in Jato](#)


Configuring Jato for RCS


◆ To set up Jato to use RCS:

1. Start Jato.
2. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **RCS**.
3. Jato 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 Jato to pause after executing each command, make sure **Pause after each command** is checked.
5. Click **OK**.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Configuring Jato for Source Integrity

Source Integrity stores files in collections called projects. Jato 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.

<p>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.</p>

◆ **To set up Jato to use Source Integrity:**

1. Start Jato.
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 Jato for a generic system

If you do not have any of the above source control systems, you can tell Jato what command(s) to execute in order to check a file in or out. When you tell Jato to use a generic source control system, it calls an external batch file to perform source control operations. It passes the type of operation and the full name of the source file as command line arguments to the batch file. You just add whatever commands your source control system needs to the batch file.

◆ To set up Jato 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 Jato 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 %1
  goto :done
```

When the batch file runs, the `%1` 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 %1
  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:


```
:doCheckOut
  c:\bin\myrcs\get %1
  goto :done
```

5. Save the batch file.
6. Start Jato.
7. On the **Tools** menu, click **Options** and then click the **Source Control** tab. Under **Source control system**, click **Generic**.
8. Click **OK**.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)


 [Source code control in Jato](#)


Local files


Jato 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.

Jato 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 Jato to check out files, check in modifications, and refresh files from the source control archive.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

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 Jato. Source control projects are an abstraction used by your source control system to help organize checked in files. Jato projects are collections of Jato targets.

Jato does not create source control projects; you must do that yourself using the tools that came with your source control system. The following sections gives 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 Jato targets. If you have already created an ObjectCycle project, you can skip the following steps. To create a new ObjectCycle project to hold your Jato 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 Jato. Note that you do not need the ObjectCycle Manager to check in or check out files; you should do that from within Jato itself.

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`) 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 Jato 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 Jato, you should open the project file (`.WXP`) in the sandbox folder.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 2. Using targets and projects](#)

[Source code control in Jato](#)

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 Jato Files window. A list of the files extensions is given below:

Extensi on	File Type
wxf	Form file
wxc	Class file
wxr	Resource file
wxt	Target file
wxp	Project file
java	Java source file
c	C source file
h	C header file
htm, html	HTML file
gif	GIF graphic file
jpg, jpeg	JPEG graphic file
ico	Icon file
bmp	Bitmap file
cur	Cursor file
rc	Resource script 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 ICO, BMP, or CUR files. All of the files listed above are text files except for BMP, ICO, and CUR. 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.
- Jato backup files. When Jato 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 `RC`, `CPP`, and `HPP` 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, Jato 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, but you can also check select which files to check in.

◆ To check in all the files that you have checked out:

1. In the main Jato **File** menu, click **Check In Project**.

This checks in all of the files that you have checked out. Jato 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.


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. Click 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: Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Checking out files

Jato 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, Jato 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, Jato 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.


◆ To check out a file or a number of files:

1. Open the Files window.
2. Click 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, Jato automatically reloads the file in the design environment (if possible).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Undoing a check out operation

After checking out files and changing them, you may decide to throw away the changes you have made. Jato can discard your changes if you have not checked them in. Jato can also unlock the files and get the latest versions of the files.


◆ 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, Jato automatically reloads the file in the design environment (if possible).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Getting the latest revision of a file

◆ **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**. Jato 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, Jato automatically reloads the file in the design environment (if possible).


The **Refresh Project** item of the main Jato **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 Files windows **File** menu will contain the item **Refresh Target**. As with **Refresh Project**, **Refresh Target** does not overwrite any files that you currently have checked out.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

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.


This feature is only supported for source control packages which are SCC-compliant.

On the **File** menu of the main Jato 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, Jato does everything needed to bring the project to your system and set things up so you can begin development on the project.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 2. Using targets and projects](#)

 [Source code control in Jato](#)

Source control options

Jato 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 1.5, or MKS Source Integrity).

◆ To set source control options:

1. On the **Tools** menu of the main Jato menu bar, click **Source Control Options**.

The Source Control Options item is disabled until you have configured Jato 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.

Summary of targets and projects

Targets and projects

A target is an applet, library or application built with Jato. Targets are built from source files. Each target must be kept in its own folder.

A project is a collection of one or more targets. The project file lists all the targets associated with a project.

When you start your Jato session, you typically begin with an untitled project that has one untitled target (an executable program named `Untitled.exe`). You can design forms for this project and run the untitled program to test it. When you save the project, Jato asks you to specify a name for each target folder and the target file.

Running a target

When you run a target, Jato checks to see if you have changed any of the target's source files since the last time you built the target. If so, Jato builds the target again, then runs the program.

During the build process, Jato may detect compilation errors or linkage errors. All errors are reported in the error log. In addition, compilation errors are reported in code editor windows, showing the point where each error was detected.

The target folder

The folder associated with a target contains a number of files and folders associated with the target. In particular, the `Debug` folder contains files associated with running the program in debugging mode and the `Release` folder contains files required for an end-user release of the target. To switch from `Debug` to `Release`, use the Targets window.

The Targets window

The Targets window displays the targets of your project, the source files that are used to build each target, and the default options used in processing the targets and source files. The menus of the Targets window let you add new targets and source files to the project, and also make it possible to specify dependencies between targets.

You can also use the Targets window to get a property sheet for each target and source file. Each property sheet shows the options that will be used in processing the associated file. For example, the property sheet for a source file containing Java code shows the options for compiling that file.

Types of targets

With Jato, you can build applets, standalone Java applications, Java libraries, web server applications, and NetImpact Dynamo applications.

Source control

Jato can coordinate its actions with a variety of source control systems. You use the **Source Control Options** item of the **Options** dialog to specify which source control package you are using. You can then open the Files window and use the right mouse button to click on a particular file to check that file in or out.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

Chapter 3. Using Jato

This chapter explains the mechanics of using Jato: 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 Jato](#).

[General usage notes](#)

[Using the form design window](#)

[Changing a form's properties](#)

[Adding objects to a form](#)

[Visual classes](#)

[Templates](#)

[Startup options](#)

[File type options](#)

[Changing an object's properties](#)

[Adding and modifying event handlers](#)


[Working with classes](#)


[Using drag-and-drop programming](#)

[Jato command line options](#)

[Summary of basic Jato operations](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

General usage notes

Jato 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 Jato menu bar contains a list of all form design windows, open or closed.

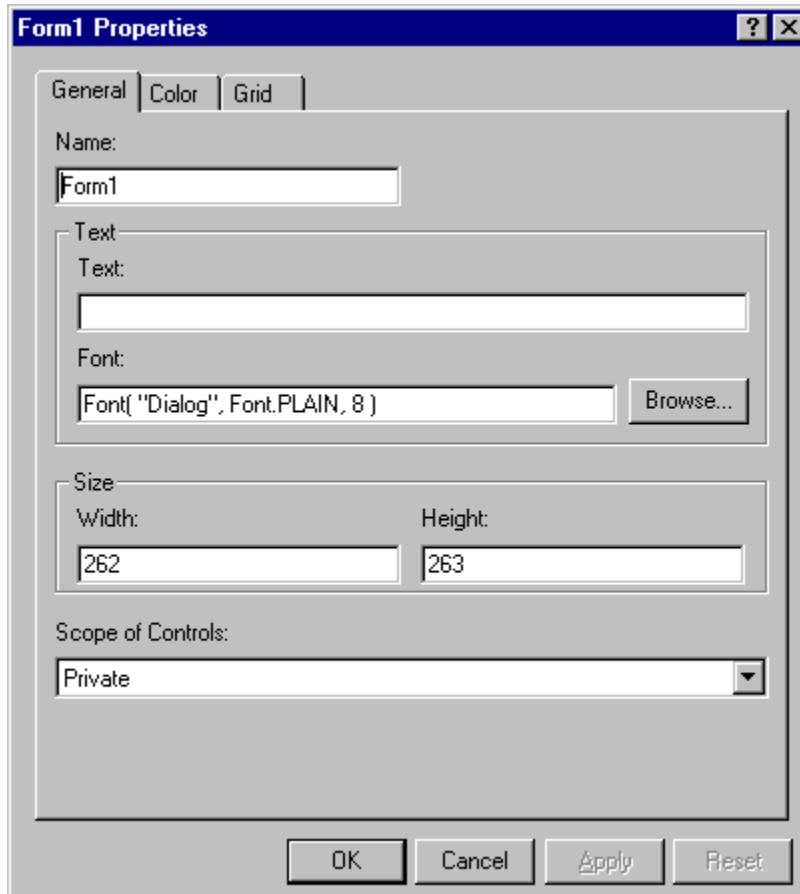
◆ To open a closed form design window:

1. Click the name of the form in the **Window** menu of the main Jato menu bar.

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, Jato 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 Jato 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

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Changing a form's properties](#)

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, Jato 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.

[!\[\]\(b39c89771cd6fb2128a8c57aa7d97f9a_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(d0a1791f26d167e866e44ebbf83efebe_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(5eb1325dfdc3f1cad8426726c0db51cd_img.jpg\) Chapter 3. Using Jato](#)

Adding objects to a form

The first step in creating a program with Jato is to design one or more forms. When you start a new project, Jato 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.

[!\[\]\(950a62bbddad88d64435fd35607dfc42_img.jpg\) The Java Component palette](#)

[!\[\]\(5a132f13505a6571904d622757b7a8f0_img.jpg\) Positioning an object](#)

[!\[\]\(10f8862fc183b400327470ea85afe9ae_img.jpg\) Sizing an object](#)

[!\[\]\(e1d6102fe77919492c04879c8450f1f5_img.jpg\) Deleting an object](#)

[!\[\]\(73002692dd5e7a64e60946be3158e719_img.jpg\) Copying an object](#)

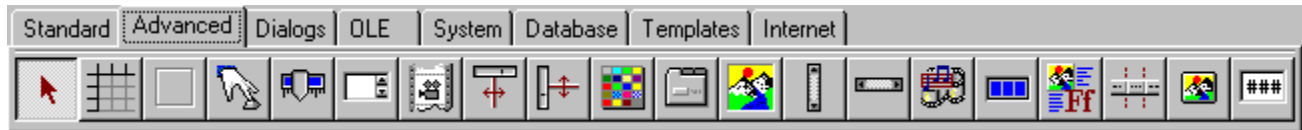
[!\[\]\(d5d7044e5caf6907399af2dced8d6ff8_img.jpg\) Aligning objects](#)

[!\[\]\(35dc653d59570f8f891c312eeece91a2_img.jpg\) Matching object sizes](#)

[!\[\]\(ab4e2b3fc7e7887b7a72f548aa6f5e60_img.jpg\) Selecting multiple objects](#)

The Java Component palette

The *Java Component palette* is a tabbed toolbar that has buttons with icons representing 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, Jato displays a *tooltip* telling what the button means.

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. Jato 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, Jato 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. Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a form](#)

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 `SHIFT` key and press an arrow key, Jato moves the active object in the direction of the arrow.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a form](#)

Sizing an object


Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a form](#)

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, Jato deletes the form. In the process, Jato deletes all objects and member functions associated with the form, as well as any files associated with the form.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a 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 slightly offset from the original so that you can see it more clearly. 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 a **Click** event handler for the original object, the copied object has an identical **Click** event handler, except that it is named `cb_2_Click` instead of `cb_1_Click`. 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a form](#)

Aligning objects


Jato 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, Jato places hollow sizing handles on all the objects.
2. Click an object whose alignment you want to match. Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a form](#)


Matching object sizes


Jato 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, Jato places hollow sizing handles on all the objects.
2. Click an object whose size you want to match. Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Adding objects to a form](#)

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.

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 [Working with databases](#)). Therefore, you could create 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 create a visual class object, Jato automatically creates all the objects that you have placed on the visual class "form". This saves you the trouble of creating such objects explicitly.

All visual classes must inherit from some standard Jato class. By default, they inherit from `Object`, which is the most general class in Java.

◆ To create a visual class:

1. From the **Files** menu of the main Jato menu bar, click **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 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 you do not want to use the default file name for this class, type a different name under **File name**.
9. If this will be a public class, make sure **Public** is checked.
10. If this will be an abstract class, make sure **Abstract** is checked.
11. If this will be an interface, make sure **Interface** is checked.
12. Click **Finish**.


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


When Jato creates an object of the visual class type, it automatically creates all the member objects that have been placed on the visual class. In the process it triggers **Create** events for those objects. Similarly, when Jato destroys the object, it triggers **Destroy** events for all the objects on the visual


class.

Visual class properties

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Visual classes](#)


Visual class properties


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


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`.

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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

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 **Click** event handlers for the two buttons). If you place that template onto a form, Jato places the two buttons onto the form. If necessary, you can then change the position of the buttons or the code in their **Click** 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, Jato 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. Jato 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**, Jato calls up the Image Editor to create an icon for this template.
6. Click **OK**.

Jato 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 Jato 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**. Jato checks to make sure you really want to delete the template.


 [Form templates](#)


 [Target templates](#)

 [The templates folder](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Templates](#)

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. Jato 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**, Jato calls up the Image Editor to create an icon for this template.
6. Click **OK**.

Jato 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 Jato 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.


<p>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.</p>


If you create a new form from a form template, and the form name associated with that template matches an existing form name, you will be asked to specify a new name for the form being created. For example, suppose you create a form template specifying a form named `TempForm`. If you use this template to create a new form for a target that already has a form named `TempForm`, you will be asked to specify a different name for the new form.


◆ 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**. Jato checks to make sure you really want to delete the template.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Templates](#)

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 codes 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**. Jato checks to make sure you really want to delete the template.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Templates](#)

The templates folder

By default, the templates you create are stored in a folder named `Template` under the main Jato 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 Jato 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, Jato will search for templates in the specified folder and will also store new templates in that folder.

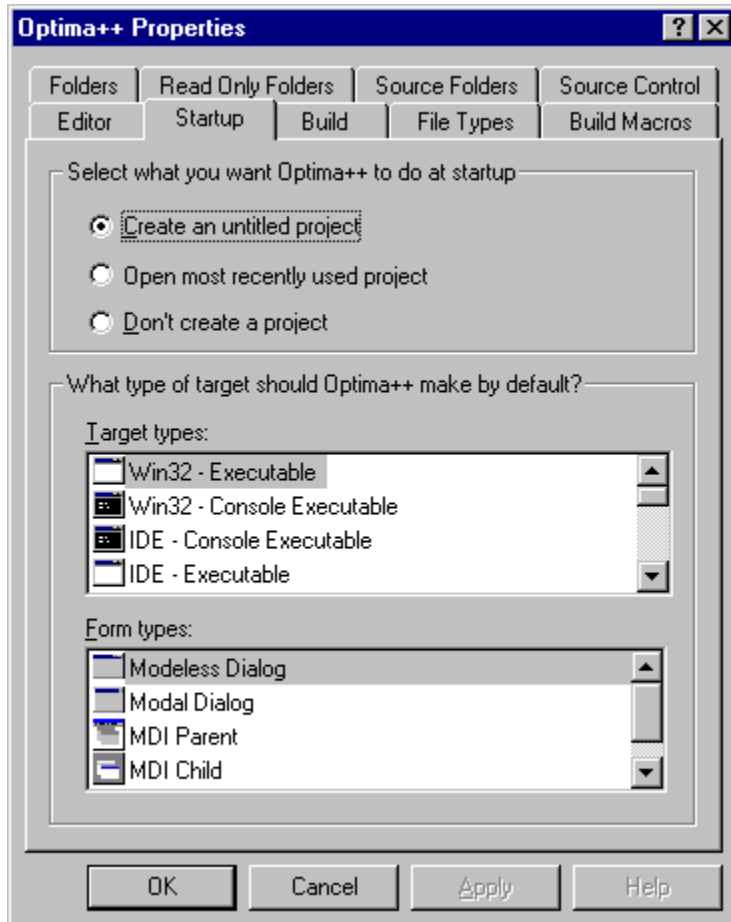
Startup options

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

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

1. From the **Tools** menu on the main Jato menu bar, click **Options** and then **Startup**.

This displays the following:



The default selection is **Create an untitled project**. If this is marked, Jato creates an untitled project every time it begins execution. If **No action** is marked, Jato 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, Jato 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, Jato will use those as the initial target and form for any new project.

File type options

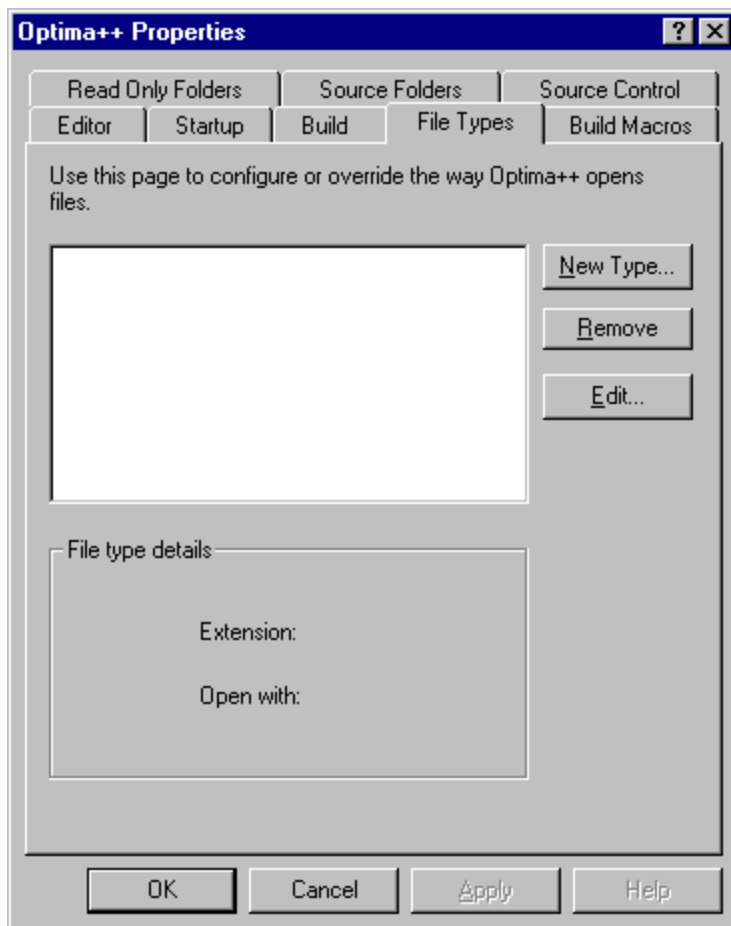
The *file type options* for Jato dictate how Jato opens various types of files. Jato determines the type of a file by examining the file name extension of the file. For example, files whose names end in `.WXC` are taken to be managed class definitions; when Jato opens such a file, it uses a code editor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

By default, Jato identifies files using information taken from the system registry. The registry lists a number of file name extensions and the programs that should be used to open them. File type option settings let you override the defaults given in the registry, creating Jato-specific options.

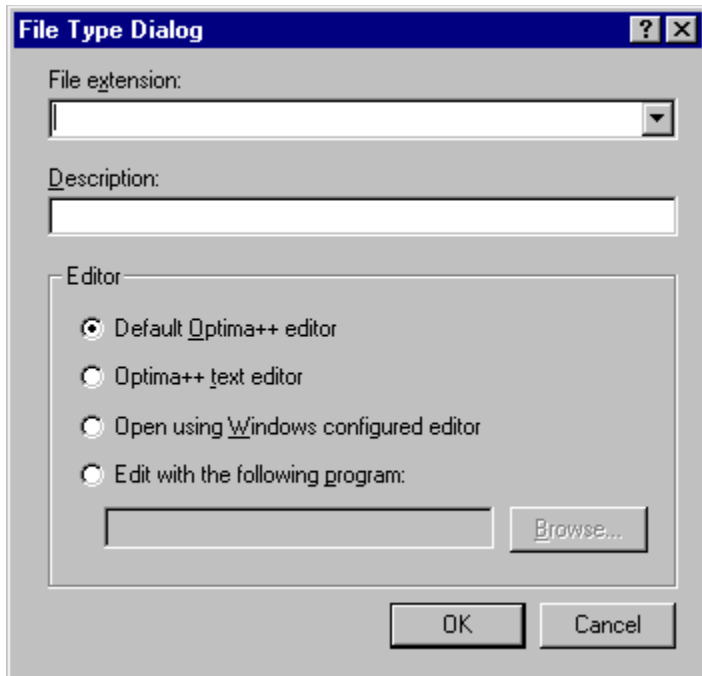
◆ To see the current file type options:

1. From the **Tools** menu on the main Jato 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. Possibilities include:

Default Jato editor

Use whatever editor Jato uses by default.

Jato text editor

Forces the use of the Jato text editor itself.

Open using Windows configured editor

Use the editor given by the system registry.

Edit with the following program

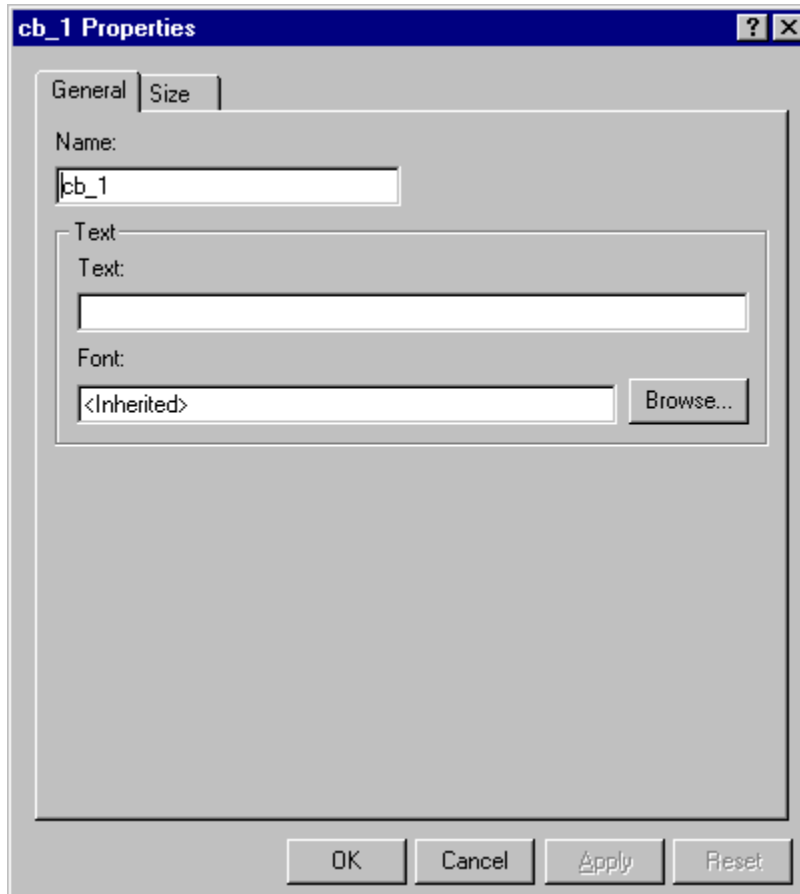
Gives a specific program that should be used when opening such files.

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

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, Jato 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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 3. Using Jato](#)

[Changing an object's properties](#)

Changing an object's name

Jato 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, Jato changes the corresponding names of all associated event handlers. For example, if you change the command button's name from `cb_1` to `okButton`, Jato automatically changes the button's **Click** handler from `cb_1_Click` to `okButton_Click`.

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

```
cb_1->setText( "Hello" );
```

into

```
okButton->setText( "Hello" );
```

However, Jato 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
```

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

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

Jato 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 Jato 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.

<p>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.</p>

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 **Click** 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, Jato displays a *code editor* which you can use to write a routine to handle that event.
3. If you click **More**, Jato 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.

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

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 **Click** event handler for the button, and ignore all the other possibilities.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 3. Using Jato](#)

[Adding and modifying event handlers](#)

The Jato code editor

The Jato code editor helps you write Java source code. Jato 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 and header files.

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

```
public boolean cb_1_Click( ClickEvent event)
{
    return false;
}
```

The most important part of this skeleton is the prototype for the handler. This is a standard Java function prototype giving several pieces of information:

- A function name constructed from the name of the object (`cb_1`) and the name of the event (**Click**).
- 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 **Options** 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 **Options** menu also lets you change the tab stops and character font used in displaying code. For more information, see the **Options** menu and press F1 to obtain a complete description of these facilities.

Saving source code

The usual method of saving source code is to click the **Save Project** button on the main Jato toolbar, or to click **Save Project** on the **File** menu of the main Jato 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, Jato 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 Jato 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, Jato 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 Jato as well as code that you wrote yourself. Code that is generated by Jato is *read-only*; you cannot change it with the editor. Read-only code is displayed in gray to distinguish it from normal code.

[!\[\]\(08a82c22d89d6b027ff69762ad096586_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(35e4f762fc1cfea5610d92e2d225d5b4_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(d84e7ea36f695d92cb39ec32c307ac93_img.jpg\) Chapter 3. Using Jato](#)

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 source files that are needed by the code in a class.

[!\[\]\(83f22ed94ec5517769dd76d702c6bfd8_img.jpg\) Adding new member functions](#)

[!\[\]\(8d0f0e0fe25b320c33272c52aec1fbca_img.jpg\) Renaming a member function](#)

[!\[\]\(642aa997563f9a325b310230bb5078b7_img.jpg\) Changing the prototype for a member function](#)

[!\[\]\(2b376d1a92330ab09dad2665d2f89bf5_img.jpg\) Adding data members to a form definition](#)

[!\[\]\(3cb60d42b10e53f9522bb0b392c1c4cd_img.jpg\) Deleting items](#)

[!\[\]\(d0262bbe9d2356661a2e89321dfcc781_img.jpg\) Importing source files](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 3. Using Jato](#)

[Working with classes](#)

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.

◆ **To create a new member function for a form:**


1. From the **File** menu of the Classes window, click **Create User Function**.
2. Choose the scope of the new function (private, protected, or public).
3. Type the prototype for the function in **Function Prototype**.
4. Click **OK**.


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


The **Create User Function** menu item is also available through the **File** menu of the code editor.

Important: You cannot define an inline function this way; the technique only works for normal (non-inline) functions. If you want to define an inline function, use the Classes window to open the **Class Declarations** for the form, and add the inline function definition in the class declarations.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Working with classes](#)

Renaming a member function

If you add a member 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 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 member functions that you have created explicitly with **Create User Function**. For example, 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Working with classes](#)


Changing the prototype for a member function


If you add a member function to a form class, you can change the prototype for the function through the Classes window.


◆ **To change the prototype of a member function previously added:**

1. In the Classes window, use the right button to click on the name of the function you want to rename, then click **Open Prototype**. Jato opens a code editor window showing the prototype declaration in the `public`, `private`, or `protected` declarations at the beginning of the form class.
2. Change the prototype declaration as desired.
3. Using the same code editor window, search for the function definition, then change the prototype that begins the function definition.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Working with classes](#)


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. Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)


 [Working with classes](#)


Deleting items


Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Working with classes](#)

Importing source files

The Classes window lets you specify definition files that should be imported when you compile a class.

◆ **To import a file for a form definition:**


1. In the Classes window, click the name of the item that needs to import a file, then double-click **Imports** for that item.
2. In the resulting window, enter an appropriate `import` directive to import the file.


For example, if you want to include the header file for Form2, type:

```
import form2;
```

(This assumes that Form2 is in the same Java package as the class that is importing Form2.)


 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)


 [Chapter 3. Using Jato](#)

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](#)


 [Methods that return values](#)

 [The object prefix](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

 [Using drag-and-drop programming](#)

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.

<p>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.</p>

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.

[Jato Programmer's Guide](#)

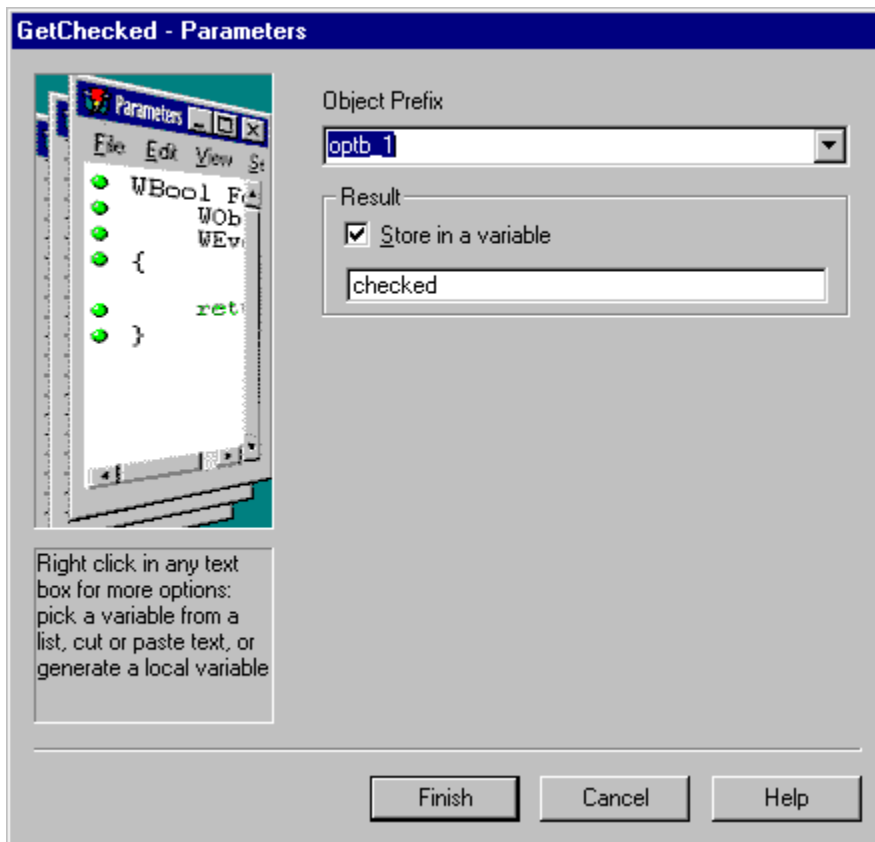
[Part I. Fundamentals](#)

[Chapter 3. Using Jato](#)

[Using drag-and-drop programming](#)

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 **getChecked** to determine if an option button is checked. The Parameter Wizard displays the following:



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

```
boolean checked;  
checked = optb_1.getChecked();
```

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

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 3. Using Jato](#)

[Using drag-and-drop programming](#)

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, Jato 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,

```
textb_1.setBackgroundColor( Color.blue );
```

sets the background color of the specified text box, but

```
setBackgroundColor( Color.blue );
```

sets the background color of the form itself.

Jato command line options

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

`-bt "Debug" -b file.wxp`

Immediately builds a `Debug` version of the project specified by the `.WXP` file.

`-bt "Release" -b file.wxp`

Immediately builds a `Release` version of the project specified by the `.WXP` file.

`-b file.wxp`

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

`-c file.wxp`

Deletes all the generated files associated with the project specified by the `WXP` 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 Jato and immediately builds a `Release` version of a project:


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


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

```
start optima -c c:\optima\projects\myproj\myproj.wxp
```

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

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 3. Using Jato](#)

Summary of basic Jato operations

Jato often provides several ways of performing the same operation. This guide makes no effort to list all the possibilities; for complete information, see the online help.

Creating a form

To add an object to a form, click on the appropriate button in the Java component palette, then click on the form shown in the form design window. After you have placed an object on the form, you can change its position by dragging it to a new location on the form. You can change its size by clicking on the object, then using the drag handles to drag the object to the desired size.

To set the properties for the form or any object on the form, use the right mouse button to click on the form or object, then click **Properties**.

To define an event handler for the form or any object on the form, use the right mouse button to click on the form or object and click **Events**. Choose the appropriate event from the short list of possibilities, or click **More** to call the Object Inspector, then double-click the event name on the **Events** page. Jato opens a code editor window where you can type Java code.

If you change an object's name, the change is propagated throughout the form that contains the object.

The Classes window lets you add new member functions to a form as well as data object declarations and `import` directives.

Templates

A template consists of one or more objects which already have their properties set and/or event handler routines defined. You can save yourself work by defining templates for objects or object combinations that you use frequently. Object templates are available on the **Templates** page of the Java component palette.

You can also define form templates. For example, you could define a template for dialog operations that you perform frequently. This makes it easy to incorporate such a form into any program you write. Form templates are available through the Form Wizard.


Drag-and-drop programming

Drag-and-drop programming simplifies the creation of Java source code. Drag from an object on the form to an open code editor. Jato displays the Reference Card to show the methods available for the chosen object. When you have chosen a method, the Parameter Wizard helps you construct a call to that function.

Command line options


When you start Jato, you can specify command line options that tell Jato which project you want to work with. You can also specify whether you want a Debug or Release version of a target. Finally, you can tell Jato to clean out all files which are built from other files.


 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)

Chapter 4. Standard types and events


This chapter examines a number of basic classes used by the Jato component library. Many of these come from the AWT library, while others are unique to Jato. The chapter also explains the principles of dealing with Jato events.

