Simplicity for Java

User Guide

Simplicity for Java

Simplicity for Java 1.2 by Data Representations, Inc.
Manual Version 1.2

Simplicity for Java contains FLEXlm license management software from Globetrotter Software, Inc. (http://www.globetrotter.com/) FLEXlm is a trademark of Globetrotter Software, Inc.

Simplicity for Java contains InstallShield installation software from InstallShield Software Corporation. (http://www.installshield.com/) InstallShield is a trademark of InstallShield Software Corporation.

Simplicity for Java contains SimpleText database software from Thought, Inc. (http://www.thoughtinc.com/)

Table of Contents

CHAPTER 1

Installation

This chapter will discuss installing Simplicity for Java.

It will cover

- System requirements
- The install program
- Operating system specific notes

System Requirements

Java Virtual Machine

Before you install Simplicity for Java, you must install a Java Development Kit (JDK). If you do not have a JDK already installed you can get one from

http://www.javasoft.com/cgi-bin/java-ports.cgi

This web page lists all of the known JDKs by operating system. Make sure that you choose one which is compliant with Java version 1.1 or later. Try to use the latest revision of the JDK for your operating system. It is preferable that your JDK be version 1.1.4 or later.

Minimum System Requirements

- Java Development Kit, version 1.1 or higher
- 10 MB free disk space for program
- HTML web browser

Suggested Minimum System Features

- CPU of similar power to a 133 MHz Pentium chip
- 32 MB RAM (64 MB on MacOS)
- 800x600 pixel display

Installing

NOTE: Before beginning the install, you must have installed a JDK1.1.4 or greater on your computer. This must be a full JDK; a JRE is not enough.

Windows (95/98/NT)

The installer is called simpjava.exe. Double click this file to begin the install. The install utility will guide you through the installation. You must accept the terms of the licence agreement to complete the install. After specifying the install location, all of the files will be copied.

Mac OS

Make sure that you have previously installed the MRJ 2.1.4, available from http://developer.apple.com/java/ . MRJ 2.2 is not recommended at the current time.

The installer is MacBinary encoded and should be automatically decoded when downloaded. If not, you can decode it using Stuffit Expander 4.5 or later. Double click the installer to begin the install.

OS/2 Warp

The distribution is called simpjava.zip. Unzip this file using an unzip utility, being careful to maintain the directory structure. Execute the simpwps.cmd Rexx script to create a Program Object on your desktop for starting Simplicity.

Linux, Solaris, AIX, IRIX, SCO UnixWare, and Other Unix

Simplicity is distributed as a .tar.gz file, named simpjava.tar.gz. To install, copy the distribution to your user directory and execute the following commands to unpack the archive:

```
gzip -d simpjava.tar.gz
tar -xf simpjava.tar
```

Starting Simplicity for Java

On most platforms, the installer will create a launcher which will start Simplicity.

- ***Windows 95, 98 or NT*** A group called 'Simplicity for Java' will be created in your Programs menu on the Start Menu.
- ***OS/2 Warp*** There is also a file called simpwps.cmd which will create a Program Object on your desktop from which you can start Simplicity.
- ***Solaris, Linux, and most other Unix systems*** a shell script will be created in the directory where Simplicity is installed, called Simplicity. You must edit the two variables in this file which correspond to the location of the java executable on your machine and the location of the Simplicity install directory. Set this script to be executable (using the chmod command), and then you can copy this file to a directory on your path.
- ***Macintosh*** A program icon called "Simplicity for Java" will be created in the install directory. Double-click this icon to start Simplicity. You may wish to make an alias of this file and put it on the desktop.

In many cases the default installation is sufficient. There are cases when you may need to customize the installation. The following information is provided to help you customize the Simplicity for Java installation.

ClassPath

Simplicity requires that three items be on your classpath. They are (in order)

1. The full path of simplicity.jar
2. The swingall.jar file. (This is not needed if you are using JDK 1.2 or later, as the swing classes are already present.)
3. The full path of datarep_common.jar

The method by which you set your system classpath varies according to the operating system.

- ***Windows 95*** The CLASSPATH is set in your autoexec.bat file with
  SET CLASSPATH=.;c:\Simplicity\simplicity.jar;c:\Simplicity\swingall.jar;
  c:\Simplicity\datarep_common.jar
- ***Windows NT*** The CLASSPATH is set in the Environment page of the System Properties in the Control Panel. Set a system variable similar to the Windows 95 example, above.
- ***OS/2 Warp*** The CLASSPATH is set in your config.sys file with a line similar to the Windows 95 example, above.
- ***Solaris/Linux/other Unix*** The CLASSPATH is set in your .profile with
  export CLASSPATH =$CLASSPATH:$HOME/Simplicity/simplicity.jar:
  $HOME/Simplicity/swingall.jar:
  $HOME/Simplicity/datarep_common.jar

Note that the CLASSPATH definitions must be on one line.

Starting Simplicity for Java

Simplicity for Java is started by invoking the Java interpreter. The main entry point is ‘datarep.Simplicity’. On most systems, the following command can be used to start Simplicity.

```
java -mx100m datarep.Simplicity
```

If you are using Microsoft’s Java Virtual Machine, the command would be

```
jview datarep.Simplicity
```

Platform Specific Notes

Windows NT 4.0

You must use JDK 1.1.7B or higher.

OS/2 Warp

The installation directory for Simplicity for Java as well as the Project directory must be located on an HPFS drive.

OS/2 users may wish to execute the simpwps.cmd file. This REXX script will create a WPS object on the OS/2 desktop for starting Simplicity.

We recommend using either jdk114 or jdk117 from IBM. These can be found at http://www.ibm.com/Java/jdk/download/. Do NOT use jdk116, as it contains a bug which will crash the WPS, when you exit from Simplicity. If you are using JDK 1.1.8, be sure to download the fixes from IBM (ftp://ftp.hursley.ibm.com/pub/java/fixes/os2/11/).

The debugger needs to have the localhost interface enabled. You can do this by typing the following command in an OS/2 window:

```
ifconfig lo 127.0.0.1 up
```

Note that this will be reset when you next reboot your computer.

Linux

Before installing Simplicity, make sure that you have the latest Linux JDK from www.blackdown.org. Simplicity for Java will work properly with either jdk1.1.7_v3 or jdk1.2.2_rc3.

Suse Linux users: the rpm packaged JDK from older Suse distributions contains an incomplete version of libjpeg.so. This causes the jdk to be unable to display jpeg files. To fix this, install the .tar.gz distribution from blackdown (jdk1.1.7), and then copy the libjpeg.so from that distribution over the one in /usr/X11R6/lib.

Macintosh

It is highly recommended that you have at least 64 MB of RAM to run Simplicity for Java. MRJ 2.1.4 is the recommended version of the MRJ.

Solaris

On some Solaris machines, Simplicity for Java will stall shortly after starting. This can be fixed by disabling the JIT. (Uncomment the appropriate line in the startup script.) Solaris 8 users will need to be sure to edit the first line of the shell script as described in the script.

SCO UnixWare

On some SCO machines, Simplicity for Java will stall shortly after starting. This can be fixed by removing the -mx100m argument from the startup script. (Uncomment the appropriate line in the startup script.)

The Personal Settings Directory

Simplicity 1.2 has introduced a centralized directory where all of the settings files for Simplicity are stored. The name for the directory is '.simplicity'. The default location for the directory is different in Unix (including Linux) and in other operating systems.

Linux, Solaris, SCO UnixWare, and other Unix

In Unix, the default location for the settings directory is in the user's home directory. For example, a user named bob with a home directory ***/home/bob***, would have the default settings directory located in ***/home/bob/.simplicity***. Multiple users in Unix can run Simplicity from the same location, but have their own unique settings. In Unix, the '.' which precedes the directory name marks the directory as a settings directory, and as such is not normally displayed by the ***ls*** command. However, ***ls -a*** will list all files, including ones which start with '.'.

Windows, MacOS, and OS/2

Windows, Macintosh, and OS/2 are primarily single-user operating systems. As such, the personal settings directory on these platforms is located in the directory where Simplicity is installed.

Files in the Personal Settings Directory

There are several files and directories which will have special meaning to Simplicity if found in the Personal Settings Directory.

- ***Project files*** Every project stores its settings in a file in the Personal Settings Directory with a name constructed by adding .Simplicity to the end of the project name. (i.e. Tutorial2.Simplicity).
- ***program.set*** This file contains the configuration settings for Simplicity, including window locations and printing preferences.

- ***IDEmenu.config*** This file is used to customize the IDE (see page 137).
- ***.lic_txt*** and the contents of the ***.license*** directory are files generated by Simplicity and used to store license information. Users should not change these files.

Technical Support and Feedback

Technical Support

Send your questions to: support@datarepresentations.com

We ask that you restrict your questions to issues specific to Simplicity for Java.

Be sure to include your name and return email address. Please also indicate which Operating System and Java Virtual Machine you are using.

Feedback

We always love to hear from our customers. Please send us your comments.

We are constantly improving our products. Please send us your ideas and suggestions for how we can improve Simplicity for Java.

Send feedback to: info@datarepresentations.com

CHAPTER 2

Tutorial 1 - Introduction to Simplicity

Welcome to Simplicity for Java. This tutorial will guide you through creating a simple text editor using Simplicity for Java.

By completing this tutorial you will learn about

- Managing files in the IDE
- Building a Graphical User Interface (GUI)
- Creating event code using the Code Sourcerer

It is estimated that this tutorial should take most people between 30 and 40 minutes to complete. All of the details in this tutorial are designed to highlight important functionality, so try to read and follow each part carefully.

A Simple Text Editor

If you haven't already done so, start Simplicity for Java in the manner appropriate for your operating system. The Simplicity IDE should be on the screen.

Open a project

You will now open a project which came with Simplicity for Java called Tutorial1.

1. Choose Open from the File menu.
   If the welcome screen is showing, you can choose 'Open an Existing Project' by clicking the mouse on the green disk icon. Otherwise, you can choose 'Open Project' from the File menu or click the 'Open' button (green disk icon) on the button bar.
2. A small window will appear listing the names of all the available projects. Choose 'Tutorial1' from the list and press the Ok button.

Create a new Application

You will now create a new Main Application (which will become your text editor). In the IDE, you see on the left the Project Tree, which has three sections. The first is called "Project Groups", and contains a list of group names. Each of these items represents a group of files which will appear in the box on the right when selected. For example, if you chose 'Java Source Files', all of the Java source files would appear in the box on the right.

1. Choose the first item in Project Groups, labeled 'Composer Files'.
2. To see what our tutorial will look like when finished, double click on Finished.Main. (You can also choose 'Open' from Finished.Main's pop-up menu or select Finished.Main with the mouse and choose 'Open' from the Edit menu.)

   Three windows appear. One of these will be a small text editor with a title bar 'Finished Text Editor'. It will have three buttons, New, Load, and Save. Try typing some text in the text area. When you press the New Button, the text will be cleared. The Load, and Save buttons work as you would expect, too. Try loading a text file from your computer into the Text Editor.
3. When you are finished, choose Exit from the File menu of the first window, titled 'Simplicity Composer'. Press No when asked if you would like to save any changes
4. In the IDE, choose '***Main App***' from the ***Create*** menu. This creates a new Application in the current project. (You can also press the 'Create a Main App' button on the button bar. It is the fourth button with the light bulb on it.)

   A new Icon appears in the 'Composer Files' group with a name similar to Untitled1.Main.

5. Click this new icon once to select it. (It will become highlighted.)
6. From the ***Edit*** menu, choose ***Rename selection***. A text field will appear to the right of the icon. Replace the old name with 'text editor'. Click the mouse anywhere else in the right hand window to accept the change. If you don't specify the '.Main' extension, Simplicity will add it for you. Simplicity will also modify the spacing and capitalization to conform to Java naming conventions.
7. Double click the new 'TextEditor.Main' icon to open the Simplicity Composer.

You have now created a new Application, and have loaded it into the Simplicity Composer. You should see three new Windows on the screen. They are the Composer window, the Object Palette, and a small window with the word 'Empty' in it. This last window is your program, which currently has nothing in it.

The composer window is the place where you will set up all of the properties of the graphical parts of your program. It is also where you will be able to supply all the code that your programs will need.

Toward the top of the composer window is a choice box. This is the part list. If you select it, you will see that there is currently only one item in it called TextEditor. This refers to the empty window that will become your application.

The bottom portion of the Composer window has a four page notebook. The first page, called 'Main Window', has the properties that are specific to a main application window.

1. Click the mouse in the text field entitled 'Title Bar'. A cursor should now be visible in that field.
2. Replace any text in the Title Bar field with 'Text Editor' .
   As you type, the title appears in the program window. (The one with the word 'Empty' in it.) You may need to resize the program window to see the full title.

Assemble the GUI

You will now begin to assemble the graphical components for your Text Editor. All of the graphical components can be found in the Object Palette window.

All graphical components in a Java application are placed into Java Layouts. The Layouts position your components so that they will be properly sized to be attractive regardless of the operating system that your application is running on. While most of the layouts are rather simple, they can be combined to create just about any component placement you desire.

We will start by using a Border Layout. A Border Layout can hold up to five components, located in the North, South, East, West, and Center. The layout tries to give the four border components their 'preferred' size, and any space left over goes to the Center component.

1. On the Object Palette, make sure that the first page, 'Layouts', is visible. (If not, click the mouse button on the 'Layouts' Tab.)
2. You will see icons for all of the available layouts. Click the first icon, named 'Border', with the mouse. (It should now appear depressed. Also, the title bar of the Object Palette will reflect your selection.)
3. In the program window, click once in the center. You have just placed a Border Layout into the window. You should now see five empty spaces, labeled North, South, East, West, and Center. Each of these spaces can hold a component (or another layout).
4. On the Object Palette select the second page's tab, labelled 'Basic'. This page contains all of the basic components which are part of the Java library.
5. Click the second to last icon, named TextArea. It should appear depressed.
   A TextArea is a component which can hold multiple lines of text, and is optionally editable and scrollable.
6. Click the mouse once in the 'Center' empty space in the Border Layout which you previously placed in the main window. This will put a TextArea in the center of the Border Layout.
7. In the composer window, you will see the properties notebook for the TextArea that you just created. Edit its properties by settings 'Rows' to 5 and 'Columns' to 40. Make sure that 'Editable' is selected, and erase any text in 'TextArea text'.
8. Each component is given a name which you use to refer to each component when you write Java code. In the properties for the TextArea you'll see that the current 'Object name' is set to 'textArea2'. Change it to a simpler, more descriptive name, 'text'.
9. On the Object Palette, choose the 'Layouts' page and click the second icon, named Flow. Click once in the 'North' empty space of the Border Layout that you previously put in the window. You have just put a new Flow Layout in the northern part of your Border Layout.
   A Flow Layout lays out components in a row and wraps to a new line if more space is needed. You see a single empty space where you can place components. As you click components into this space, the space moves to the right.

10. On the Object Palette, choose the ‘Basic’ page and click the fourth icon, named Button. Click once in the empty space inside the Flow Layout you just created. A new button appears and the empty space moves to the right. Click again in the flow’s empty space to add another button. Do this one more time to add a third button.
11. To be careful, save your work by pressing the ‘Save’ button at the top of the Composer window (the red disk icon).

If by accident you placed a part in the wrong empty space, you can remove it by using the ‘Recycle’ button. Choose the part by clicking it with the mouse button. Then push the ‘Recycle Current Part’ button located toward the top of the Composer window. The part will be removed from your program and will appear as an icon in the ‘Recycled’ page in the Object palette. You can select it from the palette if you wish to reuse it elsewhere. If you recycle a layout, the entire contents of the layout are moved to the Recycle page.

Cleaning up

You have finished placing all of the components that you will need for the simple text editor. You will now clean up the extra empty spaces, and do a little decorating.

1. Toward the top of the Composer window is the part list. It is a Choice field which contains the names of all of the parts that you have just created. From the part list select ‘flow3’. The properties for the Flow Layout appear in the Composer window.
2. On the Flow Layout Properties page, de-select ‘Show empty panel’. The empty panel in the Flow Layout disappears, leaving only the three buttons.
3. Choose ‘border1’ from the part list. De-select ‘Show empty panel’. The remaining empty panels in the Border Layout disappear.
4. Clicking on any component will cause its property notebook to appear in the Composer window. Click on button4. Change the ‘Button Text’ from ‘button4’ to ‘New’. Click on the remaining two buttons and change their text to ‘Load’ and ‘Save’, respectively.
5. From the part list choose flow3. From the flow3 properties notebook, choose the second page, labelled ‘Visual’. This page exists for every component and allows you to specify a variety of visual properties of your components. By setting the visual properties of a Layout, the changes will cascade through all of the components in the Layout.

6. In the Font box, change the Name to ‘Helvetica’, change the size to 16, and change the Style to ‘Bold’. This changes the font of all of the buttons.
7. Change the cursor to ‘hand’. This specifies that the cursor should appear as a hand whenever the mouse is over the Flow Layout.
8. Save your changes again by choosing ***Save*** from the ***File*** menu in the Composer window.

Responding to events

So far, you have created only the shell of your application. You can type in the text area, and you can press the buttons, but the buttons do not yet know what they are supposed to do.

In Java, components respond to user interaction using Event Listeners. For example, a button can respond to being pressed by *listening* for Action Events.

Simplicity for Java lets you select the type of events that each component should listen for and specify the code that should be executed in response. Simplicity will dynamically execute this code so that you can test your program as you design it.

Let’s begin with the New button. We would like the New button to clear any text that is currently in the text area, so that new text may then be entered.

1. Click the New button to show its properties in the Composer window.
2. Select the third page from the properties notebook, labelled ‘Listeners’. You should now see a list of the types of events that a button can listen for.
3. Select ‘Listen for action events’. A new page appears in the properties notebook for the New button labelled ‘Action’.
4. Select this new ‘Action’ page. You should now see an empty Java Editor where you can specify the code that you wish to have executed whenever the New button is pressed.

To write the code, we will use the Code Sourcerer. The Code Sourcerer will write Java code for you, based upon some simple choices.

5. Press the Code Sourcerer button toward the top of the ‘Action’ page for the New button.
6. A dialog appears with a list of choices. We want to clear the text in the text area. If it is not already selected, choose the first item, labelled ‘Change a property of an existing part...’. Press ***Next***.

7. The Code Sourcerer now asks you which part you would like to change. Choose ‘text’ (the TextArea in your program). Press ***Next***.
8. The Code Sourcerer now presents you with a list of the text area’s properties that can be changed. Choose ‘Change text of Text Area...’. Press ***Next***.
9. The Code Sourcerer now asks you where to get the new text for the Text Area. We want to simply erase the current text, so choose the first option, and leave the text field to the right blank. Press ***Done***.

The following code should have appeared in the Action Page.

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

Now you can test the code that you have just created. Type a few words into the text area in your program. Then press the New button. The text is cleared. Try changing the code in the New button’s Action page to

```
text.setText("Hello World!");
```

and press the New button. Each change that you make is immediately integrated into the working model of your program. Test the change by pressing the New button again.

We will now create the code to load a text file.

1. Press the Load button to show its properties in the Composer. Choose its ‘Listeners’ page and select ‘Listen for action events’. Select the Action page.
2. Press the Code Sourcerer button from the Load button’s Action page. Choose ‘File operations...’. Press ***Next***.
3. Choose the second item, “Create a new File object from a FileDialog...”. Press ***Next***.
4. The Code Sourcerer asks you to enter a title for the FileDialog. Enter ‘Choose a file to load’ for the title.
5. The Code Sourcerer asks you to enter a name for the File object which will point to the file that the user chooses. Leave the default value of ‘`theFile`’ for the destination name.
6. Select ‘Load’ mode.
7. Press ***Done***. Several lines of code should have appeared in the Action page which will launch a File Dialog and ask the user to choose a file to load. A file object has been created called `theFile`.
8. Press the Code Sourcerer button again. Choose ‘File operations...’. Press ***Next***.
9. Choose ‘Read a String from a text file...’. Press ***Next***.

10. The Code Sourcerer asks for the name of a File object to read from. Leave the default value of '`theFile`'. The Code Sourcerer also asks for the name of the string in which to store the data. Leave the default value of '`theText`'.
11. Press ***Done***. Now you have created a File object, and read the text from the file that the user chose into the String, '`theText`'.
12. We will use the Code Sourcerer once more to put the text that we just read into the text area. Press the Code Sourcerer button and choose 'Change a property of an existing part...'. Press ***Next***.
13. Choose the part, 'text' to change. Press ***Next***.
14. Choose 'Change text of Text Area'. Press ***Next***.
15. Choose 'The following variable expression'. Remove any text in the text field to the right and replace it with `theText`. (Note: It is case sensitive!) Press ***Done***.

You should see the following code in the Action page of the Load button.

```
FileDialog fd = new FileDialog(_getFrame(this),
        "Choose a file to load", FileDialog.LOAD);
fd.show();
if( fd.getFile() == null ) return; // user pressed Cancel
File theFile = new File(fd.getDirectory(),fd.getFile());
String theText = "";
try {
        FileReader fr = new FileReader(theFile);
        char charar[] = new char[(int)theFile.length()];
        fr.read(charar);
        theText = String.valueOf(charar);
        fr.close();
}
catch ( FileNotFoundException excpt0 ) {
}
catch ( IOException excpt1 ) {
}
text.setText(String.valueOf(theText));
```

You can now test this code by pressing the Load button. A FileDialog will appear. Choose any text file on your computer to load. The contents of that file should appear in the text area.

We will now create the code to save a text file.

1. Press the Save button to show its properties in the Composer. Choose its 'Listeners' page and select 'Listen for action events'. Select the Action page.
2. Press the Code Sourcerer button from the Save button's Action page. Choose 'File operations...'. Press ***Next***.
3. Choose the second item, "Create a new File object from a FileDialog...". Press ***Next***.
4. Enter 'Choose a filename to save' for the title.
   Leave the default value, '`theFile`', for the destination name.
   Select 'Save' mode.
5. Press ***Done***. Several lines of code should have appeared in the Action page which will launch a File Dialog and ask the user to choose a filename. A file object has been created called `theFile`.
6. Press the Code Sourcerer button again. Choose 'File operations...'. Press ***Next***.
7. Choose 'Write a String to a text file...'. Press ***Next***.
8. The Code Sourcerer asks for the name of a File object to write to. Leave the default value, '`theFile`', which is the name of the File object you just created. Press ***Next***.
9. The Code Sourcerer now asks where the text to be saved should come from. Select 'Another Part' and choose 'text' from the Choice box. This instructs the Code Sourcerer to save the current text from the component named 'text' (our TextArea).
10. Press ***Done***. Now you have created a File object, and saved the text to it.

You can test this code by typing a few words into the text area and pressing the Save button. A FileDialog will ask you for the filename to save it to. You can check that this worked by pressing the New button, and then using the Load button to read the text back.

Let's add one last thing to this program... A way to close it.

1. Toward the top of the Composer window, find the part list Choice box and choose the first item, 'TextEditor'. This shows the settings for the window that our program is sitting in.
2. Select the 'Listeners' page.
3. We want to respond to the user trying to close the window. Select 'Listen for window events'. A new page appears in the properties notebook called 'Window'. Select this new page.
4. There are seven kinds of window events. We want to respond to a 'Window Closing' event. From the Choice box to the left of the Code Sourcerer button,

select the third item, labelled ‘windowClosing’. (Be sure to select windowClosing and not windowClosed.)

5. Press the Code Sourcerer Button.
6. Choose ‘Change a property of an existing part...’. Press ***Next***.
7. Select the first item, labelled ‘TextEditor’. Press ***Next***.
8. Choose ‘Dispose of this Frame’. Press ***Done***.

The following code should have appeared in the Window Page for the windowClosing event.

```
_getFrame(this).dispose();
```

You can test this code by closing the program window in the default manner for your operating system. (This is often done by double clicking the upper-left icon or by pressing a close button on the upper-right). You can get the window back by choosing ***Initialize Class*** from the ***Program*** menu in the Composer.

You have now finished designing your text editor. Save your work by choosing ***Exit*** from the ***File*** menu in the Composer. Choose ***Yes*** when asked if you want to save your changes.

Completing the program

Looking back at the IDE, you should see the TextEditor.Main file in the ‘Composer Files’ group. Let’s look at the code you just created.

1. Choose the ‘Java Source Files’ group. You should see a new file called TextEditor.java. This is the java code which has been generated by the Composer. (If you are using the demo version of Simplicity for Java, the composer will not create the Java source file. Use Finished.java instead for the rest of the tutorial.)
2. Double-click the TextEditor.java icon. This opens Simplicity’s Java Source Editor with the TextEditor.java file loaded. Browse through the code. You will see toward the bottom of the file all of the code which you generated using the Code Sourcerer.
3. Exit the Java Editor. (Don’t save any changes to the file just yet.)

Now we want to compile and test the program.

4. You want to bring up the pop-up menu for the TextEditor.java icon. This is usually done by clicking the right mouse button on the icon, but this behavior varies on different operating systems. You can also show the pop-up menu by holding the shift key and clicking the icon with the mouse.
5. Choose ‘Compile’ from the pop-up menu. Simplicity for Java now invokes your Java compiler to turn the Java source code which you just created into an executable file, called a Java class file.
6. In the Project Tree, choose ‘Java Class Files’. You should see an icon named TextEditor.class. This file is your executable program.
7. Double click TextEditor.class. The Class Viewer appears. The Class Viewer can provide a variety of information about the contents of a class file.
8. At the bottom of the Class Viewer is a large button labelled ***Run***. Press this button to run the application.

Congratulations! You have completed this tutorial and learned many of the basics of working with Simplicity for Java.

CHAPTER 3

Tutorial 2 - Programming with Simplicity

This tutorial will guide you through creating a small bank account application. By completing this tutorial you will learn about

- Creating a multi-window application
- Using the GridBagLayout
- Creating variables, methods, and constructor code
- Integrating Code Sourcerer generated code with your own.

It is estimated that this tutorial should take most people between 30 and 40 minutes to complete. It is assumed that you have already completed the first tutorial. You should already be familiar with the Simplicity IDE, creating simple layouts using the Object Palette, and choosing event listeners.

A Bank Account Application

If you haven't already done so, start Simplicity for Java in the manner appropriate for your operating system. The Simplicity IDE should be on the screen.

This tutorial will make use of some network resources. If you need to start a dial-up connection in order to reach your SMTP server (this is your e-mail server) you must

do this before you start Simplicity for Java. You will also need to know the name of your SMTP server. (It often is similar to smtp.somewhere.com).

Open a project

You will now open a project which came with Simplicity for Java called Tutorial2.

1. Choose 'Open Project...' from the File menu or from the welcome screen.
2. A small window will appear listing the names of all the available projects. Choose 'Tutorial2' from the list and press the Ok button.

View the finished application

Let's begin by viewing the completed tutorial so that you'll know what we are going to accomplish. In order for the email portion to work, you will first need to enter your SMTP mail server and your email address.

1. In the IDE, choose 'Java Source Files' from the Project Tree.
2. Double-click 'FinishedBankAccount.java'. This will open the Java Editor.
3. Scroll Down about 3/4 of the file and find a set of lines which read:
```
email.setServer("smtp.somewhere.com");
email.setFrom("me@somewhere.com");
email.setTo("me@somewhere.com");
```
4. change the SMTP server name and the email address in these three lines to your SMTP server and email address.
5. ***Save*** and ***Exit*** the Java Editor using the ***File*** menu.
6. Compile the Application by right clicking the 'FinishedBankAccount.java' icon and choosing ***Compile*** from the pop-up menu. The pop-up menu can also be shown by holding down shift while clicking the icon.
7. Now choose 'Java Class Files' from the Project Tree.
8. Double-click 'FinishedBankAccount.class' and press the 'Run' button in the Class Viewer.

The bank account application is very simple to use. Press the Deposit or Withdrawal buttons to add or subtract money from the account. The Print and Email Receipt buttons will send an account statement to your printer or email account.

When you are finished, close the finished bank account program and the Class Viewer.

Creating the Transaction Window

We will now create the dialog which is used to handle deposits and withdrawals.

1. In the IDE, choose ‘Composer files’ from the Project Tree.
2. Press the ‘Create a Dialog’ button on the button bar. (The exclamation mark in the yellow triangle.)
3. Rename this new Dialog to ‘TransactionDialog.Dialog’. Then Double click it to open the Composer.
4. In the ***Title Bar*** field for the Dialog, enter ‘Transaction Dialog’.
5. Check the ‘modal’ checkbox.

Using the GridBag Layout

The GridBag Layout is one of the most versatile and flexible layouts. We will use it to build the entire Transaction Dialog.

1. From the Layouts page of the Object Palette, choose GridBag.
2. Click once in the Empty Space in the Working Model. A new GridBagLayout appears, initially with one Empty Space available.
3. Press the ***Add Empty*** button twice (at the top of the GridBag properties page). This will add an additional two Empty Spaces to the layout.
4. From the Basic page of the Object Palette, choose Label. Click once in the first Empty Space of the GridBag.
5. Change the Label’s text to “Enter amount:”.
6. From the Extended page of the Object Palette, choose ValidText. Click once in the second Empty Space of the GridBag.
7. Change its ‘Object name’ to ‘amountField’, its ‘Type of validation’ to ‘Float’ and its ‘Number of columns’ to ‘20’.
8. From the Basic page of the Object Palette, choose Button. Click once in the third Empty Space of the GridBag.
9. Change its ‘Object name’ to ‘commandButton’ and its ‘Button text’ to ‘Command’.
10. Choose ‘gridbag1’ from the partlist (at the top of the Composer window) to view the properties for the GridBag Layout.

The GridBag Layout allows you to give each part a set of constraints which govern how the parts should be positioned. The property sheet for the GridBag Layout

contains a table showing all of the constraints for each part. You can edit these directly in the table as well.

The table should have three rows, one for each part that we've added. The column on the left lists the part names. We will now adjust some of these properties.

11. For 'label2', change the X and Y both to 0. Change the Top and Left Insets both to 15. Click once on the small square in Fill column to specify a horizontal fill. (The small box changes to a horizontal bar. Clicking four times will cycle through all the states: None, Horizontal, Vertical, and Both.)
12. For 'amountField', change the X to 0 and the Y to 1. Change the Left, Bottom and Right Insets each to 15.
13. For 'commandButton', change the X to 0 and the Y to 2. Change the Bottom Inset to 15.

You've finished adjusting the constraints for the parts in the Transaction Dialog. The table should look similar to the following:

Name	X	Y	Wid...	Hei...	Anc...	Fill	X W...	Y ...	X In...	Y ...	Top I...	Left...	Botto...	Right...
label6	0	0	1	1			0	0	0	0	15	15	0	0
amountField	0	1	1	1			0	0	0	0	0	15	15	15
commandButton	0	2	1	1			0	0	0	0	0	0	15	0

Adding code

The layout for the Transaction Dialog is finished. Now we want to add some code to this Dialog.

The Dialog needs a variable to hold the amount that the user enters.

1. Choose 'Goto declaration code' from the ***Code*** menu in the Composer.
2. Press the Code Sourcerer button.
3. Choose 'Declare a new variable' and press ***Next***.
4. Choose 'float' from the 'primitive' choice box.
5. Enter 'amount' into the 'Choose its name' field.
6. Choose 'private' in the accessibility group.
7. Press ***Next***.
8. Leave the initial value at 0.0 and press ***Done***.

The Dialog needs methods to access the ‘amount’.

1. Choose ‘Goto method code’ from the ***Code*** menu in the Composer.
2. Type in the following lines. (Much of this can be generated using the Code Sourcerer, but we’ll type it to save time).

```
public float getAmount() {
     return amount;
}

public void setCommand(String s) {
     commandButton.setLabel(s);
     commandButton.invalidate();
     gridbag1.validate();
}
```

In order for declaration and method code to be integrated into the Working Model, the program needs to be initialized.

3. Choose ‘Initialize Class’ from the ***Program*** menu. (This executes any variable and method declarations, resets all components, and executes any constructor code as well.)

Now we’ll add some code to the ‘Command’ button and test the Dialog.

4. Press the Command button in the Working Model. Its properties appear in the Composer.
5. Choose the ‘Listeners’ tab and check ‘Listen for action events’.
6. Choose the newly added ‘Action’ tab.
7. Press the Code Sourcerer button and choose ‘Ask a part about one of its properties...’. Press ***Next***.
8. Choose ‘amountField’. Press ***Next***.
9. Choose ‘get value as a float’. Press ***Done***.
10. Replace the suggested new variable ‘float value’ with ‘amount’. Press ***Ok***.
11. Press the Code Sourcerer button again and choose ‘Change a property of an existing part...’. Press ***Next***.
12. Choose ‘TransactionDialog’ from the list. Press ***Next***.
13. Choose ‘Dispose of this Dialog’. Press ***Done***.

Test the Transaction Dialog

The Transaction Dialog is now finished, but let's add another line of code to test it.

1. Press the Code Sourcerer button and choose 'Java System Operations'. Press ***Next***.
2. Choose 'Write text to the standard output...'. Press ***Next***.
3. Choose the third option, 'The following variable expression' and enter
   `"You entered $" + getAmount()`

in the adjacent text field.

4. Press ***Done***.

You can now test the Dialog by entering a number in the amountField and pressing the 'Command' button. The dialog should vanish and a message should appear in the Java console. Choose 'Initialize Class' from the Program menu to reset the dialog.

When you are finished testing the Dialog, comment-out the last line by putting two slashes in front of it. The code should read

```
amount = amountField.floatValue();
this.dispose();
// System.out.println(String.valueOf("You entered $" + getAmount()));
```

You can now save and exit the Dialog Composer. All of the code you have just created is written to 'TransactionDialog.java'. You should compile this file now. (If you are using the demo version of Simplicity for Java, this file will not be created. You can use the 'FinishedTransactionDialog.java' instead for the rest of the tutorial.)

Creating the Bank Account application

We will now create the BankAccout Main Application.

1. In the IDE, choose 'Create a Main App' from button bar. (The lightbulb button).
2. Rename it 'BankAccount.Main' and then double-click it to open the Composer.
3. Change the 'Title Bar' to 'Bank Account'.
4. Choose the 'Listeners' page and check 'Listen for window events'.
5. Choose the newly added 'Window' page.
6. Choose 'windowClosing' from the window events list.

7. Press Code Sourcerer and choose 'Change a property of an existing part...'. Press ***Next***.
8. Choose 'BankAccount' from the list. Press ***Next***.
9. Choose 'dispose of this frame'. Press ***Done***.

Create the layout

We will create a compound layout using a series of layouts.

1. From the 'Extended' page on the Object Palette, choose 'InsetSizer' and click once in the Empty Space in the Working Model.
2. On the properties page of the InsetSizer, set the four Inset values to 20. Leave the Size fields blank.
3. From the 'Layouts' page on the Object Palette, choose 'Border' and click once in the Empty Space in the InsetSizer.
4. Change the Border Layout's vertical gap to 15.
5. From the 'Layouts' page on the Object Palette, choose 'Grid' and click once in the 'South' space of the Border Layout.
6. Change the Grid Layout's vertical and horizontal gaps to 15.
7. From the 'Basic' page on the Object Palette, choose 'Label' and click once in the 'North' space of the Border Layout.
8. Change the Label's 'Object Name' to 'output'.
9. Choose 'border2' from the Part List and uncheck the 'Show empty panels' checkbox.
10. From the 'Basic' page on the Object Palette, choose 'Button' and click once in each of the four grid spaces to create four buttons.
11. Change the button text on the four buttons to

Make a deposit	Make a withdrawal
Print a receipt	Email a receipt

Deposits and Withdrawals

We need to add some code to keep track of and let the user know the current account balance.

1. Choose 'Goto declaration code' from the ***Code*** menu.
2. Press the Code Sourcerer button and choose 'Declare a new variable...'. Press ***Next***.

3. Choose ‘float’ from the ‘primitive’ choice box.
4. Enter ‘balance’ into the ‘Choose its name’ field.
5. Choose ‘private’ in the accessibility group. Press ***Next***.
6. Leave the initial value at 0.0 and press ***Done***.
7. Choose ‘Goto method code’ from the ***Code*** menu.
8. Type the following method
```
public void setBalance(float f) {
   balance = f;
   output.setText("Your balance is $"+balance);
}
```
9. Choose ‘Goto constructor code’ from the ***Code*** menu.
10. Type the following
```
setBalance(0);
```

You can test the code that you’ve typed so far by choosing ‘Initialize Class’ from the ***Program*** menu. The output label should read “Your balance is $0.0”.

11. Press the ‘Make a deposit’ button to view its properties.
12. Enable the Action event for this button and goto its Action page.
13. Press the Code Sourcerer button.
14. Choose ‘Window operations...’. Press ***Next***.
15. Choose ‘Open a new Frame/Window/Dialog...’. Press ***Next***.
16. Choose ‘TransactionDialog’ or ‘FinishedTransactionDialog’ if you are using the Simplicity for Java Demo.
17. Enter ‘dialog’ where the Code Sourcerer asks for a name for this window.
18. Press ***Done***. Two lines of code are produced. The first creates the Dialog. The second shows the dialog on the screen.
19. Insert a second line and add a fourth line so that the code reads
```
TransactionDialog dialog = new
   TransactionDialog(_getFrame(this));
dialog.setCommand("Deposit");
dialog.show();
setBalance(balance += dialog.getAmount());
```
20. Select all of this text by choosing ‘Select All’ from the pop-up menu. The pop-up menu will appear when you right-click in the Sourcerer, or when you hold down the control key while you click in the Sourcerer.
21. Select ‘Copy’ from the pop-up menu.