 [Frequently used types](#)

 [The Object class](#)

 [The String class](#)


 [StringBuffer](#)

 [The Point class](#)

 [The Dimension class](#)

 [The Rectangle class](#)


 [The Color class](#)


 [The Font class](#)

 [The Toolkit class](#)

 [The URL class](#)


 [The Applet class](#)

 [The Range class](#)

 [Standard events](#)

 [Summary of standard types and events](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

Frequently used types

The Jato component library provides a class for every kind of object that can appear on a form. For example, text boxes are represented by the `TextBox` class and command buttons are represented by the `CommandButton` class.

The Jato 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 Jato. This includes some standard Java library classes as well as Jato classes.

For complete details on all Jato classes, see the [Jato Component Library Reference](#).

 [Fully qualified Java names](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[Frequently used types](#)

Fully qualified Java names

The code you write in Jato draws on many different libraries (for example, the standard Jato library and AWT). In some cases, there are conflicts between names used in these libraries. For example, both the AWT and the Jato library have a `Label` class.

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 two `Label` classes have the following fully qualified names:

```
java.awt.Label
powersoft.jcm.ui.Label
```

As shown, the first name specifies that it comes from the AWT, while the second from Powersoft Jato.

All fully qualified names in the Jato library start with `powersoft.jcm` (where `jcm` stands for Java Component Module). Within this library are several subsets:

<code>db</code>	Database-related classes
<code>event</code>	Event-related classes
<code>net</code>	Internet components
<code>ui</code>	User interface components
<code>util</code>	Utility classes

To identify where a class is defined, look in

```
java\Lib\powersoft\jcm
```

in your main Jato folder. Each group of classes (`db`, `event`, etc.) appears as a subfolder under this folder. 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.jcm.util.*
```

`import`s information about all the utility classes defined for Jato. (Remember that names used in `import` statements are case-sensitive.)

You will see fully qualified names in much of the code generated by Jato and in many of the examples in this guide.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

The Object class

The Object class (`java.lang.Object`) is the base class for all other AWT classes in Java and Jato. 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 Jato class.

The standard constructor for an Object is

```
Object obj = new Object();
```

This creates an object with no contents.

<p>Note: C and C++ programmers should note that Java typically uses the Object class in situations where C/C++ would use void *.</p>

 [Comparing Object values](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The Object class](#)

Comparing Object values

The **equals** method compares two Object values to determine if they refer to the same object. This is not the same as referring to identical objects. For example, in

```
Object oldObj = new Object();  
Object newObj = oldObj;
```

`oldObj` equals `newObj` because they both refer to the same object (the same instantiation). However, in

```
Object oldObj = new Object();  
Object newObj = new Object();
```

the two are not equal because they refer to different objects.

The standard form for using **equals** is

```
boolean result = object1.equals( object2 );
```

The result is `true` if both values refer to the same object and `false` otherwise.

No object is equal to the special `null` object.

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

The 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 constant; 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](#)).

<p>Note to C and C++ programmers: The text stored in a String object normally does not end in '\0'. Of course, there may be a '\0' present if you created the String from C or C++ string data.</p>
--

[Creating a string](#)

[Simple String methods](#)

[Comparing strings](#)

[Obtaining substrings of strings](#)

[Comparing substrings](#)

[Searching strings](#)

[Converting an object to a string](#)

[Converting data into strings](#)

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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.

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 ASCII value is less than or equal to `\u0020` (the space character).

The **replace** method replaces all the occurrences of one character with a new character:

```
String old = "Robin" ;  
String new = old.replace( 'i', 'y' );
```

produces "Robyn".

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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`.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 4. Standard types and events](#)

[_The String class](#)

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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** includes the name of the class to which the object belongs. There is one exception: when you apply **toString** to a String object, you get the string object itself. For example,

```
String str1 = "abc";  
String str2 = str1.toString();
```

assigns `str1` itself to `str2`.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The String class](#)

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]`.

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

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.

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.

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`.

[Creating a string buffer](#)

[Converting a string buffer to a string](#)

[Simple string buffer methods](#)

[Appending text to a string buffer](#)

[Inserting text into a buffer](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[StringBuffer](#)

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" );
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [StringBuffer](#)

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"
```

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[StringBuffer](#)

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".

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[StringBuffer](#)

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](#).

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[StringBuffer](#)

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](#).

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)


The 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
```


 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

The 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.

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

The 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](#)

[Changing a rectangle's size and position](#)

[Points and rectangles](#)

[Unions and intersections of rectangles](#)

[!\[\]\(cead67df4d82d6c83effe4f8699a7d8f_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(1d3a1175dd4902218e694b9c098adb83_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(c507f772dba2b921f86777f01218e570_img.jpg\) Chapter 4. Standard types and events](#)

[!\[\]\(4729e517bc6a7cd81c8025b9646574fb_img.jpg\) The Rectangle class](#)

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).

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The Rectangle class](#)

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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The Rectangle class](#)

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 determines the smallest rectangle that contains both the original rectangle and a given point:

```
Rectangle newRec = rec.add( x, y );
```

This gives the smallest rectangle that contains both `rec` and the point (x,y) . You can also specify the point as a Point value:

```
//Point pt;  
Rectangle newRec = rec.add( pt );
```

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The Rectangle class](#)

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 );
```

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

The 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.

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 white has a value of zero for the red, green, and blue components. The color black has a value of 255 for the red, green, and blue components.

[Constructing colors](#)

[Color components](#)

[Darker and brighter colors](#)

[The HSB color model](#)

[Color names](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The Color class](#)

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.

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The Color class](#)


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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The Color class](#)

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 increased 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 decreased 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.

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[The Color class](#)

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.

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The Color class](#)

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
```

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

The 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](#)

[Constructing fonts](#)

[Simple font methods](#)

[The FontMetrics class](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The Font class](#)

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.

[!\[\]\(08a82c22d89d6b027ff69762ad096586_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(35e4f762fc1cfea5610d92e2d225d5b4_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(d84e7ea36f695d92cb39ec32c307ac93_img.jpg\) Chapter 4. Standard types and events](#)

[!\[\]\(feabb98897b440bc8695a03336a6e2df_img.jpg\) The Font class](#)

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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The Font class](#)


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.

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)


 [_Chapter 4. Standard types and events](#)

 [_The Font class](#)

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.

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

The 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");
```

[Java Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

The 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.

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.


This section describes a number of simple methods associated with URLs. For more detailed information about working with URLs, see Writing Internet applications .
--

[Constructing a URL](#)


[Obtaining information from a URL](#)

[Converting a URL to a string](#)

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The URL class](#)

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" );
```

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [The URL class](#)


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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)


 [The URL class](#)

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();
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

The Applet class

The Applet class of Jato (`powersoft.jcm.ui.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.


The **resize** method lets the applet request a new size for the current form. 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).

<p>The Applet class supports a number of other methods directly related to the Applet class of Java itself. For more information, see the Jato Component Library Reference.</p>

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

The Range class

The Range class (`powersoft.jcm.ui.Range`) is defined by Jato 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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

Standard events

When you create a program using Jato, you write source code based on the *events* that can happen to objects.

Jato 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 **Click** 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 Jato run-time environment. In most cases, the default handling simply cleans up after the event, making it look like the event was ignored.

Note: The event model of Jato is based on the JavaBeans model. This release of Jato only handles selected JavaBeans events, but future releases may supply greater support for JavaBeans events.

[The EventData class and its derivations](#)

[Event handlers](#)

[Default handling](#)

[Underlying mechanisms](#)

[Chaining event handlers](#)

[Adding a new event handler](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[Standard events](#)

The `EventData` class and its derivations

`EventData` objects (`powersoft.jcm.event.EventData`) contain information about events. For example, when a **Click** event occurs on a command button, the Jato library creates an object derived from the `EventData` class, providing information about the **Click** event. In particular, the information would tell which button was clicked.

Different types of events may have different types of information associated with them. For example, if the user selects an item in a list box, the event information should include which item was selected. In order to provide this kind of extra information, Jato uses various classes derived from the basic `EventData` class. For example, the event information associated with a **Select** event is stored in a `SelectEvent` object; the `SelectEvent` class is derived from `EventData` but contains extra information specific to **Select** events.

Since all event information blocks are based on `EventData`, all these classes have certain features in common. The rest of this section examines those common features.

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 `CommandButton` object corresponding to the command button that was clicked.

The `RawEvent` property

Many (but not all) of the events triggered in Jato result from an AWT event. If a Jato event results from an AWT event,

```
Object awtEvent = event.getRawEvent();
```

obtains the event information associated with the AWT event. If the Jato 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 Jato programmers will never have to worry about this, because Jato automatically handles the low level details for you.

The `Handled` property

The **Handled** property 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 handler](#).

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[Standard events](#)

Event handlers

The code that responds to a particular event is called an *event handler*. Event handlers are usually method functions defined in the class that represents a form.

All event handler routines have similar function prototypes:

```
public boolean object_eventName( EventData event )
```

object is the name of the object and *eventName* is the name of the event that the function handles. For example, an event handler that responds to a **Click** event on a button named `cb_1` would normally have the prototype

```
public boolean  
    cb_1_Click( powersoft.jcm.event.ClickEvent event )
```

The `event` argument always has a class type derived from `EventData`. In the above example, this type is called `ClickEvent`. Different types of events use different types of `event` arguments, but all such arguments have types based on `EventData`.


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, the event handler is expected to return a boolean result. A result of `false` indicates that Jato 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`, Jato 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.

<p>If your event handler returns <code>true</code>, 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.</p>
--

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 4. Standard types and events](#)

 [Standard events](#)

Default handling

Jato 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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 4. Standard types and events](#)

[Standard events](#)

Underlying mechanisms

The Jato event interface is analogous to the JavaBeans 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 type of object wants to receive **Click** events. In Jato, the declaration of the object must state that it implements the ClickListener class (`powersoft.jcm.event.ClickListener`). This indicates that the object intends to listen for **Click** events. The object must also include a method named **click**. This function is invoked when a **Click** event is received by the object. The argument to the **click** function is a ClickEvent object providing information about the click action.

Similarly, if an object wants to receive **Select** events, it implements a class called SelectListener and must contain a method named **select**. The **select** method accepts a SelectEvent object as its argument. A similar pattern holds for all other recognized events.

It is important to understand that an event listener is an <i>object</i> , not a function. The event listener object contains a function which is invoked when the event listener is notified of that type of event.
--

Jato supports two possible ways of implementing this model: the form-based model and the relay model. These models are described in the sections that follow.

The form-based model

When you define an event handler for an object in this model, Jato automatically declares the form containing that object to be a listener for that event. For example, if you define a **Click** event handler for command button `cb_1`, Jato declares that the form containing this button implements ClickListener. Jato also creates a **click** method in the form class that will invoke `cb_1_Click` when the user clicks `cb_1`.

Now consider what happens if the form contains more than one command button. When a click action occurs anywhere on the form, the form's **click** method is invoked. The **click** method must then decide which button was clicked and invoke the appropriate event handler associated with that button.

While the form-based model is simple, it may be inefficient for a single **click** method to handle click actions anywhere on the form. If the form contains lots of buttons, the **click** method may have to perform a number of `if` statements to determine which button was clicked and which event handler should be invoked.

The relay model

The relay model associates an event listener with each individual object that can receive the event. For example, suppose a form contains command buttons `cb_1` and `cb_2`. Then the relay model creates new classes named `cb_1_Relay` and `cb_2_Relay`, each of which implements ClickListener. Each of these classes contains its own **click** method. Each of these **click** methods invokes the appropriate event handler for the associated command button. For example, the **click** method in `cb_1_Relay` invokes `cb_1_Click`.

The advantage of the relay model is that it avoids a single **click** method that serves the whole form and has to figure out which button has been clicked. The disadvantage of the relay model is that it requires the definition of more Java classes. This may increase the file space needed to store your project and possibly the amount of memory used by your program during execution.

Chaining event handlers

It is possible to specify more than one event handler for the same event. For example, consider the **Select** event for list box `lb_1`:

- You may start out with a single general event handler `lb_1_Select` which handles **Select** events in a simple way.
- During execution, the user selects an option that makes it necessary to handle the **Select** event for `lb_1` in a more comprehensive way. Therefore you specify a new event handler `lb_1_SelectSpecial` to deal with the **Select** event. This handler is placed before `lb_1_Select` in the event handler chain. Therefore, when the **Select** event is triggered, `lb_1_SelectSpecial` is invoked. This routine can handle the event as appropriate; however, `lb_1_SelectSpecial` can also decide not to handle the event, in which case the next handler in the chain (`lb_1_Select`) is invoked.
- Later in execution, you may decide to specify yet another event handler, one that is specially tuned for quick performance in a single special circumstance. Call this one `lb_1_SelectFast`. You add this handler to the chain too. Now, when the **Select** event is triggered on `lb_1`, the first routine invoked is `lb_1_SelectFast`. If this routine doesn't handle the event, control passes to `lb_1_SelectSpecial` and then `lb_1_Select`, as necessary.

If the last user-defined handler in the chain (`lb_1_Select`) does not handle the event either, control passes to the default handling supplied by Jato itself. Therefore, the default handling can be regarded as the end of the event handler chain. (The default handling may or may not be implemented as an actual event handler function. Instead of being a separate function, the default handling may be part of the routine that walks through the chain of event handlers.)

Adding a new event handler

The process of adding a new event handler has several steps:

- Defining a new event listener class
- Adding an event listener object to the current list of event listeners

In order to understand this process, it may be useful to examine the code that Jato 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 first 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 **Click** listener class. Listeners for other events follow a similar format. For the sake of example, this **Click** listener is intended to invoke a method named `cb_1_NewClick` defined for a command button on Form1:

```
class MyClickListener
    implements powersoft.jcm.event.ClickListener
{
    public MyClickListener( Form1 form )
    {
        _form = form;
    };
    public void click( powersoft.jcm.event.ClickEvent ev )
    {
        if ( _form.cb_1_NewClick( ev ) )
            ev.setHandled( true );
    };
    private Form1 _form;
};
```

In order to use standard JavaBeans tools (for example, “introspection”), the name of the class should end in the string `Listener`.

The constructor for `MyClickListener` receives `Form1` argument indicating the form that contains the command button. The constructor saves a copy of this argument so that this `MyClickListener` can refer to the form later.

The `click` method invokes `cb_1_NewClick` for the specified form. It passes the `ClickEvent` data as an argument to `cb_1_NewClick`. If `cb_1_NewClick` returns `true`, the `click` method uses `setHandled` to indicate that the event has been handled.

Adding an event listener to the list

Every Jato component object may have a list of event listeners which listen for events on that object. For example, a command button may have a list of click listener objects listening for **Click** events on that button.

When an object can receive an event, 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 `addClickListener` while a list box has a method named `addSelectListener`.

The following example shows how to add a new **Click** listener for a command button named `cb_1`:

```
cb_1.addClickListener( new MyClickListener( this ) );
```

Notice that this function call specifies an argument of `this` when creating the new `MyClickListener` object. This creates a **Click** listener which refers to the current form. Once the `MyClickListener` object has been added to the list of **Click** listeners for `cb_1`, it waits for a **Click** event to happen. If the event happens, the run-time environment invokes the `click` method in the **Click** listener which in turn invokes `cb_1_NewClick` for the current form. Presumably this method handles the **Click** event in the context of the current form.

Objects also have “remove listener” functions to remove an existing event listener. For example, command buttons have a **removeClickListener** method while list boxes have a **removeSelectListener** method. For more information, see the Jato Component Library Reference.

Summary of standard types and events

Standard types

This chapter discussed the following standard types:

Object

A generic type underlying all AWT classes.

String

String constants and strings whose contents cannot be changed.

StringBuffer

Strings whose contents can be changed.

Point

Points on a form.

Dimension

The dimensions of a rectangular area (width and height).

Rectangle

The size and position of a rectangle on a form.

Color

Screen colors.

Font

Type fonts.

Toolkit

Provides access to general-purpose methods.

Applet

Stands for the applet as a whole, and provides a way to access various types of application-wide data.

Range

A range of integer values.

Events

An event is triggered to inform the program that the user has performed some action (such as clicking a button) or that some other piece of software has a message to communicate. When an event is triggered, the Jato run-time environment calls an event handler written to handle that event.

An event handler has the prototype

```
public boolean object_event(EventData event );
```

where *event* is a block of information about the event. The *event* argument has a type derived from the EventData class. The derived classes have names like ClickEvent, SelectEvent, and so on. These derived classes may provide specific data unique to the event; for example, SelectEvent has a method that tells which item was selected.


There may be several event handlers associated with a given event. The event handlers form a “chain” of functions which are executed in reverse order of registration: the most recently registered event handler is executed first, then the next most recent, and so on.

The return value of an event handler should be `true` if the event has been completely handled and `false` otherwise. If the return value is `false`, execution continues with the next event handler in the chain; if the return value is `true`, the event is considered to be completely handled and no further event handlers are called.

Jato provides default handling for every event. If you do not define an event handler for a particular event, or if every event handler in the chain returns `false`, Jato performs the default handling. Typically, the default handling simply ignores the event; however, some types of events require more elaborate default handling.


Most of the event handlers you write should return `false`. In this way, control passes from your event handler to the default event handling provided by Jato (which performs any clean-up required after the event).


 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)

Chapter 5. Programming standard objects


This chapter examines the standard components that can be placed on Jato forms. This includes all the components that appear on the **Standard** and **Utilities** pages of the Java component palette. The chapter assumes you have read [Standard types and events](#).

 [Base classes in Jato](#)

 [Object properties](#)


 [Notes on writing Jato code](#)


 [Labels](#)


 [Command Buttons](#)


 [Picture buttons](#)

 [Check boxes](#)


 [Option buttons](#)


 [Picture boxes](#)

 [List boxes](#)


 [Combo boxes](#)

 [Text boxes](#)


 [Group boxes](#)


 [Scroll bars](#)

 [Timers](#)

 [Summary of standard objects](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

Base classes in Jato

The Component class is the root class underlying all Jato components. Classes like CommandButton, ListBox, and so on are all derived from Component.

The Container class 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 is based on Container, and is the root class for all containers that can be contained by other containers.

The PopupWindow class is the root class for all objects that can be windows on their own (for example, forms but not panels that must be contained by other containers).

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard 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 **Error! Reference source not found.**

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 **getChecked** and **setChecked** methods. A read-only property will only have a **get** method.

In Jato, **get** and **set** are written in lower case letters. Contrast this with Powersoft Optima++ (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 Jato Component Library Reference.

Note: The Jato Component Library Reference documents the **get** and **set** methods for a property in the entry for the property itself. For example, **getText** and **setText** are documented in the reference manual entry for the **Text** property.

[Text](#)

[Font](#)

[Visible](#)

[Enabled](#)

[Color](#)


[Parent](#)


[Size and position properties](#)


[ResizePercentages properties](#)

[Database properties](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)

Text

The **Text** property is a string of text used to label a visual object. Different objects are labeled in different ways. For example, the text of a button is displayed on the button itself, while the text of a text box is displayed inside the text box.

With some types of objects, the text is not displayed at all. You can still assign these objects a **Text** value, but it has no effect on what the user sees.

An object's initial **Text** 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 text is displayed: click the **Browse** button to choose a font.


Changing Text at run time


During execution, the **setText** method changes the **Text** property of an object. For example, suppose that `cb_1` refers to a command button; then


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

changes the text on the command button to the given string.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)


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 box `textb_1`:


```
Font oldF = textb_1.getFont( );
Font newF =
    new Font( oldF.getName(), oldF.getStyle(), oldF.getSize()+2 );
textb_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 [The Font class](#).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)


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, Jato makes it visible by default. During program execution, your code can use the **setVisible** method to make the object appear or disappear:

```
object.setVisible( true );    // now you see it
object.setVisible( false );  // now you don't
```


 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)

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 a **Click** event handler for the button.


When an object is disabled, its appearance is changed to show that it is unavailable. For example, a command button typically has its **Text** string 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, Jato turns on **Enabled** by default. During program execution, your code can use the **setEnabled** method to enable or disable the object:

```
object.setEnabled( true );    // now it's enabled
object.setEnabled( false );   // now it's disabled
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)

Color

Most objects have two associated colors: a *foreground color* and a *background color*. In a text box, for example, the box'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 **ForeColor** or **BackColor**. 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.


You can also change colors by clicking the **Browse** button beside **ForeColor** or **BackColor**. 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 **setForeColor** and **setBackColor**, as in


```
textb_1.setForeground( Color.blue );  
textb_1.setBackgroundColor( Color.white );
```

For further information about these properties, see the Jato Component Library Reference.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)

Parent

Objects may contain other objects. For example, a form contains all the buttons, text boxes, 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 = textb_1.getParent( );
```

determines the parent of `textb_1` as a `Container` object. Usually you would cast this to a form object, as in

```
Form1 par = (Form1) textb_1.getParent( );
```

You can also use **setParent** to change the parent of an object. **setParent** removes the object from its current parent (if necessary), then adds the object to the specified parent. This process is rarely necessary.

Size and position properties

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, Jato 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, Jato 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 Jato 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 `getRectangle` (described later in this section). You cannot just assume that the numbers will be the ones you set at design time.

Note: Jato 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.

You can also change the object's size and position 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.

Setting size and position at run time

The `Rectangle` property specifies size and position as a single `Rectangle` value. For example,

```
Rectangle r = cb_1.getRectangle( abs );
```

returns a rectangle value showing the object's size and position in pixels. The `abs` argument is a boolean value. If it is `true`, the position is given relative to the user's screen as a whole; if it is `false`, the position is given relative to the window containing the object in question. Therefore,

```
Rectangle r = cb_1.getRectangle( false );
```

determines the size and position of `cb_1` relative to the form that contains the object. If you omit the argument, as in

```
Rectangle r = cb_1.getRectangle( );
```

the default is to give position relative to the containing window (which is like specifying `false` as the argument).

The **setRectangle** method changes the size and position of an object on the form. For example,

```
Rectangle r = cb_1.getRectangle( );
r.translate( 10, 0 );
cb_1.setRectangle( r );
```

shifts the position of the command button `cb_1` ten pixels to the right. This happens because the **translate** method of `Rectangle` moves the rectangle's position by the given number of pixels. By repeating this sequence of instructions, you can eventually make the button walk off the edge of the form.

Individual settings for size and position

The **Rectangle** property specifies the size and position of an object as a single `Rectangle` object. You can also specify size and position components separately, using the following properties:

Left

The X coordinate of the left side of the object (in pixels).

Top

The Y coordinate of the top of the object (in pixels).

Width

The width of the object (in pixels).

Height

The height of the object (in pixels).

There are **set** and **get** methods for each of these properties. For example, the following code determines the height of text box `textb_1` and sets the height of text box `textb_2` to the same size:

```
int h = textb_1.getHeight( );
textb_2.setHeight( h );
```

Dialog unit measurements

The following methods convert measurements in dialog units into pixels:

```
// int duWidth, duHeight, x, y, width, height;
// Rectangle duRect;
int pixelWidth = DUWidth( duWidth );
int pixelHeight = DUHeight( duHeight );
int pixelRect1 = DURectangle( duRect );
int pixelRect2 = DURectangle( x, y, width, height );
```

These methods make it possible for you to specify heights and widths in dialog units at run time. For example, suppose you want the height of a text box to be 100 dialog units. You could write

```
textb_1.setHeight( DUHeight(100) );
```

to convert 100 dialog units into pixels and then to set the height of the text box to that number of pixels.

DUWidth, **DUHeight**, and **DURectangle** are methods defined in the `Jato Component` class. In particular, they are defined as methods in every form class, so they can be used as shown above in any code with a form definition (including event handler code). If you need to use the methods in code that does not belong to any class based on `Component`, you may need to specify a component explicitly, as in

```
int pixelWidth = form1.DUWidth( duWidth );
```

The **DURectangle** property gives the original dialog unit dimensions that you set for the object at design time:

```
Rectangle duRect = textb_1.getDURectangle( );
```

In this one case, the values returned to the `duRect` `Rectangle` specify measurements in dialog units rather than pixels.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Object properties](#)

ResizePercentages properties

The **ResizePercentages** properties control how objects on a form behave when the size of the form changes. These properties are specified on the **Size** page of each object's property sheet. The property sheet lists four resize percentages:

Left Percentage

Controls how far the left edge of the object moves when the form's size changes.

Top Percentage

Controls how far the top of the object moves when the form's size changes.

Width Percentage

Controls how much the width of the object changes when the form's size changes.

Height Percentage

Controls how much the height of the object changes when the form's size changes.

Each of these properties 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 **ResizePercentages**.

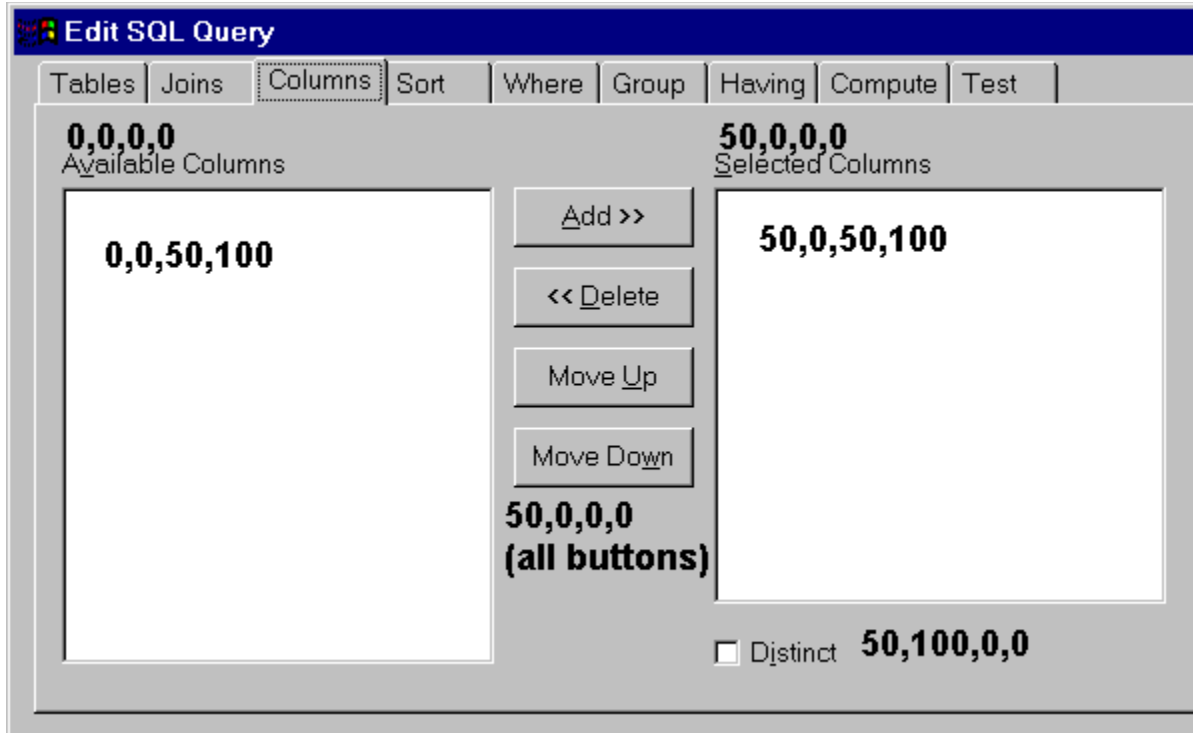
At run time, you can use **getResizePercentages** to determine the current resize percentages and **setResizePercentages** to change the values. **ResizePercentages** are expressed as a Rectangle value, as in

```
Rectangle r = textb_1.getResizePercentages( );
```

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

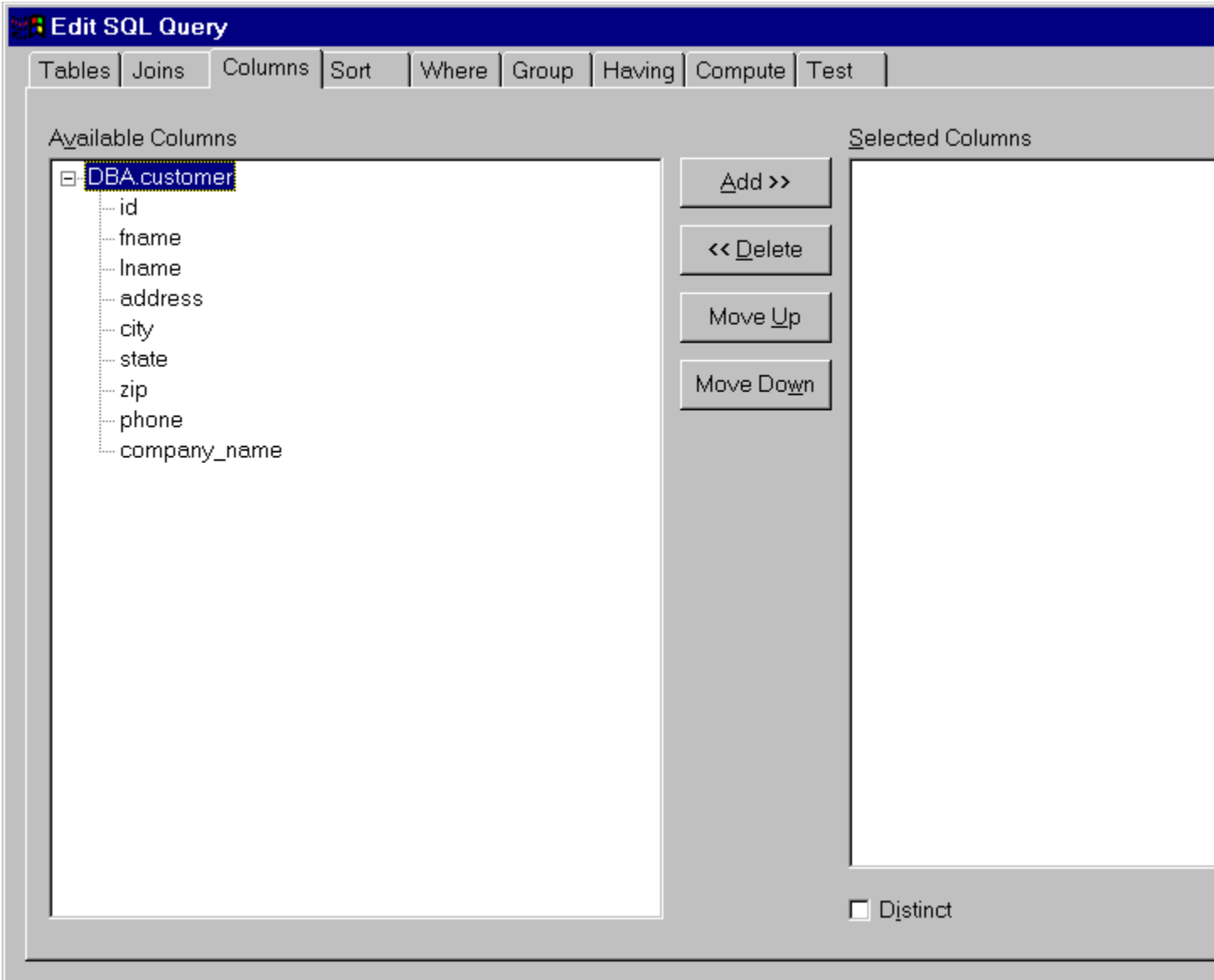
As an example of how this works in practice, the following picture shows a sample form with resize percentages displayed in bold face (in the order, **Left, Top, Width, Height**):



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 **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 tree view on the left and a list box 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 box under the **Selected Columns** label also has its width increase by 50% of the total width increase, while the height of the list box 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 box, and any change in height makes the same change in the height of the tree view and the list box. 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 box increase in size.


Proportional resizing


If you set any of the **ResizePercentages** 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 text box to -1 . If the width of the form doubles, the width of the text box will also double.

Obviously, the easiest way to deal with resizing is to set every object's **ResizePercentages** 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Object properties](#)

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](#).

[!\[\]\(cead67df4d82d6c83effe4f8699a7d8f_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(1d3a1175dd4902218e694b9c098adb83_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(c507f772dba2b921f86777f01218e570_img.jpg\) Chapter 5. Programming standard objects](#)

Notes on writing Jato code

This section offers a few hints on writing code for Jato programs.

[!\[\]\(a03a7eb2f4046e1d3c76772003e549ea_img.jpg\) Object names](#)

[!\[\]\(cbe2492b119e39e02a1dab2af4a4b296_img.jpg\) Object initialization](#)

[!\[\]\(e474458956c9a37fbf9586ddb60a7fa1_img.jpg\) Referring to the form](#)

[!\[\]\(3e2231b1ad3ca8da8658228c00dd08e0_img.jpg\) Focus](#)

[!\[\]\(5361750c22c4e047a52f4eac1ec2d4cc_img.jpg\) Opening a new form](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Notes on writing Jato code](#)

Object names

Jato 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 Jato components. You therefore use the `.` notation to refer to properties and methods associated with the objects, as in:

```
cb_1.setText( "OK" ); // change text
```

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 where the pointer variables are defined.
2. Under **Scope of Controls** on the **General** page, click **Public**.
3. Click **OK**.