22. Press the ‘Make a withdrawal’ button to view its properties.
23. Enable the Action event for this button and goto its Action page.
24. Select ‘Paste’ from the pop-up menu.
25. Change the `"Deposit"` command to `"Withdrawal"`.
26. Change the `+=` to `-=` on the last line.

Try testing the ‘Make a deposit’ and ‘Make a withdrawal’ buttons. The transaction dialog should appear, asking for an amount. The button should contain a ‘Deposit’ or ‘Withdrawal’ label.

Print a receipt

1. Press the ‘Print a receipt’ button to view its properties.
2. Enable the Action event and choose its Action page.
3. Press the Code Sourcerer button and choose ‘Printing Operations...’. Press ***Next***.
4. Choose ‘Print the contents of a part...’. Press ***Next***.
5. Enter “Print Receipt” for the PrintJob title.
6. Choose ‘output’ from the list of parts. Press ***Done***.

You can test this by pressing the ‘Print a receipt’ button. An operating system dependent dialog will appear in which you can choose a printer and any printer settings. When you confirm this dialog the receipt will be printed.

Email a receipt

1. Press the ‘Email a receipt’ button to view its properties.
2. Enable the Action event and choose its Action page.
3. Press the Code Sourcerer button and choose ‘Miscellaneous...’. Press ***Next***.
4. Choose ‘Send an E-mail message..’. Press ***Next***.
5. Enter your SMTP server name in the first entry field. Enter your email address in the ‘From:’ and ‘To:’ fields. Enter “Your account balance” in the ‘Subject:’ field. Leave the message field blank. Press ***Done***.
6. Find the sixth line of the generated code. It should read
   ```
   email.setMessage("");
   ```
   Change this to read
   ```
   email.setMessage(output.getText());
   ```

You can test this by pressing the ‘Email a receipt’ button. If you entered your email address and server correctly, an email receipt will be sent to you.

Finishing up

You can now save and exit the Composer. The Java source file that you have just created, BankAccount.java, will be written. (If you are using the demo version of Simplicity for Java this will be disabled. The finished code is provided with the tutorial, though.)

You can compile the BankAccount application by choosing ‘Java Source Files’ in the IDE Project Tree, right-clicking the ‘BankAccount.java’ (or ‘FinishedBankAccount.java’) icon, and choosing Compile.

You can run the BankAccount application by choosing ‘Java Class Files’ in the IDE Project Tree, double-clicking the ‘BankAccount.class’ (or ‘FinishedBankAccount.class’) icon, and pressing the ***Run*** button.

You can also test the application by invoking it directly from the Java interpreter. Make sure that the Project directory is on your CLASSPATH and use either of the following commands.

```
java Tutorial2.BankAccount
java Tutorial2.FinishedBankAccount
```

CHAPTER 4

Tutorial 3 - Using JavaBeans™ and Swing

This tutorial will guide you through creating a thermostat application. By completing this tutorial you will learn about

- adding JavaBeans to the Object Palette
- using JavaBeans in the Composer
- creating an application using Swing

It is estimated that this tutorial should take most people between 30 and 40 minutes to complete. It is assumed that you have already completed the first tutorial. You should already be familiar with the Simplicity IDE, creating simple layouts using the Object Palette, and choosing event listeners.

A Thermostat Application

If you haven't already done so, start Simplicity for Java in the manner appropriate for your operating system. The Simplicity IDE should be on the screen.

Open a project

You will now open a project which came with Simplicity for Java called Tutorial3.

1. Choose Open from the File menu or from the welcome screen.
2. A small window will appear listing the names of all the available projects. Choose ‘Tutorial3’ from the list and press the Ok button.

View the finished application

Let’s begin by viewing the completed tutorial so that you’ll know what we are going to accomplish.

1. Choose ‘Java Class Files’ from the Project Tree.
2. Double-click ‘FinishedThermostat.class’ and press the ‘Run’ button in the Class Viewer.

The thermostat application is very easy to use. Turn the knob by pressing the mouse button while the cursor is over the knob and then drag clockwise or counterclockwise. When you are finished, close the finished thermostat program and the Class Viewer.

Importing JavaBeans™

Our thermostat application is going to use a JavaBean which is not initially on the Object Palette in the Composer. This bean is the rotary knob which you saw in the finished application. We will now add this new part to the palette.

1. In the IDE, open the ‘import’ menu and choose the first item, titled ‘JavaBeans’. The ‘Import JavaBeans’ window will appear. On the left of the window will be a tree listing the items in your classpath. You can expand any item in the tree by double clicking it.
2. In the tree, expand the node of the item which has your Project directory. (This will be something similar to \Simplicity\Project). The tree will expand to show you the directories in your Project directory.
3. Expand the ‘datarep’ directory under the Project directory.
4. Single-click the ‘beans’ directory under the ‘datarep’ directory. All of the Java classes in the ‘beans’ directory will be listed in the ***Available Classes*** list.
5. Select ***Knob*** from this list and press the ***Add*** button at the bottom of the window.
6. Press ***Ok***.

You have just added a JavaBean to the Object Palette of any Composer in this Project. If you wish to see more details about the bean, you can double-click its icon in the ‘Java Beans’ group in the IDE. The Knob bean that we just added is

included with Simplicity as a demonstration. You can find thousands of different JavaBeans, though, on the Internet that you can use in your projects.

Creating the Thermostat application

We will now create the thermostat application.

1. In the IDE, choose ‘Create a Main App’ from the button bar. (The lightbulb button).
2. Rename it ‘Thermostat.Main’ and then double-click it to open the Composer.
3. Change the ‘Title Bar’ to ‘Thermostat’.

Create the layout

Our thermostat has a very simple layout.

1. From the ‘Layouts’ page on the Object Palette, choose ‘Bottom’ and click once in the Empty Space in the Working Model.
2. Change the Bottom Layout’s horizontal gap to 15.
3. Choose the ‘Beans’ page on the Object Palette. Here you will find any visual JavaBeans which you have imported.
4. Select ***Knob*** from the ‘Beans’ page, and click once in the Bottom layout which you previously added.
5. Choose the ‘Swing 1’ page on the Object Palette.
6. Select Progress and click once in the Bottom Layout.
7. Select Label and click once in the Bottom Layout.
8. From the Part List at the top of the Composer window, choose ‘bottom1’. Uncheck ‘Show Empty Panel’ to get rid of the last Empty Space.

We will now set up some of the properties for the parts which we have just added.

9. Click once on the progress bar.
10. Set the ***orientation*** to ‘vertical’.
11. Enter 73 for the ***current value***.
12. Enter 32 for the ***minimum value***.
13. Enter 212 for the ***maximum value***.
14. Choose the progress bar’s ‘Visual’ property page, and change the foreground color to red.
15. Click once on the label.

16. Change the label's text to "degrees Fahrenheit".

Working with JavaBeans

In the previous section we added several parts to our application, one of which is the Knob bean that we imported earlier. Working with JavaBeans is very similar to working with parts that are preinstalled on the palette. Lets look at the properties page for this bean.

1. Click once on the ***Knob*** in the Working Model.

Simplicity generates a custom property page for any bean that you import using the bean's exposed properties. From left to right, you see a list of property names, an input field, the data type, and the default value. Any properties which you leave blank will keep their default value.

2. Enter 32 for the ***minimum***.
3. Enter 212 for the ***maximum***.
4. Enter 73 for the ***value***.
5. Enter 35 for the ***radius***.

Handling events with JavaBeans works exactly the same as with other parts.

6. Select the 'Listeners' page of the Knob.
7. Check 'Listen for KnobTurn events'. A message will be briefly displayed in the Java Console as Simplicity sets up the new event type. When this finishes a 'KnobTurn' tab will appear.
8. Choose the KnobTurn tab.
9. Press Code Sourcerer and choose 'Ask a part about one of its properties...'. Press ***Next***.
10. Choose 'bean2' from the list. Press ***Next***.
11. Choose 'value'. Press ***Done***. Press ***Ok*** in the dialog which then appears.
12. Press Code Sourcerer and choose 'Change a property of an existing part...'. Press ***Next***.
13. Choose 'jProgressBar3' from the list. Press ***Next***.
14. Choose 'current value'. Press ***Done***.
15. In the generated code, replace 0 with "value".

The final code should appear as

```
int value = bean2.getValue();
jProgressBar3.setValue(value);
```

You can test this by turning the knob using the mouse. The progress bar should respond by showing the mercury level.

Creating a Swing Application

In our application, we have used two parts from the Swing pages on the palette, the JLabel and the JProgressBar. The application that we created, though, is still using the AWT's Frame and Panel to build the user interface. Since we want this application to be written using Swing's JFrame and JPanel, we have one more thing to do.

1. At the top of the Composer window, press the 'View Code' button (the one with the eye on it). Notice that the class uses Frame and Panel to build the user interface.
2. At the top of the Composer window, choose 'Thermostat' from the Part List.
3. Check the 'Generate Swing Code' checkbox.
4. Press the 'View Code' button again to refresh the code viewer window. Notice that the application now uses Swing's JFrame and JPanel to build the user interface.

Notice that the 'Default close action' field is enabled on the MainWindow property page when you select 'Generate Swing Code'. This option makes it easy to specify a default behavior when a JFrame is closed by the user.

5. Select 'Dispose' from the 'Default close action' field. This tells the window to dispose of itself when the user requests that the window close. It saves us the work of creating a WindowListener which we previously did in Tutorial 1.

Finishing up

You can now save and exit the Composer. The Java source file that you have just created, Thermostat.java, will be written. (If you are using the Tryout version of Simplicity for Java this will be disabled. The finished code is provided with the tutorial, though.)

You can compile the Thermostat application by choosing 'Java Source Files' in the IDE Project Tree, right-clicking the 'Thermostat.java' (or 'FinishedThermostat.java') icon, and choosing Compile.

You can run the Thermostat application by choosing ‘Java Class Files’ in the IDE Project Tree, double-clicking the ‘Thermostat.class’ (or ‘FinishedThermostat.class’) icon, and pressing the ***Run*** button.

You can also test the application by invoking it directly from the Java interpreter. Make sure that the Project directory and the Swingall.jar file are on your CLASSPATH and use either of the following commands.

```
java Tutorial3.Thermostat
java Tutorial3.FinishedThermostat
```

CHAPTER 5

Tutorial 4 - Advanced Swing and JDBC

This tutorial will guide you through creating a database application. By completing this tutorial you will learn about

- using the Swing data models to control the content of a JTable
- accessing databases using JDBC
- managing groups in the IDE

It is estimated that this tutorial should take most people between 30 and 40 minutes to complete. It is assumed that you have already completed the first tutorial. You should already be familiar with the Simplicity IDE, creating simple layouts using the Object Palette, and choosing event listeners.

A Database Application

If you haven't already done so, start Simplicity for Java in the manner appropriate for your operating system. The Simplicity IDE should be on the screen.

Open a project

You will now open a project which came with Simplicity for Java called Tutorial4.

1. Choose Open from the File menu or from the welcome screen.
2. A small window will appear listing the names of all the available projects. Choose 'Tutorial4' from the list and press the Ok button.

Creating the Database application

We will now create the database application. This application will allow users to type an SQL query and view the results in a table.

1. In the IDE, choose 'Create a Main App' from the button bar. (The lightbulb button).
2. Rename it 'QueryTable.Main' and then double-click it to open the Composer.
3. Change the 'Title Bar' to 'SQL Query Table'.
4. Check 'Generate Swing Code'.

Create the layout

Our application has a very simple layout.

1. From the 'Layouts' page on the Object Palette, choose 'Border' and click once in the Empty Space in the Working Model.
2. Select ***JScrollPane*** (Scroll) from the 'Swing 2' page, and click once in the Center of the Border Layout. a 'viewport' will appear in the Center.
3. Select ***Table*** from the 'Swing 2' page, and click once in the viewport in the Center of the Border Layout.
4. From the 'Layouts' page on the Object Palette, choose 'Flow' and click once in the Empty Space in the 'North' of the Border Layout.
5. Select ***Label*** from the 'Swing 1' page, and click once in the Flow Layout.
6. Select ***TextField*** from the 'Swing 2' page, and click once in the Flow Layout.
7. From the Part List at the top of the Composer window, choose 'border1'. Uncheck 'Show Empty Panel'. Do the same for 'flow4'.

We will now set up some of the properties for the parts which we have just added.

8. Click once on jLabel5.
9. Change the ***JLabel text*** to "Enter a query:".
10. Click once on jTextField6.
11. Change the ***Object name*** to query.
12. Erase any text in ***JTextField text***.

13. Change the ***Number of columns*** to 30.
14. Select 'jTable3' from the Part List. Change its ***Object name*** to 'table'.

Using JDBC (Java Database Connectivity)

We will now add a few lines of code which will connect our application to a database using JDBC. In this application we will use the SimpleText™ Database Server and JDBC Driver. (The SimpleText database is a free database server and JDBC implementation from Thought, Inc. written completely in Java. SimpleText is included with this tutorial so that the tutorial will operate consistently everywhere. We wish to thank Thought, Inc. for allowing us to redistribute SimpleText.)

To start we will need some variables to store data.

1. Select ***Goto declaration code*** from the ***Code*** menu in the Composer.
2. Type the following code. (All of the text after the "//" is a comment. It is provided to help you understand the code. You don't have to type it if you don't want to.)

```
// import various standard Java classes
import java.sql.*;
import java.util.*;

Connection db;          // The JDBC connection
Statement statement;    // an SQL statement
ResultSet rs;           // an SQL ResultSet
ResultSetMetaData meta; // meta data describing the results
Vector data;            // storage for the results
```

Next, we will connect to the database server in the constructor of our class.

3. Select ***Goto constructor code*** from the ***Code*** menu in the Composer.
4. Type the following code.

```
try {
        // Load the database driver
        new jdbc.SimpleText.SimpleTextDriver();
```

5. Press the Code Sourcerer button.
6. Choose 'File operations...' and press ***Next***.
7. Choose 'Create new file object from pathname...' and press ***Next***.

8. Press the ***Browse*** button. Browse to /Project/Tutorial4/Databases/ADDRESS.SBF in the Simplicity install directory. Press ***Done***. The Code Sourcerer will produce the appropriate code.
9. Press the Code Sourcerer button.
10. Choose 'File operations...' and press ***Next***.
11. Choose 'Get the directory from a File object...' and press ***Next***.
12. Leave the default value of 'theFile' and press ***Done***.
13. Leave the default value of 'dir' and press ***Ok***. The Code Sourcerer will produce the following line of code.

```
String dir = theFile.getParent();
```

14. Type the following code.

```
    Properties prop = new Properties();
    prop.put("Directory",dir);

    // Connect to the database
    db = DriverManager.getConnection("jdbc:SimpleText",prop);
    statement = db.createStatement();
} catch ( SQLException e ) {
}
```

15. Choose 'Initialize Class' from the Program menu to initialize the code you just typed.

We want to allow the user to type any valid SQL query in the JTextField and display the results of the query in the JTable.

16. Click once on the JTextField named 'query' to view its properties.
17. Choose its ***Listeners*** page.
18. Check 'Listen for Action events'. Select the new ***Action*** page which appears.
19. Type the following code.

```
// try to execute the query. If it fails, notify the user.
try {
        // execute whatever the user typed in query
        rs = statement.executeQuery(query.getText());
        meta = rs.getMetaData();// get the Meta Data
        data = new Vector();// initialize the data storage
        while( rs.next() ) {
             for(int i=0;i<meta.getColumnCount();i++) {
```

```
                    // store the data in row order
                    data.addElement(rs.getString(i+1));
                }
            }
        } catch ( Exception e ) {
            getToolkit().beep(); // bad query
            System.out.println("Bad SQL query: "+e.getMessage());
            data = null;// indicate failure
        }
        // tell the table to update its contents
        ((AbstractTableModel)table.getModel())
            .fireTableStructureChanged();
```

Test this code by typing "`select * from address`" in the TextField, and then press enter. If any error occurs, a message will displayed.

Creating a TableModel

So far, we've connected to a database, we've set up code to send an SQL query to the database, and we've stored the results. We now want to be able to display the data. To do this we will create a TableModel.

1. From the Part List, select 'table' to view the properties of our JTable.
2. Choose the ***TableModel*** tab. This code area contains the methods which describe the data we wish to place in the table.
3. Choose 'getColumnCount' from the method choice field (above the code area).
4. Enter the following code:
   ```
   try {
       return meta.getColumnCount();
   } catch ( Exception e ) { return 0; }
   ```
5. Choose 'getColumnName' from the method choice field.
6. Enter the following code:
   ```
   try {
       return meta.getColumnName(columnIndex+1);
   } catch ( Exception e ) { return "Error"; }
   ```
7. Choose 'getRowCount' from the method choice field.
8. Enter the following code:
   ```
   try {
   ```

```
        return data.size()/meta.getColumnCount();
    } catch ( Exception e ) { return 0; }
```

9. Choose ‘getValueAt’ from the method choice field.

10. Enter the following code:

```
    try {
        return data.elementAt(
          meta.getColumnCount()*rowIndex + columnIndex);
    } catch ( Exception e ) { return "Error"; };
```

The TableModel code is executed on the fly in the same manner as the event code which you are already familiar with. As you enter each of the above pieces of code, the table will show the results.

Working with SQL queries

Our program is finished. We can test it by entering SQL queries into the TextField and pressing Enter. Try the following queries to see what they do.

```
select * from address
select Name,State from address
select * from address where state='OH'
select Name,City from address where state='OH'
```

Finishing up

You can now save and exit the Composer. The Java source file that you have just created, QueryTable.java, will be written. (If you are using the Tryout version of Simplicity for Java this will be disabled. The finished code is provided with the tutorial, though.)

You can compile the QueryTable application by choosing ‘Java Source Files’ in the IDE Project Tree, right-clicking the ‘QueryTable.java’ (or ‘FinishedQueryTable.java’) icon, and choosing Compile.

You can run the QueryTable application by choosing ‘Java Class Files’ in the IDE Project Tree, double-clicking the ‘QueryTable.class’ (or ‘FinishedQueryTable.class’) icon, and pressing the ***Run*** button.

You can also test the application by invoking it directly from the Java interpreter. Make sure that the Project directory and the Swingall.jar file are on your CLASSPATH and use either of the following commands.

```
java Tutorial4.QueryTable
java Tutorial4.FinishedQueryTable
```

CHAPTER 6

Tutorial 5 - Using the Canvas Composer

This tutorial will guide you through creating a small traffic light JavaBean, and inserting it into a program. By completing this tutorial you will learn about

- The Canvas Composer
- Creating and using your own JavaBeans

It is estimated that this tutorial should take most people 40 minutes to complete. It is assumed that you have already completed the first tutorial. You should already be familiar with the Simplicity IDE, creating simple layouts using the Object Palette, choosing event listeners, and using the Code Sourcerer.

A Traffic Light JavaBean

If you haven't already done so, start Simplicity for Java in the manner appropriate for your operating system. The Simplicity IDE should be on the screen.

Open a project

You will now open a project which came with Simplicity for Java called Tutorial5.

1. Choose Open from the File menu or from the welcome screen.

2. A small window will appear listing the names of all the available projects. Choose ‘Tutorial5’ from the list and press the Ok button.

Creating the Traffic Light

We will now create the traffic light.

1. In the IDE, choose ‘Create a Canvas’ from the button bar (The button with the Square, Circle, and line). A new file will be created with a name similar to ‘Untitled0.Canvas’.
2. Control-click on the new Canvas file, and choose ‘Rename’ from the pop-up menu which will appear. You will now be able to edit the name of the new Canvas file.
3. Rename the new file ‘TrafficLight’. Simplicity will add .Canvas to the name for you, if needed.
4. Double-click TrafficLight.Canvas. The Canvas Composer will open your new file.

Set the Size of the Canvas

The Canvas is the area where you will be working. It is possible to edit the size of the Canvas whenever you like, but in this tutorial, you will only change it once.

1. Make sure you are looking at the Canvas Composer.
2. Choose the “Canvas Methods” tab on the Canvas Composer. This is a special tab which only appears in the Canvas Composer.
3. Use the choice box to the left of the Code Sourcerer button to choose getPreferredSize().

You should see the following code:

```
return new Dimension(200,200);
```

This is the code which tells Simplicity how large the Canvas should be. It will be easier to work with a larger Canvas. Edit the code so that it reads

```
return new Dimension(300,300);
```

The Canvas window will change to reflect the new, larger, size.

Set up initial variables.

Next you will set up some variables for the traffic light.

1. Choose “Goto declaration code” from the Code menu of the Canvas Composer.
2. Type the following lines into the text area. This text could be created using the Code Sourcerer, using “Declare a new variable...” and then adjusting the various settings.

```
static final int RED = 0;
static final int YELLOW = 1;
static final int GREEN = 2;
int state = 0;
```

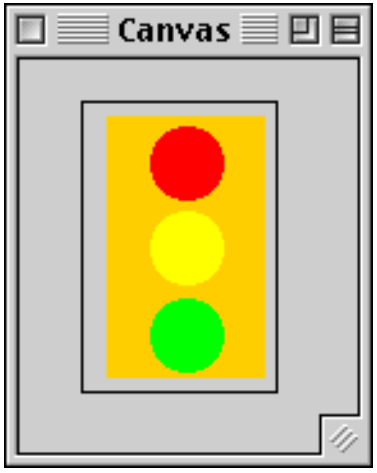
These lines of code create a variable called ‘state’ which stores which color the traffic light should display, and define three possible states, RED, YELLOW, and GREEN.

3. Choose “Initialize Class” from the Program menu. This will cause Simplicity to restart the working model, giving it access to the variables you just created.

Drawing the Traffic Light

We will now start to draw the traffic light itself.

1. Choose “Rectangle” from the Graphics page of the Object Palette.
2. Drag on the working model to get a rectangle that looks like the outer rectangle in the picture below.



Next we will change the color to orange and draw the inner part of the traffic light.

3. Choose “Orange” from the Color page of the Object Palette.
4. Click once anywhere on the working model. Now the next thing which is drawn will be orange.
5. Choose “Filled Rectangle” from the Graphics page of the Object Palette.
6. Draw a Filled rectangle inside the first similar to the inner rectangle in the figure.

Now you will start to set up some code which will let the traffic light display the three lights.

7. Click on the Code Sourcerer button in the Simplicity Composer.
8. Choose Java Language Statements... and press the ***Next*** button.
9. Choose if ... and press the ***Next*** button.
10. type “state == RED” in the conditions box, and press the ***Done*** button.

Now you will draw the first of the bulbs of the traffic light.

11. Choose “Red” from the Color page of the Object Palette.
12. Click once anywhere on the working model.
13. Choose “Filled Oval” from the Graphics page of the Object Palette.
14. Hold down the control key and drag the oval near the top third of the rectangle. The control key constrains the oval to be a circle.

If you wish to change the location or the size of the circle, you may edit the line that was just generated in the Simplicity Composer. If you dislike the circle that you drew, you could also erase the line that was generated and try again.

The second and third bulbs are set up and drawn in a very similar manner.

15. Click on the Code Sourcerer button in the Simplicity Composer. Choose Java Language Statements... and press the ***Next*** button.
16. Choose if... and press the ***Next*** button.
17. type “state == YELLOW” in the conditions box, and press the ***Done*** button.
18. Choose “Yellow” from the Color page of the Object Palette.
19. Click once anywhere on the working model.
20. Choose “Filled Oval” from the Graphics page of the Object Palette.
21. Hold down the control key and drag over the middle third of the rectangle, to draw a circle.

22. Press the Code Sourcerer button in the Simplicity Composer. Choose Java Language Statements... and press the ***Next*** button.
23. Choose if... and press the ***Next*** button.
24. type “state == GREEN” in the conditions box, and press the ***Done*** button.
25. Choose “Green” from the Color page of the Object Palette.
26. Click once anywhere on the working model.
27. Choose “Filled Oval” from the Graphics page of the Object Palette.
28. Hold down the control key and drag the oval over the bottom third of the rectangle to draw a circle. Your traffic light should now look like the figure.
29. Now you must do some editing of the generated code. The last part of the code which is generated will look like this:

```
if (state == RED) {
    // type statements that will happen if the above is true.
}
g.setColor(Color.red);
g.fillOval(79,33,29,29);
if (state == YELLOW) {
    // type statements that will happen if the above is true.
}
g.setColor(Color.yellow);
g.fillOval(79,66,29,29);
if (state == GREEN) {
    // type statements that will happen if the above is true.
}
g.setColor(Color.yellow);
g.fillOval(79,100,29,29);
```

You need to change the code so that the setColor and fillOval methods are inside their respective if statements. This is easily accomplished by deleting each close bracket, “}”, and typing it after the following fillOval method. You may also wish to adjust the tabs in the code. This section of your code will now look like this:

```
if (state == RED) {
    // type statements that will happen if the above is true.
    g.setColor(Color.red);
    g.fillOval(79,33,29,29);
}
if (state == YELLOW) {
    // type statements that will happen if the above is true.
    g.setColor(Color.yellow);
```

```
    g.fillOval(79,66,29,29);
}
if (state == GREEN) {
    // type statements that will happen if the above is true.
    g.setColor(Color.green);
    g.fillOval(79,100,29,29);
}
```

Light changing methods

We will now create the methods to control the state of the traffic light.

1. Choose “Goto method code” from the Code menu in the Canvas Composer.
2. Type the following code into the method section:

```
public int getState() {
    return state;
}
public void setState(int state) {
    this.state = state;
    repaint();
}
public void cycle() {
    state++;
    if (state > GREEN) state = RED;
    repaint();
}
```

3. Close the Canvas Composer by choosing ‘Exit’ from the file menu in the Composer. When you are asked if you wish to save your changes, answer “yes”. If you are running the tryout version of Simplicity for Java, you may use FinishedTrafficLight.java for the rest of this Tutorial.
4. In the Simplicity IDE, click on the ‘Java Source Files’ Group in the Project Tree to see the file that was just created, TrafficLight.java.
5. Control-click TrafficLight.java in the IDE.
6. Choose ‘Compile’ from the pop-up menu which will appear.
7. Choose ‘Java Class Files’ or ‘All Files’ in the Project Tree to see the new class file that you’ve created.

Creating the main application

You have just created a JavaBean! Now you will make a simple application which will use the traffic light.

1. Choose the 'Composer Files' group in the Project Tree.
2. Create a new Main Application by choosing "Main App" from the Create menu.
3. Control-click on the new Main application file in the IDE, and then choose "Rename" from the pop-up menu that appears.
4. Rename the newly created main application "RunTraffic".
5. Double-click the RunTraffic icon to open the Composer.
6. From the Layouts page of the Object Palette, choose Border.
7. Click once in the Empty space in the Working Model. A new Border Layout appears. You will see five empty spaces, labeled North, South, East, West, and Center.
8. Select Button from the Basic page of the Object Palette
9. Click once in the South of the Border Layout. The button you created will be used to cycle the traffic light.
10. Choose the 'Beans' page of the Object Palette. Your JavaBean, TrafficLight, has been automatically placed on this page.
11. Choose TrafficLight and then click in the Center section of the Border Layout.
12. Click in the Composer window. Change the 'ObjectName' of your bean to 'theLight'.
13. Choose "Hide all empty panels" from the Parts menu. This is a fast way to hide the empty panels in the Working Model.

Now you will make the button change the traffic light.

14. Press the button at the bottom of the working model. Its properties notebook will appear in the Composer.
15. Change the 'Button text' to "Cycle Light".
16. Choose the Listeners page from the properties notebook of the button.
17. Select 'Listen for Action events'.
18. A new page appears in the properties notebook, called "Action". Choose this page, and then add the following code to the actionPerformed area:

```
theLight.cycle();
```

The traffic light demo is now almost finished. You can test it out by clicking on the "Cycle Light" button a few times.

You still need to tell your program what it should do when its window is closed. In this Tutorial, the program will quit when its window is closed.

1. Toward the top of the Composer window, find the part list Choice box and choose the first item, 'RunTraffic'.
2. Select the 'Listeners' page.
3. Select 'Listen for window events'. A new page appears in the properties notebook called 'Window'. Select this new page.
4. Choose 'windowClosing' from the Choice box to the left of the Code Sourcerer button.
5. Press the Code Sourcerer button.
6. Choose 'Java system operations...', and press the ***Next*** button.
7. Choose 'Exit the program with termination code', and press the ***Done*** button.

The Code Sourcerer will generate the appropriate code for the program to quit when its window is closed. If you run this from within Simplicity, Simplicity will indicate to you that the program has tried to quit by displaying a dialog.

Finishing up

There are now just a few steps left.

1. Close the composer. When asked if you want to save your changes, answer "yes".
2. In the IDE, choose 'Java Source Files' in the Project Tree. Control-click the RunTraffic.java file (use FinishedRunTraffic if you are using the Tryout version). Choose 'Compile & Run' from the pop-up menu. Simplicity will compile the source file and then run the program.

CHAPTER 7

Integrated Design Environment

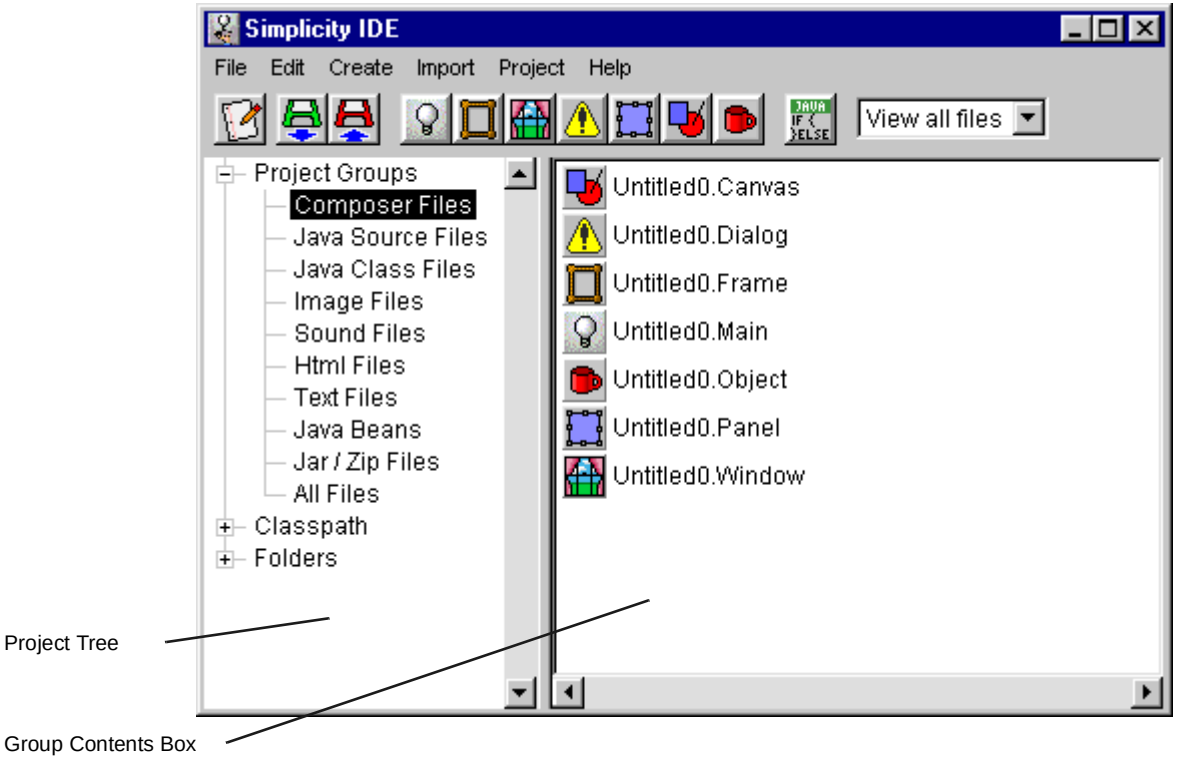
This chapter will discuss Simplicity's Integrated Design Environment (IDE) and how to use it to build and manipulate projects.

It will cover

- The IDE window, button bar, and menu bar
- File Groups
- Editing the Classpath
- Adding Folders
- Creating and importing files
- Program settings

The IDE Window

The first window which appears after Simplicity for Java is started is the IDE Window.



The IDE is centered around the Project Tree. The Project Tree organizes all of the elements in your project into three key areas:

- Project Groups
- Classpath
- Folders

The Classpath and the Folders areas will be discussed first, then the Project Groups.

The Classpath

The Classpath is where Simplicity looks for Java classes on your hard drive. It can include directories, Zip files, and Jar files. The Classpath should indicate directories at the root of the subtree of your java files. If a user wishes to access the class datarep.common.Util, and the datarep directory is located inside a directory named MyProjects, the Classpath would need an entry similar to `/home/user/MyProjects`. Simplicity will search through the Classpath starting at the top, and stopping when the specified class is found.

Simplicity is started using a "primordial Classpath" which contains the core classes that Simplicity requires. This primordial classpath cannot be edited from within Simplicity, and so it is not displayed in the Project Tree. Simplicity will always check the dynamic Classpath first when looking for classes, then the primordial Classpath. It is recommended, therefore, that the Classpath used to start Simplicity be as simple as possible (just the Simplicity.zip file, the Swingall.jar file, and the datarep_common.jar file), and all other changes to the Classpath be made within Simplicity.

Simplicity will generate package statements based on where a file is with respect to the Classpath. In the example above, files created in the "MyProjects" directory will not have any package statements. Files created in the "datarep" directory will be made part of the "datarep" package. Complex packages can be created by having folders inside of folders.

The Folders area

The Folders area is a place to put any directory that you want quick access to. It can be used to quickly traverse and view folders which are commonly used, or to find other files in the user's file system.

Using the Classpath and the Folders area

It is possible to navigate the Classpath or the Folders area by expanding the nodes in the tree view of the directories which is displayed in the Groups List box area. A node with a minus sign is expanded. A node with a plus sign is collapsed.

Items may be added or removed from the Classpath and the Folders area using the Classpath editor and the Folders editor, both found on the Edit Menu in the IDE.

Project Groups

A project can also be organized into groups of related items. The Groups in a project are listed in the first area of the Project Tree and can be modified by choosing ***Edit Groups*** from the ***Edit*** menu

A group consists of fields.

1. Group Name
2. Group Directory
3. Group Extensions

The user can edit any Group and modify the directory to be searched and the filename extensions to be included. The directory may be specified relative to the project's directory, or as a fully qualified path. This allows a project to access files located anywhere on the user's file system. Any number of extensions may be specified. If no extensions are specified, all files in the directory are included as well as the parent directory ('..') allowing complete file system navigation. Any group whose extensions include gif, jpg, or xbm will be included in any Image Selection Dialog.

Groups can also be used to filter the files which are viewed in the Classpath or the Folders area. If a group is chosen from the drop-down menu on the menubar, only files with the appropriate extensions will be viewed. This can be useful when directories with many files in them are being viewed.

Editing parts of the Project Tree

The Project Groups, the Classpath, and the Folders area can all be edited in the IDE.

Editing Groups using the IDE Group Editor

The IDE Group Editor lists the current groups. When the user chooses a Group, the group's fields are displayed. The values can be edited in the entry fields on the right of the Group Editor.

The following commands are available.

- ***New*** Create a new Group, initially named 'new group'.

- ***Up*** Move the selected Group one position upward in the Project Tree.
- ***Down*** Move the selected Group one position downward in the Project Tree.
- ***Delete*** Erase the selected Group.
- ***Ok*** Accept the changes.
- ***Cancel*** Dismiss the changes.

Editing The Classpath

The Classpath Editor lets the user manipulate the items on the Classpath and add new directories or Jar/Zip files. The following commands are available.

- ***Up*** Move the selected directory or Jar/Zip file one position upward in the Classpath.
- ***Down*** Move the selected directory or Jar/Zip file one position downward in the Classpath.
- ***Remove*** Remove the selected directory or Jar/Zip file from the Classpath.
- ***Add directory*** Opens a new dialog to choose a directory to add to the Classpath. The current directory is shown at the top of the dialog. The special files ".." and "." are displayed. It is possible to move to the parent directory by double-clicking on the ".." file. The "." file signifies the current directory.
- ***Add Jar/Zip file*** Opens a system-dependent dialog to choose a Jar or Zip file to add to the Classpath.
- ***Ok*** Accept the changes to the Classpath.
- ***Cancel*** Dismiss the changes to the Classpath.

Editing The Folders area

The Folder List Editor lets the user manipulate the items in the Folders area and add new directories or Jar/Zip files. The Folder List Editor is similar to the Classpath Editor. The following commands are available.

- ***Up*** Move the selected directory or Jar/Zip file one position upward in the Folders area.
- ***Down*** Move the selected directory or Jar/Zip file one position downward in the Folders area.
- ***Remove*** Remove the selected directory or Jar/Zip file from the Folders area.
- ***Add directory*** Opens a new dialog to choose a directory to add to the Folders area. The current directory is shown at the top of the dialog. The special files

".." and "." are displayed. It is possible to move to the parent directory by double-clicking on the ".." file. The "." file signifies the current directory.

- ***Add Jar/Zip file*** Opens a system-dependent dialog to choose a Jar or Zip file to add to the Folders area.
- ***Ok*** Accept the changes.
- ***Cancel*** Dismiss the changes.