This specifies that the pointer 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. The 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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Notes on writing Jato code](#)

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 **Create** event handler for the object. The **Create** event takes place just before the object is displayed to the user. For example, the **Create** event for a list box takes place just before the list box is displayed. Therefore, you might write a **Create** event handler to initialize the list box (for example, to set up the initial items displayed).
2. You can initialize an object in the **Create** 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 **Create** events for separate objects taking place in any particular order. For example, you cannot be sure that the **Create** handler for one object is executed before the **Create** 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 **Create** 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 **Create** handler for the form. That way you can write the initialization instructions in the required order.

The **Create** event for a form takes place after the **Create** events for each object on the form. Usually, these **Create** 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 **Create** events; for more information, see [Event timing](#).

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_Notes on writing Jato code](#)

Referring to the form

In the code for an event handler routine, the C++ keyword `this` stands for a pointer to the form. For example,

```
textb_1.setParent( this );
```

makes the current form the parent of `textb_1`. Similarly,

```
this.setText( "My Pretty Form" );
```

sets the title for the form. The preceding function call could also be written

```
setText( "My Pretty Form" );
```

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,

```
textb_1.setBackgroundColor( Color.blue );
```

sets the background color of the given text box, but

```
setBackgroundColor( Color.blue );
```

sets the background color of the whole form.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Notes on writing Jato code](#)

Focus

Focus is the ability to receive input from the keyboard or from the mouse. Only one object on a form can have focus at any time. For example, if a form contains three text boxes, only one of them can have the focus. When the user types input from the keyboard, the resulting text appears in the text box that has the focus. In order to type into another text box, the user must move the focus from the old box 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 **setFocus** method. For example, the following code changes the focus for a text box named `textb_1`:

```
textb_1.setFocus( true );      // gives box the focus
textb_1.setFocus( false );    // removes the focus
```

Many versions of Java 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 gets 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.

[!\[\]\(082f818d99f166a3ba574d9284d73064_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(d263118e0bfd47dc6bc704167d936b83_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(34b4f260a8587d2e97eeaee361cc357b_img.jpg\) Chapter 5. Programming standard objects](#)

[!\[\]\(3d8c13c92b853674f749aac6fa869926_img.jpg\) Notes on writing Jato code](#)


Opening a new form


The usual way of opening a new form is to create an object of the form type, and then apply the **create** method to it. For example, an object of the type Form1 could open one of the type Form2 with

```
Form2 f2 = new Form2( );  
f2.create( );
```

%%% At present, there is no way for forms to share data except by assigning values to `public` data members within each other. The final version of Jato will support the FDX data exchange mechanisms.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)


 [Chapter 5. Programming standard objects](#)


Labels

A 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. Labels are represented by Label objects (`powersoft.jcm.ui.Label`). Jato gives command buttons names of the form *label_N*. On the Java component palette, labels are represented by the following button:





A label may be a bound control for a database. For more information, see [Bound controls](#).

 [Label properties](#)

 [Labels at run time](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Labels](#)

Label properties

The following list discusses commonly used 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Labels](#)

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.setVisible( false );
```


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 );
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

Command Buttons

Command buttons are also called “push buttons”. When the user clicks a command button, the picture on the screen changes slightly to make it look like the button has been pressed.

Command buttons are represented by `CommandButton` objects

(`powersoft.jcm.ui.CommandButton`). Jato gives command buttons names of the form `cb_N`. On the Java component palette, command buttons are represented by the following button:





Command buttons support the usual properties of components (for example, **ForeColor** and **BackColor**). The **Text** property specifies the text that appears on the button.


The **Click** event is triggered on a command button when the user clicks the button.

 [A typical command button application](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Command Buttons](#)

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_Click(powersoft.jcm.event.ClickEvent event)
{
    count = (count + 1) % 3;
    switch (count)
    {
        case 1:
            setBackgroundColor( Color.red );
            break;
        case 2:
            setBackgroundColor( Color.green );
            break;
        default:
            setBackgroundColor( Color.blue );
    }
    return true;
}
```


Note that this routine is part of the `Form1` class. Therefore, the instruction


```
setBackgroundColor( Color.blue );
```

changes the background color of the current form.

Also notice that the `count` variable is declared outside the event handler routine and therefore 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

Picture buttons

A picture button is similar to a command button; however, it combines both a picture and a text caption.


Picture buttons are represented by `PictureButton` objects (`powersoft.jcm.ui.PictureButton`). Jato gives picture buttons names of the form `pictbtn_N`. On the Java component palette, picture buttons are represented by the following button:




The **Click** event is triggered on a picture button when the user clicks the button.

 [Picture button properties](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Picture buttons](#)

Picture button properties

Picture buttons support all the properties of command buttons. They have the following additional properties:

TextPosition [**General** page]

Where the text should appear in relation to the picture on the button.

Insets [**Picture** page]

The amount of space between the picture and the edges of the button (expressed in DLUs).


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.

Type [**Picture** page]

The type of picture on the button. Possibilities are `Absolute`, `CodeBased` and `DocumentBased`. `CodeBased` and `DocumentBased` are only supported for applet targets.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

Check boxes


Check boxes can be checked or unchecked to show whether an option is turned on or off. Typically, check boxes are used for options which are independent of other options. When you have a set of options that are mutually exclusive, it is common to use option buttons instead.


Check boxes are represented by `CheckBox` objects (`powersoft.jcm.ui.CheckBox`). Jato gives check boxes names of the form `checkboxbox_N`. On the Java 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](#). The **Click** event is triggered on a check box when the user clicks on the box.


 [Check box properties](#)

 [Using check boxes](#)

 [The Toggle function](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Check boxes](#)

Check box properties

The following list discusses commonly used check box properties:

Text [General page]

Specifies the text that appears to label the check box.

Checked [General page]


Determines whether the check box is checked or left blank. If you turn on **Checked** at design time, the check box is checked when the form is first displayed; if **Checked** is turned off, the check box is left blank.


You can determine whether a check box is currently checked or unchecked using **getChecked**. For example,

```
checkbox_1.getChecked( )
```

is `true` if the check box is currently checked and `false` if the box is blank.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Check boxes](#)


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 a **Click** event on the check box.

Since the box automatically toggles itself, your **Click** event handler does not need to set the state of the check box. However, the **Click** 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 **setChecked**, it does *not* trigger a **Click** event on the check box being changed.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Check boxes](#)

The Toggle function

The function

```
checkbox_1.Toggle( );
```

turns the check box off if it is currently on, and vice versa.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

Option buttons

Option buttons are typically used for groups of options which are mutually exclusive—when one button in the group is clicked, that option should be turned on and all the other options turned off.

Option buttons are represented by `OptionButton` objects (`powersoft.jcm.ui.OptionButton`). Jato gives option buttons names of the form *optb_N*. On the Java component palette, option buttons are represented by the following button:




An option button may be a bound control for a database. For more information, see [Bound controls](#).


The **Click** event is triggered on an option button when the user clicks on the button.


[Option button properties](#)

[Using option buttons](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Option buttons](#)

Option button properties

The following list discusses commonly used option button properties:

Text [General page]

Specifies the text that appears to label the option button.

Checked [General page]

Determines whether the option button is checked or left blank. If you turn on **Checked** at design time, the button is checked when the form is first displayed; if **Checked** is turned off, the button is left blank.

Group [General page]

Specifies which option button group this button belongs to. All of the buttons in a given group are considered mutually exclusive: when the user clicks on one, your program automatically marks that option button and clears all the others in the group.

When you place the first option button on a form, Jato creates an option button group for that form named `group_1`. You can create new option button groups by clicking the **New** button beside **Group** on any option button's property sheet. You will be asked to type a name for the new group.


To set the **Group** for an option button, click on an existing group name or create a new one.


You can determine whether an option button is currently marked or clear using **getChecked**. For example,


```
optb_1.getChecked( )
```

is `true` if the option button is currently marked and `false` if the button is blank.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Option buttons](#)


Using option buttons


When the user clicks an option button, the button is automatically marked and all the other buttons in the same group are cleared.

Since the buttons in the option button group automatically turn themselves on or off appropriately, your **Click** event handler does not need to set the state of the option buttons. The **Click** event handler simply responds to the user turning on the selected option. For example, suppose a group of option buttons control the background color of the form. When the user clicks the option button for a particular color, the **Click** event handler for that button should set the background color appropriately.

When you change the state of an option button with **setChecked**, it does *not* trigger a **Click** event on the button being changed.

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 5. Programming standard objects](#)

Picture boxes

A picture box displays a graphic image. The image is specified using a URL (for example, referencing a GIF or JPEG file).


Picture boxes are represented by `PictureBox` objects (`powersoft.jcm.ui.PictureBox`). Jato gives picture boxes names of the form `pictb_N`. On the Java component palette, picture boxes are represented by the following button:





Typically, you do not define event handlers for picture boxes.

 [_Picture box properties](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Picture boxes](#)

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.

Insets [Picture page]

Specifies the distance between the area available to draw the picture and the edges of the picture box (in DLUs).

URL [Picture page]

Gives a URL where the picture can be found.

Type [Picture page]

The type of picture. Possibilities are `Absolute`, `CodeBased` and `DocumentBased`. `CodeBased` and `DocumentBased` are only supported for applet targets.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

List boxes

A list box 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 box to let the user scroll through the available items. Each line in a list box 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.

Jato supports the following types of list boxes:

- *Single selection list boxes*, which only allow the user to select a single item from the list.
- *Multiple selection list boxes*, 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](#).

List boxes are represented by `ListBox` objects (`powersoft.jcm.ui.ListBox`). Jato gives list boxes names of the form `lb_N`. On the Java component palette, list boxes are represented by the following button:



Note: Jato list boxes are based on the list box components defined in the Java AWT. AWT list boxes automatically resize themselves, depending on the data they contain. Therefore, list boxes may change their size at run time; this cannot be controlled.

[Simple list box properties](#)

[Adding items to list boxes](#)

[User data](#)

[Changing the contents of a list box](#)


[TopIndex and VisibleCount](#)


[Determining selections](#)

[Multiple selections](#)

[The Select event for list boxes](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [List boxes](#)

Simple list box properties

The **Count** property determines the number of items currently in a list box, as in

```
int numItems = lb_1.getCount( );
```

The **Sort** property determines whether the items in a list box are sorted (in alphabetical order) or unsorted. To specify that the list box should be sorted, use

```
lb_1.setSort( true );
```

To have an unsorted list box, use

```
lb_1.setSort( false );
```

The **MultipleSelection** property determines whether this is a single selection or multiple selection list box. The property can be set at design time using the property sheet for the list box, or at run time by using

```
lb_1.setMultipleSelection( true ); // multi select  
lb_1.setMultipleSelection( false ); // single select
```

You can also use

```
boolean multi = lb_1.getMultipleSelection( );
```

to determine whether the list box is single selection or multiple selection.

The **getText** method determines the text associated with a particular item. For example,

```
String str = lb_1.getText( index );
```

obtains the text of the item that has the given `index`.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[List boxes](#)

Adding items to list boxes

At design time, items can be specified for the list box on the **Items** page of the property sheet. Each line you type on that page will be a separate item in the list box. The **Save** button lets you save a list of items in a file, and the **Load** button lets you load a list from a file.

During execution, the **add** method adds a new entry to the list box. For example,

```
int pos = lb_1.add( "New line" );
```

adds the specified string to the list box. If the **Sort** property of the list box is turned on, the new item is added in its correctly sorted position; if **Sort** is turned off, the new item is added at the end of the list. The return value of **add** is the index assigned to the newly added item; if an error occurs, **add** returns -1.

Another form of **add** lets you place an item into the list at a specific location. For example,

```
int pos = lb_1.add( "New line", index );
```

inserts the given line into the list so that the new line has the specified index. In this case, the **Sort** property is ignored. If you want to add the line at the end of the list, specify an index of -1.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_List boxes](#)

User data

Each item in a list box can have an associated block of data. For example, if each item in a list has an action associated with it, each item might have a block of data which provides information for performing the relevant action. The data associated with a list box item is called *user data*. The user data for a list box item is stored as an object of the `Object` class.

You can specify the user data for a list box item when you call **add** to add the item to the list:

```
// Object data;  
int pos = lb_1.add( "Item 1", -1, data );
```

The user data object is specified as a third argument for **add**.

You can determine the user data associated with a list box item using **getUserData**:

```
Object userData = lb_1.getUserData( index );
```

This obtains the user data object associated with the item that has the given index. Similarly, you can change the user data associated with an item using

```
// Object newData;  
lb_1.setUserData( index, data );
```

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_List boxes](#)

Changing the contents of a list box

There are a variety of method functions for changing the contents of a list box. To delete a particular item from the list box, use

```
lb_1.delete( index );
```

To delete all the items in the list box, use either

```
lb_1.deleteAll( );  
lb_1.reset( );
```

To change the text displayed for an item, use


```
// String newString;  
lb_1.replace( newString, index );
```


There is a second form of **replace** which replaces both the text of the item and its associated user data:


```
// Object newUserData;  
lb_1.replace( newString, index, newUserData );
```

With both forms of **replace**, the 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).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [List boxes](#)

TopIndex and VisibleCount

During execution, you can use **setTopIndex** to specify which line should be the first one visible at the top of the list box. For example,

```
lb_1.setTopIndex( 10 );
```

places item 10 at the top of the list box. This means that items 0 to 9 are scrolled off the top.


Similarly,


```
int topItem = lb_1.getTopIndex( );
```


determines which index item is currently shown at the top of the list box.

To determine how many items are currently visible inside the list box, use

```
int numberSeen = lb_1.getVisibleCount( );
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [List boxes](#)

Determining selections

If a list box currently has one and only one item selected, the **getSelected** function returns the index of that item, as in

```
int lineNumber = lb_1.getSelected( );
```

This version of **getSelected** returns `-1` if no line has been selected or if there have been several lines selected (in a multiple selection list box).

Another version of **getSelected** determines whether a given item is selected. For example,

```
boolean selected = lb_1.getSelected( index );
```

returns `true` if the item is currently selected and `false` otherwise. If the given `index` is out of range, **getSelected** returns `false`.

The **setSelected** function selects a specified item, as in

```
lb_1.setSelected( index );
```

In a single selection list box, this selects the item with the given `index` and unselects everything else. In a multiple selection list box, this selects the given item in addition to anything that is currently selected. As a special case, if `index` is less than zero, **setSelected** unselects every item in the list box, both for single selection and multiple selection boxes. For example,

```
lb_1.setSelected( -1 );
```

clears all current selections.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_List boxes](#)

Multiple selections

With a multiple selection list box, **getSelectedCount** returns the number of items that are currently selected:

```
int count = lb_1.getSelectedCount( );
```

To get a list of which items are selected, you can use

```
int list[] = lb_1.getSelectedList( );
```

The list contains the indexes of the items which are currently selected.

Multiple selection list boxes support a special form of **setSelected** which provides more control over selecting and unselecting items:

```
// boolean onOff;  
lb_1.setSelected( index, onOff );
```

selects the given item if **onOff** is **true** and unselects the item if **onOff** is **false**. For example, the following code selects every odd-numbered item and unselects every even-numbered item:

```
int i;  
int count = lb_1.getCount( );  
for ( i=0; i < count; i++ ) lb_1.setSelected( i, (i%2)==1 );
```

As a special case, if **index** is less than zero, this version of **setSelected** selects or unselects every item in the list box. For example,

```
lb_1.setSelected( -1, true );
```

selects every item in the box.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_List boxes](#)

The Select event for list boxes

A list box generates a **Select** event whenever the user selects an item. Your **Select** event handler will have a prototype of the form

```
public boolean  
    lb_1_Select(powersoft.jcm.event.SelectEvent event);
```

The `event` argument is an object of the `SelectEvent` class. This class is derived from `EventData` but supports an additional method:

```
int i = event.getIndex( );
```

returns the index of the item that was just selected in the list box.

In a multiple selection list box, there will be a **Select** event each time the user selects an item.

Therefore, there will be a separate **Select** event for each item selected. You can use **getSelectedList** to determine the complete list of items that have been selected.

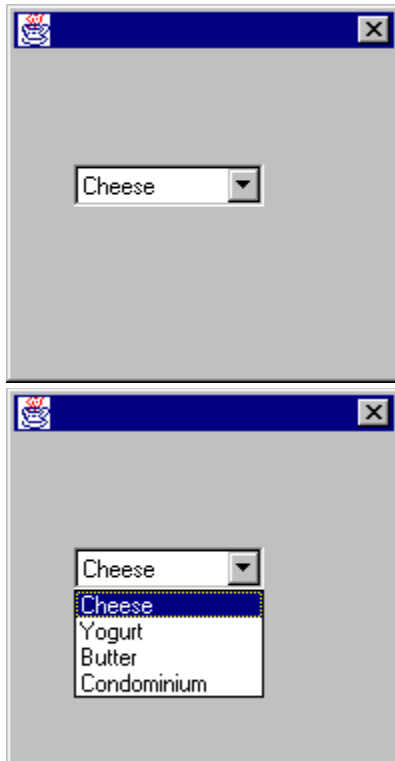
Note: In many cases, you should not take immediate action when the user selects a list box item. This makes it possible for users to change their minds or to correct a mistaken click. For example, you might have users select an item from a list box, then click a command button to indicate that they have made their final selection. In this case, you would ignore **Select** events on the list box and only respond to the **Click** event on the button.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

Combo boxes



A combo box is similar to a single-selection list box, but takes up less space on a form. In its closed state, the combo box only shows one list item (the item currently selected). In its open state, the combo box shows all of the list (unless the list is so long that it requires scroll bars).

Note: Unlike Powersoft Optima++, Jato only supports “read-only” combo boxes. Users can only select items from the list offered; they cannot type their own text into the area at the top of the combo box.

Combo boxes are represented by `ComboBox` objects (`powersoft.jcm.ui.ComboBox`). Jato gives combo boxes names of the form `combo_N`. The Text of a combo box is associated with the text box part of the combo box.

On the Java Component palette, combo boxes are represented by the following button:



When you first place a combo box on a form, it is shown in its closed form (without the list showing).

Note: Jato combo boxes are based on the combo box components defined in the Java AWT. AWT combo boxes automatically resize themselves, depending on the data they contain. Therefore, combo boxes may change their size at run time; this cannot be controlled.

[Combo box properties and methods](#)

[The Select event for combo boxes](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Combo boxes](#)

Combo box properties and methods

Combo boxes support many of the properties and methods supported by list boxes. These include the following:

```
// String str;
int num = combo_1.getCount( );           // number of items
int item = combo_1.getSelected( );       // currently selected item
combo_1.setSelected( index );           // select item
String s = combo_1.getText( index );    // text of item
combo_1.setUserData( obj );             // user data
Object obj = combo_1.getUserData( );
combo_1.add( str );                      // add new item
combo_1.add( str, userData );
combo_1.delete( index );                 // delete item
combo_1.reset( );                       // delete all items
combo_1.select( str );                   // select item by name
```

In addition, combo boxes support a find method to find the index of an item in the combo box's list. It has the prototype

```
int find( String str, int startAfter, boolean exact );
```

where:

str

Specifies the string you're looking for.

startAfter


Is the index of an item in the list. The search begins immediately following the specified item. To search from the beginning of the list, specify -1 for startAfter. %%% Does the search wrap around?

exact


If this is true, find looks for an item that exactly matches str. If this is false, find looks for an item that begins with the given string but may contain extra characters. %%% Verify

The result of find is the index of the first item matching the given string. If the list contains no such item, find returns -1.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)


 [Chapter 5. Programming standard objects](#)


 [Combo boxes](#)

The Select event for combo boxes


Combo boxes receive the Select event in the same way as list boxes. For more information, see [The Select event for list boxes](#) on [The Select event for list boxes](#).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

Text boxes





A text box is an area that displays text. In a text box, the *caret* serves the same function as a cursor: it 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 boxes are represented by `TextBox` objects (`powersoft.jcm.ui.TextBox`). Jato gives text boxes names of the form `textb_N`. On the Java component palette, text boxes 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](#)


 [Determining the text in a text box](#)

 [MultiLine text boxes](#)


 [Deleting from the text box](#)


 [Caret position](#)


 [Working with selected text](#)

 [The Change event](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Text boxes](#)

Text box properties and styles

The following list discusses design-time text box properties:

MultiLine [General page]

Indicates that the text box can contain more than one line of text. If you do not choose **MultiLine**, the text box can only contain a single line.

ReadOnly [Style page]


Specifies that the user cannot type or edit text directly in this box.


PasswordCharacter [General page]


Can mark this text box as a *password box*. If the user types text into this box, 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 a password character, type the character inside single quotes; for example, '*' sets the password character to an asterisk. You can also enter the decimal ASCII value of the character instead of the character itself.

If you do not want this text box to be a password box, the **PasswordCharacter** should be zero.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Text boxes](#)

Determining the text in a text box

The **getText** method returns the current text contained by a text box, as in

```
String str = textb_1.getText( );
```

Similarly, the **setText** method changes the contents of the text box, as in

```
textb_1.setText( "New text" );
```

The **getTextLength** method returns the number of characters in the current text:

```
int numChars = textb_1.getTextLength( );
```

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_Text boxes](#)

MultiLine text boxes

When the **MultiLine** property is turned on, the text box can contain more than one line. When setting text, you use `\n` characters to separate lines, as in

```
textb_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 box's property sheet. For example, you might type in

```
First line\nSecond line
```


In the form design window, this text will be shown as a single line, with the `\n` characters shown in the middle of the text. However, when the program actually runs, the text will be broken into lines correctly, as specified by the `\n` characters.


<p>Note: <code>\n</code> characters are only recognized as line separators when the text box has MultiLine turned on. If MultiLine is turned off, <code>\n</code> characters are interpreted as unprintable characters and shown as such in the middle of the Text string. This is true both at design time and at run time.</p>
--


The **getLineCount** method determines the number of lines in a **MultiLine** text box:

```
int numLines = textb_1.getLineCount( );
```


 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 5. Programming standard objects](#)


 [_Text boxes](#)


Deleting from the text box


The **clear** method deletes the current contents of the text box:

```
textb_1.clear( );
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Text boxes](#)

Caret position

The **CaretPosition** property specifies the position of the caret. The beginning of the text box is position 0, the position after the first character is 1, and so on. The function

```
int pos = textb_1.getCaretPosition( );
```

determines the current caret position, and

```
textb_1.setCaretPosition( pos );
```

sets the caret to the given position.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_Text boxes](#)

Working with selected text

The user may select all or part of the text in a text box (for example, by dragging the mouse across the desired portion). When the user has selected some text, the function

```
String str = textb_1.getSelectedText( );
```

obtains the text that has been selected. If no text is currently selected, **getSelectedText** returns `null`.

The **getEditSelection** method returns a `Range` value, giving the character positions of the first and last characters in the selected range:

```
Range r = textb_1.getEditSelection( );  
// r.start is start of range  
// r.end is end of range
```

The character position of the first character is 0, the next character is 1, and so on.

The **setEditSelection** method selects some or all of the current text. As arguments, you specify a `Range` value giving the start and end character positions of the range you want to select:

```
Range r = new Range( start, end );  
textb_1.setEditSelection( r );
```

For example,

```
textb_1.setEditSelection( new Range(0,2) );
```


selects the first three characters in the text box.


Note: In order to specify a `Range` value, you may have to place


```
import powersoft.jcm.ui.Range;
```

in the declarations at the beginning of your source code.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Text boxes](#)

The Change event

The **Change** event is triggered on a text box whenever the user changes the box's text. For example, if the user types in a string of text, the **Change** event is triggered for each character typed. The **Change** event is also triggered for deletions, replacements, and so on.

Because the **Change** event is triggered for every character, applications typically ignore the **Change** event. Instead, they provide the user with a button to click to after the full text has been entered.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

Group boxes


A group box is a border that can surround a number of other objects. For example, you might put a group box around a set of check boxes to show that the boxes are logically connected.


Group boxes are represented by `GroupBox` objects (`powersoft.jcm.ui.GroupBox`). Jato gives group boxes names of the form `groupb_N`. On the Java component palette, group boxes are represented by the following button:




[Notes for working with group boxes](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Group boxes](#)

Notes for working with group boxes

When designing a form, place the group box first and then place the objects that lie inside the group box. In this way, the objects lie “on top of” the group box in the form window. If place objects in the other order, the group box lies on top of the other objects, and you cannot click on the other objects.

When you move a group box in the form design window, objects inside the group box do not normally move. If you want to move the group box and all the objects inside it together, follow these steps:

1. Click the form design window just outside one corner of the group box, then drag the cursor to surround the group box.
2. Release the cursor. This puts empty sizing handles on every object in the selected area.
3. Drag any of the objects in the selected area. This drags the entire selected group.

The **Text** of a group box appears on the upper border of the group box. It can be changed at run time with

```
groupb_1.setText( "New text" );
```

Most applications do not define any event routines for group boxes. They are only used for making the form easier to understand.

Scroll bars

Scroll bars are used for scrolling through information. For example, you might create a scroll bar whose actions affect an associated text box. When the user moves the scroll bar, the text displayed in the text box moves appropriately to show different parts of the text.

Note: Many objects automatically create their own scroll bars as needed. For example, a list box 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 [The Scroll event](#).

The Java component palette contains both vertical and horizontal scroll bars. Vertical scroll bars are represented by `VScrollBar` objects (`powersoft.jcm.ui.VScrollBar`), while horizontal scroll bars are represented by `HScrollBar` objects (`powersoft.jcm.ui.HScrollBar`). Both of these object types are derived from the `ScrollBar` class (`powersoft.jcm.ui.ScrollBar`). All of these types have similar properties and methods.

Jato gives vertical scroll bars names of the form `vscroll_N`, and horizontal scroll bars names of the form `hscroll_N`.

On the Java component palette, vertical scroll bars are represented by the following button:



Horizontal scroll bars are represented by the following button:



Note: The properties and methods discussed in the sections that follow apply to both horizontal and vertical scroll bars.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Scroll bars](#)

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:

ScrollRange

The range of the scroll bar, expressed as a Range value.

ScrollPosition

The integer value corresponding to the slider's position in the range.


These properties have the usual **get** and **set** methods, as in:


```
Range r = new Range( 0, 100 );
scroll_1.setRange( r );
scroll_1.setPosition( 0 );
```


You can determine the current scroll position using

```
int val = scroll_1.getPosition( );
```


 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Scroll bars](#)

Increment and Step

Scroll bars may have associated **Increment** and **Step** values:

- The **Increment** is the amount that the **ScrollPosition** should change if the user clicks one of the end arrows of the scroll bar.
- The **Step** is the amount that the **ScrollPosition** should change if the user clicks in the blank area above or below the scroll indicator.

You can set these values using appropriate methods, as in:

```
scroll_1.setIncrement( 1 );  
scroll_1.setStep( 10 );
```

If you set the increments for a scroll bar, the default event handlers for the scroll bar automatically change the **Position** 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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

[Scroll bars](#)

The Scroll event

When the user clicks on a scroll bar, it generates a **Scroll** event. The argument to a **Scroll** event handler has the `ScrollEvent` type. This type is based on the usual `EventData` type but supports several additional methods.

The most important method of `ScrollEvent` is

```
// ScrollEvent event;  
int msg = event.getMessage( );
```

The result of `getMessage` tells what the user did with the scroll bar. Possible values are:

`ScrollEvent.ABSOLUTE`

The user dragged the scroll indicator to a new position.

`ScrollEvent.LINE_UP`

The user clicked the arrow at the top or left of the scroll bar.

`ScrollEvent.LINE_DOWN`

The user clicked the arrow at the bottom or right of the scroll bar.

`ScrollEvent.PAGE_UP`

The user clicked in the blank area above or to the left of the scroll indicator.

`ScrollEvent.PAGE_DOWN`

The user clicked in the blank area below or to the right of the scroll indicator.

If the user has dragged the scroll indicator to a new position,

```
// ScrollEvent event;  
int dir = event.getPosition( );
```

tells you the position that the user chose.

Default Scroll handling

Jato supplies default handling for the **Scroll** event. The default handling responds to messages in the following ways:

`ScrollEvent.ABSOLUTE`

Change the **ScrollPosition** value to reflect the new position of the scroll indicator.

`ScrollEvent.LINE_UP`

Subtract **Increment** from the current **ScrollPosition**.

`ScrollEvent.LINE_DOWN`

Add **Increment** to the current **ScrollPosition**.

`ScrollEvent.PAGE_UP`

Subtract **PageSize** from the current **ScrollPosition**.

`ScrollEvent.PAGE_DOWN`

Add **PageSize** to the current **ScrollPosition**.

The default handling performs appropriate range-checking. For example, if subtracting **PageSize** from **ScrollPosition** would move the position outside the scroll range, **ScrollPosition** is set to the minimum value in the valid range.

Setting the ScrollPosition

If you write your own **Scroll** event handler, it can use `setScrollPosition` to set a different scroll position

from the one specified in the ScrollEvent block. If you do this, the scroll indicator stays 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 **ScrollPosition** if you want to prevent the user from scrolling to a particular region or a particular position value.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 5. Programming standard objects](#)

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.jcm.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. Jato gives timers names of the form *timer_N*.

[The HighPriority property](#)


[Setting the timer](#)


[Setting a timer at design time](#)


[Stopping a timer](#)

[The Timer event](#)

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 5. Programming standard objects](#)

 [_Timers](#)

The HighPriority property

Timers only have one property: **HighPriority**. If this 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](#).

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 5. Programming standard objects](#)

[_Timers](#)

Setting the timer

The **start** method starts a timer running. It is typically used with a call of the form:

```
boolean success = timer_1.start( interval, tickCount );
```

The parameters are:

`long interval`

Specifies a length of time in thousandths of a second.

`int tickCount`


The maximum number of times the timer should go off. For example, if you only want the timer to go off once, set a `tickCount` of 1. If you set `tickCount` to zero, the timer will keep going off at the set `interval` until the timer is explicitly stopped.


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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Timers](#)

Setting a timer at design time


You can use the property sheet of a timer to start the timer running as soon as the form is created.


◆ **To create a timer that starts running as soon as the form is created:**


1. On the **General** page of the timer's property sheet, set the **Interval** and the **TickCount** values as you would for the **start** method.
2. Click the **Running** check box so that it is checked.

When the form is created, Jato automatically issues a **start** call that starts the timer running.

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 5. Programming standard objects](#)

 [_Timers](#)


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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

 [Timers](#)

The Timer event


The **Timer** event is triggered when the timer goes off. If the timer is set to go off at repeated intervals, it immediately begins counting down for the next interval; it does *not* wait for the **Timer** event to finish execution. If you don't want the timer to start until the **Timer** event is finished, you must **stop** the timer at the beginning of the **Timer** event routine and **start** the timer again just before the **Timer** returns.


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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 5. Programming standard objects](#)

Summary of standard objects

A property controls the appearance and behavior of an object. A style is similar to a property, but is represented internally by a bit setting within a group of styles.

The **Text** of an object typically serves as a caption for the object, although the text is not always visible when an object is placed on a form.


The **Visible** and **Enabled** properties determine whether an object can be seen or used by the user.

Color properties determine the foreground and background colors of an object. Size and position properties determine where the object appears on a form and how big it is.

At design time, properties are set through the property sheet of an object. At run time, the current value of a property may be obtained with a **get** method and the value may be changed through a **set** method.


For example, the **getRectangle** method obtains the rectangle which specifies the size and position of an object. The **setRectangle** method changes the size and/or position of the object (which means that **setRectangle** can be used to make an object move about the screen).

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)


Chapter 6. Using and programming menus


This chapter shows how to add menus to a form and how to design such menus using the Jato menu editor.

 [The MenuBar object](#)

 [The menu editor](#)


 [Menus and menu items](#)


 [Menu events](#)

 [Menu item methods](#)

 [Menu container methods](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)


The MenuBar object

A menu bar object is a single object representing all the menus displayed on the menu bar of a form. Menu bar objects belong to the `MenuBar` class (`powersoft.jcm.ui.MenuBar`). Jato gives `MenuBar` objects default names of the form `menu_N`. On the **Standard** page of the Java 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.


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 can only be placed on forms](#)

 [The default menu for a form](#)

 [Java Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [The MenuBar object](#)


Menus can only be placed on forms


A menu bar object can only be placed on a *form* or a frame, not on a dialog—this is a limitation of AWT. For example, you cannot place a menu bar on a standard Applet window, since it is not a form. 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( );
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [The MenuBar object](#)

The default menu 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 menu structures, 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.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

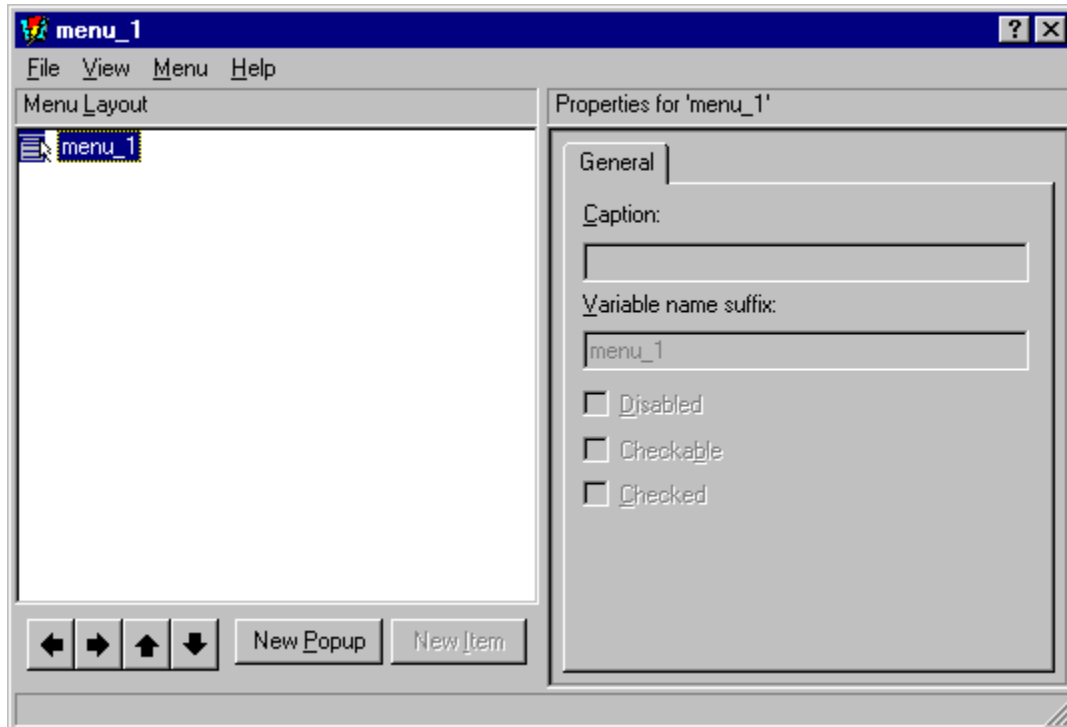
[_Chapter 6. Using and programming menus](#)

The menu editor

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 6. Using and programming menus](#)

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).

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.
--

[Creating a menu or menu item](#)

[Specifying the menu structure](#)

[Menu and menu item properties](#)

[Separators](#)

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 6. Using and programming menus](#)

[_Menus and menu items](#)

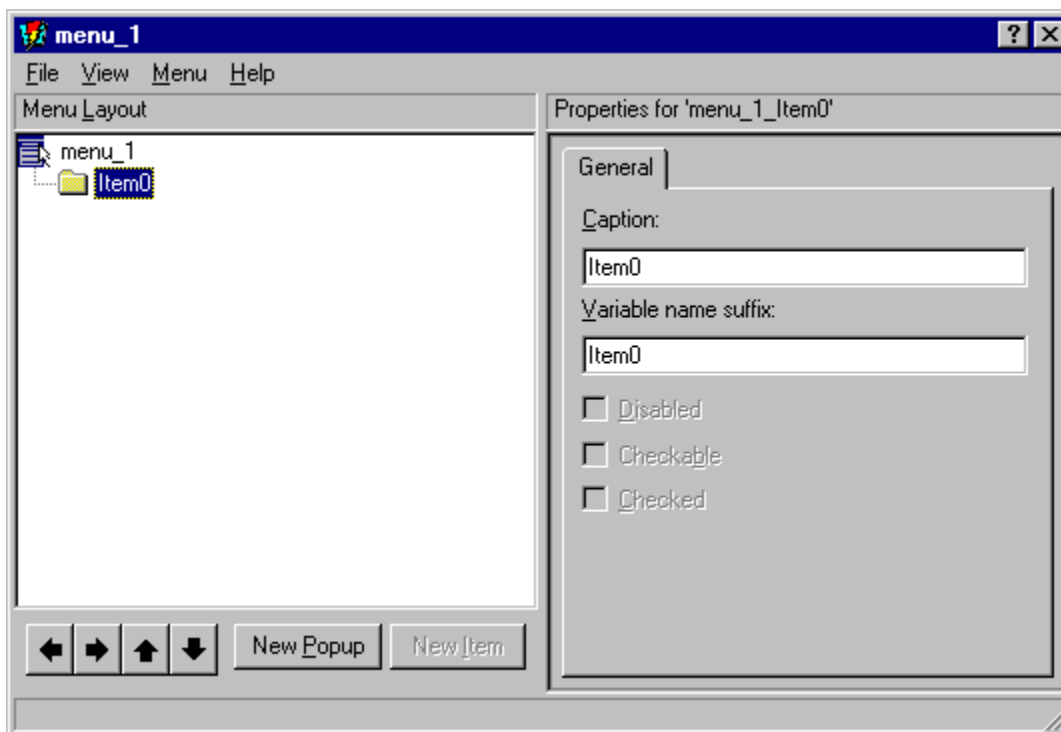
Creating a menu or menu item

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:



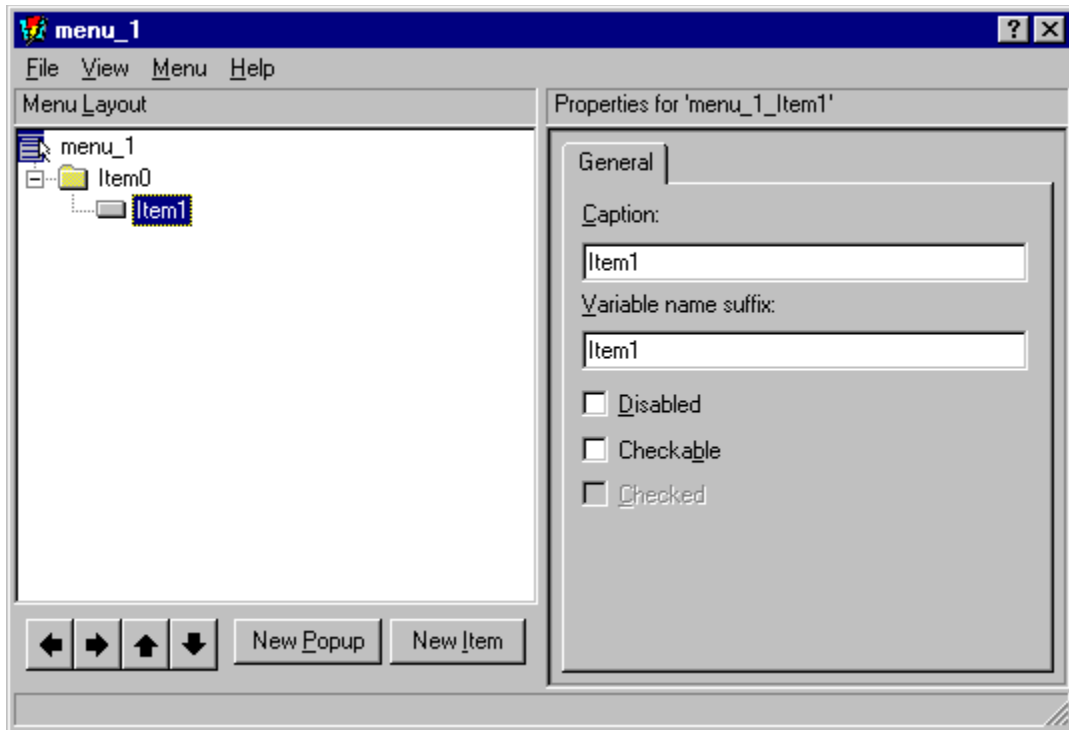
As shown in the property sheet on the right side of the menu editor, the default variable name suffix is the same as the caption.

The process of adding a new menu item to a popup menu is similar:

◆ To 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. Click **New Item**.

The new item appears in the menu layout as shown:



◆ **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**.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

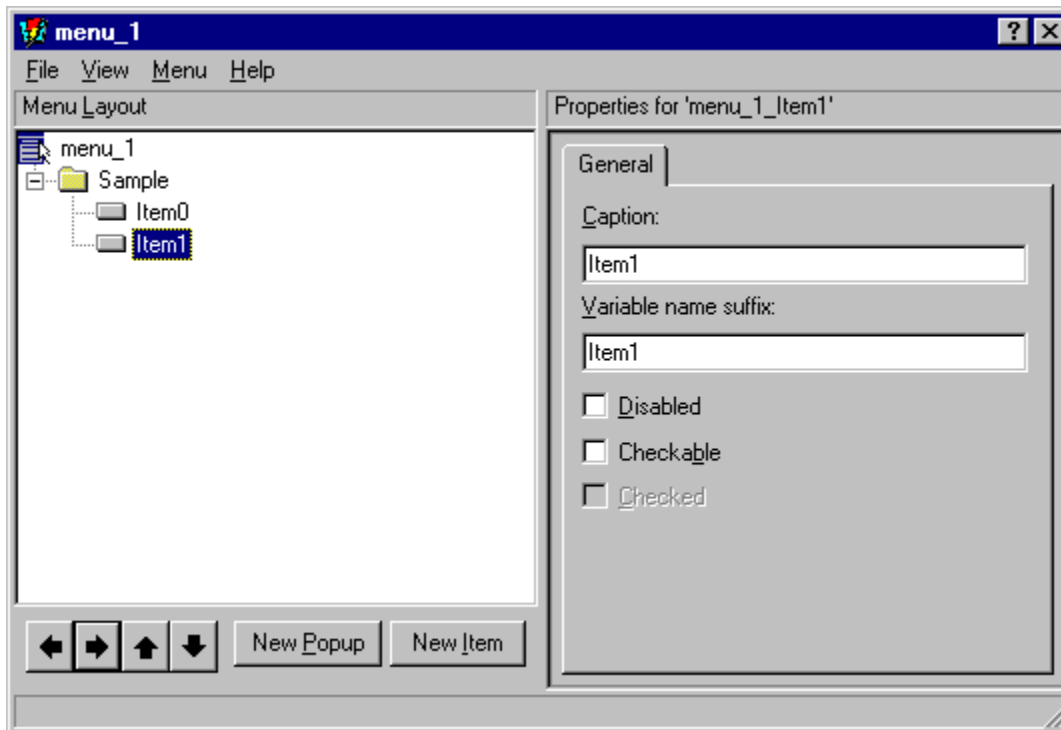
[_Chapter 6. Using and programming menus](#)

[_Menus and menu items](#)

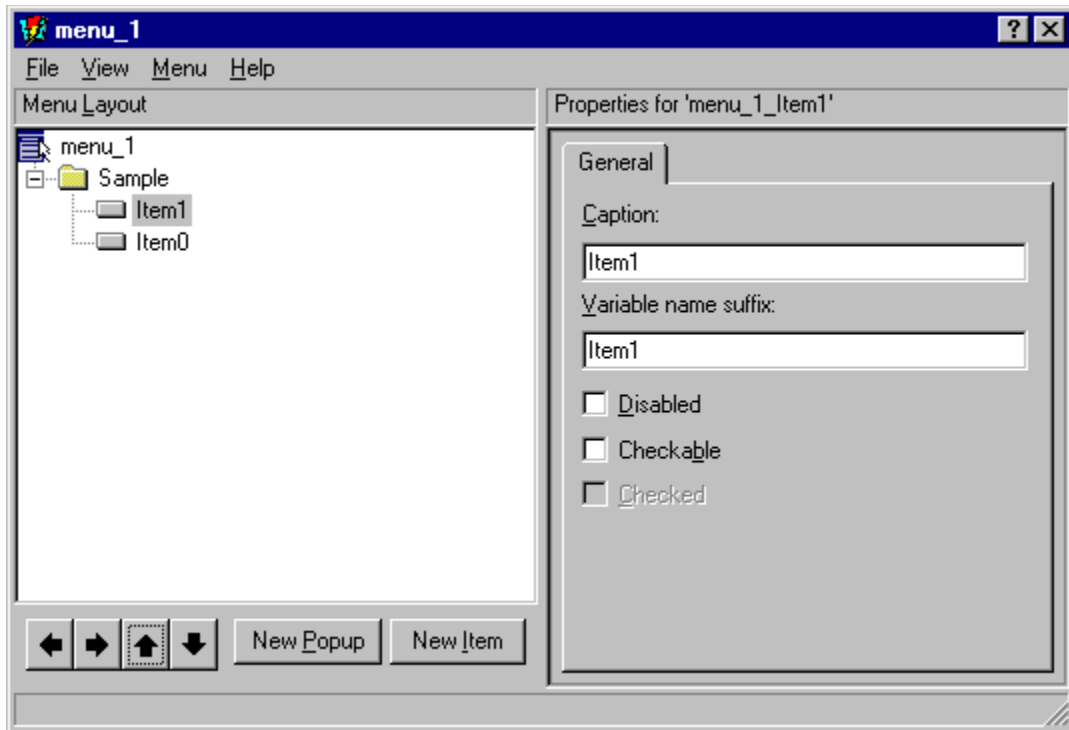
Specifying the menu structure

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.

[_Jato Programmer's Guide](#)

[_Part I. Fundamentals](#)

[_Chapter 6. Using and programming menus](#)

[_Menus and menu items](#)

Menu and menu item properties

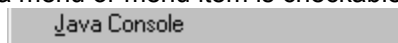
Menus and menu items may have the following properties (specified by clicking the appropriate check boxes in the menu editor):

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 **setChecked** method.

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 6. Using and programming menus](#)

 [_Menus and menu items](#)

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. From the **Menu** menu of 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

Menu events

There is only one event defined for menu and menu items:

Click


The **Click** event takes place when the user clicks the menu or menu item.


◆ To create a Click event handler for a menu or menu item:

1. In the menu layout of the menu editor, use the right mouse button to click the menu or menu item.
2. Click **Events**, then **powersoft.jcm.event.Click**.

This opens a code editor window where you can specify the code for your event handler.

 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)


 [Chapter 6. Using and programming menus](#)


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.


 [Checking an item](#)

 [Enabling an item](#)


 [Changing the caption](#)


 [The font of a menu item](#)


 [The parent of a menu item](#)

 [User data for a menu item](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu item methods](#)

Checking an item

The **Checkable** property determines whether an item may be marked with a check mark. For example,

```
menu_1_Item1.setCheckable( true );
```


makes the given item checkable.


If an item is checkable, the **Checked** property determines whether a check mark is currently displayed:

```
menu_1_Item1.setChecked( true );           // check mark  
menu_1_Item1.setChecked( false );        // no check mark
```

There are corresponding **get** functions to determine if an item is **Checkable** and **Checked**.

 [_Jato Programmer's Guide](#)

 [_Part I. Fundamentals](#)

 [_Chapter 6. Using and programming menus](#)

 [_Menu item methods](#)

Enabling an item


The **Enabled** property determines whether an item is enabled:


```
menu_1_Item1.setEnabled( true );    // enabled
menu_1_Item1.setEnabled( false );  // disabled
```

When an item is disabled, it is “grayed out” and cannot be clicked.

There is a corresponding **get** function to determine if an item is currently enabled.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu item methods](#)

Changing the caption


The **Text** property specifies the caption for an item. For example,


```
menu_1_Item1.setText( "New caption" );
```

changes the caption of the item.

There is a corresponding **get** function to obtain the current caption of the item.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu item methods](#)


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 );
```

sets the font of the given menu item.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu item methods](#)

The parent of a menu item


The **Parent** property specifies the parent of a menu item. For example,


```
Menu mc = menu_1_Item1.getParent( );
```


determines the parent of the given item. There is also a **setParent** method that can switch a menu item from one parent menu to another:

```
// Menu newMenu;  
menu_1_Item1.setParent( newMenu );
```

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu item methods](#)

User data for a menu item

Each menu item can have user-defined data associated with it. This data is specified as an Object object in the **UserData** property:

```
//Object data;  
menu_1_Item1.setUserData( data );  
Object ud = menu_1_Item1.getUserData( );
```

[!\[\]\(08a82c22d89d6b027ff69762ad096586_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(35e4f762fc1cfea5610d92e2d225d5b4_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(d84e7ea36f695d92cb39ec32c307ac93_img.jpg\) Chapter 6. Using and programming menus](#)

Menu container methods

This section examines methods that can be performed on menu bar, menu and submenu objects (objects that contain menu item objects).

[!\[\]\(83f22ed94ec5517769dd76d702c6bfd8_img.jpg\) Counting children](#)

[!\[\]\(8d0f0e0fe25b320c33272c52aec1fbca_img.jpg\) Obtaining children by index](#)

[!\[\]\(642aa997563f9a325b310230bb5078b7_img.jpg\) Removing an item from a menu](#)

[!\[\]\(2b376d1a92330ab09dad2665d2f89bf5_img.jpg\) Adding an item to a menu](#)

[!\[\]\(082f818d99f166a3ba574d9284d73064_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(d263118e0bfd47dc6bc704167d936b83_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(34b4f260a8587d2e97eeaee361cc357b_img.jpg\) Chapter 6. Using and programming menus](#)

[!\[\]\(3d8c13c92b853674f749aac6fa869926_img.jpg\) Menu container methods](#)

Counting children

The **Count** property determines the number of children directly contained by the menu object. For example,


```
int num = menu_1.getCount( );
```


determines the number of menus that appear on the menu bar. This does not count the number of items that are included in those menus. Similarly,

```
int num = menu_1_Submenu.getCount( );
```

counts the number of items in the given submenu, but does not include items in submenus of the submenu.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu container methods](#)

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 );
```


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 menu_1_submenu getMenuItem( 0 );
```

returns a MenuItem object representing the first item in the submenu.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu container methods](#)

Removing an item from a menu

The `removeMenuComponent` 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,


```
boolean success = menu_1.removeMenuComponent( 0 );
```


removes the first menu from the menu bar, while


```
boolean success =  
    menu_1_Submenu.removeMenuComponent( menu_1_Item1 );
```

removes the specified component. In both cases, the result is `true` if the operation succeeds and `false` otherwise.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 6. Using and programming menus](#)

 [Menu container methods](#)

Adding an item to a menu

The `addMenuComponent` 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

```
boolean success =  
    menu_1_Submenu.addMenuComponent( menu_1_Item1 );
```

puts the item back on the menu again.

If you want 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 **Create** 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 a **Click** event handler for the item. This is why it is easier to set everything up at design time (including the **Click** event handler) and then remove them temporarily than to try building items from scratch.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

Chapter 7. Debugging

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

[Introduction to debugging](#)

[Specifying your virtual machine](#)

[Breakpoints](#)

[Debug windows](#)

[Stepping through your code](#)

[Breaking program execution](#)

[Resuming normal program execution](#)

[Source code folders](#)

[Remote debugging](#)


[Run options](#)


[Debugging techniques](#)

[Performance tips](#)

[Summary of debugging](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)


Introduction to debugging


Finding bugs is an art, not a science. However, Jato 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 Jato 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.

Note: In order to use certain debugging tools effectively, you must be familiar with assembly language and the way in which programs use the computer hardware.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

Specifying your virtual machine

%%% In this beta release, the debugging facilities only work when you run your Java code with the Microsoft Java interpreter (virtual machine). You specify the interpreter in the run options for the target you are debugging.

◆ To set run options for a target:

1. Open the Targets window.
2. Use the right mouse button to click the target you want to run. Click **Run Options** in the resulting menu.

This opens the run options dialog for the target.

Different types of targets accept different types of run options.

- For Java applications, specify the virtual machine on the **General** page of the run options dialog. To use the debugging facilities discussed in this chapter, you must use the Microsoft interpreter.
- For Java applets, specify the virtual machine on the **General** page of the run options dialog. You can either choose the Microsoft interpreter, or use a web browser that uses the Microsoft interpreter (for example, Microsoft's Internet Explorer).

[!\[\]\(08a82c22d89d6b027ff69762ad096586_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(35e4f762fc1cfea5610d92e2d225d5b4_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(d84e7ea36f695d92cb39ec32c307ac93_img.jpg\) Chapter 7. Debugging](#)

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. Jato 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.


[!\[\]\(8d0f0e0fe25b320c33272c52aec1fbca_img.jpg\) Setting a simple breakpoint](#)


[!\[\]\(642aa997563f9a325b310230bb5078b7_img.jpg\) When a breakpoint is encountered](#)


[!\[\]\(2b376d1a92330ab09dad2665d2f89bf5_img.jpg\) Breakpoints dialog box](#)

[!\[\]\(3cb60d42b10e53f9522bb0b392c1c4cd_img.jpg\) Advanced breakpoint options](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Breakpoints](#)

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 Jato 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. The breakpoint condition is evaluated. If you have set a condition for the breakpoint and it is not met, the following steps are skipped and program execution is resumed.
2. 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.

3. If you have set a code patch, it is executed. If you specified that execution should continue after the code patch, the following step is skipped and program execution resumes.
4. Program execution is suspended.

For information on breakpoint conditions and code patches see [Advanced breakpoint options](#).

When a breakpoint is encountered that suspends execution, Jato 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 Jato window or in the code editor window.

Repainting at a breakpoint

While a program is paused at a breakpoint, it's common to place various windows on top of the program's forms. For example, you might have a code editor window showing you the program's source code, another window showing you a memory dump of the program's data, and so on. Some or all of these windows may be placed on top of the program's forms.

When you move one of these windows off such a form, you may expect the form to restore itself to its previous appearance. Unfortunately, this can't be done. When you move an overlapping window, you trigger a **Paint** event on the underlying form, telling it to repaint itself. However, your program isn't running—it's paused for a breakpoint. Therefore, the form can't respond to the **Paint** event and can't restore its usual appearance. The form won't restore itself until you resume execution of the program.

The same principle applies to other visual effects that might happen when you're stopped at a breakpoint. For example, suppose you change the **Text** of an object on a form. Since the form can't repaint itself, it can't show you the changed text. The change can only become visible on the form when you resume execution and give the form a chance to revise its appearance.

[Jato Programmer's Guide](#)

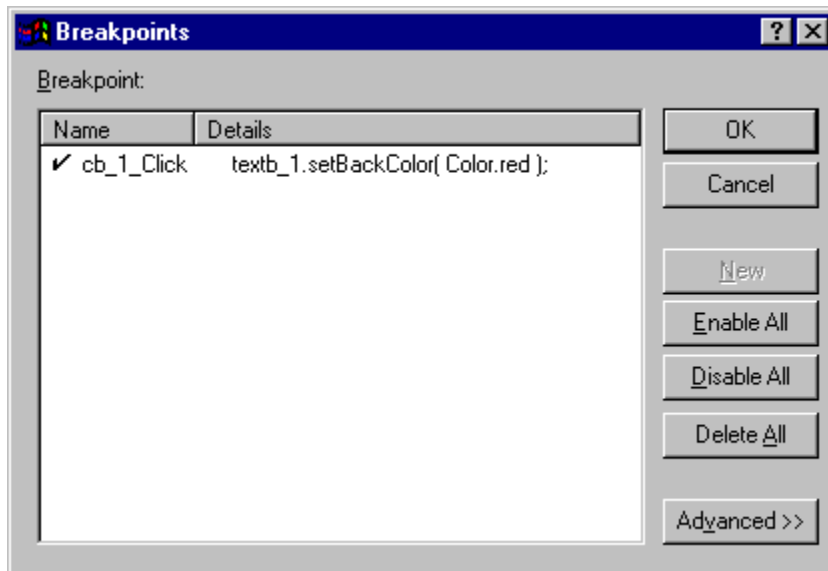
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Breakpoints](#)

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 Jato menu bar, click **Breakpoints**.

When you use the right button to click the name of an entry in the breakpoint list, Jato 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.

[Jato Programmer's Guide](#)

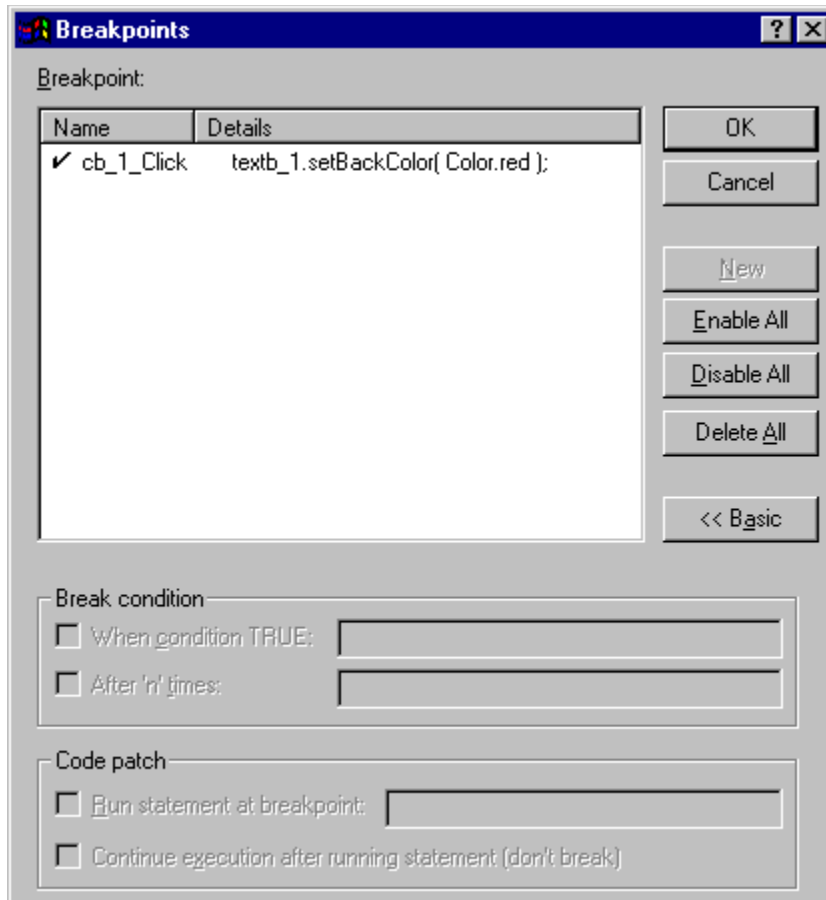
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Breakpoints](#)

Advanced breakpoint options

If you click the **Advanced** button of the Breakpoints dialog box, Jato 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, Jato evaluates the specified expression. If this expression is `TRUE` (non-zero), Jato triggers the breakpoint normally. However, if Jato finds the expression is `FALSE` (zero), Jato 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 Jato passes through the breakpoint position, it increments a counter by 1. When the counter finally reaches the value *n*, Jato triggers the breakpoint. For example, if you type a value of 10 for *n*, Jato 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 is TRUE** together, the breakpoint only happens after *n* executions when the condition is true. For example, suppose you specify

```
Break when condition is TRUE: p != NULL
After 'n' times: 5
```

Each time execution passes through the break location, Jato 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 Jato finds a non-null p .

Run statement at breakpoint

If you check this option and type a Java statement in this box, Jato 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.

Breaking on an address

As well as setting breakpoints on lines that you can see in the code editor, you can also set breakpoints on addresses specified by Java expressions. For instance, if your program causes an execution fault and the details give an address where the problem occurred, you can set a breakpoint at a preceding address.

In order to set such a breakpoint, Jato must have your program loaded so that it can resolve the address. You can do this by setting a breakpoint in the editor near the beginning of your program, or by specifying in the run options that the program should break after starting.

◆ To break at an address:

1. While your program execution is paused, click **Breakpoints** on the **Run** menu.
2. Use the right button to click a blank area in the Breakpoints list, then click **New**.
3. Type in a Java expression that can be evaluated to an address, or click **Symbol Lookup** to choose a breakpoint location from a list of symbols.

Note: If you type a symbol name, the address of that symbol will be used. To prevent this, precede the symbol name with an asterisk.

[!\[\]\(f4912148590488019602cab6e009e597_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(8af806fb1314382d09bc5ec5b767526c_img.jpg\) Part I. Fundamentals](#)

[!\[\]\(2e897e890e69d81eae4503a8342c36b0_img.jpg\) Chapter 7. Debugging](#)

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 Jato 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.

[!\[\]\(0aff635c4179ba9e710b00f4b01d3b20_img.jpg\) Call Stack window and commands](#)

[!\[\]\(830769b31eeeaca920791081939ff8ba_img.jpg\) Locals window](#)

[!\[\]\(0b5e7e25e8775f7e7e80906ada4f0021_img.jpg\) Watches window](#)

[!\[\]\(8bba887393ca45b761e5cb49e755e762_img.jpg\) Assembly window](#)

[!\[\]\(6bb0e4f14c4133b37d2887cb37e67ddd_img.jpg\) The Registers window](#)

[!\[\]\(47734e4656765d20df4fdbd5b7aff048_img.jpg\) FPU Registers window](#)

[!\[\]\(bd3b31712ad9bab5a241210fa6925cdd_img.jpg\) Threads window](#)

[!\[\]\(0fb13ad0bfa3d86868cdd3883e5665b3_img.jpg\) Memory window](#)

[!\[\]\(799877f5c2f906134441300079881630_img.jpg\) The Stack window](#)

[!\[\]\(41aea2746216b27a6939d696d8e035da_img.jpg\) Drag-and-drop in debugging windows](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

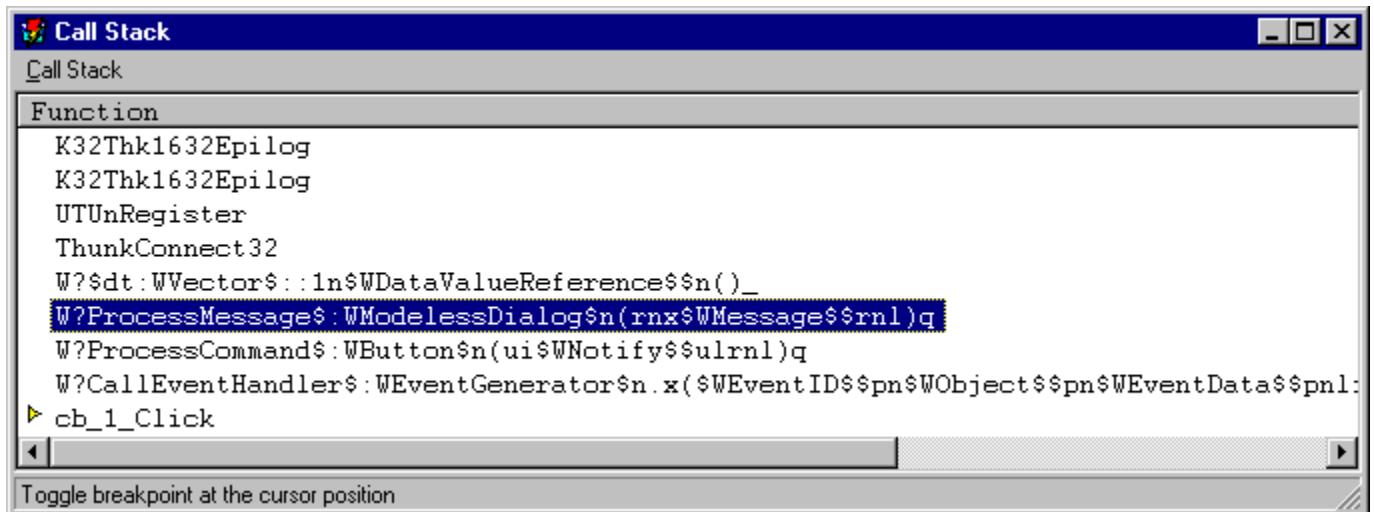
[Debug windows](#)

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_Click`, **Up Call Stack** displays information about the function that called `cb_1_Click`. If possible, Jato opens a code editor window to display the function; however, if Jato does not have Java source code for the function, Jato shows an assembly code version of the function as well as the hardware register values.

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_Click`, **Down Call Stack** moves back down to `cb_1_Click` 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_Click`).

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. The one exception to this is the Registers window: the ESP, EBP, and EIP values are displayed correctly relative to the new function, but the other register values may not reflect the values they had when the function was active.

If you use the right mouse button to click an entry in the Call Stack, Jato displays a menu of actions

you can perform on that entry:

Toggle Breakpoint

Sets a breakpoint immediately 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

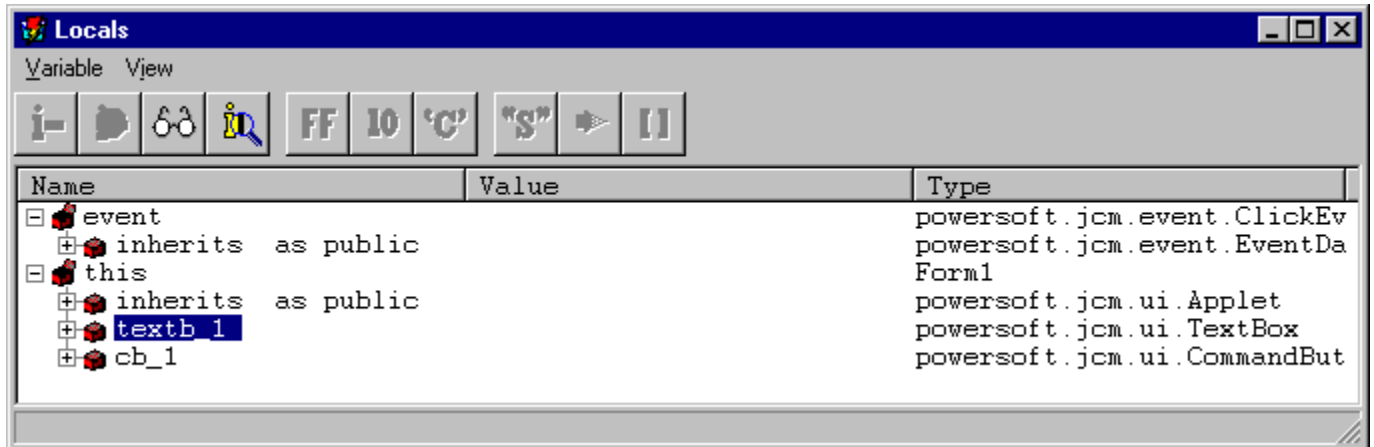
Runs the program up to the instruction immediately following the point where the entry called the next routine on the call stack. Therefore, execution stops as soon as it returns to the specified function. This may be in the middle of a Java statement.

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:

- Pointers are marked with arrows.
- 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 a pointer, the window displays the object pointed to by the pointer. 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.

The Variables menu

The **Variables** 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. Possible **Show** options are:

Pointer Memory displays the memory pointed to by a pointer, using a Memory window (see [Memory window](#)).

Variable Memory displays the memory containing the variable, using a Memory window.

Pointer Code displays the memory that a pointer points to, using a code editor or an Assembly window (see [Assembly window](#)).

Type

Can display the value of the variable in different formats. If you ask to display the value as an array, Jato 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 WString 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, Jato displays a context menu offering the same choices as the **Variables** menu.

[Jato Programmer's Guide](#)

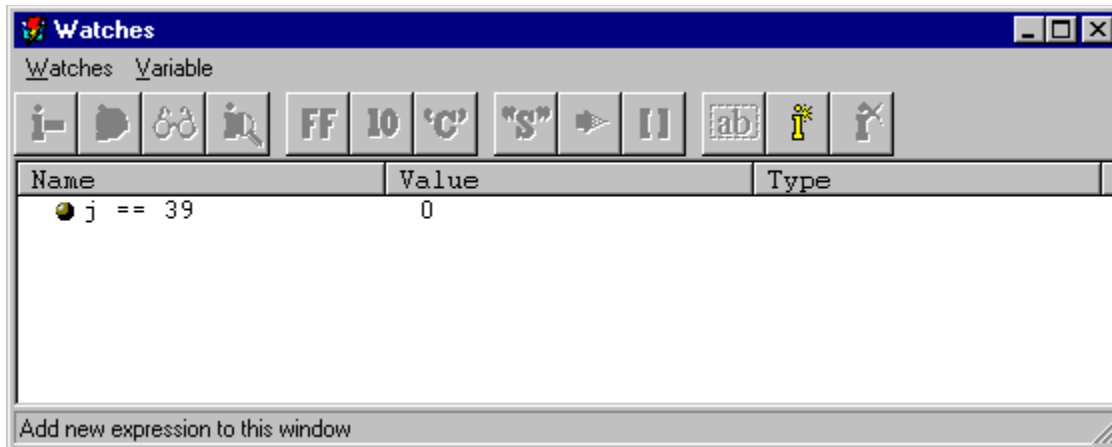
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Debug windows](#)

Watches window

A *watch expression* is a Java expression whose value is displayed by Jato 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.

Jato 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**. Jato 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**.

Jato 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 and a **View** menu which duplicate the menus of the Locals window. In particular, **Break when Modified** on the **Variable** menu triggers a breakpoint when the

value of the watch expression changes.

Note: You can only specify **Break when Modified** for L-value expressions with a size of 1, 2, or 4 bytes. In other words, the expression must refer to a specific location in memory. For example, the expression could be the name of a variable or a pointer value. The breakpoint occurs when the specified memory location changes.

If you use the right mouse button to click an entry in the Watches List, Jato displays a context menu offering the same choices as the **Variable** menu.

[Jato Programmer's Guide](#)

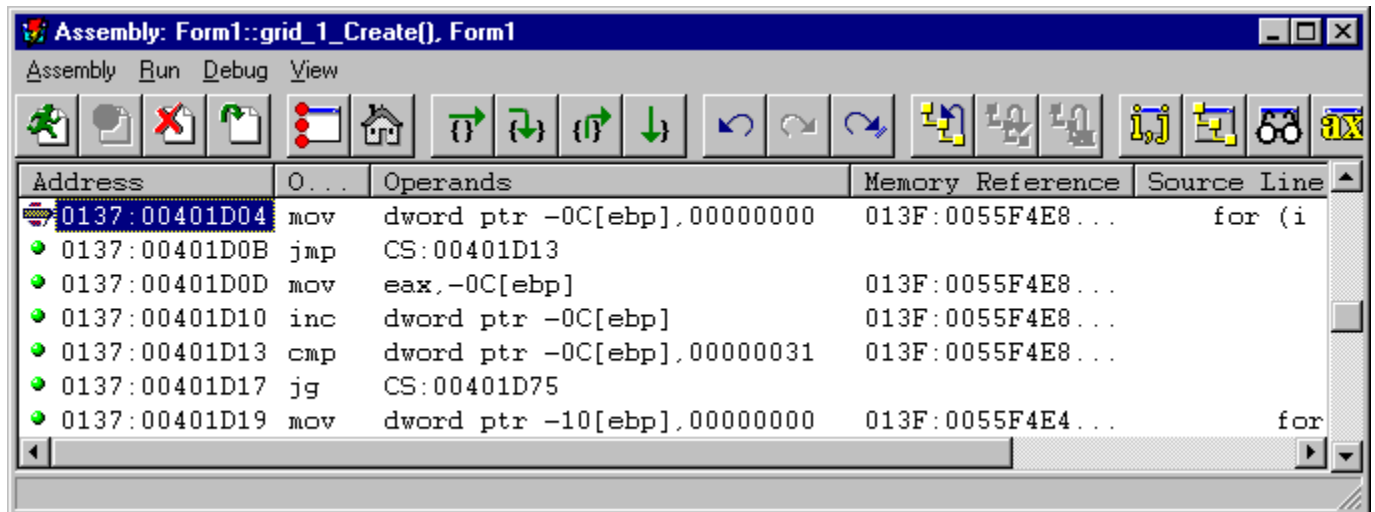
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Debug windows](#)

Assembly window

The *Assembly* window displays an assembly language version of executable code in your program.



◆ 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**. Jato 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, Jato 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](#).

[Jato Programmer's Guide](#)

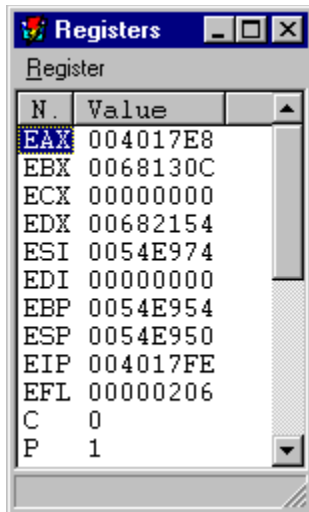
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Debug windows](#)

The Registers window

The *Registers window* displays the contents of the computer's hardware 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 Jato 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.) Jato 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.

[Jato Programmer's Guide](#)

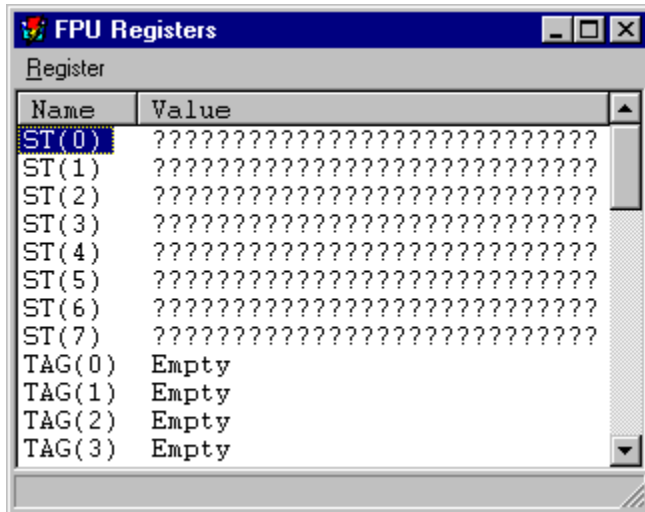
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Debug windows](#)

FPU Registers window

The *FPU Registers* window displays information about the 80x87 FPU of your computer.



◆ **To see the FPU Registers window:**

1. From the **Debug** menu of the code editor, click **FPU Registers**.

The FPU Registers window lets you change floating-point register values in the same way as the Registers window changes normal register values.

If you use the right mouse button to click any register value in the FPU Registers window, Jato displays a context menu of actions you can perform on that register.

[Jato Programmer's Guide](#)

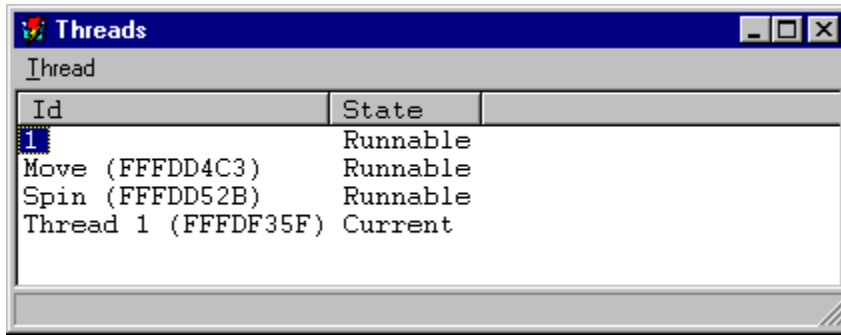
[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Debug windows](#)

Threads window

The *Threads* window displays information on the execution threads of your program.



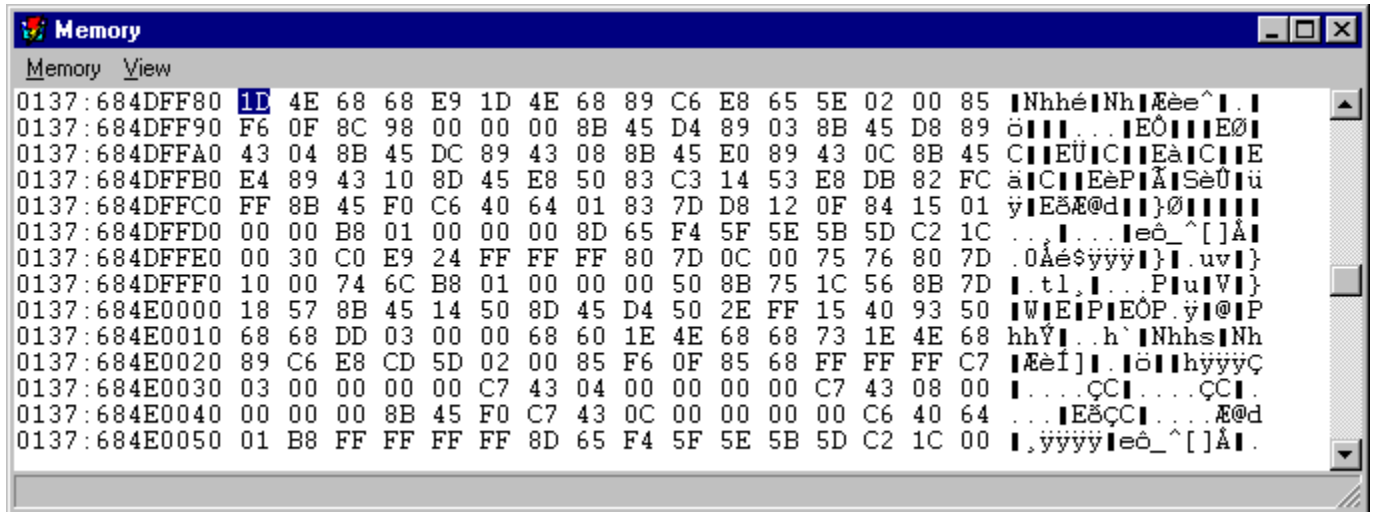
◆ **To see the Threads window:**

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.



◆ To see the Memory window:

1. From the **Debug** menu of the code editor, click **Memory**. Jato 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 using the format *segment:address* (hexadecimal) or with any Java expression referring to a memory location (for example, `&var` or the name of a function).
3. Click **OK**.

The beginning of each line in the Memory window gives a memory address, in the form *segment:address*. The remainder of the line shows the contents of 16 bytes, beginning at the specified address. The Memory window has scroll bars to let you scroll forward and backward in memory.

Data formats

By default, the Memory window displays memory byte by byte. Each line in the window shows 16 bytes in hexadecimal format, then the same bytes as ANSI characters. If a byte value does not correspond to a printable character, it is shown as a thick vertical bar in the ANSI display.

The **Type** menu of the Memory window lets you choose a different format for displaying the contents of memory. Possible formats are:

Byte

The default, showing bytes in hexadecimal and as ANSI characters.

Word

16-bit words are shown as hexadecimal integers.

DWord

32-bit double-words are shown as hexadecimal integers.

Char

Each byte is shown as a signed decimal integer.

Short

Each 16-bit word is shown as a signed decimal integer.

Long

Each 32-bit double-word is shown as a signed decimal integer.

Unsigned Char

Each byte is shown as an unsigned decimal integer.

Unsigned Short

Each 16-bit word is shown as an unsigned decimal integer.

Unsigned Long

Each 32-bit double-word is shown as an unsigned decimal integer.

0:16 Pointer

Each 16-bit word is shown as a hexadecimal integer (representing a plain pointer).

16:16 Pointer

Each pair of 16-bit words is shown in hexadecimal format, displayed together in the form *segment:address*.

0:32 Pointer

Each 32-bit word is shown as a 32-bit pointer value.

16:32 Pointer

Triplets of 16-bit words are shown as *segment:address* pointers.

Float

Each 4-byte double-word is shown as a floating-point value (using scientific notation).

Double

Each 8-byte quadruple-word is shown as a double precision floating-point value (using scientific notation).

Extended Float

Each 10-byte quantity is shown as a single extended floating-point value (using scientific notation).

Patching memory

The Memory window lets you change any value shown in the window. This process is called *patching* the memory location.

◆ To change a value in memory:

1. In the Memory window, double-click the value you want to change. Jato displays a dialog box that lets you type a new value for that memory location.
2. Type the value you want to place in that location, using the same format as the original value. Then click **OK**.

If you use the right mouse button to click an entry in the Memory window, Jato displays a context menu containing the same options as the **Memory** menu. These are:


Modify


Asks you to enter a new value for the selected memory location.


Show Address

Lets you use the Memory window to look at a different area of memory. Enter the address of the area you want to look at.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Debug windows](#)

The Stack window


The Stack window is actually a Memory window of the current stack. (The stack is the area of memory whose address is found in the ESP register.) Jato programs use the stack to store function arguments, local variables, and function call information (for example, the return address).


◆ **To see the Stack window:**


1. From the **Debug** menu of the code editor, click **Stack**.

If you use the right mouse button to click an entry in the Stack window, Jato displays a context menu containing the same options as the **Memory** menu.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Debug windows](#)


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, Jato 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, Jato 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, Jato assigns that value to the object. This works for all relevant debugging windows: the Locals window, Watches window, Assembly window, Registers window, FPU Registers window, and Memory window. You can also use drag-and-drop between debugging windows and code editor windows.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)


Stepping through your code


When a breakpoint is encountered, Jato suspends your program so that you can examine its code and data. At this point, Jato offers several ways in which you can resume program execution:


- **Run:** Jato lets the program run normally from the point it was suspended. If the program hits another breakpoint, it will suspend execution again.
- **Restart:** Jato 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. Jato confirms that you really want to do this before restarting the program.
- **Terminate:** Jato kills the program entirely, returning to your original Jato session. If you choose this, you will lose any unsaved data that you may have provided for the program. Jato 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](#)

 [Skip to Cursor](#)


 [Step over](#)


 [Step into](#)

 [Step out](#)

 [Step next](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Stepping through your code](#)

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**, Jato 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Stepping through your code](#)

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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Stepping through your code](#)

Step over

The **Step Over** action steps through your code one source statement at a time. In the code editor or Assembly window, Jato displays a triangular arrow, 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**, Jato 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, Jato 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**, Jato 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 Jato 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 Jato, 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.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Stepping through your code](#)

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.

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Stepping through your code](#)

Step next

The **Step Next** action tells Jato 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, Jato 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;
}
```

Jato 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, Jato 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, Jato displays an Assembly window showing the assembly code that was being executed.

Note: The **Break** operation will only work if your program is responding to messages. For example, you can't break into your program if it is deadlocked on semaphores.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

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).


 [Jato Programmer's Guide](#)


 [Part I. Fundamentals](#)


 [Chapter 7. Debugging](#)

Source code folders

The **Source Folders** page of the **Options** dialog lets you specify folders where Jato should look for source code, other than the folders referenced in the OBJ files of your project. For example, if you are linking your program to a library whose source code is given in some other folder, you can use **Source Folders** to specify the pathname of that folder. This lets Jato find the source code, so that you can debug at the source-code level instead of the assembly-language level.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

Remote debugging

Remote debugging lets you use the debugging facilities of Jato on your own computer while running the program on another computer connected via TCP/IP networking. There are a number of reasons why you might want to do this. For example, if you are developing your program on a Windows 95 system but want to debug its execution on a Windows 3.1/Win32s system, you must use remote debugging. In this case, you use the debugging facilities of Jato on your Windows 95 system while the program itself runs on the remote Win32s system.


Terminology: When you are running a remote debugging session, the system running Jato is called the *local* system. The system running the program that you want to debug is called the *remote* system.

You must have access to the other computer to operate the program running on it. You may find it most convenient to have the two computers side by side while using remote debugging.


Remote debugging works by setting up a TCP/IP connection between your system and the remote system. When you tell your system to run an Jato target, the system copies the target program to the remote system and starts the program executing on that system. The larger the program, the longer it will take to copy the executable file over the TCP/IP connection.


Once the program begins executing, you can debug it in much the same way that you debug a program running on your own system, except that you must use the other computer to operate the other program's user interface. Jato interacts with the remote program through the TCP/IP connection. It sends any debugging instructions you enter (for example, operations that examine or patch memory in the remote program). Because data is being sent back and forth, there may be delays in the speed with which the program responds to input. Apart from this, however, there is no essential difference between debugging on your own system and debugging a remote program.


 [Requirements for remote debugging](#)

 [Starting a remote debugging session](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Remote debugging](#)

Requirements for remote debugging

In order to run a remote debugging session, you must have TCP/IP running on both computers. The remote system must also have the following files:

```
tcpserv.exe  
std.dll  
PView.dll
```

You should place all these files in the same folder so that `tcpserv.exe` can locate the DLLs. This folder does not need to be in your execution search path.

The remote system does not need to have Jato installed on it. However, your system must have the full Jato package installed.

You must start the `tcpserv` program on the remote computer before you can begin a remote debugging session.

When you run a remote debugging session, Jato checks for the presence of a number of DLL libraries on the remote computer. If the libraries are not there, Jato automatically transfers them to the remote computer over the TCP/IP connection. This means that it may take some time to start the first remote debugging session on another system. After the first time, the library files will be present on the remote system, so they won't have to be shipped again.

The same process takes place with DLL libraries needed by the program to be debugged. Jato ships these libraries to the remote computer so that they are available when the program runs. The libraries are shipped the first time you run a remote debugging session, and whenever the library version on the remote system is out of date with respect to the version on the local computer.

Starting a remote debugging session

If you want to use remote debugging, you must configure the target to use remote debugging. This setting will be saved for the target between Jato sessions.

◆ **To specify remote debugging for a target:**

1. On the remote computer, start the `tcpserv.exe` program. If you want to connect via a particular socket, you must specify that socket number in the options dialog. (To do this, click **Disconnect** and **Options**, type the socket number, then click **Connect**).
2. In the Targets window, use the right mouse button to click the name of the target, then click **Run Options** and click **Remote**. Jato displays the Remote Debugging Options box.
3. Click **Run on Remote Machine** so that it is checked.
4. Under **Remote machine name or IP address**, type the IP address of the remote computer. This can be specified as a host name (as in `galahad.camelot.com`) or a numeric IP address (as in `123.123.123.123`).
5. If you want to connect to the remote computer via a particular socket, type the socket number under **TCP/IP socket number**.
6. Click **OK**.
7. From the **Run** menu, click the **Run** menu item. This starts executing the program on the remote computer, under the debugging control of Jato running on your local computer.

When a target is configured to use remote debugging, the target program is copied to the remote computer and executed there every time you run the program. Although you can control the debugging on the local computer, you must use the remote computer to use the interface of the program.


When you no longer want to have that target set up to use remote debugging, open the Remote Debugging Options dialog box and click **Run the program locally**.


Note: If you change the socket number in **Specify Socket**, you must specify that socket number in the remote computer's `tcpserv` options dialog box every time you start `tcpserv.exe`. You can also specify the socket number as a command line argument when you start `tcpserv`, as in

```
tcpserv 1031
```

In most cases, there is no reason to change sockets. By default, Jato uses socket 3563 (0xDEB) which is generally unused.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

Run options


Run options let you control the way that Jato runs your application. The section [Remote debugging](#) showed one use of these run options: to run a program on a remote system. This section looks at other ways to use run options.


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

1. In the Targets window, use the right mouse button to click the name of the target, then click **Run Options**.

Jato displays the Run Options dialog box. This box has different forms depending on the type of target. The sections that follow describe the different forms this box may take.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)


 [Chapter 7. Debugging](#)

Debugging techniques

The preceding sections describe the basic debugging tools of Jato. 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.

<p>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 <code>Debug</code>.</p>

 [Modular testing](#)

 [Searching for bugs](#)

[Jato Programmer's Guide](#)

[Part I. Fundamentals](#)

[Chapter 7. Debugging](#)

[Debugging techniques](#)

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.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`.

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Debugging techniques](#)

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.


 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)


 [Chapter 7. Debugging](#)


Performance tips


This section offers tips for improving performance: the performance of Jato at design time, and the performance of the generated program at run time.

 [The Windows temporary folder](#)

 [Jato Programmer's Guide](#)

 [Part I. Fundamentals](#)

 [Chapter 7. Debugging](#)

 [Performance tips](#)

The Windows temporary folder

The Windows system creates temporary files for various purposes during program execution. Temporary files are created in a folder specified by the `TEMP` environment variable; if you do not define this environment variable, the system uses a subfolder named `Temp` under the main windows folder (for example, `C:\Windows\Temp`).

Normally, the system deletes temporary files when they are no longer needed. However, files may not be deleted if the program that is using them crashes during execution. If you have a lot of crashes during the process of debugging a program, this can lead to a build-up of useless files in the temporary folder. This has two undesirable effects:

- The files waste disk space.
- Roughly speaking, the more files there are in a folder, the longer it takes to create a new file in that folder. Therefore, if you already have a lot of files in your temporary folder, the system takes longer to create a new temporary file. This slows down every program that needs temporary files, including programs created with Jato, and the Jato program itself.

To avoid these problems, you should regularly check the temporary folder used by the system and delete any files that may be left over from program crashes. Such files have the extension `.TMP`.

Make sure that you don't have other programs running when you go in to clear out the old temporary files. That way, you don't have to worry about files that are actively in use by other programs.

Summary of debugging

Debugging Tools

A breakpoint pauses program execution at a particular code location. While the program is paused, you may examine and manipulate the program using a number of tools:

- The Call Stack window displays what functions were executing at the time of the break.
- The Locals window displays the local variables defined within an executing function.
- The Watches window displays a number of expressions whose value you want to watch.
- The Assembly window displays an assembly code version of executable code.
- The Registers window displays the values of hardware registers.
- The FPU Registers window displays information about the 80x87 FPU of your computer.
- The Threads window displays information about the threads of your program.
- The Memory window displays the contents of memory in a variety of formats.

You can use these windows to modify (patch) values in memory or in the registers of your program.

The debugging tools have extensive drag-and-drop capabilities. For example, you can patch a memory location by dragging a value from some other window and dropping it in a window showing that memory location.

Stepping

Jato lets you step through your program in various ways:

- **Step Into** executes a single instruction. If this instruction contains a function call, **Step Into** stops at the first instruction of the function.
- **Step Over** executes a single instruction. **Step Over** does not step into functions; in other words, it executes the current instruction as a whole, including any function calls.
- **Step Out** executes up to the point where the current function returns to its caller.
- **Step Next** executes up to the next line of source code. It is typically used at the end of a loop construct, to run through any remaining iterations of the loop.
- **Run to Cursor** executes up to the line of code containing the cursor.
- **Skip to Cursor** begins execution with the line of code containing the cursor, skipping any intervening code.

If you are paused at a breakpoint, **Run** resumes normal execution of your program, **Restart** begins program execution from the start again, and **Terminate** terminates the program.

Remote debugging







Remote debugging lets you run Jato on one computer (the local system) in order to debug a program that is running on another computer (the remote system). For example, the local system might be running Windows 95 while the remote system is running Win32s.

Remote debugging works by establishing a TCP/IP connection between the local system and the remote system. In order for this to be possible, the remote system must be running `tcpserv` and must have `std.dll` installed. If Jato needs any other files present on the remote system, it ships the files via the TCP/IP connection before beginning the debugging session.

 Jato Programmer's Guide

Part II. Advanced topics

This part describes advanced Jato programming topics.

-  Chapter 8. Working with databases
-  Chapter 9. Writing Internet applications
-  Chapter 10. Using and creating JavaBeans
-  Chapter 11. Using ActiveX components and servers
-  Chapter 12. Using threads
-  Chapter 13. Using graphics and printers

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

Chapter 8. Working with databases

This chapter discusses how you can use Jato to work with JDBC databases, and ODBC databases accessed by the JDBC-ODBC bridge. This guide only makes no attempt to explain the JDBC or ODBC standards themselves. It only discusses the features of Jato that let you access JDBC and ODBC functionality.

Note: For a tutorial example of using databases, see the Getting Started guide.

[Transaction objects](#)

[Query objects](#)

[Bound controls](#)

[Moving through the result set](#)

[Bound list boxes and combo boxes](#)

[Making changes in the database](#)

[Database events](#)


[The data navigator](#)


[The Query Editor](#)

[Bound parameters](#)

[Database dialog forms](#)

[Summary of working with databases](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

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 JDBCTransaction class. Jato gives transaction objects default names of the form *transaction_N*. On the **Database** page of the Java component palette, transaction objects are represented by the following button:





The JDBCTransaction class is derived from a base Transaction class that specifies methods for dealing with many types of databases. Transaction defines generic database operations while JDBCTransaction is an implementation of those operations specifically for JDBC. Therefore, this guide usually discusses transaction objects in terms of the base Transaction class rather than the specific JDBCTransaction class.

If a form's code interacts with a database, you must place a transaction object on the form or on a parent of the form. 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 properties](#)

 [Setting up transaction information at run time](#)

 [Connecting to the database](#)

 [Managing transactions manually](#)

 [A hint for setting up transactions](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Transaction objects](#)

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. The **General** page of the property sheet contains the following items:

JDBCDriver

The JDBC driver used to interact with your database. Click the arrow in the **JDBCDriver** combo box to see which drivers are currently supported.

Note: To access an ODBC database, use the driver named `sun.jdbc.odbc.JdbcOdbcDriver`. This driver is often called the JDBC-ODBC bridge because it performs the translations needed to execute JDBC operations on an ODBC database.

DataSource URL

A URL specifying the location of the database.

UserID

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

The password that the program should use to connect to the database. For security, Jato 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

Any extra information needed for connecting to the database. The nature of this information depends on the type of database management system. The information is passed to the database system in the manner expected by that type of database.

AutoConnect

If this is checked, Jato 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

If this is checked, each database operation is committed as it is completed.

TraceToLog

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. The default is `false`.


TraceToLog only has an effect in Debug mode; it does nothing in Release mode.


Note that this property controls information written to the Jato debug log, not the database's trace log.


DataSource (ODBC)

May be filled in if you want to use the query editor to create your queries. The current implementation of the Jato query editor only works with ODBC databases. Therefore, you can specify an ODBC database for the query editor to read at design time, even if the application will use a JDBC database at run time. This means that the ODBC database specified as **DataSource (ODBC)** may not be the database that the program actually uses (although it could be, if your program uses the JDBC-ODBC bridge to access the ODBC database).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Transaction objects](#)

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 );
```

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 8. Working with databases](#)

[_Transaction objects](#)

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 );
```

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();
```

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Transaction objects](#)

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**. One way around this is to **close** the query object before you commit.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 8. Working with databases](#)


 [Transaction objects](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

Query objects


A *query object* represents a query on a specific database. 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. Jato 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 Jato Query Editor. For more information, see [The Query Editor](#).


 [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](#)

 [Executing a query](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)


 [Query objects](#)


Associating a query object with a transaction object

Every query object must be associated with a transaction object. The transaction object specifies the database on which the query will act.


In the simplest case, the query object is placed on the same form as the transaction object. The query object can also be placed on a “descendant” form of the form containing the transaction object. This means that the form containing the query must be a child of the form containing the transaction object, or a child of a child, and so on.

In order to make this possible, all the transaction objects in an application must have different names. For example, if both Form1 and Form2 contain a transaction object, the transaction objects can't have the same name.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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](#).

PrimaryKeyColumn [Options page]

Specifies the name of the primary key column in the SQL statement. If this property is not set, the query object cannot go into Edit mode.

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


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. The default is `false`.


TraceToLog only has an effect in Debug mode; it does nothing in Release mode.

Note that this property controls information written to the Jato debug log, not the database's trace log.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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 );
```


As another example, **setTransactionObject** sets the transaction object associated with the query:


```
query_1.setTransactionObject( transaction_1 );
query_1.setTransactionObject( transaction_2 );
```


You might change transaction objects if you want to use the same query object to query two different databases.

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`.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

Run-time only Query properties

A number of Query properties can only be set at run time. The following list discusses some of these:

AutoRefresh

Automatically refreshes the contents of all bound objects after any update or delete operation. For further information, see [Refreshing after a change](#).


Opened


getOpened returns `true` if the query is open and `false` otherwise. There is no **setOpened** function.


ReadOnly

getReadOnly returns `true` if the statement is not open or the concurrency level is read-only. It returns `false` otherwise. There is no **setReadOnly** function.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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** when the query object is being created.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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](#).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 8. Working with databases](#)


 [Query objects](#)


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](#).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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.isFirstRow( );  
boolean last  = query_1.isLastRow( );
```


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.getBOF( );  
boolean end   = query_1.getEOF( );
```

let you test whether you are at “beginning of file” (before the first row in the result set) or “end of file” (after the end of the first row in the result set).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Query objects](#)

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 **setSQL** method automatically closes the query before it assigns a new SQL statement to the query object.

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.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 8. Working with databases](#)

[_Query objects](#)

Executing a query

The **execute** method of Query lets you execute a SQL statement directly:


```
query_1.  
    execute( "delete from dba.employee where emp_id=12" );
```

The argument of **execute** is a String value giving the database command you want to execute. This command must not return a result set.

If the query object is open, **Execute** automatically closes it before executing the given SQL statement.

If the argument of **execute** is a null string, **execute** executes the string given by the query's **SQL** property.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

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
- Labels
- Check boxes
- List boxes
- Combo boxes


If the user changes the value of a bound control, it typically changes the corresponding value in the database. 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.

If **AutoCommit** is turned on for the transaction object, changes are immediately made permanent (committed) in the database. If **AutoCommit** is turned off, the changes do not become permanent until you invoke the **commit** method of the transaction object; before you call **commit**, you can cancel changes using the **rollback** method of the transaction object.

If updates are not allowed on a database, the user will not be permitted to change the values in bound controls. For example, if the concurrency level for the query object is read-only, the user will not be permitted to change the values displayed by the bound controls.

 [Setting up bound controls](#)

 [Checked and unchecked values](#)

 [How controls display values](#)

Setting up bound controls

If you intend to use an object as a bound control, you must click the **Bound Control** checkbox on the **General** page 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 **Create** 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 **Create** event handler for the form, because the query object is opened before the form's **Create** event is triggered.

The DataSource property

The **DataSource** property specifies the query object to which the control will be bound. For example, suppose that `textb_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

```
textb_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 `textb_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 `textb_1`. Set **DataColumns** to `emp_id`.
- To set this up at run time, use the **setDataColumns** of `textb_1`, as in:


```
textb_1.setDataColumns( "emp_id" );
```


If you specify a column by its number, the number is given as a string value:

```
textb_2.setDataColumns( "2" );
```

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Bound controls](#)

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 that the check box should be unchecked when the associated column has a particular value. 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". A three-state check box will be in its indeterminate state (grayed where the check mark would appear) if the value is neither "Here" nor "There".