Opening items in the Group Contents Box

Choosing any item in the Project Tree displays its contents in the Group Contents Box.

The Group Contents Box shows the name of each item in the group alongside an icon which indicates the type of item and the behavior of the item when opened. The Group Contents Box is aware of and has special behaviors for

- Composer Files (*.Main, *.Frame, *.Window, *.Dialog, *.Panel, *.Canvas, and *.Object)
- Image files (*.gif, *.jpg, *.xbm)
- Sound files (*.au, *.wav)
- Java source files (*.java)
- Java class files (*.class)
- Java archives (*.zip, *.jar)
- Text and HTML files (*.txt, *.html, *.htm)
- Java Beans
- Folders

Each item has a pop-up menu associated with it which contains those actions that are appropriate for the selected item. For example, a Java class file's pop-up menu has the default options: Open, Copy, Rename, and Delete as well as an option to invoke the compiler.

The first action on an item's popup menu is always Open. This action can also be performed by double clicking the item.

- Opening a Composer file will launch the Simplicity Composer. These files store the contents of a Main Application or Applet, a Frame, a Window, a Panel, or an Object Composer.

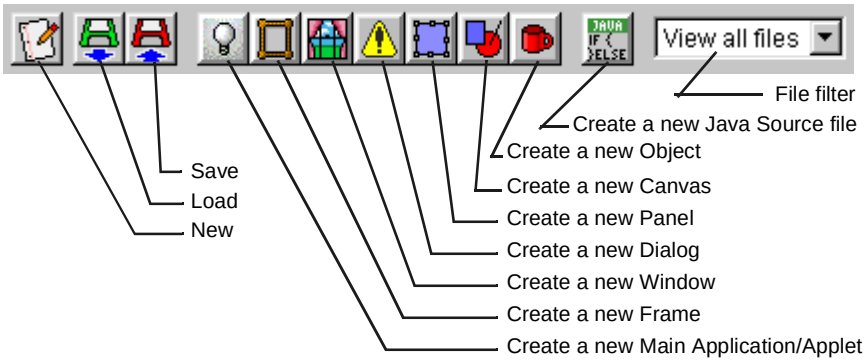
- Opening a Java source file will launch the Simplicity Java Editor. This default action may be overridden in the Program Settings in order to specify a preferred Editor.
- Opening a Java class file or Java Bean will launch the Simplicity Class Viewer which will allow inspection of the contents of that class. If the class has a `public static void main(String[])` method, the Class Viewer will also include a button which will execute this program entry point.
- Opening an HTML file will launch the preferred web browser specified in the Program Settings.
- Opening an Image file will launch the Simplicity Image Viewer. This default action may be overridden in the Program Settings in order to specify a full-featured image editor.
- Opening a Sound file will launch the Simplicity Sound Player. This default action may be overridden in the Program Settings in order to specify a full-featured sound editor.
- Opening a folder will display the contents of that folder in the Group Contents Box. Folders only appear when all files are being displayed.
- Opening all other files will launch a Text Editor.

IDE Menu Bar

The IDE Menu Bar provides a set of commands for manipulating items in a project. The Menu Bar also provides commands for opening, saving, and closing projects as well as exiting from Simplicity and getting help.

IDE Button Bar

The IDE Button Bar provides easy access to the most commonly used menu items.



File Menu

The File menu has the following options.

- ***New Project...*** This option will prompt the user for the name of a new project to create, and the project's main directory. The main directory can be anywhere the user chooses. A ***Browse...*** button is available to help the user choose the location. If needed, the project's main directory will be created at the specified place. Once the main directory is specified, Simplicity will then create a set of directories for the new project. If a project was previously open, the user will be given the option of saving any changes before it is closed.
- ***Open Project...*** This option will prompt the user to select one of the projects that has previously been created. Projects may be anywhere the user wishes. The Project Directory (used in Simplicity 1.1 and earlier) may still be used, but it is not necessary. If a project created in Simplicity 1.1 does not appear in the list, try using the ***Migrate Old Projects...*** menu item. If this menu item is chosen when a project is already open, the user will be given the option of saving any changes before it is closed.
- ***Save Project*** This option will save any changes to the current project.
- ***Close Project*** This option will close the current project. The user will be given the option of saving any changes before it is closed.
- ***Migrate Old Projects...*** This option will allow Simplicity to import projects from earlier versions of Simplicity. The user may choose a Project directory, and then pick which projects to import. After using this option, the projects will

appear in the list given by ***Open Project...*** To share projects from Simplicity 1.2 or later, simply copy the .Simplicity file corresponding to the desired project into the Personal Settings Directory.

- ***Program Settings...*** This option will launch the Program Settings Dialog (page 64) which allows the user to modify many aspects of how Simplicity for Java operates. All changes are applied after the user presses the ***Ok*** button and affect all projects.
- ***Show Console/Hide Console*** These options will make visible or hide the Java Input/Output Console.
- ***Exit*** This option will exit from Simplicity for Java. If a project was previously open, the user will be given the option of saving any changes before it is closed.

Edit Menu

The Edit menu has the following options. Those options which modify items will make permanent changes to files on your hard drive.

- ***Open Selection*** This option will open the item in the Group Contents Box which has been selected with the mouse. This has the same effect as double clicking an item or choosing ***Open*** from an item's pop-up menu.
- ***Close Selection*** This option will close the item in the Group Contents Box which has been selected with the mouse.
- ***Rename Selection*** This option will allow the user to change the name of an item. A text field will appear in place of the item containing the old name. Pressing the mouse button anywhere outside the text field will accept the change. This has the same effect as choosing ***Rename*** from an item's pop-up menu.
- ***Copy Selection*** This option will make a copy of an item. This has the same effect as choosing ***Copy*** from an item's pop-up menu.
- ***Delete Selection*** This option will delete an item.This has the same effect as choosing ***Delete*** from an item's pop-up menu.
- ***Refresh*** This option will cause the Group Contents Box to update itself with any changes that have been made to the project directory. This may be necessary when external programs create or modify files in a project.
- ***Edit Groups*** This option will launch the Group Editor. This will allow the list of groups to be modified.

- ***Edit Classpath*** This option will let the user interactively add folders and/or Zip files to the Classpath. They will be available ***immediately***. They will also be shown immediately in the IDE's main window.
- ***Edit Folders*** This option will let the user interactively add folders to the IDE's main window. This can be a useful way to access files which do not have to be in the Classpath.

Create Menu

The Create menu has the following options. The user may need to select an appropriate group in order to see the new item.

- ***Main App*** This option will create a new Main Application/Applet Composer file. This will allow the user to create a new java program.
- ***Frame*** This option will create a new Frame Composer file. This will allow the user to create a framed window which can be launched from a program.
- ***Window*** This option will create a new Window Composer file. This will allow the user to create an unframed window which can be launched from a frame.
- ***Dialog*** This option will create a new Dialog Composer file. This will allow the user to create a (possibly modal) dialog which can be launched from a frame.
- ***Panel*** This option will create a new Panel Composer file. This will allow the user to create a reusable compound object which can be inserted into other Composer files.
- ***Canvas*** This option will create a new Canvas Composer file.
- ***Object*** This option will create a new Object Composer file. This will allow the user to create a non-graphical object which can be used in other Composer files.
- ***Java File*** This option will create a new empty Java source file.
- ***Text File*** This option will create a new empty text file.
- ***Data File*** This option will create a new empty binary data file.
- ***HTML File*** This option will create a new empty HTML file.
- ***Image File*** This option will create a new graphic file. It is useful as a template or place holder for a graphic that will be created later.
- ***Sound File*** This option will create a new sound file. It is useful as a template or place holder for a sound that will be created later.
- ***Folder*** This option will create a new folder.

Import Menu

The Import menu has the following options. The user may need to select an appropriate group in order to see the new item.

- ***JavaBeans*** This menu item will launch the 'Import JavaBeans Dialog'. The user will be presented with a tree view of the current classpath. Beans can be selected from the 'Available Classes' list and added by pressing the 'Add' button. Those beans which subclass java.awt.Component will be added to any Composer's Object Palette.
- ***Text File*** This option will import a text file.
- ***Data File*** This option will import a binary data file.
- ***HTML File*** This option will import an HTML file.
- ***Image File*** This option will import an Image file.
- ***Sound File*** This option will import a Sound file.

Project Menu

The Project menu has options for operations affecting an entire project

- ***Compile project*** Compile all Java source files in the Project directory.
- ***Compile current group*** Compile all Java source files in the currently selected group.
- ***New command window...*** Opens a new Java Command window (see page 127).

Help Menu

The Help menu has the following options.

- ***User Guide...*** This option will show the user a hyper-linked table of contents of the Simplicity for Java User Guide (This book!).
- ***Tutorials...*** This option will show the user the table of contents for the Simplicity for Java Tutorials.
- ***About Simplicity...*** This option will show the user the Simplicity for Java 'About Box' and version information. Clicking the mouse button anywhere will dismiss the about box.

Program Settings

The program settings dialog has five pages.

- Directories
- External Editors
- Object Palette
- Java Editor
- Printing

Directories

The Directories page contains the following fields.

- ***Simplicity Installation Directory*** This is the directory where Simplicity's core files are stored. You cannot change this information here.
  Example: C:\Program Files\Simplicity
- ***Personal Settings Directory*** This is the directory where the information that Simplicity creates about projects will be stored. The personal settings file and other files that Simplicity needs will also be stored here. You cannot change this information here.
  Example: C:\Program Files\Simplicity\.simplicity
  See page 6 for more details.
- ***Preferred Java Compiler*** If this field is left blank, then Simplicity for Java will attempt to use the Java compiler which is built into your platform's Java Development kit. The user can override this behavior to use a different compiler by specifying a command here. For most purposes it is preferable to leave this field blank.
  Example: C:\jdk1.1.5\bin\java.exe
- ***Preferred JDB Command*** If this field is left blank, then Simplicity for Java will attempt to use the JDB debugger which is built into your platform's Java Development kit. The user can override this behavior to use a different JDB-compatible debugger by specifying a command here. In most cases this field should be left blank, as the debugger is very sensitive to different environments.
- ***Show Opening Dialog*** If checked, a welcome screen will appear when Simplicity is first started, presenting the user with three choices: Open an existing project, create a new project, or view the tutorials.

External Editors

The External Editors page contains the following text fields.

- ***Preferred Web Browser*** This is the full path to the web browser in which the user has chosen to view HTML pages.
  Example: /usr/local/bin/netscape
- ***Preferred Image Editor*** If this field is left blank, then images will be viewed with Simplicity for Java's Image Viewer. Otherwise, the image editor specified here will be launched to view images.
  Example: C:\Program Files\PhotoShop\PhotoShop.exe
- ***Preferred Sound Editor*** If this field is left blank, then sounds will be played with Simplicity for Java's Sound Player. Otherwise, the sound editor specified here will be launched to edit sound files.
  Example: C:\WinNT\System32\Sndrec32.exe
- ***Preferred Java Editor*** If this field is left blank, then Java source files will be edited with Simplicity for Java's Java Editor. Otherwise, the user may specify their favorite editor here.
  Example: /usr/bin/vi
- ***Preferred Text Editor*** If this field is left blank, then text files will be edited with Simplicity for Java's Text Editor. Otherwise, the user may specify their favorite editor here.
  Example: C:\Win95\Notepad.exe

Object Palette

Video display size and quality can vary greatly between computers. The Object Palette Customizer lets the user adjust the size and behavior of the Object Palette so that it will be clear and easy to use without wasting unnecessary space on the user's video display. A small sample palette is provided, so that the user may observe the effect of changes immediately. The following settings may be adjusted.

- ***Spacing*** The icons in the palette will be spaced this many pixels apart.
- ***Shadow*** The icons will cast a shadow this many pixels deep. The depressed icon will cast a shadow half this number.
- ***Enable Flyouts*** If selected, icons will popup a small window containing the text of the icon. This is particularly useful if Pictures Only is selected.
- ***Show palette as*** These three radio buttons allow the user to indicate that icons should be shown using pictures, text, or both.

- ***Size*** This slider lets the user indicate how large icons should be. This is very useful for small displays.
- ***Scaling Algorithm*** This choice field lets the user choose which algorithm to use to scale the icons. While "Smooth Scaling" is often the best choice, the choice which appears most attractive may vary on different platforms.

Java Editor

The Java Editor Customizer lets the user customize how Simplicity for Java's Java Editor should operate. A small sample editor is provided so that the user may observe the effect of changes immediately. The following settings may be adjusted.

- ***Background*** The background color.
- ***Foreground*** The color for class names and variable names.
- ***Keywords*** The color for Java Keywords (e.g. `int`, `for`, `class`, `void`,...).
- ***Numbers*** The color for numbers (e.g. `3`, `3.14`, `3e-5`, `0x0011`,...).
- ***Strings*** The color for text strings (e.g. `"hello"`, `"\""`,...).
- ***Characters*** The color for characters (e.g. `'a'`, `'\\'`, `'\011'`,...).
- ***Operators*** The color for Java operators (e.g. `+, -, =, ==, [],` ...).
- ***Comments*** The color for comments (e.g. `/* comment */`).
- ***Errors*** The color for lexical errors (e.g. `bad@name`).
- ***Enable Auto-tab*** Enabling this option inserts tab characters when the Return/Enter key is pressed so that the new line lines up with the previous line.
- ***Substitute spaces for tabs*** Enabling this option will replace tabs with spaces, the number of which is determined by the ***Tab Size***.
- ***Font*** The display font.
- ***Tab Size*** The number of characters in each tabstop. The width of an 'n' character of the chosen font is used to compute the tab stop width.

Printing

This settings page complements the ***Java Editor*** page by providing settings specific to printing Java Source code to a printer. A preview is provided to serve as a guide to the appearance of the printed page.

- ***Margins*** These four fields specify the top, bottom, left, and right margins on the printed page.

- ***Tab size*** This field specifies the size, in inches, of tabs. This can have a large effect on the printing of source code.
- ***Wrap long lines*** If selected, lines which extend beyond the right margin will be wrapped to the next line. This option may make printed output appear less attractive, though it ensures that all text is printed.
- ***Print page headers*** Select this option to print a header on the top of each page. The header will include the filename and the page number. The first page will also include the current date.
- ***Printer font*** The font to be used for printing

CHAPTER 8

Java Source Code Editor

This chapter will discuss Simplicity's Java Source Code Editor. The Editor is designed to provide many powerful tools for working writing Java code, while being easy to learn and use.

This chapter will cover:

- Basic Editing
- Printing
- Search and Replace
- Using the Sourcerer's Apprentice
- Configuration

Editing

The general design of the Java Source Editor is intended to be familiar for users. General text editing, cursor movement with the keyboard and mouse, as well as selecting text using the mouse all operate in the standard fashion as most other text editors.

The Java Source Editor can be used as a standalone editor within the Simplicity IDE. It is also used within the Composer anywhere that code areas appear. The standalone editor has a menubar at the top for accessing its features, as well as a pop-up menu containing most features. When the editor is used in a Composer, only the popup-menu is available. In addition most menu items have keyboard shortcuts available.

File Menu

- ***New*** clears any text in the editor in order to start a new document. The user will be prompted to save any unsaved changes.
- ***Open...*** displays a dialog for the user to select a new file to edit. The user will be prompted to save any unsaved changes.
- ***Save*** saves the current document.
- ***Save As...*** saves the current document to a new file name.
- ***Print...*** lets the user print the current document. If the printer supports color, the color-syntax display will be printed in color. This item first displays an operating system-specific print dialog, allowing the user to customize the print job.
- ***Exit*** closes the editor. The user will be prompted to save any unsaved changes.

Edit Menu

- ***Undo*** undoes the last operation. This can be used as many times as the user wishes to undo changes. All operations in the editor can be undone.
- ***Redo*** is the opposite of Undo. Once an edit has been undone, it can be redone until another edit is performed.
- ***Cut*** removes the selected text from the editor and places it on the system clipboard.
- ***Copy*** copies the selected text to the system clipboard.
- ***Paste*** pastes any text in the system clipboard into the editor at the cursor.
- ***Delete*** deletes the selected text.
- ***Select All*** selects all text in the editor.
- ***Sourcerer's Apprentice...*** displays the methods and fields in a particular class. See Sourcerer's Apprentice on page 73.
- ***Goto...*** displays a dialog letting the user jump to a specific line in the code.

- ***Search & Replace...*** displays a dialog letting the user search for some text and optionally replace it with new text. See Search & Replace.

To facilitate fast editing, the following keyboard shortcuts are available:

TABLE 1.

Feature	Keyboard Sequence	Alternate Sequence
Cut	Ctrl-X	Shift-Delete
Copy	Ctrl-C	Ctrl-Insert
Paste	Ctrl-V	Shift-Insert
Delete	Delete	
Select All	Ctrl-A	
Undo	Ctrl-U	Ctrl-LeftArrow
Redo	Ctrl-R	Ctrl-RightArrow
Goto...	Ctrl-G	
Search & Replace...	Ctrl-F	Ctrl-S
Sourcerer's Apprentice	Ctrl-Space	F1

Holding down the Shift key while using the arrow keys can also be used to select text using the keyboard.

Indentation Features

The Java Source Editor can optionally auto-indent when the user presses the Enter key. This saves time by automatically adding the same number of tabs at the beginning of a new line as the previous line.

The user can also change the indentation level of a block of code by selecting the code and then pressing tab. The entire block will be indented. Shift-Tab will remove an indentation tab from the block.

The tab size and the auto-indent feature can be configured in the IDE's program settings.

Color and Printing Features

The Java Source Editor can identify various Java language elements by displaying each in a different color. The language elements that it can identify are: keywords, number, strings, characters, operators and comments. Each of these may be assigned an individual color in the Program Settings. Additionally, the background and foreground colors can be specified, and a screen font can be chosen.

When printing a Java source file, all of the color settings (with the exception of background) will be used if the printer supports color printing. The user can also specify page margins and a printer font in the Program Settings' Printing page. The user can also indicate whether long lines should be wrapped (they will be wrapped at word boundaries), and whether a header should be printed at the top of each page.

To print a Java source file, select the print command on the File menu of the Java Source Editor. An operating system-specific dialog will appear displaying any options for the available printer(s). After the user confirms this dialog, the document will be printed.

Search & Replace

The Java Source Editor's 'Search & Replace' dialog lets the user find any occurrences of a text string and optionally replace them with a new text string. A single 'Search & Replace' dialog is shared between all Java Source Editors, making it easy to perform the same search on multiple files. The dialog will always perform the search in the Editor which currently has the focus.