At design time, you can set these properties on the **Database** page of the check box's property sheet.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Bound controls](#)

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. The check box is unchecked if the value of the associated column equals the value of the check box's **DataValueUnchecked** property. Three-state check boxes are grayed out if the value does not match either **DataValueChecked** or **DataValueUnchecked**.
- List boxes and combo boxes can display values in several different ways. For more information, see [Bound list boxes and combo boxes](#).

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

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 a boolean argument named `notify`. This is explained later in this section.

The **move** methods are:

`move(row, notify)`

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.

`moveFirst(notify)`

Moves the cursor to the first row of the result set. The `row` argument is ignored.

`moveLast(notify)`

Moves the cursor to the last row of the result set. The `row` argument is ignored.

`moveNext(notify)`

Moves the cursor to the next row of the result set. The `row` argument is ignored.

`movePrevious(notify)`

Moves the cursor to the previous row of the result set. The `row` argument is ignored.

`moveRelative(offset, notify)`

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.

The **move** methods always execute an **update** action before moving from the current row. If the `notify` argument of the **move** method is `true`, the **update** action notifies all bound controls that they should refresh themselves; otherwise, **update** does not do this. For more information on the **update** method, see [Making changes in the database](#).

[Getting a value from the current row](#)

[Finding the column index](#)

[Column information](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Moving through the result set](#)

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;  
DataValue dv = query_1.getValue( column );
```

where `column` is the number of the column whose value you want. The result of **getValue** is a `DataValue` object.

`DataValue` is a type that encapsulates all the data types that can be returned from the database; it has methods to cast values into other types. For example, suppose `dv` is a `DataValue` object; the following lines show a few ways to interpret the value as various Jato types.

```
String s = dv.getCHAR();  
%%% Many more to come
```

As an example, if you want to display data value from column 2 in a text box, you can write

```
textb_1.setText( query_1.getValue(2).getCHAR() );
```

For more information on `DataValue` and its conversion methods, see the Jato Component Library Reference.

[!\[\]\(cead67df4d82d6c83effe4f8699a7d8f_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(1d3a1175dd4902218e694b9c098adb83_img.jpg\) Part II. Advanced topics](#)

[!\[\]\(c507f772dba2b921f86777f01218e570_img.jpg\) Chapter 8. Working with databases](#)

[!\[\]\(4729e517bc6a7cd81c8025b9646574fb_img.jpg\) Moving through the result set](#)

Finding the column index

The **getColumnIndex** method determines the column index corresponding to a given name. For example,

```
int i = query_1.getColumnIndex( "Employee_ID" );
```

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Moving through the result set](#)

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 );
```

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 index = dc.getIndex();           // column number
int length = dc.getLength();         // column length
int dSize = dc.getDisplaySize();     // display size
long prec = dc.getPrecision();       // precision
int scale = dc.getScale();           // scale
String label = dc.getLabel();         // column label
String name = dc.getName();          // column name
boolean readOnly = dc.getReadOnly(); // read-only?
boolean uns = dc.getUnsigned();      // unsigned?
```

For more information on DataColumn, see the Jato Component Library Reference.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

Bound list boxes and combo boxes


This section discusses the use of list boxes and combo boxes as bound controls.


There are two different ways to use list boxes:

- 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 box 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 box 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 list box property named **DataBindAsLookup**. For further information, see [Using list boxes 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 list boxes in list mode](#)

 [Using list boxes in lookup mode](#)

 [Bound combo boxes](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Bound list boxes and combo boxes](#)

Using list boxes in list mode

In list 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.

Note: Users may have trouble working with a list box that contains a huge number of entries. If you expect a query to have a large result set, consider using a list view instead of a list box.

To put a list box into list mode, set **DataBindAsLookup** property to `false` (leave it blank on the property sheet). 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. You can specify either one or two columns. If you only specify one column, the list box is bound to that column and displays values from that column. If you specify two columns, the list box displays values from the first column; values from the second column are retrieved as `DataValue` values and stored as the `userdata` value for each list box item.

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 changes to a different row. 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.

By default, **DataTrackRow** is `false`.

DataLookupSource, DataLookupColumns

Ignored in list mode. If you try to change the values of either of these properties in list mode, the operation fails.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Bound list boxes and combo boxes](#)

Using list boxes in lookup mode

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 **DataBindAsLookup** 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 **add** or **dataAddLookupItem** method of the list box.
- Obtaining the list with a database query. For example, if the list box is suppose 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.

You can also combine these two approaches. For example, you can obtain a basic list of possibilities using a database query, then add more items manually. You can also do this in the opposite order: start off by adding some items manually, then obtain more possibilities using a database query.

Note: For an example of using bound list boxes, see Database Data Lookup List in the Jato sample programs.

Manually setting up lookup lists

To place a lookup value directly into the list box, use the **dataAddLookupItem** method. This method has the prototype:

```
int dataAddLookupItem( DataValue value,  
    String string,  
    int itemNumber,  
    Object itemUserData,  
    boolean addInSortedOrder );
```

The parameters are:

value

An actual value that can appear in the column. Note that this is a `DataValue` object.

string

The actual string that should appear in the list box item. If this value is `null`, the list box shows the given *value*.

itemNumber

Specifies where this value should be placed in the list box list. An item number of `-1` means the end of the list.

itemUserData

Specifies any user data that should be associated with the item.

addInSortedOrder

`true` if you want to add the item in alphabetic (or numeric) order, and `false` otherwise. If this

argument is `true`, **dataAddLookupItem** ignores the *itemNumber* argument.

As an example of the relationship between *value* and *string*, suppose that list box is going to list all the departments in a company. The *value* might be an ID number used to identify a particular department, while the *string* 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.

Instead of using **dataAddLookupItem**, you can also use the normal **add** method of `ListBox` to add items to this list box. In this case, the "value" of an item is its text.

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. This example appears as `Lookup Mode List Box` in the `Jato` sample programs. 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.

emp_id	emp_fname	emp_lname	dept_id
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
195	Marc	Dill	200

Employee first name: Philip

Employee last name: Chin

dept_id	dept_name
100	R & D
200	Sales
300	Finance
400	Marketing
500	Shipping

Lookup list box shows employee department

- R & D
- Sales
- Finance
- Marketing
- Shipping

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** 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Bound list boxes and combo boxes](#)


Bound combo boxes


A bound combo box is similar to a bound list box in lookup mode. The text box part of the combo box shows the value of a column in the current row, and the list box part shows all the possible values for that column. The difference is that a lookup mode list box only lets you select entries that are already in the list; a bound combo box lets you select existing entries or enter a new value in the text box part of the combo box.

A bound combo box 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 combo box using the **DataAddLookupItem** method of the combo box. The method has the same format as for list boxes.

If you edit the contents of the combo box's text box directly or by selecting a lookup item, the same change is made in the associated column value of the current row of the database.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

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; 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. This means you must set an appropriate access mode for the transaction object and concurrency level for the query object.


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 immediately. 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](#)

 [Adding new rows](#)

 [Modifying existing rows](#)

 [Refreshing after a change](#)

 [Canceling changes](#)

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 8. Working with databases](#)

[_Making changes in the database](#)

Deleting existing rows

The **delete** method of the query object deletes the current row:

```
// boolean notify;  
boolean success = query_1.delete( notify );
```

If `notify` is `TRUE`, bound objects are updated to show that the row has been deleted. This is done by triggering a **DataAvailable** event on each bound object. For more information on the **DataAvailable** event, see [DataAvailable](#).

If `notify` is `FALSE`, bound objects are not updated.

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Making changes in 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:

```
query_1.add( false, false, false );
textb_1.setText("Value 1");
textb_1.setText("Value 2");
query_1.update( true );
```

The **add** method has the prototype:

```
boolean add( boolean copyValues,
            boolean append,
            boolean copyIntoBuffer );
```

The parameters are:

copyValues

If this 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).

append

If this is `true`, **add** puts the new row at the end of the current result set. If *append* is `false`, **add** places the new row after the current row.

copyIntoBuffer

If this is `true`, **add** copies column values from the new row into any arrays bound to those columns. If *copyIntoBuffer* is `false`, **add** does not copy the values.

The **update** method has the full prototype:

```
boolean update( boolean notifyTargets );
```

where:

notifyTargets

If this is `true`, **update** triggers a **DataAvailable** event on each bound control. For further details, see [DataAvailable](#).

If the **AutoRefresh** property is turned on, the columns of the new row will be shown in the position where they are placed by the database. This position depends on the way that the database is sorted. For more information, see [Refreshing after a change](#).

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: If you open a query with an `ORDER BY` modifier in the SQL statement, the database creates a special temporary table when you add a new row to the database. The new row stays in that temporary table until you close the query; at that point, the row is added to the database in its proper place. Because of this, you can't see the any new rows you add to the database until you close and reopen

the query, even if your cursor type is dynamic.

The same thing happens in various other situations: for example, when there is a `GROUP BY` or `DISTINCT` clause that can't be satisfied by an index, or when you specify `UNION` but not `UNION ALL`. Other database products typically create temporary tables in similar situations.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[Making changes in 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 or bound arrays.
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:

```
query_1.edit();  
textb_1.setText( "Value 1" );  
textb_1.setText( "Value 2" );  
query_1.update( true );
```

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.

Even if **AutoEdit** is `true`, you must call **edit** if you are changing the row by directly modifying data in bound columns. In other words, if you use **setValue** to change a value, **setValue** calls **edit** automatically (when **AutoEdit** is `true`); however, if you store values in bound arrays without using **setValue**, you must call **edit** yourself.

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](#).

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 8. Working with databases](#)

[_Making changes in the database](#)

Refreshing after a change

The **refresh** method of a query object synchronizes the contents of bound objects with the current contents of the database:

```
query_1.refresh();
```

The effect is similar to closing the query then opening it again, but is more efficient. The **refresh** method also attempts to set the current row to the row you were at before calling **refresh**.

If the **AutoRefresh** property is turned on for the query object, your program automatically performs a **refresh** operation after every **update** or **delete** operation. If **AutoRefresh** is turned off, the query objects are not refreshed until you call **refresh** explicitly. In this case, the run-time support environment makes its best guess at the effects of each operation, without actually checking the real effects against the database.

A **refresh** operation can take a good deal of time, especially with large databases or queries that return a lot of data. If you are performing a series of operations that should have a predictable effect (such as adding an ordered sequence of rows), you can improve performance by turning off **AutoRefresh** and calling **refresh** explicitly at the end of the operations.

However, if you are performing operations that may change the order of the database rows, it could be safer to **refresh** after every change. Otherwise, bound controls like list boxes may get out of synch with the database, and subsequent operations may not be performed correctly.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 8. Working with databases](#)

[_Making changes in the database](#)

Canceling changes

The **cancelUpdate** 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.cancelUpdate( true );
```


This resets the bound controls to their previous values. If you do not need to reset the bound controls, you can use


```
query_1.cancelUpdate( false );
```

You must call **cancelUpdate** before calling **update**; once you call **update**, you can't cancel the changes.

The **cancelUpdate** 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, **cancelUpdate** cancels the operation.

 [Jato Programmer's Guide](#)


 [Part II. Advanced topics](#)


 [Chapter 8. Working with databases](#)

Database events


This section examines events related to working with databases.


 [DataAvailable](#)

 [AdjustCursor](#)

 [Event timing](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)


 [Database events](#)


DataAvailable


The **DataAvailable** event is triggered on all bound controls whenever the data in the current row changes: when current row values are modified or the cursor moves.

Most users will never have to write a **DataAvailable** event handler—the default handlers automatically update bound objects as desired.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 8. Working with databases](#)


 [Database events](#)

AdjustCursor

The **AdjustCursor** event of Query takes place immediately after a **move** operation changes the current row. Most user programs will not respond to this event, but it is used by the data navigator. (See [The data navigator](#) for more details.)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Database events](#)

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 **Create** event is triggered, since the **Create** event is not triggered until all the objects on the form have been properly created and initialized.

This means that bound controls will receive a **DataAvailable** event (and possibly other database-related events) before the **Create** event for the form. By the time the form's **Create** event is triggered, the bound controls will control their first values, as obtained from the database.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

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 buttons with arrows on them; clicking these buttons automatically moves back and forth through the database in a manner similar to the **move** methods of 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 **cancelUpdate** on the query).
- Refresh the result set (by executing **refresh** 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. Jato gives data navigators default names of the form *dataNavigator_N*. On the **Database** page of the Java 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](#)

[Binding the data navigator to a query](#)

[Action properties](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[The data navigator](#)

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 **General** page of the data navigator's property sheet. For example, if you turn off **ShowRefresh**, the **Refresh** button will not appear on the data navigator. You can also use


```
dataNavigator_1.setShowRefresh( false );
```


to make the **Refresh** button disappear at run time, or

```
dataNavigator_1.setShowRefresh( true );
```

to make the button appear.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The data navigator](#)

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 navigator's property sheet.
- At run time, you can bind the data navigator with **setDataSource**:

```
dataNavigator_1.setDataSource( query_1 );
```


[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[The data navigator](#)

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 in 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 in the data navigator's property sheet, or at run time with **setBOFAction**.

In order to handle EOF and BOF actions, the data navigator specifies an **AdjustCursor** event handler for the query object. If the cursor is about to be 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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

The Query Editor

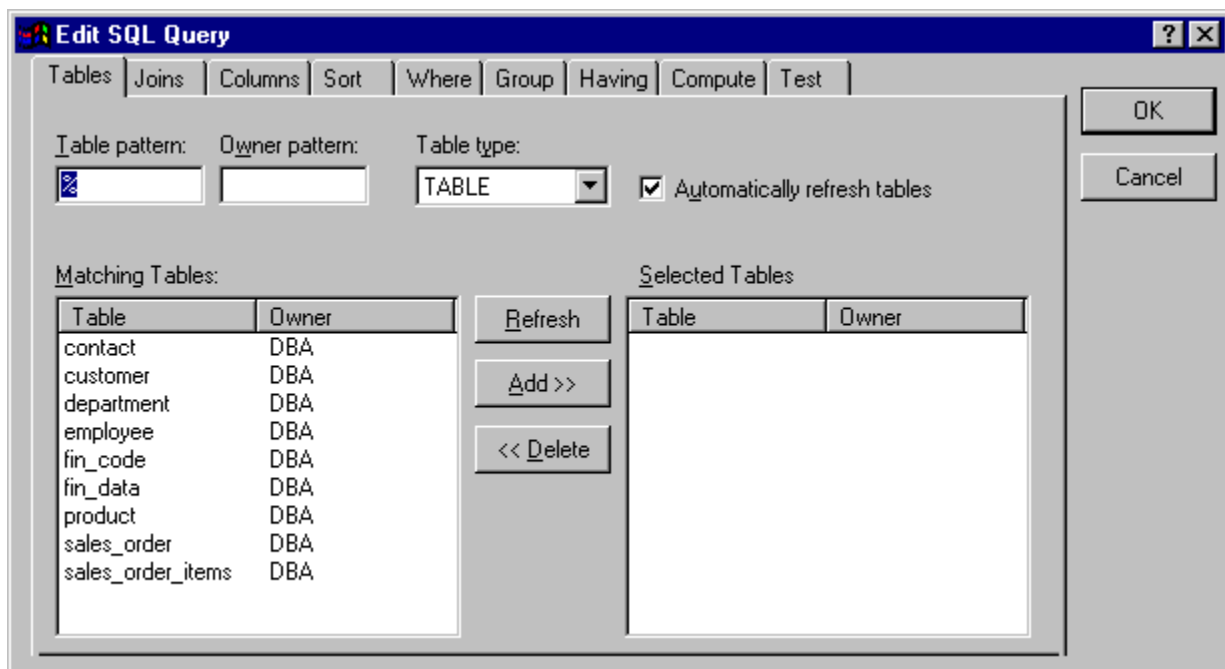
The *Query Editor* helps you construct the SQL statement that is associated with a Query object. The Query Editor is opened at design time, once you have placed the Query object on a form and have bound the object to a valid transaction object. The Query Editor uses the information specified in the transaction 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.

Important: %%% In this beta version of Jato, the query editor must work with an ODBC database—the query editor does not support JDBC databases. Some applications may be able to construct a query with the query editor using the ODBC database at design time; then at run time, the applications can access the ODBC database using the JDBC-ODBC bridge. For more information about the JDBC-ODBC bridge, see [Transaction properties](#).

◆ 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**, Jato 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. The sections which follow examine each page of the Query Editor.

[The Tables page](#)

[The Joins page](#)


[The Columns page](#)

[The Sort page](#)


[The Where page](#)

- [Using the Criterion Editor](#)
- [The Group page](#)
- [The Having page](#)
- [The Compute page](#)
- [The Test page](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The Query Editor](#)

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).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The Query Editor](#)


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**.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The Query Editor](#)

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**.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

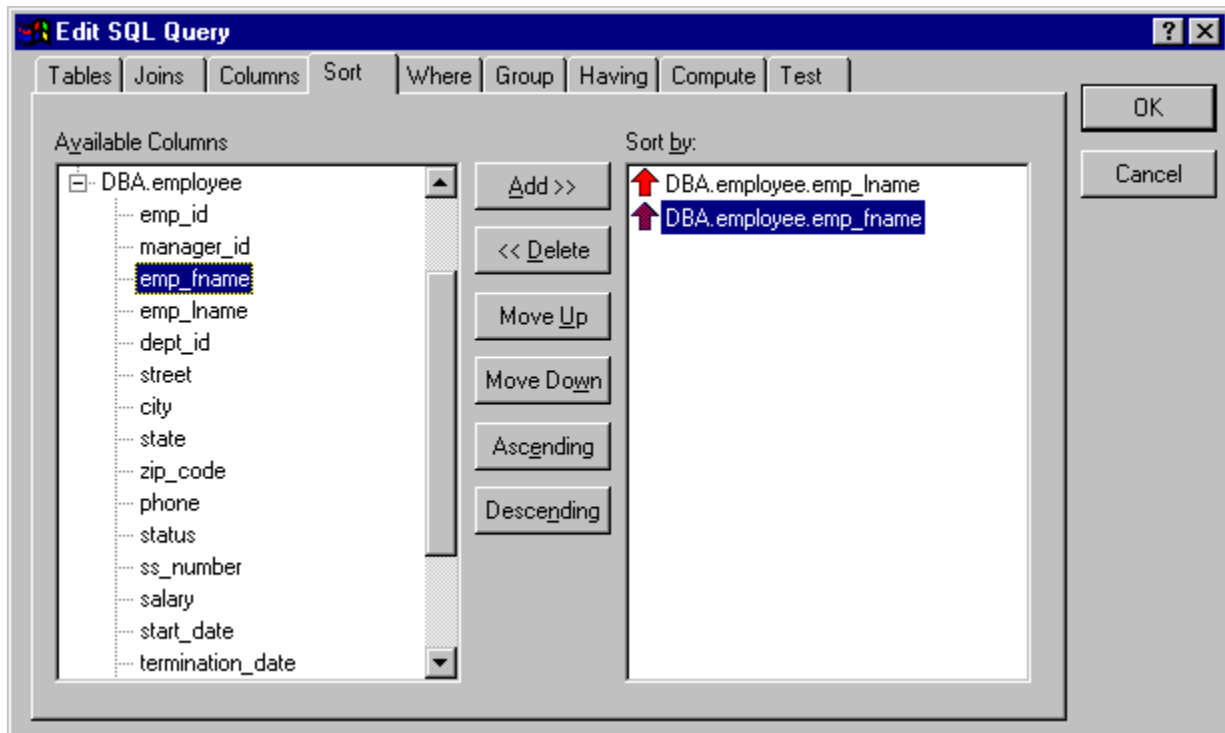
[Chapter 8. Working with databases](#)

[The Query Editor](#)

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**.

[Jato Programmer's Guide](#)

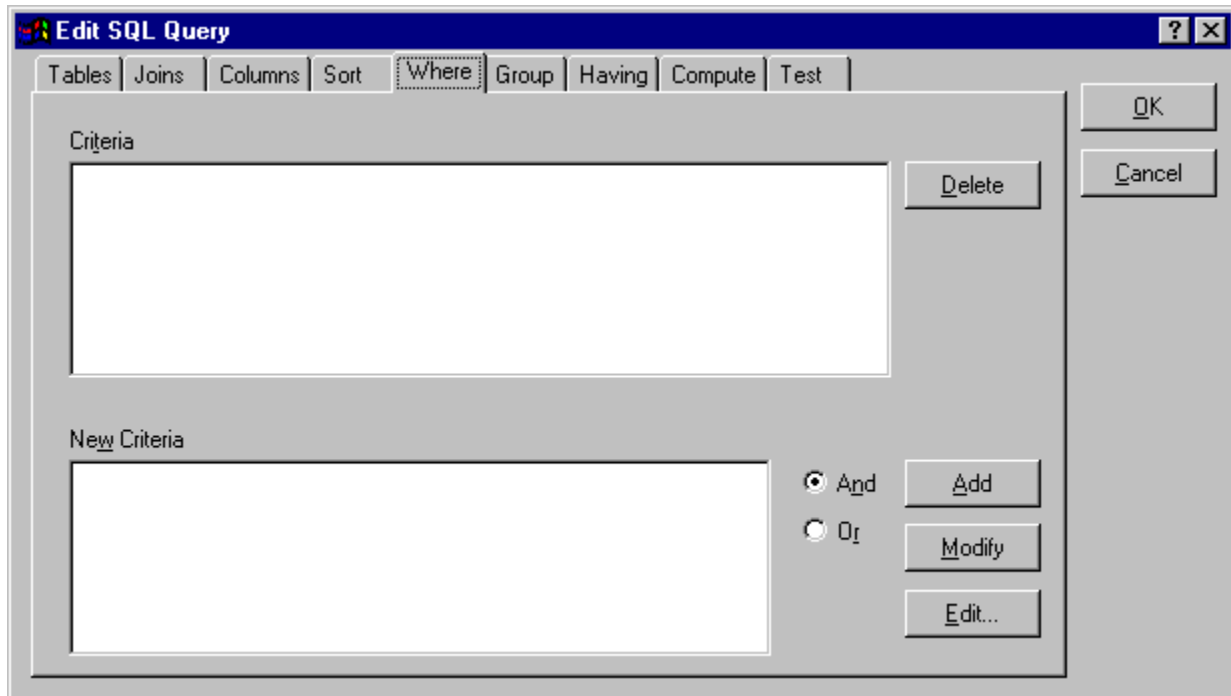
[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[The Query Editor](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The Query 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.

[Jato Programmer's Guide](#)

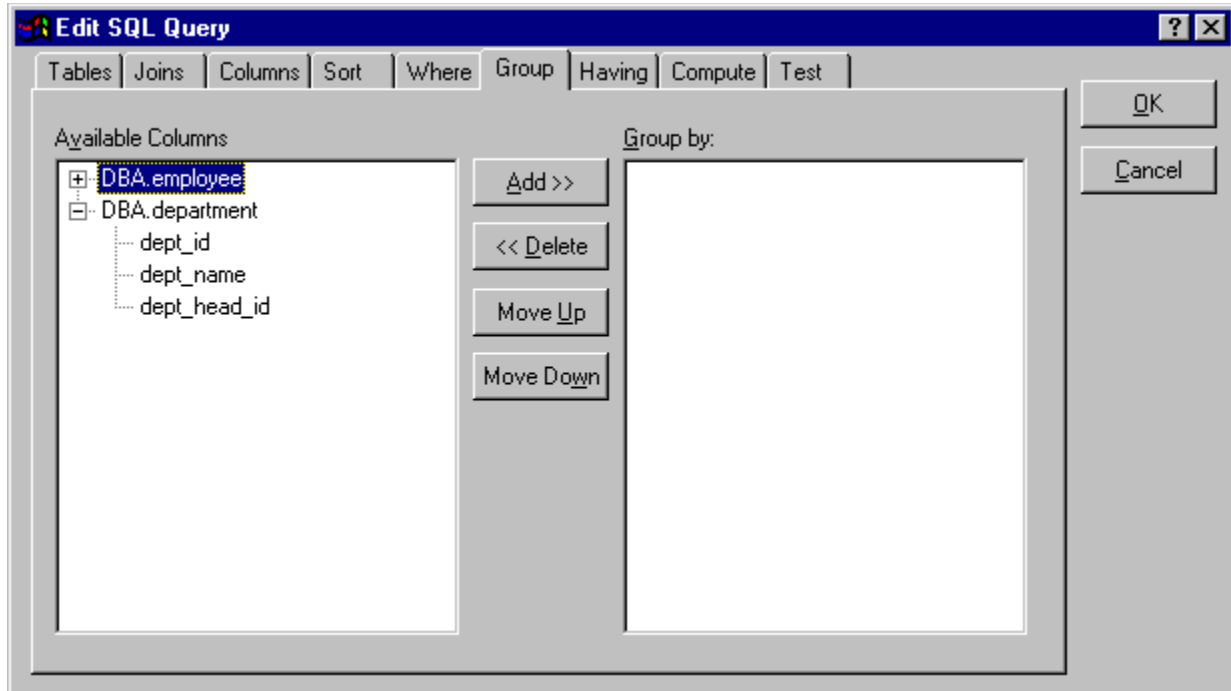
[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[The Query Editor](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The Query Editor](#)

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](#).

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 8. Working with databases](#)

[The Query Editor](#)

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](#).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [The Query Editor](#)

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.

Bound parameters

SQL queries may use the question mark (?) as a placeholder in statements like

```
SELECT * FROM employee WHERE manager_id = ?
```

These placeholders are called *parameters*. The 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.

To specify values for a parameter, 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, DataValue( 200 ) );
```

sets the first parameter in the query to a value of 200.

If you change the value of a bound parameter, you can use the **resubmit** method of the query object to submit the query again with the new parameter values. The **resubmit** method is faster than the usual **open** operation.

For more information about **setParameter** and **resubmit**, see the Jato Component Library Reference.

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 = ?
```

You would bind this to a list box with the `lname` showing and the `id` in the `userdata` field. In the **Select** event for the list box, you would fetch the `id` of the selected item and do something to associate it with the parameter in the detail query. The association is done either by setting the value of a variable which has been bound to the parameter, or perhaps by setting the parameter value directly using **setParameter**. After fetching the `id` value you would then resubmit the detail query which would fetch and display the details for the specified `id`.

Bound parameters are also useful in filling the lookup values in drop-down lists.

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.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 8. Working with databases](#)

[_Bound parameters](#)

Using bound parameters for output

You can also use bound parameters to obtain information from the database. In this case, you use **setParameter** to specify information about the parameter before opening the query, then use **getParameter** to determine the value of the parameter.

The following statement shows a typical use of **setParameter** for setting up an output parameter:

```
query_1.setParameter( 3, "input/output", SQL_CHAR,  
    WQPTInputOutput, 200 );
```

The first argument is the number of the bound parameter in the query's SQL statement. The next is an initial value for the parameter (in this case, the string "input/output". The next argument is the type of value in the data column. The second last argument specifies how the bound parameter is being used; possible values are:

```
WQPTInputOutput  
WQPTInput  
WQPTOutput
```


The final argument is the maximum size of the value. It only applies to character or binary data, and only if this bound parameter is going to receive output.


After the query is opened, you can obtain the value returned to the bound parameter using the **getParameter** method, as in


```
DataValue dv = query_1.getParameter( 3 );
```

This sample statement gets the value that was assigned to parameter 3.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)


 [Bound parameters](#)


Stored procedures

You can use stored procedures in the SQL statement associated with a query. To do this, you enclose the procedure call in braces, as in

```
query_1.setSQL( "{call test_output(?,?)}" );
```

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

Database dialog forms


%%% Not yet implemented in the beta version of Jato.


The easiest way to set up a form that uses database is to select **DatabaseDialog** from the Form Wizard. This 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](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 8. Working with databases](#)

 [Database dialog forms](#)

Master detail views

When you create a **Database Dialog** 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.

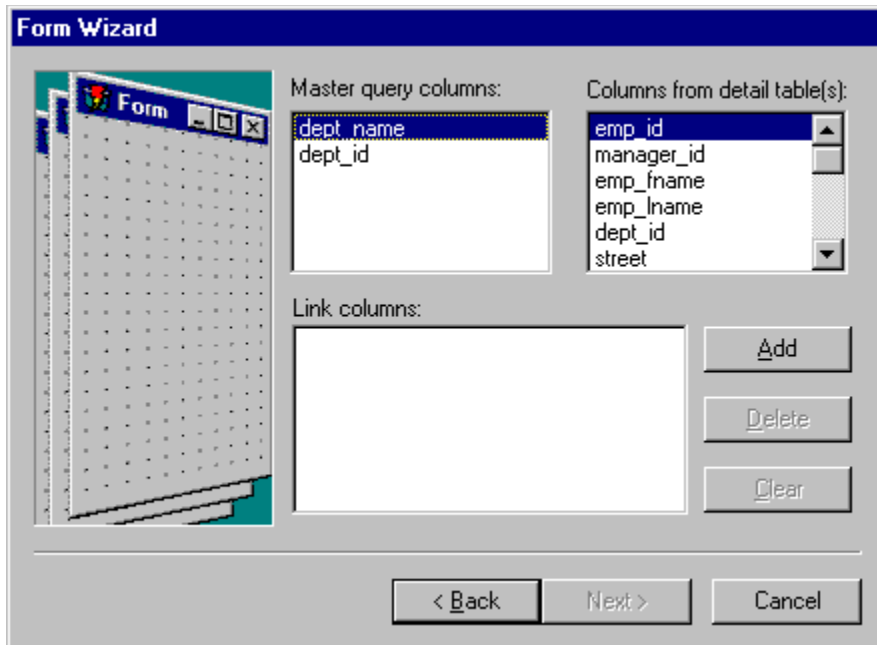
For example, the master query may obtain information about company departments and the detail query may obtain information about employees in each department. Both queries may have bound controls displaying information. For example, there might be several text boxes for displaying the department name 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.

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.

Summary of working with databases

Transaction and query objects

Any program that works with a database needs at least one transaction object and one query object:

- The transaction object specifies information for connecting with the database (for example, the name of the database, the database management system, the userid, and the password).
- The query object specifies the SQL statement used to fetch information from the database. It also specifies options for processing the query. The easiest way to create the SQL statement in a query object is to use the Query Editor.

Result sets

The **open** method opens a query, fetching the requested information from the database. The total amount of information fetched from database is called the result set. A result set contains a number of rows, and each row contains a number of columns.

At any one time, one row in the result set is considered the current row. The **move** methods for the query object move change the current row from one row to another.

The data navigator object gives the user simple controls to **move** from one row to another, and to make simple changes in the database.

Bound controls and arrays

A bound control is an object which displays information obtained by a query. For example, you can bind a text box to a particular column in the query; once the text box is bound, it always shows the value of that column in the current row. Various types of data objects can be used as bound controls, and they display the data in various ways.

A bound list box has two ways to display data:

- In list mode, the list box displays the values of a particular column in all the rows of the result set.
- In lookup mode, the list box displays all the possible values for a particular column, and the highlighted item of the list box shows the value of that column in the current row. The list box may be initialized by placing in items manually or by obtaining a list of items with another query into the database.


A bound combo box displays information in a manner similar to a list box in lookup mode. A bound list view (in Report mode) can display multiple columns of data from the result set.

Changing the database

If you set appropriate property values for the transaction and query object, you can use the query object and bound controls to modify the database. For example, if a particular text box is a bound control showing the value of a column from the current row, changing the value of the text box modifies the corresponding row in the database.

To add a new row, you put the query object into Add mode using the **add** method. You can then create the contents of the new row in the bound controls and bound arrays. Similarly, to change an existing row, you put the query object into Edit mode using the **edit** method. You can then modify existing values in the bound controls and bound arrays. To end either Add mode or Edit mode, you execute **update**, which actually delivers the new or modified row to the database.


 [Jato Programmer's Guide](#)


 [Part II. Advanced topics](#)


Chapter 9. Writing Internet applications

This chapter explains how to use Jato to write an application that makes use of the Internet, either via the World Wide Web or with Windows socket facilities. For additional information about many of the topics discussed in this chapter, see the Internet-related books mentioned in the [Bibliography](#).