The search can be performed in three modes:

- ***Ignore case*** finds any matches with the same letters, but ignoring the case of the letters. For example, 'java' would find occurrences of 'java', 'JAVA', and 'jAvA'.
- ***Match case*** finds any matches with the same letters. For example, 'java' would find only occurrences of 'java', and not 'JAVA' or 'jAvA'.
- ***Match Perl 5 regular expressions*** finds matches based on the regular expressions syntax of the Perl 5 language. For example, ' [a-c]* ' matches any word starting with 'a', 'b', or 'c'.

The following buttons are available for searching:

- ***Find Next*** finds the next occurrence of the string in the 'Find what:' text field. The search is always performed starting at the cursor location. If a match is found, it will be highlighted in the editor, otherwise a beep sound will indicate no match.
- ***Replace*** will replace the highlighted text in the editor with the text in the 'Replace with:' text field. It will then automatically perform a find operation.
- ***Replace All*** will replace all occurrences of the search string with the replacement string.
- ***Cancel*** will dismiss the Search dialog.

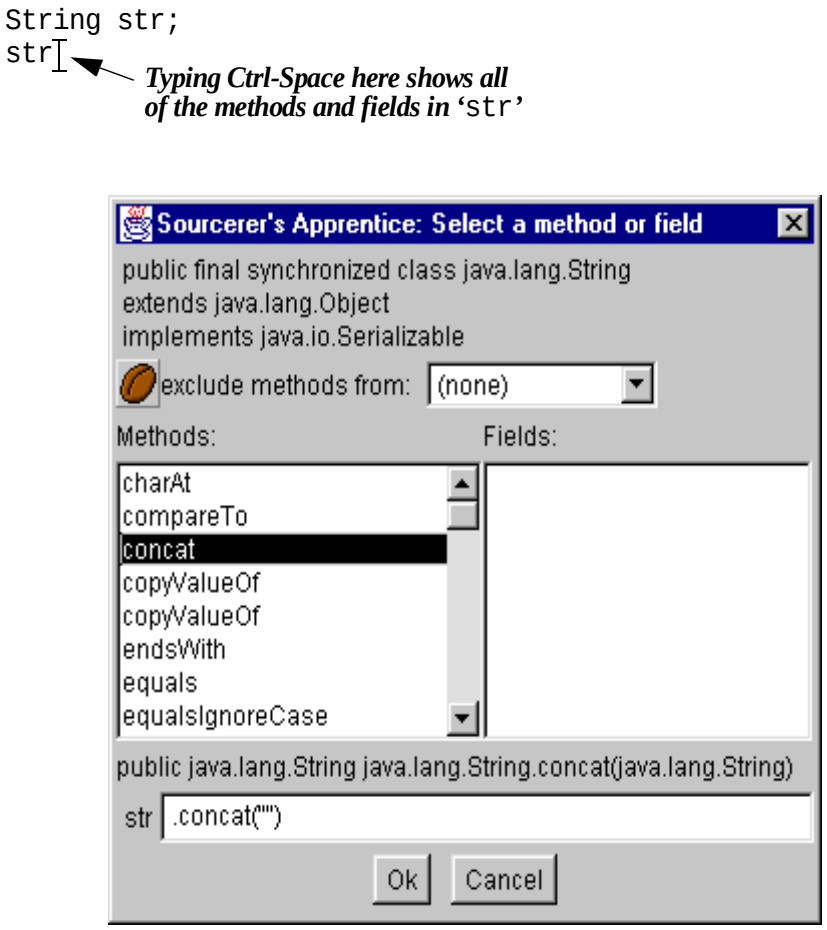
All replacement operations can be undone in the editor.

The Sourcerer's Apprentice

The Sourcerer's Apprentice lets the user quickly view and select from the available methods and fields in a class. This feature is invoked from the menubar or popup-menu, or by pressing Ctrl-Space or F1. When this is done, the Sourcerer's Apprentice takes the Java identifier to the left of the cursor and determines its class. It then displays a dialog showing all the methods and fields in that class. The user can select a method or field which will then be displayed at the bottom of the dialog. If a method is selected, default arguments will be provided. The user can edit this code if they wish. When the OK button is pressed, this code will be inserted in the editor at the cursor's position. The Cancel button dismisses the dialog without making any changes to the code.

For example, if a section of code has a String object named 'str', typing str and then pressing Ctrl-Space will launch the Sourcerer's Apprentice. See Figure 2.

Figure 2. Example of Sourcerer's Apprentice



If the Sourcerer's Apprentice is invoked next to something which is not an instance of class, it will beep, but not display a dialog box.

CHAPTER 9

Composers

This chapter will discuss the Simplicity Composer. The Composer lets the user specify properties of any parts, as well as enter user code.

This chapter will cover

- Choosing and creating composers
- The Composer window, button bar, and menu bar
- Property notebooks

Creating a New Composer

A new composer file can be created by choosing one of the first seven items on the ***Create*** menu in the Simplicity IDE. These items are also on the Button Bar in the IDE.

Types of Composers

The seven composer types are

1. ***Main Application***

2. ***Frame***
3. ***Window***
4. ***Dialog***
5. ***Panel***
6. ***Object***
7. ***Canvas***

Each type of composer allows the user to create a different type of compound object. A ***Main Application*** Composer is used to create either a Java Applet (usually to be embedded in a web page) or the primary frame of a Java Application. The ***Frame***, ***Window***, and ***Dialog*** Composers are used to create secondary windows which can be launched from a Main Application. The ***Panel*** Composer is used to create compound parts which can be reused in other composers. The ***Object*** Composer is used to create non-graphical parts. The ***Canvas*** Composer is significantly different from the other composers and will be described in its own chapter.

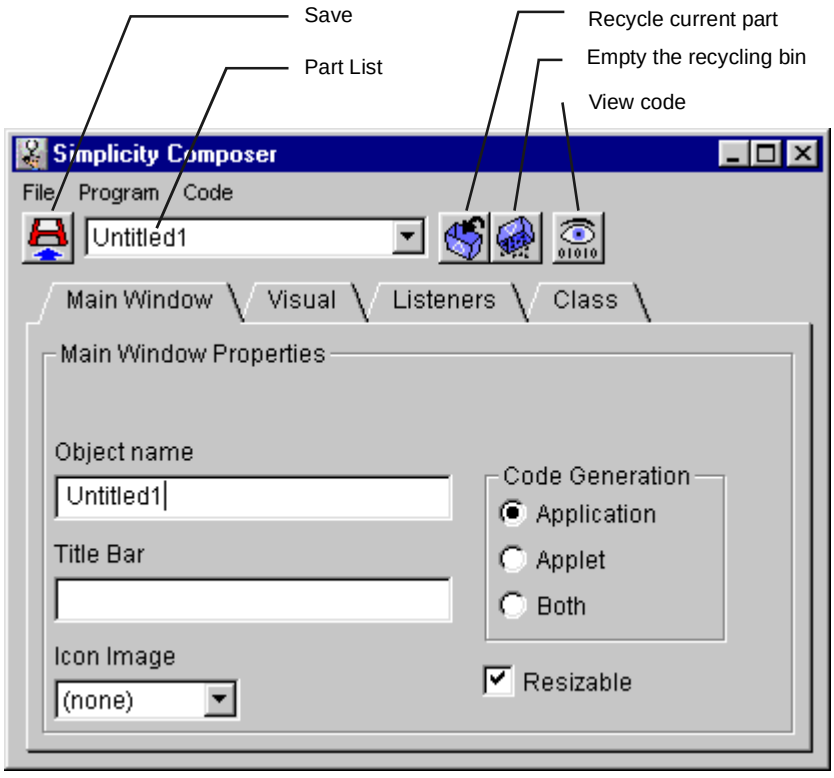
Each composer will generate a single Java source file corresponding to a public class as well as any supporting inner classes.

When a Composer is opened from within the IDE, three windows appear (except in the case of an Object Composer). These three windows are the Composer window, the Object Palette, and the Working Model window.

The Composer Window

The Composer window is a dynamic, context-sensitive property editor in which the user can assign default properties and behaviors for all parts in a program. All user supplied code that a program requires is entered into the composer, including event response code, class declaration and constructor code, and method code which can be called from within the class or from external code which will use the class.

The Composer window contains a menu bar, a button bar, and a context sensitive set of tabbed pages.



Composer Button Bar

The Composer button bar contains the following items:

- ***Save*** This button is identical to the Save option on the file menu. It saves the current state of the composer to disk.
- ***Part List*** This choice field contains a list of all of the parts being used in the current program. The user can view the properties of any part by selecting it from the Part List.
- ***Recycle Current Part*** Pressing this button removes the part currently displayed in the Part List from the working model and stores it in the Recycled page in the Object Palette. If the part is a layout, all of its contents are moved with it to the Recycled page. The part is still active while in the Recycled page. It can be modified from its properties page as well as referenced by event code.

- ***Empty The Recycling Bin*** Pressing this button deletes all parts from the Recycled parts page. The parts are no longer accessible from event code. If a part in the Recycled parts page is a layout, any parts contained within it are deleted as well.
- ***View Code*** Pressing this button opens the Code Viewer window, which shows the current generated code. If the window is already visible, pressing this button will update the code and bring the window to the front.

File Menu

The File menu has the following options.

- ***Save*** This item saves the current state of the composer to disk, as well as the generated Java code.
- ***Generate code*** This item saves the current generated Java code, but does not save the current state of the composer.
- ***View code*** This item is equivalent to pressing ***View Code*** in the button bar.
- ***Exit*** This item exits the Composer. It will query the user to save any changes first. Upon exiting, a Java source file will be created containing all of the generated code.

Program Menu

The Program menu has the following options. Depending on the type of Composer, only a subset of these may be available.

- ***Command window...*** This item opens a new Java Command Window (see page 130).
- ***Initialize Class*** This item restores all parts to the current settings in the Composer window. Any code in the Code Declaration and Constructor pages is also executed.
- ***Execute init()*** This item will execute any code in the init() method. It should be used to simulate a web browser or applet viewer. When an applet is first loaded, the browser will execute the init() method.
- ***Execute start()*** This item will execute any code in the start() method. It should be used to simulate a web browser or applet viewer. A browser will execute the start() method when the applet is started.

- ***Execute stop()*** This item will execute any code in the stop() method. It should be used to simulate a web browser or applet viewer. A browser will execute the stop() method when the applet is stopped.
- ***Execute destroy()*** This item will execute any code in the destroy() method. It should be used to simulate a web browser or applet viewer. When an applet is removed from memory, the browser will execute the destroy() method.
- ***Execute finalize()*** This item will execute any code in the finalize() method. It should be used to simulate the Java Virtual Machine, when it runs the garbage collector on the class.

Code Menu

The Code menu has the following options.

- ***Goto declaration code*** This item will bring the declaration code area into view in the Composer window. The user should use this code area to declare any instance variables as well as static variables.
- ***Goto constructor code*** This item will bring the constructor code area into view in the Composer window. The user should use this code area for any code that should be added to the class constructor.
- ***Goto method code*** This item will bring the method code area into view in the Composer window. The user should use this code area for any instance methods or static methods that should be included in the class.
- ***Goto init() code***
- ***Goto start() code***
- ***Goto stop() code***
- ***Goto destroy() code***
- ***Goto finalize() code*** These methods will bring the chosen method into view in the Composer window.

Parts Menu

The Parts menu has the following options.

- ***Recycle current part*** This places the part currently displayed in the Part List from the working model and stores it in the Recycled page in the Object Palette, just as if the button with the same name had been pressed.
- ***Empty Recycling bin*** This deletes all parts from the Recycled parts page, just as if the button with the same name had been pressed.

- ***Hide all Empty Panels*** This item has the same effect as unchecking the "Show empty panel" checkbox in all the layouts in the partlist.
- ***Enable Events*** When this item is unchecked, parts in the working model will not respond to events. This will allow you to temporarily suspend all code execution, to allow easier editing. Be sure to re-enable this checkbox to continue testing your code.

Property Notebooks

Notebook pages

Each time a part is added to the Working Model, its Properties Notebook is displayed in the Composer window. At any future time, a part's Properties Notebook can be viewed either by selecting the part's Object Name from the Part List or by clicking the part with the mouse.

Every part in the Composer has a minimum of three pages in its properties notebook. These three pages are

1. The part-specific properties page
2. The ***Visual*** properties page
3. The ***Listeners*** page

All properties specified in the properties notebook are the initial properties that a part will have. Any properties may be changed in the working model by either event code or by user interaction with the working model. The user may restore all parts to the state specified in the properties pages by choosing the ***Initialize Class*** menu item on the ***Program*** menu.

The first property page for any part contains the name which will identify the declared object and the part-specific properties for that object. For example, the first page for a Checkbox would contain the Object Name plus the checkbox label and the initial checkbox state.

The first property page may also contain controls which have no effect on the operation of the final program, but will assist the user in using the part while in the Composer. For example, the Flow Layout part has a checkbox to indicate whether the Empty Space should appear. The user should check this checkbox to add

additional parts to the Flow Layout, and uncheck it when finished. Further, the user can specify the position of the Empty Panel using the Position choice field.

The second property page is the ***Visual*** properties page. This page is identical for every part. It contains the visual properties that are common to all parts. Initially, these properties are set to 'default' which means that they should inherit the properties of the layout in which they are sitting. By changing one of these properties, the user overrides this default setting. If a visual property is changed for a layout, that property cascades through all of the parts within that layout as long as that property is set to 'default' for each of those parts.

Visual properties include

- ***Font*** This specifies the font to use when displaying the part. When the font name is set to default, the font size and style will be disabled.
- ***Color*** This specifies the foreground and background colors to be used when displaying the part.
- ***Cursor*** This specifies the mouse cursor to be displayed when the mouse is placed over the part.
- ***Visible*** This specifies whether the part is initially visible.
- ***Enabled*** This specifies whether the part is initially enabled.

Parts from the Swing library also include

- ***Tool Tip*** This is the text that should be displayed when the user holds the mouse over a component without activity for a period of time.
- ***Double Buffer*** This specifies whether graphics should be drawn using an off-screen buffer, rather than directly on the screen. It uses more memory, but creates much smoother scrolling. Any part which is doubled buffered will also double buffer its children.
- ***Auto-scroll*** This option lets a part do internal scrolling based on mouse and cursor movement. It only has meaning for parts which support it, such as JTextField.

The third properties page is the ***Listeners*** page. This page contains a set of checkboxes corresponding to the kinds of events that a part is capable of 'Listening' for. The events which are most often used are presented in bold.

When a listener is selected, a new page will appear in that part's properties notebook. This new page will contain a code area where the user can specify code

which should be executed in response to events of that Listener type. Many types of events have several kinds of events defined within them. Each of these events are listed in the choice field at the top of the event page. The code which is displayed in the code area corresponds to the event listed in the choice field.

The user can enter any legal Java statements in the code area. Simplicity watches for any changes made and updates the working model so that the working model always reflects the current state of the code. When the user generates an event in the Working Model, the event code will be executed. If there is an error in the code, a message will be printed to the console.

At the top of every event page is a ***Code Sourcerer*** button. The Code Sourcerer will write Java statements for the user based upon the user's choices.

All parts listen for at least five types of events.

1. ***Component Events*** These events are generated when a part is shown, hidden, resized, or moved.
2. ***Focus Events*** These events are generated when a part gains or loses the focus.
3. ***Key Events*** These events are generated when the user presses or releases a key.
4. ***Mouse Events*** These events are generated when the mouse enters or exits a part, or when a mouse button is pressed or released over a part.
5. ***Mouse Motion Events*** These events are generated when the mouse is moved or dragged over a part.

All of the Layout parts also listen for a sixth type of event.

6. ***Container Events*** These events are generated when components are added or removed from the layout.

Some parts generate additional events which are specific to their functionality. For example, a Button generates an Action Event whenever the user presses the Button.

The properties notebooks for the top-level containers (Main Application, Frame, Window, Dialog, Panel, Object) also have a fourth page titled ***Class***. This page contains code areas for user code. These code areas include

- ***declaration code*** This code area should be used to declare any static variables or instance variables that a program may need. Any Java statement which is legal to exist within the outermost scope of a Java class can be put here.

- ***constructor code*** This code area should be used for any additional initialization code that a class may need. Any legal Java statements may be placed here. This code will be appended to the class constructor.
- ***method code*** This code area should be used to declare any static methods or instance methods that a program may need.
- ***finalize() code*** This code area should be used for any de-construction code. It will be executed asynchronously by the Java garbage collector when the class is no longer in use.

A Main Application Composer will also have code areas for the four special methods that a web browser uses to control an Applet.

Each of these code areas can be also be reached using the ***Code*** menu in the Composer. They can be tested by being used in event code or by the commands on the ***Program*** menu in the Composer.

Code Generation

Each of the five graphical Composers (Main, Frame, Dialog, Window, Panel) generate code based on the AWT architecture by default. They each have a checkbox on their properties page labeled ‘Generate Swing code’ which changes the generated code to the JFC/Swing architecture.

Specifically, selecting the ‘Generate Swing code’ checkbox converts all Frame, Dialog, Window, and Panel classes to JFrame, JDialog, and JPanel classes. All of the Menu-related classes are also generated to correspond to their appropriate parent Frame or JFrame.

The three Composers which include title bars (Main, Frame, and Dialog) have an additional option called ‘default close action’. This choice, available only when Swing code generation is enabled, is a shorthand to specify what should happen when the user attempts to close a Frame or Dialog. The most common actions (hide or dispose) can be chosen. Using these options can replace the need to listen for window events in many cases.

The Main Application Composer has an addition set of three radio buttons. These buttons indicate what kind of Java program should be created. Choose ‘Application’ to create a stand-alone Java application. Choose ‘Applet’ to create a

Java program that can be embedded in a web page. Choose ‘Both’ to create a Java program that can run stand-alone or be embedded in a web page.

CHAPTER 10

Object Palette

This chapter will discuss the Object Palette and how to use it to assemble Graphical User Interfaces.

It will cover

- Assembling layouts and parts
- Each of the layouts
- Each of the basic AWT parts
- Each of Simplicity's extended parts
- Each of the Swing parts
- Each of the Menu parts
- Importing JavaBeans

Assembling A Program Using The Object Palette

When a Composer is opened from the IDE, two additional windows are created: the Object Palette and the Working Model.

Object Palette

The Object Palette contains all of the graphical parts which are available to be used in a project. It is organized into a set of pages containing related parts. When the user clicks the mouse button over one of the icons in the palette, that icon will appear depressed to indicate that it is selected.

Working Model

The Working Model is a live prototype which is built interactively. Because the model is built using the actual Java controls and layouts, the model reflects exactly how the application will function and behave. All of the controls are active and will respond to user interaction. The layouts will show the true part placement.

The Working Model initially is a window containing a box labelled Empty. Whenever the user clicks the mouse in an Empty box, the part which is selected in the Object Palette will be placed in that space.

Object Palette Pages

The pages in the Object Palette are

- ***Layouts*** These parts each contain a series of empty spaces in which the user can place an additional part.
- ***Basic*** These parts are the graphical components which make up the Abstract Windowing Toolkit (AWT). They are part of the standard Java libraries and can be assumed to be present on any Java-enabled machine.
- ***Extended*** These parts are additional parts which come with Simplicity for Java to provide extended functionality to the AWT.
- ***Swing 1 & 2*** These parts are the graphical components from the JFC/Swing library. They are included with JDK 1.2. They can also be used with JDK 1.1 if the swingall.jar file is included in the classpath.
- ***Menus*** These parts are used to build menus in your applications. They represent both the AWT menu classes, as well the Swing menu classes. If the ‘Generate Swing code’ checkbox is checked, then the generated code for these parts will be based on the Swing library. Otherwise, the AWT menu classes will be used.
- ***Beans*** This page is initially blank. Java Beans can be imported in the IDE. If a bean is a subclass of Component, it will be added to the Beans page. Any Panel Composers, once compiled, will also be available here.

- ***Recycled*** This page is the 'Recycling Bin' for parts. When the Recycle button in the Composer is pressed, the part which is showing in the part list will be removed from the working model and moved to the Recycled page in the Palette. It can then be placed back into any Empty Space in the Working Model.

The Object Palette can exist in its own window or it can be placed at the bottom of the Composer window. The user can switch between these two positions by closing the window or by pressing the ***Detach*** button, respectively.

Layout Parts

Border Layout

This Layout can hold up to five parts, arranged in the North, South, East, West, and Center positions. The center part expands to fill any excess room. The properties page has controls to specify the horizontal and vertical gaps.

A Border Layout holds five Empty Spaces. When the user is finished adding parts, the remaining Empty Spaces can be removed by unchecking the 'Show empty panels' checkbox.

Flow Layout

This layout places parts in a centered, horizontal row, wrapping if necessary. The properties page has controls to specify the alignment (left, right, or centered) and the horizontal and vertical gaps.

A Flow Layout starts out with a single Empty Space. Each time the user clicks a parts into the layout, the Empty Space advances to the right. The user can specify where the Empty Space should appear using the Position choice field. When the user is finished adding parts, the Empty Space can be removed by unchecking the 'Show empty panel' checkbox.

Grid Layout

This Layout holds parts arranged in a grid of user specified dimensions. The properties page has controls to specify the number of rows and columns in the grid as well as the horizontal and vertical gaps.

A Grid Layout holds (rows*columns) Empty Spaces. When the user is finished adding parts, the remaining Empty Spaces can be removed by unchecking the 'Show empty panels' checkbox. When the grid dimensions are enlarged, Empty Spaces are added to the bottom and right. When the grid dimensions are diminished, Empty Spaces are removed from the bottom and right. If parts are sitting in these spaces, those parts are moved to the 'Recycling Bin'.

GridBag Layout

The GridBag Layout is the most flexible and versatile layout among the AWT Layout Managers. Once a set of parts have been added, the user can specify a set of constraints to be applied to each part to indicate how the parts should be laid out with respect to each other.

A GridBag Layout starts out with a single Empty Space. The user can add or remove Empty Spaces, as well as change the order of parts using the buttons at the top of the properties page. The rest of the properties page is a table which shows the constraints for each part that is placed in the layout. Any of these constraints can be edited in this table.

The GridBag Constraints are

- ***gridx, gridy*** The part's position in the grid.
- ***width, height*** The number of rows and columns the part should occupy.
- ***anchor*** A part can be anchored to any side of its cell. The user can choose a value by clicking the mouse in the grid in the appropriate table cell.
- ***fill*** How the part should expand to fill extra space. The possible values are none, horizontal, vertical, and both. The user can cycle through these values by clicking the mouse on the appropriate table cell.
- ***xweight, yweight*** Specifies how extra space should be distributed among parts.
- ***x internal..., y internal...*** Internal padding of the part.
- ***top inset, left inset, bottom inset, right inset*** External padding of the part.

Card Layout

The Card Layout can hold any number of parts, but only displays one of them at a time. Each part is given a name. The visible part can be chosen programmatically using the name or by advancing through the parts in order.

The properties page for a Card Layout has a list of the available cards. Clicking any item in the list will make that card visible. The name of the visible card appears in the ‘Current Card’ text field and can be changed. New cards can be added by pressing the ‘Add a new card’ button. A card can be removed by choosing it from the list and pressing the ‘Recycle’ button. The order of the cards can be changed by choosing a card and pressing the ‘Up’ or ‘Down’ buttons. The Card Layout’s internal borders can also be set using the Horizontal and Vertical Gaps entry fields.

Tabbed Card Layout

The Tabbed Card Layout can hold any number of parts, but only displays one of them at a time. Each part is given a name. The visible part can be chosen programmatically by name or by user interaction with the name tabs.

The properties page for a Tabbed Card Layout has a list of the available cards. Clicking any item in the list will make that card visible. The name of the visible card appears in the ‘Current Card’ text field and can be changed. New cards can be added by pressing the ‘Add a new card’ button. A card can be removed by choosing it from the list and pressing the ‘Recycle’ button. The order of the cards can be changed by choosing a card and pressing the ‘Up’ or ‘Down’ buttons. The Tabbed Card Layout’s internal borders can also be set using the Horizontal and Vertical Gaps entry fields.

A Tabbed Card Layout will generate an ItemEvent whenever a new page is selected.

Left Side Layout

This layout places parts in a left justified vertical column. The properties page has controls to specify the alignment (top, bottom, or centered), the vertical gaps, and whether the components should fill horizontally any extra space.

A Left Side Layout starts out with a single Empty Space. Each time the user clicks a parts into the layout, the Empty Space advances downward. The user can specify where the Empty Space should appear using the Position choice field. When the user is finished adding parts, the Empty Space can be removed by unchecking the ‘Show empty panel’ checkbox.

Bottom Layout

This layout places parts in a bottom justified, horizontal row. The properties page has controls to specify the alignment (left, right, or centered), the horizontal gaps, and whether the components should fill vertically any extra space.

A Bottom Layout starts out with a single Empty Space. Each time the user clicks a parts into the layout, the Empty Space advances to the right. The user can specify where the Empty Space should appear using the Position choice field. When the user is finished adding parts, the Empty Space can be removed by unchecking the 'Show empty panel' checkbox.

ScrollPane Layout

This layout simplifies the use of scrollbars. It holds a single part which can be scrolled programmatically, or by user interaction with the scrollbars. The scrollbars can be chosen to be always visible, visible only if the contained part is larger than the display size, or never visible. The display size can also be fixed to a given size.

Absolute Layout

This layout allows the user to specify the exact location and size of each component in the layout. The layout can be given a preferred size. Additionally, if the layout is resized the parts can optionally be scaled proportionally to fit the new size. Should two parts overlap, the Z-order will determine which part will be on top. (Note: any 'heavyweight' part, such as most AWT components, will always appear on top of a lightweight part, such as a Swing component.)

The mouse editing option allows the user to adjust the size and position of components in the Absolute Layout.by dragging on the components within the layout. To move a component, drag on the edge of it. To adjust the size of a component, drag the rectangles displayed in the corners and sides of the component's frame. If the preferred size is being used, then the user can move the components, but not resize them.

An absolute Layout starts out with no Empty Spaces. The user can add or remove Empty Spaces, as well as change the Z-order of the parts using the buttons at the left of the properties page. The user can also set the preferred size and scaling using the controls at the left. The rest of the properties page is a table which shows the location and size for each part that is placed in the layout. Any of these constraints can be edited in this table.

Basic Parts

Label

A Label can hold a single line of text. The alignment of the text can be specified as left, right, or centered. The alignment is particularly useful when the Label is sized larger that the text.

Checkbox

A Checkbox holds a single line of text and a graphical On/Off indicator. The text and the initial state of the Checkbox can be specified in its properties page.

A Checkbox will generate an ItemEvent whenever its state is changed.

Radio Button

A Radio Button holds a single line of text next to a graphical On/Off indicator. Each Radio Button is also assigned a group. Only one Radio Button in each group can be in the On state at a time.

The text and the initial state of the Radio Button can be specified in its properties page. Pressing the 'Group Editor' Button will launch a dialog displaying the current group and allowing the user to add, edit, or remove groups. All Radio Buttons are initially set to a group named 'defaultGroup'. Once the user has created other groups in the Group Editor, these groups can be chosen from the Group choice field.

All changes in the Group Editor effect all of the Radio Buttons in that Composer. Once a group is created it is available to all Radio Buttons.

A Radio Button will generate an ItemEvent whenever its state is changed.

Button

A Button allows the user to indicate when an action should be performed. It can hold a single line of text, which can be specified in the properties page.

A Button will generate an ActionEvent when the Button is pressed.

Text Field

A Text Field is a single line entry field for text input. In its properties page, the user can specify the default text to appear and the number of columns wide the field should be. The Text Field can be set read-only by disabling the 'Editable' checkbox. The user can also choose an echo character if the Text Field will contain sensitive information such as a password. When the 'Enable' checkbox is checked, the character to the right of the checkbox will become the echo character.

A Text Field will generate a TextEvent whenever the user modifies the field's text.

A Text Field will generate an ActionEvent whenever the user presses the 'Enter' key while the field has the focus.

Choice

The Choice field lets the user choose one item from a set. The properties page includes an area in which the user may enter the items to appear in the Choice.

A Choice field will generate an ItemEvent whenever the user chooses an item.

Listbox

The Listbox lets the user choose one item or multiple items from a list. The properties page includes an area in which the user may enter the items to appear in the Listbox. The user can also specify the number of visible lines in the Listbox and whether multiple selections should be allowed

A Listbox will generate an ItemEvent whenever the user chooses an item.

A Listbox will generate an ActionEvent whenever the user presses the 'Enter' key while the Listbox has the focus.

Text Area

The Text Area is a multi-line editable text component. In its properties page, the user can specify the default text to appear and the number of rows high and columns wide the Text Area should be. The Text Area can be set read-only by disabling the 'Editable' checkbox. The user can also specify which scrollbars should appear.

A Text Area will generate a TextEvent whenever the user modifies the text.

Scrollbar

A Scrollbar allows the user to select from a range of values. In its properties page, the user can specify the scrollbar minimum and maximum values, the visible portion of the scrolled object, and the initial scrollbar value. The user can also choose whether the scrollbar should be oriented horizontally or vertically.

A Scrollbar will generate an AdjustmentEvent whenever the user moves the control.

Extended Parts

Spacer

The Spacer is an empty part used to adjust the spacing between parts in a layout. The user can specify its width and height.

Inset Sizer

The Inset Sizer is a container which can hold one part. It allows the user to specify the height and width of the part that it holds, as well as the insets (gaps) around that part. This is very useful for overriding the default sizing of parts by a layout.

For example, a Button's size is usually based upon the text inside the Button. If the user wants to create a large Button whose size is always 150 by 150 pixels, the Button can be placed inside an Inset Sizer with the dimension set to 150 by 150. If the user want a gap around the Button, it can be entered as a set of Insets.

The Inset Sizer will generate a ContainerEvent whenever a part is added or removed.

Validated Text Field

A Validated Text Field has all of the features of a Text Field plus the ability to ensure that the user can only enter text which matches a particular data type. Its

properties page includes a choice field that lets the user choose the validation type. The validation types are

- ***Null*** no validation
- ***Byte*** integers from -128 to 127
- ***Short*** integers from -32768 to 32767
- ***Integer*** integers from -2147483648 to 2147483647
- ***Long*** integers from -9223372036854775808 to 9223372036854775807
- ***Character*** a single character
- ***Float*** floating point from 1.4e-45 to 3.4e+38
- ***Double*** floating point from 4.9e-324 to 1.8e+308
- ***String*** any string, no validation
- ***Boolean*** true or false
- ***Phone*** (000)000-0000
- ***Social Security*** 000-00-0000
- ***Zip*** 00000
- ***Zip+4*** 00000-0000
- ***Dollar Amount*** $0.00
- ***Time*** 00:00am
- ***Date*** 00/00/00
- ***Date2000*** 00/00/0000
- ***Identifier*** an identifier (variable name) according to the rules of Java

A Validated Text Field will generate a TextEvent whenever the user modifies the field's text.

A Validated Text Field will generate an ActionEvent whenever the user presses the 'Enter' key while the field has the focus.

Wrap Label

A Wrap Label is a multi-line label. Its preferred size is to fit the longest line of text. If its parent layout should size it smaller, the text will wrap to fit as much as possible in the allocated space.

Image Button

An Image Button is similar to a Button, except it holds an image rather than a text string. Images are loaded through the Simplicity IDE's ***Import*** menu. The properties page for an Image Button contains a choice field listing all the images that are loaded into the IDE. The user can choose among the images using this choice field. The user can also specify a text string which will be appear in a tool-tip flyout window. The tool-tip only works on platforms which are capable of displaying a Window (without a Frame).

An Image Button will generate an ActionEvent when pressed.

Image Canvas

An Image Canvas is a part which displays an image. It attempts to size itself to the size of the image. Images are loaded through the Simplicity IDE's ***Import*** menu. The properties page for an Image Canvas contains a choice field listing all the images that are loaded into the IDE. The user can choose among the images using this choice field.

Group Box

A Group Box is a container which can hold one part, often a layout. It draws a border around its contents to indicate a related set of parts. The user can specify a text string to appear at the top of the box.

The Group Box will generate a ContainerEvent whenever a part is added or removed.

Progress Bar

A progress bar graphically displays a percentage. Often it is used to show the percent of a process that is complete. The user can specify the minimum, maximum, and current values.

Flyer

A Flyer is a container which can perform some simple marque-style animation. Any part or set of parts in a layout can be placed in a Flyer. The user can indicate in the properties page the number of pixels per second in the vertical and horizontal

directions that the part should move. The user can also indicate the frame rate, as well as the size of the bounding box.

The horizontal and vertical bounce checkboxes indicate whether the part should reverse directions when hitting the side of the bounding box or wrap to the other side. The horizontal and vertical runoff checkboxes indicate whether the part should be allowed to completely move out of the bounding box or only touch the side. Start and Stop buttons are provided so that the user may easily test the animation. A Flyer should be started and stopped programmatically, though, using its start() and stop() methods.

Frame Animator

A Frame Animator takes a series of images and displays them sequentially at a specified frame rate. Start and Stop buttons are provided so that the user may easily test the animation. A Frame Animator should be started and stopped programmatically, though, using its start() and stop() methods.

Swing 1 & 2

The Swing pages contain the components from the Swing graphical user interface library.

Button (JButton)

A JButton allows the user to indicate when an action should be performed. It can hold a single line of text, whose alignment and position can be specified relative to an icon. Several different icons can be specified to indicate the various states of the JButton. A keyboard mnemonic can also be specified.

A JButton will generate an ActionEvent when the JButton is pressed.

A JButton will generate a ChangeEvent whenever the button changes state (i.e. is pressed, is released, etc...)

Toggle (JToggleButton)

A JToggleButton allows the user to indicate an on/off condition. A set of ToggleButtons can be grouped together, such that only one can be selected at a

time. The button can hold a single line of text, whose alignment and position can be specified relative to an icon. Several different icons can be specified to indicate the various states of the JToggleButton. A keyboard mnemonic and the initial state of the JToggleButton can also be specified.

A JToggleButton will generate an ActionEvent when the JToggleButton is pressed.

A JToggleButton will generate a ChangeEvent whenever the button changes state (i.e. is pressed, is released, etc...)

A JToggleButton will generate an ItemEvent when the JToggleButton is selected or deselected.

CheckBox (JCheckBox)

A JCheckBox allows the user to indicate an on/off condition. A set of checkboxes can be grouped together, such that only one can be selected at a time. The checkbox can hold a single line of text, whose alignment and position can be specified relative to an icon. Several different icons can be specified to indicate the various states of the JCheckBox. A keyboard mnemonic and the initial state of the JCheckBox can also be specified.

A JCheckBox will generate an ActionEvent when the JCheckBox is pressed.

A JCheckBox will generate a ChangeEvent whenever the JCheckBox changes state (i.e. selected, rolled over, disabled, etc...)

A JCheckBox will generate an ItemEvent when the JCheckBox is selected or deselected.

RadioButton (JRadioButton)

A JRadioButton allows the user to indicate an on/off condition. A set of radio buttons can be grouped together, such that only one can be selected at a time. The button can hold a single line of text, whose alignment and position can be specified relative to an icon. Several different icons can be specified to indicate the various states of the JRadioButton. A keyboard mnemonic and the initial state of the JRadioButton can also be specified.

A JRadioButton will generate an ActionEvent when the JRadioButton is pressed.

A JRadioButton will generate a ChangeEvent whenever the JRadioButton changes state (i.e. selected, rolled over, disabled, etc...)

A JRadioButton will generate an ItemEvent when the JRadioButton is selected or deselected.

Label (JLabel)

A JLabel can hold a single piece of text as well as an icon. An icon and a disabled icon can be specified, as well as the position and alignment of the text relative to the icon. The gap (spacing) between the icon and text can be indicated. A part can be chosen as the 'Labeled part' that this label is labeling. The chosen keyboard mnemonic will set the focus to this labeled part.

ComboBox (JComboBox)

A JComboBox is a combination of a Choice and a TextField. The user can select from a set of pre-defined values or edit the contents directly (if the editable checkbox is selected). The Maximum rows entry field indicates the number of rows that are visible when the ComboBox is opened.

The ComboBox can be used in two ways. The first is by specifying a static set of text entries in the 'JComboBox items' area (each one on a line). The second, and more powerful, is to use a ComboBoxModel. A ComboBoxModel lets the user fully describe the contents of the ComboBox in an abstract manner to create dynamic contents. See the JDK API reference and documentation for further details.

A ComboBox will generate an ItemEvent whenever the user chooses an item.

A ComboBox will generate an ActionEvent whenever the user presses the 'Enter' key while the Listbox has the focus.

Listbox (JList)

A JList shows a list of a set of items. In order to include a scrollbar, a JList must be placed inside of a JScrollPane. The user can select either one or multiple items depending on the 'Selection Mode'. The Visible rows entry field indicates the number of rows that are displayed when a JList is placed inside of a JScrollPane. The 'Prototype Value for Sizing' is a text string whose display width determines the preferred width of the JList.

The JList can be used in two ways. The first is by specifying a static set of text entries in the 'JList items' area (each one on a line). The second, and more powerful, is to use a ListModel. A ListModel lets the user fully describe the contents of the JList in an abstract manner to create dynamic contents. See the JDK API reference and documentation for further details.

A List will generate an ItemEvent whenever the user chooses an item.

A List will generate an ActionEvent whenever the user presses the 'Enter' key while the Listbox has the focus.