 [Web server basics](#)

 [Support for interface environments](#)


 [NetImpact Dynamo server applications](#)


 [ISAPI web server applications](#)

 [Sockets](#)


 [Server sockets](#)

 [Web application targets](#)


 [Internet components](#)


 [The Internet component class](#)

 [The HTTP component class](#)

 [The FTP component class](#)

 [Integrating Jato and Microsoft FrontPage](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

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.powersoft.com/products/internet/optima.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.powersoft.com`, many web browsers will automatically convert this to the URL `http://www.powersoft.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.powersoft.com`. The web server locates the file `products/internet/optima.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*.


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).


 [Forms](#)


 [CGI](#)

 [NSAPI and ISAPI](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Web server basics](#)

Forms

HTML makes it possible to create *forms* on the user's machine. These are not the same as Jato 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`


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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Web server basics](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Web server basics](#)

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 Jato provides access to the three environments. For information on the environments, see the following URLs:

CGI:

<http://hoohoo.ncsa.uiuc.edu/cgi/>


NSAPI:

http://www.netscape.com/newsref/std/server_api.html

ISAPI:

<http://www.microsoft.com/win32dev/apiext/isaphome.htm>

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

Support for interface environments

%%% Jato doesn't yet support all the interface environments discussed in the following section.

Jato supports CGI, NSAPI, and ISAPI. In all three cases, you use the Target Wizard to create an appropriate target:

- CGI targets are executables which can be invoked by the web server.
- NSAPI and ISAPI targets are DLLs. The web server loads the DLL, which then provides extended services on the web server.

The templates offered through the Target Wizard set linkage flags and other options for creating the target application, and set up the basic structure needed to work in the selected environment.

Although you must choose the environment at the time you create the target, Jato does as much as possible to make environmental differences transparent. In particular, you make use of a common interface class to refer to information exchanged between the User and Server sides. For example, you use the same methods to obtain information from the user, no matter which environment you choose. If you want to write different versions of an application to run under NSAPI and ISAPI, the two versions can use much of the same code.

All three types of targets run on the Server side. They interface with the web server in order to interact with the User side.

None of these targets have forms associated with them. Since they do not interact with users on the Server side, they do not have graphical interfaces.


<p>Warning: NSAPI and ISAPI targets are DLLs which run by being attached to the web server. If the web server receives multiple requests to the same URL using the same DLL, the DLL code may be entered by multiple users at the same time. This means that you must write the code to provide adequate protection in multi-user situations. For example, you should use critical sections to protect shared resources if the resources cannot be used by multiple users simultaneously. For more information about critical sections, see Synchronizing threads.</p>


Installation instructions

The NSAPI target template contains a file named `INSTALL.HTM`. This file contains HTML text explaining providing supplementary information on creating an NSAPI target. Once you create your NSAPI target, you can read this file by opening it with your web browser.

 [Non-visible forms for web services](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)


 [Support for interface environments](#)


Non-visible forms 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 visible forms associated with them. Typically, they use visual classes instead.

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

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.

◆ **To create a NetImpact Dynamo server application:**


1. From the **File** menu on the main Jato 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**, 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 [Non-visible forms for web services](#).

<p>Important: For complete details about the capabilities of NetImpact Dynamo, see the documentation that accompanies that product. This guide makes no attempt to explain how Dynamo works.</p>


 [The RunApp method](#)

 [Data members defined in the primary class](#)

 [The Session class for Dynamo targets](#)

 [The Document class](#)

 [The DBConnection class](#)

 [Dynamo exceptions](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[NetImpact Dynamo server applications](#)

The RunApp method

When you create a Dynamo servlet target, Jato creates a number of files, class definitions, and so on which are used to build the target. Most of these files are maintained by Jato; 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 a 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.

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 **RunApp**. The **RunApp** method is the heart of your Dynamo servlet: this is where you write the code that does the main work of the servlet.

The **RunApp** method has the prototype

```
public void RunApp( String args[] )
```

The arguments to **RunApp** are the command line arguments received by your application when it was invoked.

When you create a Dynamo servlet, the **RunApp** 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 RunApp runs

Your servlet will be invoked when the user sends a URL to the web server, specifying that the servlet should run. The servlet creates an object of the primary class and invokes three methods for that class:

- **StartApp** to initialize the servlet.
- **RunApp** to perform the actual work of the servlet.
- **EndApp** to clean up after the servlet.

You can add your own user code to any of these methods. You can also add your own data members to the primary class.


Once the servlet has run **StartApp**, **RunApp**, and **EndApp** for the primary class, the servlet terminates. If the User sends another URL which invokes the servlet, the servlet creates a new object of the primary class and goes through the same procedure again. Since this is a new object and a new invocation of the servlet, it does not have direct access to any data used by the previous invocation of the servlet.


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.
- 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [NetImpact Dynamo server applications](#)

Data members defined in the primary class

The primary class defines a number of data members that you'll find useful for writing your servlet:

`Session session;`

The `Session` class offers methods for maintaining an ongoing session with the User. For more information, see [The Session class for Dynamo targets](#).

`Document document;`

The `Document` class offers methods for preparing an HTML document to send back to the User system, in response to the URL that invoked this servlet. For more information, see [The Document class](#).

`DBConnection connection;`

The `DBConnection` class offers methods for interacting with the database that you connect with by going through NetImpact Dynamo. For more information, see [The DBConnection class](#).

The data members listed above are all used to provide access to the functionality of their associated class. For example, if you want to send HTML code to the User, you invoke the appropriate method on the `document` object.

When you create a Dynamo servlet target, Jato generates Java class definitions for the classes listed above (`DynamoConnection`, `Document`, and so on). To see the full definition of one of these classes, open the `Classes` window, expand the servlet target, and open the appropriate class definition.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 9. Writing Internet applications](#)

[_NetImpact Dynamo server applications](#)

The Session class for Dynamo targets

The Session class 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. In this way, you can preserve data from one invocation of the servlet to the next.

The **setValue** method has the prototype

```
boolean setValue( String name, String value )
```

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 )
```

For example,

```
boolean value = session.getValue( "myvar" );
```

obtains the value that was previously assigned to `myvar`.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 9. Writing Internet applications](#)

[_NetImpact Dynamo server applications](#)

The Document class

The Document 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 Document object. In this way, you start with a “clean” HTML document every time you react to a URL from the user.

Sending HTML code to the user

The most important methods of Document 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 writes to a new line.

The following shows a simple example of **writeln**:

```
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 Document class, see the SQL Anywhere documentation on NetImpact Dynamo.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 9. Writing Internet applications](#)

[_NetImpact Dynamo server applications](#)

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.

The most important method of DBConnection is the **createQuery** method. This has the prototype


```
Query createQuery( String SQLStatement )
```


where SQLStatement is a normal SQL statement. For example,


```
Query query_1 =  
    connection.createQuery( "select id, lname from customer" );
```

creates a Jato Query 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 9. Writing Internet applications](#)

 [NetImpact Dynamo server applications](#)

Dynamo exceptions

The classes associated with Dynamo servlets will throw a `DynamoException` in case of error. For example, if you pass an invalid argument to a method in the `Document` class, the method typically throws a `DynamoException` in response to the error. The sample **RunApp** code demonstrates a format for catching such exceptions.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

ISAPI web server applications

An ISAPI web server application (ISAPI servlet) works in conjunction with an ISAPI server on the Server system. ISAPI 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.

◆ **To create an ISAPI web server application:**

1. From the **File** menu on the main Jato 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**, then click **Next**.
3. Select the folder where you want to store the Dynamo target, then click **Finish**.

An ISAPI 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 [Non-visible forms for web services](#).


Important: For complete details about the capabilities of ISAPI web server, see


<http://www.microsoft.com/win32dev/apiext/isaphome.htm>

This guide makes no attempt to explain how the ISAPI server works.


 [The RunApp method](#)


 [Data members defined in the primary class](#)

 [Response headers](#)

 [Sending data to the user](#)


 [Form variables](#)

 [Query variables](#)

 [Result codes](#)

 [Redirecting the user](#)

 [Preserving data from one servlet invocation to the next](#)

 [Web service exceptions](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[ISAPI web server applications](#)

The RunApp method

When you create an ISAPI servlet target, Jato creates a number of files, class definitions, and so on which are used to build the target. Most of these files are maintained by Jato; 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 item which has the same name as the target itself. For example, if the target is called `MyServ`, there will be an item called `MyServ` in the expansion for the target. This is called the *primary class* of the servlet target.

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 **RunApp**. The **RunApp** method is the heart of your servlet: this is where you write the code that does the main work of the servlet.

The **RunApp** method has the prototype

```
public void RunApp( String args[] )
```

The arguments to **RunApp** are the command line arguments received by your application when it was invoked.

When you create an ISAPI servlet, the **RunApp** 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 RunApp runs

Your servlet will be invoked when the user sends a URL to the web server, specifying that the servlet should run. The servlet creates an object of the primary class and invokes three methods for that class:

- **StartApp** to initialize the servlet.
- **RunApp** to perform the actual work of the servlet.
- **EndApp** to clean up after the servlet.

You can add your own user code to any of these methods. You can also add your own data members to the primary class.

Once the servlet has run **StartApp**, **RunApp**, and **EndApp** for the primary class, the servlet terminates. If the User sends another URL which invokes the servlet, the servlet creates a new object of the primary class and goes through the same procedure again. Since this is a new object and a new invocation of the servlet, it does not have direct access to any data used by the previous invocation of the servlet.

You cannot preserve data from one invocation to the next inside the servlet itself. However, Jato provides facilities that let you store data outside the servlet and retrieve it later. For more information, see [Preserving data from one servlet invocation to the next](#).

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 9. Writing Internet applications](#)

[_ISAPI web server applications](#)

Data members defined in the primary class

The primary class defines the following data member that you'll find useful for writing your servlet:

```
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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[JSAPI web server applications](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[JSAPI web server applications](#)

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 writes to a new line.

The following shows a simple example of **`writeln`**:

```
// WebConnection server;  
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.

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.getStream( );
```

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[JSP web server applications](#)

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 `POSTed` variables:


```
boolean posted = server.getIsPostMethod( );
```

returns `true` if the form used `POSTed` variables and `false` otherwise.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [JSP web server applications](#)

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" );
```

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 9. Writing Internet applications](#)

[_ISAPI web server applications](#)

Result codes

Your program may pass a result code to the server, to indicate status. Different servers accept different result codes:


- CGI: a zero exit code indicates success; non-zero indicates failure
- NSAPI: one of the values defined for the interface, represented by symbolic constants beginning with the characters `PROTOCOL_`
- ISAPI: one of the values defined for the interface, represented by symbolic constants beginning with the characters `HSE_`


The **setResultCode** method of `WebConnection` sets the result code for your application. For example, with an ISAPI servlet you might use

```
SetResultCode( HSE_STATUS_SUCCESS );
```

You may set the **ResultCode** property at any time during execution. The current result code is returned to the web server the first time you send output to the User system (for example, with **write**).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [ISAPI web server applications](#)

Redirecting the user

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[ISAPI web server applications](#)

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.

The first step in this process is to obtain an object of the `Session` class, using the following `WebConnection` method:

```
// WebConnection server;  
Session session = server.getSession();
```

The `Session` class supports two important methods:

```
boolean success = session.setValue( "varName", "value" );  
String 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 session = server.getSession( );
```


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.

If there has been no previous session with the current User, **`getSession`** returns `null`. This lets you know that this is a new session (or that too much time has elapsed since the last interaction, so you have to start over again from scratch).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [JSPAPI web server applications](#)

Web service exceptions

The classes associated with web service servlets will throw a `WebServiceException` in case of error. For example, if you pass an invalid argument to a method in the `WebConnection` class, the method may throw a `WebServiceException` in response to the error. The sample **RunApp** code demonstrates a format for catching such exceptions.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

Sockets

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).

Socket services are provided through a library named `WSOCK32.DLL` which comes as part of your Windows environment. Some application packages may replace the standard Microsoft sockets library with their own version of the library. For example, many Web browsers overwrite the existing `WSOCK32.DLL` with their own version. Different versions of the library may provide different services.

Socket services are supported by socket libraries in accordance with the current sockets specifications—Windows Sockets: An Open Interface for Network Programming under Microsoft Windows, Version 1.1. The Jato programmer's guide makes no attempt to reproduce the information of the specification document; it only explains how Jato makes socket functionality available to your programs.

Normal sockets are represented by Socket objects. Jato gives sockets default names of the form `socket_N`. On the **Internet** page of the Java 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](#)

[Types of sockets](#)

[Design-time socket properties](#)

[Other socket properties](#)

[Connecting to a remote system](#)

[Writing on a socket](#)

[Reading data from a socket](#)


[Closing a socket](#)


[Handling socket errors](#)

[Socket events](#)

[Datagram sockets](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Sockets](#)

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 `WSOCK32` library. This does not mean that the data has been successfully delivered to the remote machine; it just means that the `WSOCK32` 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 Jato programs. However, blocking sockets have their uses too; they are especially appropriate in server applications where multithreading is likely.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Sockets](#)

Types of sockets

A *connection-based socket* is a socket that must be explicitly connected before you can send data to a remote system.

Jato defines the following classes derived from the Socket class:

DatagramSocket

Represents a non-connection-based *datagram* socket. 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 `SOCK_DGRAM`.

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 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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[Sockets](#)

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 socket. `SOCKDGRAM` indicates a non-connection-based datagram socket and `SOCKUNKNOWN` represents an unknown type of socket. For a complete list of possible types, see the Jato 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.

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. Therefore, you must explicitly call the **create** method for the socket in order to initialize the socket, as in

```
socket_1.create( SOCKSTREAM );
```

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[Sockets](#)

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 Jato 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 Jato 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


While 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Sockets](#)

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.

[_Jato Programmer's Guide](#)

[_Part II. Advanced topics](#)

[_Chapter 9. Writing Internet applications](#)

[_Sockets](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[Sockets](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Sockets](#)

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 `SocketEIsConnected` to indicate that the socket is already connected to another system.

Possible error code values are defined in the `SocketExceptionCode` interface. For more information, see the [Jato 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 clears 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".

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Sockets](#)

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.

SocketHostResolved is also triggered when **getRemoteHostName** is called for the first time after a call to **setRemoteHostAddress**. In this case, **getRemoteHostName** returns a null string when it is first called. If you call it again after the default handling for **SocketHostResolved**, **getRemoteHostName** returns the correct host name.

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 `WSOCK32` library 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 Jato Component Library Reference.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[Sockets](#)

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( 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( 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

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. Jato gives server sockets default names of the form `srvsocket_N`. On the **Internet** page of the Java 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 (`WSockDGRAM`).


 [Server socket events](#)

 [Server socket methods](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Server sockets](#)

Server socket events

Server sockets support the following events:

SocketError

An error occurred during an operation on the server socket.

SocketHostResolved

The socket support software has determined the true Internet address of the remote system.

SocketConnectionRequest

The server socket has received a connection request from a client.

For further information on these events, see the Jato Component Library Reference.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[Server sockets](#)

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 );
}
```

 [Jato Programmer's Guide](#)


 [Part II. Advanced topics](#)


 [Chapter 9. Writing Internet applications](#)

Web application targets


A Web application 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 services). You can create a web application target which represents all this software (as well as data files) as a single object. This offers a number of advantages:


- When you build the web application target, Jato can build everything as needed. You don't have to worry about different parts of the project getting out of synch with each other.
- Web application targets have simple mechanisms for preparing the package for a debugging test, as well as packaging everything for production use.
- Web application 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.
- Web application targets automatically interface with any source control system supported by Jato (for example, RCS, PVCS, Object Cycle, and so on).

 [Web projects](#)

 [Creating a web application target](#)

 [Actions on a web application target](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Web application targets](#)

Web projects

A *web project* consists of the following:

- A web application target
- Any number of dependent targets—these are targets such as Java applets and web services that Jato builds as separate “sub-targets” of the web application
- A staging web site
- A source control archive

The files associated with the web application target represent all the files in a web site. The web application target also contains a small number of extra Jato generated files that are not needed in the final production web site. Jato uses these to maintain information about the structure of the target.

Dependent targets: Dependent targets are built in their own folders, separate from the web application target. As with normal Jato targets, dependent targets have a target folder, a `Release` folder and a `Debug` folder.

Whenever Jato builds the web application, it also builds any dependent targets that need to be built. After dependent targets have been built, Jato copies the executable files to a location specified for the web application target.

Staging web site: The staging web site is a web site where the developer runs the web application for testing purposes. This web site may be any of the following:


- A web site on a different machine
- A specified folder on the same machine as the web application target
- The folders used by the web application target itself.


The process of copying files from the web application target folders to the staging web site is called *Publishing*.

You can also Publish a web application to a production web site, in which case Jato omits any files that are only needed for debugging.

Source control archive: The source control archive contains the master revision control copies of the files in the web application target and the dependent targets. The archive can be implemented with any source control system Jato supports.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Web application targets](#)


Creating a web application target


Before creating a web application target, create the dependent targets that will make up the web application and store them in a single project.

◆ **To create a web application target:**

1. From the **File** menu of the main Jato menu bar, click **New** and then **Target**.
2. In the Target Wizard, click **Web Application**, then click **Next**.
3. Enter a name for the web application's target folder, then click **Next**.
4. Click the names of the targets that will be the dependent targets for this web application (or click **Select all** if all the targets listed will be dependent targets).
5. Click **Finish**.

Jato creates a file named `main.htm` to serve as the starting web page for this target. The contents of this file is a typical HTML 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 a link which goes to a page containing a Java applet that you want to test.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Web application targets](#)

Actions on a web application target

A web application target supports the following actions:

Build

Building a web application means building all the dependent targets, then copying the resulting files to appropriate locations in the web application's target folder.

Publish

Publishing the web application means copying all files in the target folder to the staging web site. If the staging web site is the same as the web application target folder, **Publish** does nothing. Otherwise, **Publish** copies the files to a specified location, either on the local system or on another system.

Jato lets you write JavaScript instructions to specify the operations that should be performed in a **Publish** action.

Publish automatically builds the web application (if necessary) before copying the files.

Run

Running a web target means starting a web browser to browse the staging web site and/or starting a web server on the system containing the staging web site. This allows you to test User system programs, Server system programs, or both.

Run automatically performs **Build** and **Publish** actions (if necessary) before running the application.

These actions are controlled through the run options for the web application, accessed through the Targets window. 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

Internet components

Jato supports several components that can be used to establish various types of Internet connections:

- The Internet class (`powersoft.jcm.net.Internet`)—this can be used to open a connection to a remote Server system using a variety of protocols.
- The HTTP class (`powersoft.jcm.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.jcm.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.

Note: The three classes discussed in this section are derived from standard classes in `java.net.*`.

[!\[\]\(125d701e9425b54c764340b5671b38cd_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(21199eb166cc97331a0c54c649195dcc_img.jpg\) Part II. Advanced topics](#)

[!\[\]\(2bdfe261b986065ee0ac76460d6528c9_img.jpg\) Chapter 9. Writing Internet applications](#)

The Internet component class

An Internet object makes it easy to establish a connection with remote servers on the Internet. Jato gives Internet objects default names of the form *internet_N*. On the **Internet** page of the Java 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.

[!\[\]\(ec9132f1d27c8919987d92907322654d_img.jpg\) Internet object properties](#)

[!\[\]\(05be7c7a8995decd503647c99211f7c2_img.jpg\) Establishing an Internet session](#)

[!\[\]\(aa53ad6fea213b8b2226d3077e30533a_img.jpg\) Reading the input stream](#)

[!\[\]\(dd161862f9164df98f62b726e9846241_img.jpg\) Closing a connection](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The Internet component class](#)

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.powersoft.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.powersoft.com" );  
internet_1.setFile( "help/contact.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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [The Internet component class](#)

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.powersoft.com" );
internet_1.setFile( "help/contact.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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The Internet component class](#)

Reading the input stream

You 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.


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**.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [The Internet component class](#)

Closing a connection

The **close** method of `Internet` closes a connection that has previously been opened with **open**:

```
internet_1.close( );
```


[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

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.

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

```
ftp://ds.internic.net/rfc    or  
ftp://nic.merit.edu/documents/rfc
```

Jato gives HTTP objects default names of the form *http_N*. On the **Internet** page of the Java 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](#)


[HTTP requests](#)

[Closing an HTTP connection](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [The HTTP component class](#)

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 **readFile**. 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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The HTTP component class](#)

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.powersoft.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_ADDRQ_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 request 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`.

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 9. Writing Internet applications](#)


 [The HTTP component class](#)

Closing an HTTP connection

The **closeConnection** method of HTTP closes a connection that has previously been opened with **connect**:

```
http_1.closeConnection( );
```

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

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.

This guide makes no attempt to provide technical details about FTP requests and other interactions with web servers. For information, see RFC 959, available at:


```
ftp://ds.internic.net/rfc  
ftp://nic.merit.edu/documents/rfc
```


Jato gives FTP objects default names of the form *ftp_N*. On the **Internet** page of the Java 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 Jato Component Library Reference.

 [Establishing an FTP connection](#)

 [Obtaining a directory listing](#)

 [Changing the current directory](#)

 [Retrieving a file from the remote system](#)

 [Sending a file to a remote system](#)

 [Closing an FTP connection](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The FTP component class](#)

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.

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](#).

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The FTP component class](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [The FTP component class](#)

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).

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The FTP component class](#)

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:

```
TYPE_ASCII  
TYPE_BINARY  
TYPE_EBCDIC  
TYPE_IMAGE  
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", TYPE_ASCII, false );
```

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[The FTP component class](#)

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 );
```

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 9. Writing Internet applications](#)


 [The FTP component class](#)

Closing an FTP connection

The **closeConnection** method of FTP closes a connection that has previously been opened with **connect**:

```
ftp_1.closeConnection( );
```

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

Integrating Jato 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 Jato is based on this division of tasks. FrontPage and Jato 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 Jato 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 Jato will not work correctly.


 [Control Files](#)

 [Creating a Web Site](#)


 [File Management](#)


 [Revision Control](#)


 [Publishing](#)

 [HTML Editing in Jato](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Integrating Jato and Microsoft FrontPage](#)

Control Files

Each product creates its own control files, used to store information about the files in the web site. Jato generates WXT, WXP and WXU files. FrontPage stores control files in folders that begin with `_vti_`. Neither Jato 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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 9. Writing Internet applications](#)

[Integrating Jato and Microsoft FrontPage](#)

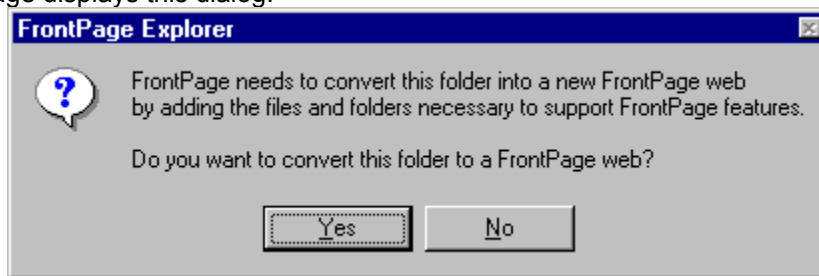
Creating a Web Site

The web site can be initially created using either Jato 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 tools will warn that there are already files in the folder you have chosen. Jato displays the following dialog.





FrontPage displays this dialog.




In both cases, the dialog is only a warning and you will be allowed to continue creating the web site. Any files created by the tool that initially created the web site will not be disturbed.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)


 [Integrating Jato and Microsoft FrontPage](#)


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 Jato appear as part of the web site when it is viewed with FrontPage.

Jato 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 Jato Files window. They will not appear in the Targets window unless you specifically add them to the Jato web target.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Integrating Jato and Microsoft FrontPage](#)


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.


Jato 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 Jato 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 Jato. However, you must not have the same web site open in both tools on one machine at a time.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)

 [Integrating Jato and Microsoft FrontPage](#)


Publishing


Publishing is the process of copying files from the folder where Jato or FrontPage edits them to the folder where a web server accesses them. Both FrontPage and Jato will recognize and publish files added to the web site by the other. Publishing can be accomplished using either tool.


Jato recognizes control files created by either tool and does not publish them. FrontPage publishes control files created by Jato.

FrontPage supports publishing by copying all the web site files to another folder or using the Web Post API.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 9. Writing Internet applications](#)


 [Integrating Jato and Microsoft FrontPage](#)

HTML Editing in Jato

The **File Types** page of the Jato Options dialog box lets you choose the editor used to edit a file with a particular file name extension. For further information, see **Error! Reference source not found.**

Since the FrontPage HTML editor is a separate executable, you can use the dialog to specify that it should be used to edit HTML files. If you install FrontPage before Jato, you can configure Jato to use FrontPage as its HTML editor.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


Chapter 10. Using and creating JavaBeans


This chapter explains how to:


- Use JavaBeans components or other Java components.
- Create JavaBeans components.

%%% Note: This chapter describes features that may not have been thoroughly tested in the pre-release version of Jato. Please report any bugs that you find!


 [Using an existing JavaBean](#)


 [Creating your own JavaBean](#)

 [JavaBean property sheets](#)

 [Summary of using and creating JavaBeans](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 10. Using and creating JavaBeans](#)

Using an existing JavaBean


A *JavaBean* defines a component that can be used in the same way as components like command buttons and list boxes. A JavaBean is basically a Java class definition (a `.class` file) which conforms to the JavaBean standard for specifying properties, methods, and so on.


JavaBeans 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 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. For more information, see the most recent release of the JDK. The “official” web site for JavaBean specifications is


<http://splash.javasoft.com/beans>

This provides the technical specifications for JavaBeans, as well as an overview of the JavaBeans concept and tips for using JavaBeans.

 [Attaching a JavaBean to Jato](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 10. Using and creating JavaBeans](#)

 [Using an existing JavaBean](#)

Attaching a JavaBean to Jato

Jato makes it easy to incorporate existing JavaBeans into your Jato session. For example, suppose that someone in your company has written a JavaBean designed to work with your company's database. You can attach this Bean to your Jato session by adding the Bean to the **Database** page of the Java components palette. Once you have done this, you can use the Bean just like any of the standard Jato components: you can place it on forms, set properties by property sheets, and so on.

Before you can attach a JavaBean to Jato, you must have the following information:

- The name of the `.class` file defining the JavaBean.
- The page on the Java components palette where you want to place the Bean. By default, Jato creates a new page named **Classes** and places the Bean there.
- The name of a folder where Jato can place various files that are used in building the JavaBean from the `.class` file.
- An icon to represent the Bean on the Java components palette. Jato 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).

<p>Note: The large and the small icon are stored together in a single <code>ICO</code> file. If you invoke the Image Editor to edit the icons, you always see the 16x16 version of the icon first. To edit the 24x24 icon, use the Select Icon Image entry of the Edit menu in the Image Editor to switch from image to the other.</p>


The process of attaching a JavaBean to Jato is controlled by the Java class component wizard.


◆ To attach an existing JavaBean to Jato:

1. In the **Components** menu of the main Jato menu bar, click **Add Java Component**. This opens the Java class component wizard.
2. Under **Page on the component palette**, click the page where you want the Bean to appear.
3. Under **Class file**, type the name of the `.class` file containing the Bean definition or click **Browse** to locate the file on your computer.
4. Click **Next**.
5. Under **Folder name**, type the name of the folder where Jato can build the Bean or click **Browse** to locate such a folder on your computer.
6. Make sure the **Use JavaBeans naming conventions** is checked, then click **Next**.
7. Specify an icon file for the Bean, either by clicking **Browse** and choosing an existing icon file or by clicking **Edit** and designing your own.
8. Click **Finish**.

After you click **Finish**, Jato attempts to build the JavaBean from the `.class` file and attach it to your program. If this is successful, the icon for the Bean will appear on the components palette page that you specified.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 10. Using and creating JavaBeans](#)


Creating your own JavaBean


A JavaBean is just a `.class` definition that follows the JavaBean model. For example, properties for the class are controlled using **get** and **set** methods, events follow the listener model used by Jato itself, and so on. To create your own JavaBean with Jato, you simply create a class that obeys these rules.


Jato has a number of features which make it easier to define a class that fits the JavaBean model. These features are described in the sections that follow.


 [Defining a JavaBean class](#)

 [Adding a property to a JavaBean class](#)


 [Adding a method to a JavaBean class](#)

 [Adding an event to a JavaBean class](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 10. Using and creating JavaBeans](#)

 [Creating your own JavaBean](#)

Defining a JavaBean class

The first step in creating a JavaBean is to create the class. You do this the same way you would create any other class: using the Class wizard.

The Class wizard requires the following information:

- A name for the JavaBean class.
- An existing class on which the JavaBean is based. The default is Object, the most general Java class. However, you can choose a different class if appropriate. For example, suppose you want to create a JavaBean which is an enhanced type of list box. You can base your JavaBean on `powersoft.jcm.ui.ListBox`, then add new properties, methods, and events to support the enhancements you want to implement.
- A name for the package that will contain the JavaBean. You can leave this blank if you don't intend the Bean to be part of a package.
- Whether or not the JavaBean implements an interface.

◆ To create a JavaBean class:

1. In the Classes window, click the target where you will define the JavaBean class.
2. In the **File** menu of the Classes window, click **New**, then **Class**. This opens the Class wizard.
3. Click **Visual Class** if the JavaBean will be visible when it is used on forms; otherwise, click **Standard Java**.
4. Click **Next**.
5. If the JavaBean will be part of a package, type the name of the package under **Package**.
6. Under **Class name**, type a name for the JavaBean.
7. Under **Extends**, type the name of the class on which the JavaBean will be based.
8. If the JavaBean implements an interface, type the name of the interface under **Implements**.
9. If the JavaBean will be a public class, make sure **Public** is checked.
10. If the JavaBean will be an abstract class, make sure **Abstract** is checked.
11. If the JavaBean will be an interface, make sure **Interface** is checked.
12. Click **Finish**.

After you finish, Jato opens a code editor window where you can enter a definition for the class. Specifically, you can enter declarations for any data members in the class and any import statements that the class may require.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 10. Using and creating JavaBeans](#)

[Creating your own JavaBean](#)

Adding a property to a JavaBean class

In a JavaBean class, properties are controlled by *property functions*. This means a **get** method which determines the current value of the property, and/or a **set** method to assign a new value. A property that only has a **get** method is called a read-only property.

Often, the value of a property will be represented by a data member in the JavaBean 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 functions, 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.

If a property has its value stored in a data member as shown above, Jato can automatically generate inline property functions using the form of the preceding example. In more complex situations, you must write your own **get** and **set** functions to work with the property. You may also want to create several overloaded versions of a **get** or **set** function, in applications where there may be several useful calling sequences for the property function.

Before you add a property to a JavaBean 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 Jato to generate simple inline **get** and **set** functions or you prefer to write your own functions.

Once you have determined this information, you can create the property.

◆ To add a property to a JavaBean 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 **Property**. This opens the Property Wizard.
2. Under **Name**, type the name of your property.
3. Under **Data Type**, click the name of the data type for your property.
4. Click **Next**.
5. If you want to write your own property functions, click **Member functions**, then type prototypes for the property functions you want to define.
6. If you want Jato to automatically create simple inline **get** and **set** functions, click **Member variable with inline Get-Set** and type in the name of the data member that will hold this property's value.
7. Click **Finish**.

Jato will open a code editor window that will let you enter code for the property functions.

In some cases, you may want to define a property before you're ready to write the property functions. You should note that the code will not compile until you give the **get** function a `return` statement that returns a value of the appropriate type.

Deleting properties from a JavaBean


You can delete a property in the same way you delete any other object.


◆ To delete a property from a JavaBean 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, Jato does *not* delete the data member when you delete the property. You must delete the data member manually in the code for the JavaBean class.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 10. Using and creating JavaBeans](#)

 [Creating your own JavaBean](#)

Adding a method to a JavaBean class

The process of adding a method to a JavaBean 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.
- One or more categories in the Reference Card where you want this method to be listed. The default is **Essential**.

<p>Note: The Essential category of the Reference Card is specified as " Essential" (with a blank before the first character of the name). When categories are sorted in alphabetical order, the blank ensures that the Essential category appears first in the Reference Card list.</p>
--

◆ **To add a method to a JavaBean 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 **Next**.
5. Under **Reference Card Categories**, type one or more categories where you want the method to appear in the Reference Card.
6. Click **Finish**.

Jato opens a code editor window where you can begin entering code for the method.

Adding an event to a JavaBean class

You may define events for JavaBeans components, using the event model described in [Standard events](#). You can use an existing event (like **Click** or **Select**) or define a new event.

If you define a new event, you must specify an event identifier (event ID) for the event. This is a symbolic name that will be used to differentiate your event from other events. Jato automatically assigns a unique numeric value to the event ID name you choose.

For every event, you must also specify an event data structure that will be used to pass information to event handlers. This will either be an existing event data structure (for example, `powersoft.jcm.event.ClickEvent`) or a new event data structure based on Jato `EventData` class (`powersoft.jcm.event.EventData`).

If you intend to use a new event data structure, you must create a class defining this structure before you define an event that uses the structure.

Before you add an event to a JavaBean class, you need to decide the following information:

- Whether you intend to use an existing event or define a new one.
 - An event ID name for the event (if you are not using an existing event).
 - A brief description of the event to appear in the Object Inspector and the Events menu. (This is not needed if you are using an existing event.)
 - The event data structure type that will be used to hold information about the event.
- ◆ **To add an event to a JavaBean 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 **Event**. This opens the Event Wizard.
 2. If you intend to use an existing event, click **Use an existing event**, then click the name of the event under **Existing Event**.
 3. If you intend to define a new event, click **Define a new event**, then type a name for the event's event ID.
 4. Click **Next**.
 5. If you are defining a new event, type a brief description under **Description**.
 6. Under **Event Data Structure**, click the data structure that will contain information about this event.
 7. Click **Finish**.

Adding an event to a JavaBean class defines several methods within the class:

addEventListener

Adds a listener for the event to the list of listeners already registered. The name of this method depends on the event being defined. For example, if you are defining a **Click** event, the name of this method is **addClickListener**.

removeEventListener

Removes a listener for the event from the list of listeners already registered. The name of this method depends on the event being defined. For example, if you are defining a **Click** event, the name of this method is **removeClickListener**.

fireEvent

Triggers the event on the current object. The name of this method depends on the event being

defined. For example, if you are defining a **Click** event, the name of this method is **fireClick**.

As an example of how this comes together, suppose you create a class named `MyButton` as an enhanced type of command button, and you define a **Click** event for this class. You build the `.class` file defining `MyButton` and attach this to your Jato session, as described in [Attaching a JavaBean to Jato](#).

Once you have done this, you can place `MyButton` objects on forms in other Java applications. Suppose you place a `MyButton` named `mb_1` on a form. Then you may take the following steps to enable **Click** events on `mb_1`:

1. The form that contains `mb_1` should be declared as an implementation of `ClickListener`.
2. The **create** method for the form should contain the code

```
mb_1.addClickListener( this );
```

to indicate that the form is listening for **Click** events on `mb_1`.

3. The form should contain a **click** method with the prototype

```
public void  
    click(powersoft.jcm.event.ClickEvent event )
```

This method is invoked when the form receives a **Click** event on `mb_1`. Typically, the **click** method uses `event.getSource()` to verify that the **Click** event was triggered on `mb_1`, then invokes an appropriate **Click** event handler (for example, `mb_1_Click`).

Alternatively, you can set up a relay function to listen for **Click** events on `mb_1`. You do this with the following steps.

1. Create an `mb_Relay` class that implements `ClickListener`. This class should define a **click** method with the same prototype as given in the previous approach.
2. In the **create** method for the form, create an `mb_Relay` object and make it the **Click** listener for `mb_1`, as in:

```
mb_1_Relay = new mb_Relay();  
mb_1.addClickListener( mb_1_Relay );
```

In this case, the **click** method for `mb_1_Relay` should be set up to invoke an appropriate **Click** event handler for `mb_1`.

User-defined events

The previous section discussed how to deal with JavaBeans which used an existing event (**Click**). If you define your own event, the process is similar but requires some extra steps. As an example, suppose you have defined a new event named **MyEvent** in a JavaBean named `MyBean`. You have also defined an event data structure class named `MyEventData`, derived from the basic Jato `EventData` class.

To work with this new event, you must begin by creating an interface named `MyEventListener`. This should be an extension of the `powersoft.jcm.event.EventListener` class. A typical definition would be

```
import powersoft.jcm.event.EventListener;  
  
public interface MyEventListener extends EventListener {  
    void myEvent( MyEventData eventData );  
}
```

This specifies that a listener for this event must include a method named **myEvent**.

When you use `MyBean` in an actual application, the application must include an object that implements `MyEventListener` and contains a method named **myEvent**. This object can either be the form that

contains the object where **MyEvent** is triggered, or a relay that is specifically associated with the object. Whichever you choose, you would use

```
mb_1.addMyEventListener( ... );
```

to specify which object will be the event listener.

Now suppose you have a MyBean object named `mb_1` and want to trigger a **MyEvent** on the object. You do this with

```
// MyEventData event;  
mb_1.fireMyEvent( event );
```

Your program must initialize the `event` argument to specify event information that you want to pass to the event handler(s).

The **fireMyEvent** method determines the first `MyEventListener` object associated with `mb_1`, and invokes **myEvent** in the event listener. The **myEvent** method typically invokes an event handler routine to handle the event, as in

```
event.setHandled( mb_1_MyEvent( event ) );
```

This passes the `event` data block to an event handler named `mb_1_MyEvent`. If the event handler returns `true`, **myEvent** uses **setHandled** on the event block to indicate that the event has been handled; otherwise, the event is still considered unhandled. Control passes back to **fireMyEvent**. If the event is still unhandled and there are more event listeners for the same object, **fireMyEvent** invokes **myEvent** in the next event listener. In this way, **fireMyEvent** works through the list of registered event listeners until one uses **setHandled** on the event data block to indicate that the event has been handled (or until **fireMyEvent** has gone through all the registered event listeners).

JavaBean property sheets

Jato maintains property sheets for JavaBean classes, and for properties, methods, and events defined for those classes. These property sheets summarize the information that was gathered by the wizard that created the item. For example, if you define a method in a JavaBean class, the property sheet for the method summarizes the information that was gathered by the Method Wizard when you first created the method.

You can use the property sheets to change information about JavaBean classes, properties, methods, and events. For example, if you want to change the Reference Card categories where a method appears, you can open the property sheet for the method and make a change on the appropriate page of the property sheet.

◆ **To open the property sheet for a class:**


1. In the left part of the Classes window, use the right mouse button to click on the name of the class, then click **Properties**.


Opening a property sheet for an item defined within a class (for example, a property) is similar.

◆ **To open the property sheet for a property, method, or event:**

1. In the right part of the Classes window, use the right mouse button to click on the name of the item, then click **Properties**.

 [Jato Programmer's Guide](#)


 [Part II. Advanced topics](#)

 [Chapter 10. Using and creating JavaBeans](#)

Summary of using and creating JavaBeans

This chapter has explained how to use Java components, including JavaBeans, and how to create JavaBeans components.

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

Chapter 11. Using ActiveX components and servers

This chapter explains how to:


- Use ActiveX controls (also called ActiveX 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. ActiveX controls were formerly referred to as OCXs, because their files have the `ocx` extension. ActiveX components can be smaller and faster than OCXs. As a result, they can be downloaded and used more easily over the internet.


Note: ActiveX controls cannot be hosted by every browser. Users of your applications will need either Microsoft Internet Explorer version 3.0 or later or Netscape Navigator with an ActiveX plug-in.
--


 [OLE and ActiveX](#)

 [Getting started with ActiveX controls](#)

 [Using ActiveX controls and server components](#)

 [Summary of using ActiveX components and servers](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

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 Jato components in that the interface to them is implemented as properties, methods, and events. Unlike native components, however, 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. Jato 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.

ActiveX server components are used by a web server to perform server-side operations. 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 Jato and any development environment that supports ActiveX.

ActiveX controls and server components are handled similarly in Jato, as detailed in the following sections.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

Getting started with ActiveX controls

ActiveX controls are used in much the same way as other components in Jato. 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, Jato creates a native interface to it. The methods and properties of that control are added to the Jato 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 Jato component except that ActiveX controls are not visible on Java forms. ActiveX controls are embedded in a web page and controlled by a Java applet on the same page.

 [Registering ActiveX controls](#)


 [Adding and ActiveX component to Java palette](#)

 [Place component on a Java form](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

 [Getting started with ActiveX controls](#)

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 Jato) 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 been registered by some other program, skip this section.


◆ **To register an ActiveX control:**

1. In the **Tools** menu of the main Jato 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 click **Close**. 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. Click **Close**.


Note: Registration is a system-wide operation: registered controls become available to all programs, not just Jato. Similarly, Jato 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

 [Getting started with ActiveX controls](#)

Adding and ActiveX component to Java palette

Adding an ActiveX control to the Jato 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, Jato 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. On the **Components** menu of the main Jato menu bar, click **Add ActiveX Component(s)**. You are presented with the ActiveX Component Wizard.
2. Choose the control that you want to add to the Component palette.
3. Select the Java Component Palette from the Component palette list.
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**.

The first page of the Component Wizard has two options: **Show type libraries** and **Show automation servers**. The only difference between these two options is how they are registered with the system. 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 the 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. Type libraries are usually stored as a file with the `TLB` extension in the same folder as the automation server.

When Jato 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 any project you make.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 11. Using ActiveX components and servers](#)

[Getting started with ActiveX controls](#)

Place component on a Java form

ActiveX components reside 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 Java virtual machine.

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 Jato to automatically attach the component your Java applet.

Attaching components to Java forms

The new ActiveX component is represented by a member variable of the form. Since the ActiveX component resides on the web page and not on the Java form, your web page must invoke a method of your Java to provide your Java applet with the component.

When you turn on the **Externally Created** property, Jato creates a method for your form that allows the web page to provide the form with a component. The web page should normally invoke this method when it is loaded. When you turn on the **Externally Created** property, Jato generates Visual Basic script to invoke the method to attach the Java applet to a component.

For example, if you place a Formula One worksheet component on your form and turn on the **Externally Created** property, Jato will declare a private data member named `_DVCF1_1` as part of your Java form:

```
private _DVCF1 _DVCF1_1;
```

Jato will also create a public method called **Set_DVCF1_1**:

```
public void __Set_DVCF1_1( Object o )
{
    _DVCF1_1 = (_DVCF1)o;
}
```

Since the component resides on the web page, Jato automatically generates an `OBJECT` tag in the default web page for your applet:


```
<OBJECT ID="HTML_DVCF1_1" WIDTH=200 HEIGHT=100
CLASSID="CLSID:042BADDC5-5E58-11CE-B610-524153480001">
</OBJECT>
```

This tag will be replaced with a Formula One worksheet when it is viewed with an ActiveX-enabled web browser such as Internet Explorer 3.0 or Netscape Navigator equipped with an ActiveX plug-in. When the web page is loaded by the browser, the **Set_DVCF1_1** method must be invoked to provide your applet with the component. Jato generates the following Visual Basic script which invokes the **Set_DVCF1_1** method when the web page is loaded:

```
<SCRIPT LANGUAGE=VBScript>
Sub window_onLoad
    document.applet.Set_DVCF1_1( HTML_DVCF1_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.

 [_Jato Programmer's Guide](#)


 [_Part II. Advanced topics](#)


 [_Chapter 11. Using ActiveX components and servers](#)

Using ActiveX controls and server components

Once you have placed an ActiveX control on 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](#)

 [_Handling events](#)

 [_Running your applet or application](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 11. Using ActiveX components and servers](#)

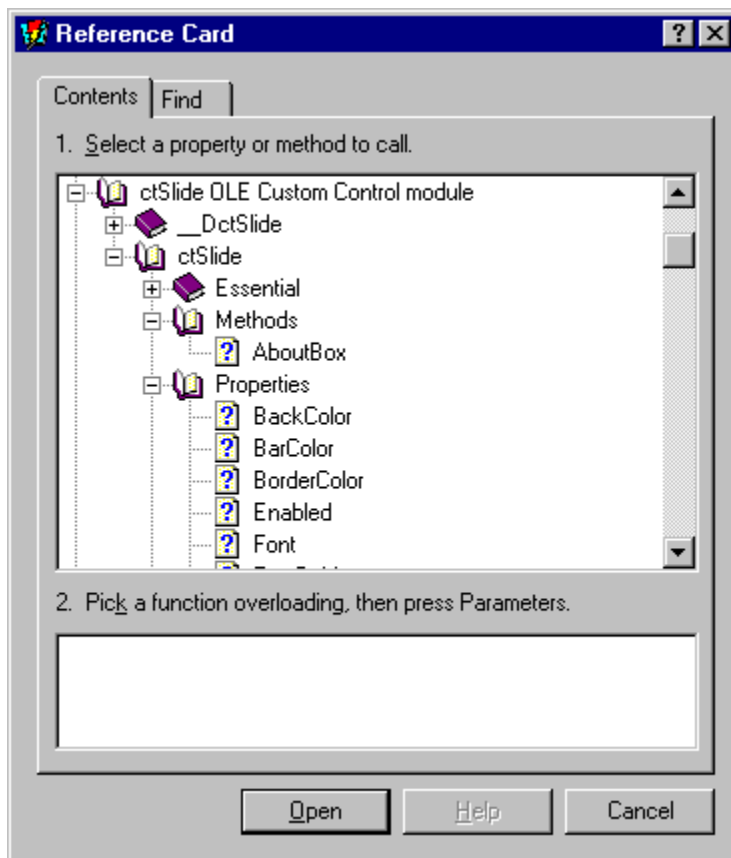
[Using ActiveX controls and server components](#)

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 Jato. 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 (provided with Jato):




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 **put** method to change the value, or the property's **get** method to retrieve the value.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

 [Using ActiveX controls and server components](#)

Handling 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. Jato 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 on the design-time Java form.
2. On the popup menu for the control, point to **Events** and click **More....** Jato 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. Jato 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. Jato 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_DVCF1_1_Click( P0, P1 )
    document.applet.Call_DVCF1_1_ClickEvent()
End sub
</SCRIPT>
```

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

 [Using ActiveX controls and server components](#)

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. Both Microsoft Internet Explorer version 3.0 and Netscape Navigator version 3.0 with the ActiveX plug-in offer support for ActiveX controls, and are available on the internet.


◆ **To change the environment that is used to run your applet:**

1. From the **View** menu, select **Targets**. Jato will display the Targets View.
2. In the Targets View, select the Java applet target.
3. From the **File** menu, click **Run Options**.
4. On the General option page, select the third option labeled **Use a web browser**.
5. In the web browser field, enter the full path of your web browser, or click **Browse** to select it from disk.
6. Click **OK**.

Once you have configured the web browser, you can press F5 to run your applet. Jato 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 know that the control will not cause damage to your computer, you can allow the control to execute.

 [Java Programmer's Guide](#)


 [Part II. Advanced topics](#)

 [Chapter 11. Using ActiveX components and servers](#)

Summary of using ActiveX components and servers

This chapter has explained how to use ActiveX components with your Java applets and ActiveX automation servers with your Java applications.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


Chapter 12. 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` and `Threads and User Events` sample projects demonstrate the use of threads.


 [Processes, threads and multitasking](#)

 [Designing for multiple threads](#)


 [Specifying threads](#)


 [Terminating a thread](#)

 [Suspending and resuming thread execution](#)

 [Thread properties](#)

 [Synchronizing threads](#)

 [Debugging threads](#)

 [Summary of threads](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using 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 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.

One thread in a multithreaded program is designated as the *primary thread*. This is the thread that will begin executing when you invoke the program (unless you explicitly specify a different thread). In other words, the primary thread corresponds with the default entry point of the program.


Jato 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 the standard Java documentation.


[Multitasking and multiprocessing](#)

[Differences between processes and threads](#)

[When to use multiple threads](#)

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Processes, threads and multitasking](#)

Multitasking and multiprocessing

Operating systems typically use *multitasking* to allow a single processor to run multiple threads concurrently. This means that 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 (approximately 20 milliseconds), 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 runtime environment can still provide a form of multithreaded execution, but general program behavior may be different. For more information, see [Thread priority](#).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Processes, threads and multitasking](#)


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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Processes, threads and multitasking](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using 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 process “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. While deadlock may be easy to predict in this circularly-dependent case, it can also arise in more subtle ways. The key to avoiding deadlock is to structure your program so that it cannot occur.


[Synchronization](#)


[Master-worker model](#)


[Daemon threads](#)

[Accessing resources](#)

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Designing for multiple threads](#)


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](#).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Designing for multiple 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.


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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Designing for multiple threads](#)


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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Designing for multiple threads](#)

Accessing resources

A frequent cause of deadlock is having different threads access the same resources in different order. For instance, consider two processes 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.

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)


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 Runnable interface](#)

 [The current thread](#)

[Java Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Specifying threads](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Specifying threads](#)

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 Jato 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:

```
public class MyForm extends Form 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 and execute the **start** method on that object, as in


```
MyForm f = new MyForm( );
f.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.

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Specifying threads](#)

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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

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 the same object.


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** on certain threads. 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](#)

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Terminating a thread](#)

Daemon threads and program termination

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.

[Java Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

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,

```
Thread t = Thread.currentThread( );  
t.suspend( );
```

immediately suspends the current thread. Execution does not resume until another thread re-activates this thread. When execution does resume, it will look as if **suspend** just returned normally. Therefore, 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.


[The resume method](#)


[The sleep method](#)

[Interrupting a sleeping thread](#)

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Suspending and resuming thread execution](#)

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.

[Java Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Suspending and resuming thread execution](#)

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.

[Java Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Suspending and resuming thread execution](#)

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 interrupted = 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 interrupted = 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.


 [Java Programmer's Guide](#)


 [Part II. Advanced topics](#)


 [Chapter 12. Using threads](#)


Thread properties

This section discusses some of the properties that a thread may have.

 [Thread name](#)

 [Thread priority](#)

 [The Daemon property](#)

 [Thread states](#)

[Java Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Thread properties](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Thread properties](#)

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.

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](#).

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Thread properties](#)

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).

[!\[\]\(b39c89771cd6fb2128a8c57aa7d97f9a_img.jpg\) Java Programmer's Guide](#)

[!\[\]\(d0a1791f26d167e866e44ebbf83efebe_img.jpg\) Part II. Advanced topics](#)

[!\[\]\(5eb1325dfdc3f1cad8426726c0db51cd_img.jpg\) Chapter 12. Using threads](#)

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.

[!\[\]\(950a62bbddad88d64435fd35607dfc42_img.jpg\) Synchronized critical sections](#)

[!\[\]\(5a132f13505a6571904d622757b7a8f0_img.jpg\) The notify and wait methods](#)

[!\[\]\(10f8862fc183b400327470ea85afe9ae_img.jpg\) Volatile data](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Synchronizing threads](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Synchronizing threads](#)

The notify and wait methods

The **notify** and **wait** methods may be used in `synchronized` routines to coordinate the activities of multiple threads.

The **notify** and **wait** methods are defined in the `Object` class. This means they can be used on any object that inherits 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 ) wait( );
        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 ) wait( );
        newData = false;
        notify( );
        return data;
    }

    public synchronized void store( int value ) {
        while ( newData ) wait( );
        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.

[Java Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 12. Using threads](#)

[Synchronizing threads](#)


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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

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](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 12. Using threads](#)

 [Debugging 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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 12. Using threads](#)

Summary of threads

A multithreaded program has several independently running threads of execution. Each thread shares the same address space, so that they can interact with each other.


The most common model for multithreaded programs is the master-worker model, where a single master thread creates separate worker threads to perform individual tasks. The workers do not interact with each other; they only interact with the master thread.


 [Jato Programmer's Guide](#)


 [Part II. Advanced topics](#)


Chapter 13. Using graphics and printers

This chapter discusses the use of graphics in a program: how to create graphics and display them on a form.


 [The Graphics class](#)


 [Drawing on a graphic context](#)

 [The Image class](#)

 [The MediaTracker class](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)


 [Chapter 13. Using graphics and printers](#)


The Graphics class


The AWT Graphics class is an abstract class representing an area where a graphical image may be drawn. Graphics is a platform-independent abstract class which is used as the basis for appropriate system-dependent classes. For example, the AWT library on a particular computer may define one Graphics-derived class for drawing pictures on the monitor screen, another class for drawing pictures that will be sent to a printer, and so on.

Any class derived from Graphics must support the standard methods defined for the root class Graphics. Therefore, the rest of this section describes various methods which can be applied to any Graphics object.


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.


 [Creating a graphic context](#)


 [Common Graphics properties](#)

 [Disposing of a graphic context](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The Graphics class](#)

Creating a graphic context

%%% The **getGraphics** method is not yet implemented.

The usual way to create a graphic context is to use the **getGraphics** method for a component. For example, suppose that you want to be able to draw on the current form:

```
Graphics g = getGraphics( );
```

obtains a graphic context for the form using the form's **getGraphics** method. Similarly, suppose you want to be able to draw on command button `cb_1` (so that you can place a picture on the button rather than text). You can obtain a suitable graphic context with


```
Graphics cbg = cb_1.getGraphics( );
```


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.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The Graphics class](#)

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. (Most printers cannot print all the way out to the edge of the page, since they need space on each side to grip the paper.)

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 Jato 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 placing text on the object.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The Graphics class](#)

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( );
```

[!\[\]\(125d701e9425b54c764340b5671b38cd_img.jpg\) Jato Programmer's Guide](#)

[!\[\]\(21199eb166cc97331a0c54c649195dcc_img.jpg\) Part II. Advanced topics](#)

[!\[\]\(2bdfe261b986065ee0ac76460d6528c9_img.jpg\) Chapter 13. Using graphics and printers](#)

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.
--

[!\[\]\(c694a3ff3b077d76910920a6a1593ab4_img.jpg\) Drawing lines](#)

[!\[\]\(ec9132f1d27c8919987d92907322654d_img.jpg\) Drawing rectangles](#)


[!\[\]\(05be7c7a8995decd503647c99211f7c2_img.jpg\) Drawing ellipses \(ovals\)](#)


[!\[\]\(aa53ad6fea213b8b2226d3077e30533a_img.jpg\) Drawing arcs and pie shapes](#)

[!\[\]\(dd161862f9164df98f62b726e9846241_img.jpg\) Drawing polygons](#)

[!\[\]\(758ebdf4629c903da74c2e079717ae32_img.jpg\) Drawing text](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [Drawing on a graphic context](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[Drawing on a graphic context](#)

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 );
```

[Jato Programmer's Guide](#)

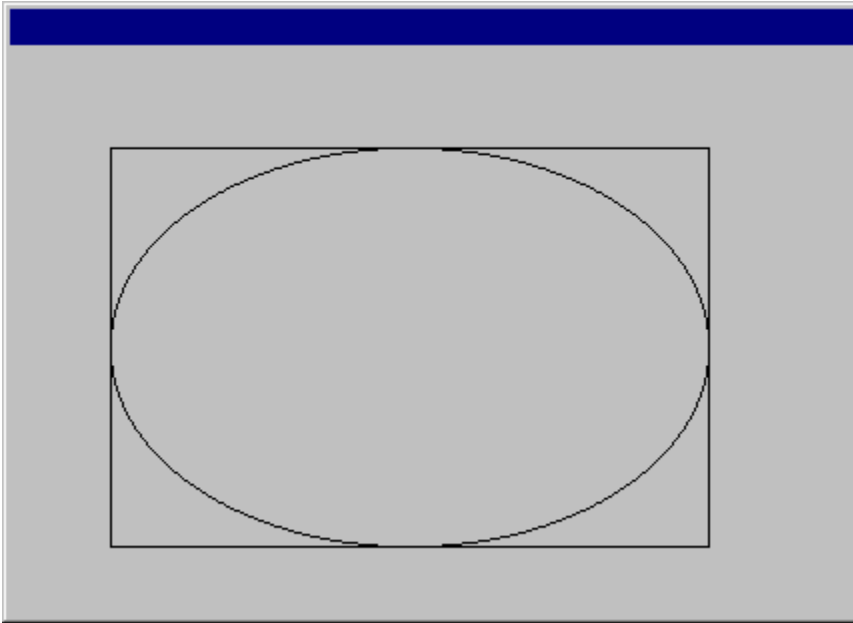
[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[Drawing on a graphic context](#)

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.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[Drawing on a graphic context](#)

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 );
```

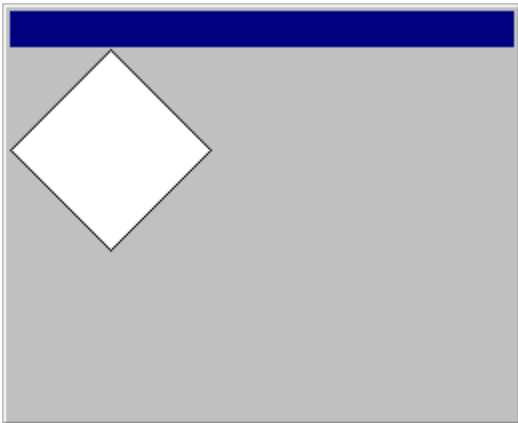
where N is the number of vertex points for the polygon. The first vertex of the polygon is (xPoints[0], yPoints[0]), the next is (xPoints[1], yPoints[1]), and so on. Similarly, the **fillPolygon** method fills the interior of a polygon with the color given by the **Color** property of the Graphics object:

```
// 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 );
```

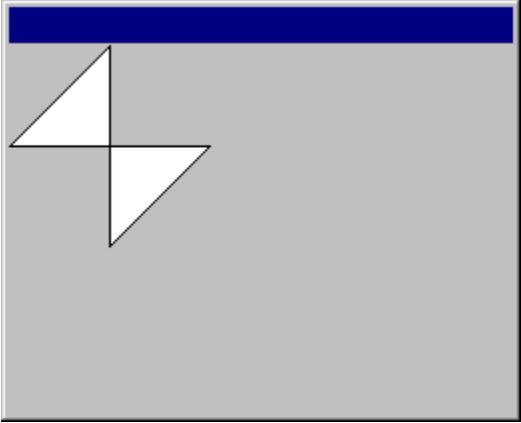
The result is the following.



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.



[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[Drawing on a graphic context](#)

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 );
```

 [Java Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)


The Image class

The AWT Image class is an abstract class representing graphical images. These can be used as “pictures” on Java forms.

Image is intended to be a platform-independent basis class. The AWT library on a given computer will contain system-dependent classes based on Image to work with actual image types. For example, a Windows 95 system will have specific classes based on Image to handle bitmaps, icons, and so on. These classes will handle such images using whatever techniques are standard on the system.

 [Loading images from files](#)

 [Image size](#)

 [Drawing images](#)

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[The Image class](#)

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 [The Applet class](#) and [The Toolkit class](#).

Here are some sample uses of the method:


```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img1 = tk.getImage( "picture.jpeg" );
Image img2 =
    tk.getImage( new URL("http://www.site.com//pic.gif");
```


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. For more information, see [The MediaTracker class](#).

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The Image class](#)

Image size

You can determine the size of an image using

```
// Image img;  
int h = img.getHeight( );  
int w = img.getWidth( );
```

Both of these dimensions are given in pixels.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[The Image class](#)

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 Jato 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*.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

The MediaTracker class

The MediaTracker class provides a simple way to determine whether Image objects have finished loading. MediaTracker provides methods to:


- Specify one or more images whose status you want to check.
- Determine whether selected images have finished loading.
- Wait for selected images to finish loading.
- Force selected images to be loaded immediately, even if you haven't tried to draw them yet in a graphic context.


The usual approach to creating a MediaTracker object is


```
MediaTracker tracker = new MediaTracker( this );
```


The argument for the constructor refers to the component with which the image is associated. This argument will almost always be `this`, referring to the current form.

Note: The ImageObserver interface provides an even greater degree of control over the loading of images. For more information, see the standard AWT reference.

 [Associating images with a MediaTracker](#)


 [Checking whether an image has been loaded](#)


 [Errors while loading](#)

 [MediaTracker status methods](#)

 [Waiting for images to be loaded](#)

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The MediaTracker class](#)

Associating images with a MediaTracker


The **addImage** method of MediaTracker associates an image with a MediaTracker object, and assigns that image an identifier that will be used in future MediaTracker methods:


```
// int id;  
// Image img;  
tracker.addImage( img, id );
```

The `id` can be any integer, but it is common practice to use 0 for the first identifier, 1 for the next, and so on.

A MediaTracker object may have any number of images associated with it. You may also specify the same identifier for several Image objects. In this case, the images are treated as a group for the purposes of other MediaTracker methods.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The MediaTracker class](#)

Checking whether an image has been loaded

The **checkID** method of `MediaTracker` checks whether all the images associated with a particular identifier number have finished loading:

```
boolean loaded = tracker.checkID( id );
```

The result is `true` if they have all finished loading and `false` otherwise.

The **checkAll** method checks whether all the images associated with the `MediaTracker` object have finished loading:

```
boolean loaded = tracker.checkAll( );
```

Again, the result is `true` if they have all finished loading and `false` otherwise.

It is possible that the run-time environment has not even started to load one or more images associated with the `MediaTracker` object. In this case, these **check** methods simply return `false`; they do not force the run-time environment to start loading the image.

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[The MediaTracker class](#)

Errors while loading

Errors may occur when loading an image. For example, suppose that when you create the Image object, you specify an incorrect file name or URL. The program does not check whether the file name or URL is valid until the image actually begins loading; therefore, the error is not discovered until loading. In such cases, **checkID** and **checkAll** will never return `true` because the image can never be loaded successfully.

The **isErrorAny** method determines whether any of the images associated with a MediaTracker object are in an error state:

```
boolean err = tracker.isErrorAny( );
```

The result is `true` if any of the images encountered an error while loading. The result is `false` if there is no error state yet. However, if there are still some images being loaded, there is still a chance that one of those images will enter an error state later; for example, if you are loading from a URL on a remote system, an error will occur if you get disconnected partway through the loading process.

The **isErrorID** method determines whether any of the images associated with a specific identifier number are in an error state:

```
boolean err = tracker.isErrorID( id );
```

Again, the result is `true` if any of the images is in an error state and `false` otherwise.

The **getErrorID** method returns a list of the images which are in an error state:

```
Object errList[] = tracker.getErrorID( id );
```

The list specifies all the Image objects which are associated with the given `id` and which are currently in an error state.

The **getErrorAll** method is similar to **getErrorID**, but it examines all the images associated with the MediaTracker object:

```
Object errList[] = tracker.getErrorAll( );
```

[Jato Programmer's Guide](#)

[Part II. Advanced topics](#)

[Chapter 13. Using graphics and printers](#)

[The MediaTracker class](#)

MediaTracker status methods

The **statusID** method of `MediaTracker` offers another way to check the progress of loading images. It has the following format:

```
// MediaTracker tracker;  
// int id;  
// boolean load;  
int status = tracker.statusID( id, load );
```

This examines the status of all images with the given `id`. Possible statuses are represented by static values:

`MediaTracker.LOADING`
Image is loading.

`MediaTracker.ABORTED`
The loading process has been aborted for at least one image with the given `id`.


`MediaTracker.ERRORRED`
The loading process has encountered an error for at least one image with the given `id`.


`MediaTracker.COMPLETE`
The loading process has completed successfully for at least one image with the given `id`.

When an `id` has more than one associated image, the status values are ORed together to produce the status value returned by **statusID**.

The second argument of **statusID** is a boolean value `load`. If this argument is `true`, the run-time environment immediately attempts to start loading all the appropriate images, if the loading hasn't started already. If this argument is `false`, the run-time environment does not try to load images which have not begun loading. Therefore, **statusID** can be used to force the loading of an image as well as for checking on the loading status.

 [Jato Programmer's Guide](#)

 [Part II. Advanced topics](#)

 [Chapter 13. Using graphics and printers](#)

 [The MediaTracker class](#)

Waiting for images to be loaded

The **waitForID** method immediately starts loading all images with the specified ID (if they have not been loaded already). It then waits for every image to be loaded or to receive an error:

```
tracker.waitForID( id );
```

After **waitForID** returns, you should use **statusID** or **isErrorID** to determine if any of the images received an error.

The **waitForAll** method is similar to **waitForID** but starts loading all images associated with the MediaTracker object and waits for the process to finish:

```
tracker.waitForAll( );
```

These **wait** methods do not return until all appropriate images have been loaded or have received an error during loading.


 [Jato Programmer's Guide](#)

Appendices

The appendix describes the coding style used in Jato examples and generated code.

 [Bibliography](#)

 [Jato Programmer's Guide](#)

 [Appendices](#)

Bibliography

The following is a selection of books and World Wide Web pages on topics related to programming with Jato:

Internet applications

Windows Sockets: An Open Interface for Network Programming under Microsoft Windows, Version 1.1 [1995].

Windows Sockets 2 Application Programming Interface, Revision 2.1.0 [1996].

Web sites

<http://www.powersoft.com/products/internet/optima.html>
The latest information about Optima++.

<http://www.watcom.on.ca/optdev/>
Optima++ technical product information, tips and tricks.

<http://www.earthchannel.com/binpub/archives/optimapp/>
Archive of an independent Optima++ email list.

<http://www.total.net/~bklein/optima/>
A user's site with Optima++ information.

<http://www.optimapp.pbe.com/>
A user's site with Optima++ information.

Search sites such as <http://www.altavista.digital.com> and <http://www.yahoo.com> can also help you to find relevant information.