Slider (JSlider)

A JSlider lets the user select a numeric value from a range of integer values. It can be drawn either horizontally or vertically. The user can specify the initial value for the slider, the minimum and maximum values, and the major and minor tick values. The user can also indicate whether ticks and/or labels should be drawn under the slider, whether the slider knob will automatically jump to the tick values, and whether the ticks are drawn in ascending or descending order.

A JSlider will generate a ChangeEvent when the Slider's knob is moved. By default, the event will only be fired when the knob's final position is selected. If the 'Fire Events While Adjusting' checkbox is selected, then events will be fired continuously as the knob is moved.

ScrollBar (JScrollBar)

A JScrollBar is a scroll bar which can be drawn either horizontally or vertically. The user can specify the initial value of the JScrollBar, as well as the minimum and maximum values. In most cases, the JScrollPane is the easiest way to add scroll bars to a Swing object.

ProgressBar (JProgressBar)

A JProgressBar displays a value from a range of values. It can be drawn either horizontally or vertically. The user can specify the initial value for the ProgressBar, as well as the minimum and maximum values. Optionally, a text string can be drawn inside the bar, and/or a border can be painted around the bar.

TextField (JTextField)

A JTextField is a single line entry field for text input. In its properties page, the user can specify the default text to appear and the number of columns wide the field should be. The text can be left, right or center justified. The JTextField can be set read-only by disabling the ‘Editable’ checkbox.

A JTextField will generate a CaretEvent whenever the user modifies the field’s text, or moves the cursor via the mouse or keyboard.

A JTextField will generate an ActionEvent whenever the user presses the ‘Enter’ key while the field has the focus.

TextArea (JTextArea)

A JTextArea is a multi-line entry field for text input. It can also be used as a multi-line label. In its properties page, the user can specify the default text to appear and the number of rows tall and columns wide the area should be. The size of tabs can be set. The JTextArea can be set read-only by disabling the ‘Editable’ checkbox. The JTextArea can optionally wrap long lines to the next line. If this line wrap is enabled, a word wrap feature can be selected, specifying that the text should be wrapped at word boundaries, rather than at any arbitrary character.

A JTextArea will generate a CaretEvent whenever the user modifies the field’s text, or moves the cursor via the mouse or keyboard.

PasswordField (JPasswordField)

A JPasswordField is a single line entry field for text input whose characters are disguised by an ‘echo’ character. In its properties page, the user can specify the default text to appear and the number of columns wide the field should be. The text can be left, right or center justified. The JPasswordField can be set read-only by disabling the ‘Editable’ checkbox.

A JPasswordField will generate a CaretEvent whenever the user modifies the field’s text, or moves the cursor via the mouse or keyboard.

A JPasswordField will generate an ActionEvent whenever the user presses the ‘Enter’ key while the field has the focus.

EditorPane (JEditorPane)

A JEditorPane is used to edit formatted content. The user can specify whether this content is plain text, HTML, or Rich Text Format. The user can load a new document via a URL. The JEditorPane can be set read-only by disabling the 'Editable' checkbox.

A JEditorPane will generate a CaretEvent whenever the user modifies the pane's text.

A JEditorPane will generate an HyperlinkEvent whenever the user clicks on a hyperlink when the pane is in HTML mode and is set to non-editable.

TextPane (JTextPane)

A JTextPane is used to display stylized text. The JTextPane can be set read-only by disabling the 'Editable' checkbox.

A JTextPane will generate a CaretEvent whenever the user modifies the pane's text.

ScrollPane (JScrollPane)

A JScrollPane is used to add scrollbars to any of the Swing components. (By default, none of the Swing components draw their own scrollbars. It is up to the user to put these components inside of a JScrollPane if scrolling is desired.) The Horizontal and vertical scrollbar display policies can be set individually.

In most cases, a single component will be placed in the center viewport of the JScrollPane. Optionally, row and column header components can be added, as well as corner components.

SplitPane (JSplitPane)

A JSplitPane allows the user to dynamically modify the location of a divider between two parts. The two parts can be displayed horizontally or vertically. The divider size and location can specified. Optionally, the child components can be redrawn while the divider is being moved.

TabbedPane (JTabbedPane)

The JTabbedPane can hold any number of parts, but only displays one of them at a time. Each part is given a name. The visible part can be chosen programmatically by name or by user interaction with the named tabs.

The properties page for a JTabbedPane has a list of the available pages. Clicking any item in the list will make that card visible. The name of the visible card appears in the 'Current Card' text field and can be changed. New cards can be added by pressing the 'Add a new card' button. A card can be removed by choosing it from the list and pressing the 'Recycle' button. The order of the cards can be changed by choosing a card and pressing the 'Up' or 'Down' buttons. The user can also specify whether the tabs should appear on the top, bottom, left, or right of the pane.

A JTabbedPane will generate an ChangeEvent whenever a new page is selected.

ToolBar (JToolBar)

A JToolBar displays a series of components in a row. The components are laid out in a similar fashion to a FlowLayout. The JToolBar adds the special capability that it can be dragged off of the application window, and displayed in a floating dialog. The dialog can then be positioned anywhere on the user's screen. The initial orientation of the JToolBar can be chosen (horizontal or vertical) and the border can be optionally drawn.

The JToolBar is especially versatile when placed in a BorderLayout. If it is initially placed in any of the border locations (not the Center) and no other components are in the border locations (uncheck 'Show empty panels' for this to work!) the user can drag the toolbar to any of these four locations. This behavior can be disabled by deselecting 'Allow Floatable'.

Tree (JTree)

A JTree displays a set of hierarchical data in a tree. The user can specify whether these items should be editable, whether the tree should scroll when a node is expanded, whether the root of the tree is visible, and whether the root handles should be visible. Also, the user can specify the number of rows displayed and the height each row should have.

A JTree is most often placed inside of a JScrollPane.

The contents of a JTree are specified using a TreeModel. A TreeModel lets the user fully describe the contents of the JTree in an abstract manner to create dynamic contents. See the JDK API reference and documentation for further details.

Table (JTable)

A JTable displays a set of data in a table. The user can specify how cells are selected (single/multiple and cell/column/row), as well as how the table is drawn (colors and lines).

A JTable is most often placed inside of a JScrollPane.

The contents of a JTable are specified using a TableModel. A TableModel lets the user fully describe the contents of the JTable in an abstract manner to create dynamic contents. See the JDK API reference and documentation for further details.

Menus

The Menus page contains the parts used to build menus. By default they are generated as the Swing menu components, and can be placed anywhere in an application. Also, any component can be placed within a menu. The MenuBar has the additional use of being added to a Frame, JFrame, JDialog, or JApplet. As such, a MenuBar can be added to the special ‘menubar empty space’ of a Main, Frame, or Dialog composer. If the composer is generating AWT, then AWT menu code will be generated. In this case, many of the (swing-specific) properties will be ignored.

MenuBar (JMenuBar)

A MenuBar is used to hold a series of menus that the user can navigate with the mouse or keyboard. It is the component that must be added to a Frame in order to build menus. In the case of a Swing application, though, it can also be added to any empty space. The user can optionally specify margins around the menubar, and indicate that a border should be painted.

Menu (JMenu)

A Menu is used to hold a series of MenuItems that the user can navigate with the mouse or keyboard. It can display a single line of text, whose alignment and

position can be specified relative to an icon. Several different icons can be specified to indicate the various states of the Menu. A keyboard mnemonic can also be specified.

MenuItem (JMenuItem)

A MenuItem displays a single line of text, whose alignment and position can be specified relative to an icon. It usually is placed in a Menu. Several different icons can be specified to indicate the various states of the Menu. A keyboard mnemonic and a keyboard accelerator can also be specified.

CheckBoxMenuItem (JCheckBoxMenuItem)

A CheckBoxMenuItem displays a single line of text, whose alignment and position can be specified relative to an icon and a checkmark to indicate on and off. It usually is placed in a Menu and is used to indicate that a feature is enabled. Several different icons can be specified to indicate the various states of the Menu. A keyboard mnemonic and a keyboard accelerator can also be specified.

A set of CheckBoxMenuItems can be grouped together, such that only one can be selected at a time.

RadioButtonMenuItem (JRadioButtonMenuItem)

A RadioButtonMenuItem displays a single line of text, whose alignment and position can be specified relative to an icon and a radiobutton to indicate on and off. It usually is placed in a Menu and is used to indicate that a feature is enabled. Several different icons can be specified to indicate the various states of the Menu. A keyboard mnemonic and a keyboard accelerator can also be specified.

A set of RadioButtonMenuItems are usually grouped together, such that only one can be selected at a time.

Separator (JSeparator)

A Separator is a component whose sole purpose is to indicate visually a separation between a set of parts. It is most often used in a menu, but can be placed anywhere. The user can specify whether the separator should display vertically or horizontally.

JavaBeans

The ***Beans*** page contains any Java Beans which the user has imported into a Simplicity project. Any JavaBeans located in the primary directory of a project (such as created by a Panel or Canvas Composer) will be imported automatically.

In order for a bean to appear in the Object Palette, it must subclass java.awt.Component, and not be a subclass of java.awt.Window.

Importing Beans into Simplicity

JavaBeans are imported into Simplicity through the IDE. When the user chooses the ***JavaBeans*** item from the ***Import*** menu, a dialog appears which allows the user to choose classes from a tree-view of their classpath. The buttons at the bottom of this dialog will add or remove beans from the 'imported' list. When the user presses 'OK', these beans will appear in the 'Java Beans' group in the IDE.

Using Beans

Java Beans can be clicked into a layout in the same manner as any other part. The properties page for a bean contains the Object Name field plus those properties which have 'set' methods. The visual properties for a bean can be modified on the ***Visual*** page. The bean is queried by the Composer for any events it will listen for. These appear on the ***Listeners*** page.

Any additional properties and setup code for a bean should be placed in the ***constructor*** code page. (Choose ***Goto constructor page*** from the ***Code*** menu.)

The Code Sourcerer can probe the methods of a bean. Bean properties can be queried or changed using the first two options in the code sourcerer. These options are

- 'Change a property of an existing part...'
- 'Ask a part about one of its properties...'

Both of these options will present the user with a list of the parts which have been added to the Working Model. If the user chooses a bean part, the properties of the bean will be displayed on the following page.

The Working Model

The Working Model is a live prototype which is built interactively. Because the model is built using the actual Java controls and layouts, the model reflects exactly how the application will function and behave. All of the controls are active and will respond to user interaction. The layouts will show the true part placement.

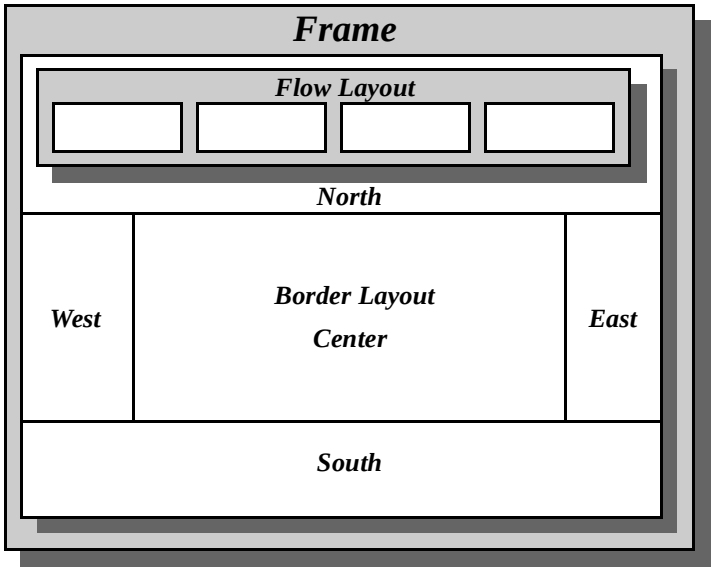
Frames

For Main, Frame and Dialog Composers the Working Model is the Frame or Dialog which is being constructed by the user. For Window and Panel Composers, the Working Model is contained within a Frame in order that the user can easily move it on the screen, as well as view it at different sizes. This Frame is not part of the generated code, though.

Building Layouts

All Working Models initially consist of a single Empty Space. Parts are added to the Working Model by choosing them from the Object Palette and clicking them into an Empty Space using the mouse. Each of the Layout parts can hold multiple additional parts. Complex layouts can be achieved by nesting Layouts within other Layouts. In the following diagram, a Frame contains a Border Layout. In the

northern part of the Border Layout is a Flow Layout. The Flow Layout is containing four other parts.



Once a part has been put into the Working Model it is alive and can respond to user interaction. The user can specify which types of events a part should respond to by selecting them in the part's Listeners page in its properties notebook.

Clicking any part with the mouse will also cause its properties notebook to appear in the Composer window. Some parts are difficult or impossible to click with the mouse. The properties notebooks for those parts can be viewed by choosing the part name from the part list at the top of the Composer window.

CHAPTER 11

Code Sourcerer

This chapter will discuss the Code Sourcerer. The Code Sourcerer will generate Java statements based upon user choices.

This chapter will cover

- How to use the Code Sourcerer
- The major categories in the Code Sourcerer.

Using the Code Sourcerer

Every code area in the Composer has a Code Sourcerer button. When the user presses this button, a dialog will appear which will guide the user through a context-sensitive set of pages. By making a few selections, and filling in a few entry fields the user can construct Java statements to perform a wide variety of activities.

The Code Sourcerer presents the user with a series of panels. Once finished with a panel, the user can proceed to the next panel by pressing the ***Next*** button. The user can return to a previous panel using the ***Back*** button. At any time the user can quit the Code Sourcerer by pressing the ***Cancel*** button. Once the Code Sourcerer has obtained enough information to completely write the requested code, the ***Done***

button will become enabled. Pressing ***Done*** will write the code. Each of these buttons will become enabled/disabled in a context-sensitive manner. In some cases, both ***Next*** and ***Done*** will be disabled. This indicates that there are still critical fields to be filled in before the Code Sourcerer can proceed.

Each time the Code Sourcerer is invoked, the code which is produced is appended to the code area. This code can be used as is, combined with other user code, or edited to meet particular needs.

The first panel of the Code Sourcerer contains eleven categories. The user can choose from

- ***Change a property of an existing part...*** Choose a part and then modify any attribute of the part.
- ***Ask a part about one of its properties...*** Choose a part and then retrieve the value of any attribute.
- ***Declare a new variable...*** Create a new variable of any type and optionally initialize it.
- ***Applet-only operations...*** These options can only be performed by an Applet. (They require a web browser.)
- ***File operations...*** Read, write, or modify files. These operations cannot be performed by an Applet.
- ***Printing operations...*** Print a part, some text, or create a complex print job.
- ***Window operations...*** Create or destroy secondary windows.
- ***Java system operations...*** Interact with the Java Virtual Machine or the Operating System. (Some of these cannot be performed by an Applet.)
- ***Network operations...*** Set up and use a network connection, using either the TCP or UDP protocol.
- ***Miscellaneous...*** Font, Image, Beep, Clipboard, Email, more...
- ***Java Language operations...*** Conditional statements (if, if else) and loop statements.

Change a property of an existing part

This option presents the user with a list of the parts which have been added to the Working Model. The user should choose the part that they wish to modify and press the ***Next*** button. The panel which follows will let the user choose from among the specialized properties for that part.

At the end of this list is always ‘Common properties’. Choose this to modify any of the common visual properties for any part, or to revalidate it. If a part is a Swing part, ‘Common Swing properties’ is added as well. Choose this to modify common swing properties like opacity and the Tool tip text

Ask a part about one of its properties

This option presents the user with a list of the parts which have been added to the Working Model. The user should choose the part that they wish to query and press the ***Next*** button. The panel which follows will let the user choose from among the specialized properties for that part.

At the end of this list is always ‘Common properties’. Choose this to get the common visual properties for any part. If a part is a Swing part, ‘Common Swing properties’ is added as well. Choose this to get properties which are specific to Swing.

After the user has chosen the property that they wish to retrieve, a dialog will ask where the value should be stored. The input field will contain a suggestion which will create a new local variable. While the variable type must remain the same, the name may be changed to any unused, legal Java name. If the user would like to store the value in a variable that has been previously declared, the entire contents of this field can be changed to that name.

Declare a new variable

This option lets the user create a new variable. The user must enter a name for the new variable. The user can choose a primitive type or a class type for the variable. If a class type is chosen, the name for the class should be entered. The full class name should be specified unless the class is a member of the following packages which are imported by default.

- java.lang.*
- java.awt.*
- java.awt.event.*
- java.io.*

Any array dimensionality and set of modifiers can be chosen. The accessibility radio buttons must be set to ***default*** for all code areas except the ‘***Declarations***’ code area.

When the ***Next*** button is pressed, the user will be able to indicate how the variable should be initialized. For primitive types, the user should enter a value. For class types, the user will be given a list of the available constructors as well as the ***null*** value. If a constructor is chosen, the next panel will contain fields for the parameters.

Applet-only operations

This option contains operations that require a web browser or an Applet viewer to work properly. They will only be available in a Main Application Composer where the user has chosen the ***Applet*** or ***Both*** code options. They include

- Ask for the URL for this Applet
- Ask for the URL for the web page containing this Applet
- Ask for the value of an HTML <PARAM> tag...
- Get an Image from a URL...
- Get an AudioClip from a URL...
- Play an AudioClip...
- Stop an AudioClip...
- Loop an AudioClip..
- Load a URL into the browser...
- Display a message in the browser's status bar..
- Ask the Applet if it is currently active

File operations

The file operations panel contains options for manipulating files. These operations are available to any Java Application, but are usually denied to an Applet by the web browser's security policy.

All of the file operations require a File object. This is a class which can refer to a user file in an operating system independent way. The first two options are used to create File objects. The first creates a File object using a hard-coded path. The second lets the end-user choose a file using a FileDialog. In both cases, an unused, legal Java name must be specified to refer to the File object. All other file operations will use this name to refer to the file. These include

- Create a new File object from a pathname...

- Create a new File object from a FileDialog...
- Delete a File object...
- Rename a File object...
- Copy a File object...
- Ask if the File exists...
- Ask if the File is a normal file...
- Ask if the File is a directory...
- Ask if the user has read permission for a File...
- Ask if the user has write permission for a File...
- Get the filename from a File object...
- Get the directory from a File object...
- Get the full pathname from a File object...
- Get the file size from a File object...
- Get the last modified date from a File object...
- Create a new directory...
- Write a String to a text file...
- Read a String from text file...
- Write an Object to a File...
- Read an Object from File...

Printing operations

The printing operations panel contains options for printing some text. The first two options

- Print some text...
- Print the contents of a part...

are very simple. In both cases, all that is needed is to specify the text, or the part, which will be printed. If a layout part is chosen in the second choice, all of the parts that it contains will be printed as well.

The next options let the user set up and use a datarep.common.TextPrinter object. A TextPrinter lets the user control how the printed text will appear. It is important to remember that a TextPrinter object must be created before any options are set, and

that printing will not occur until the TextPrinter object is sent to the printer. The other options can occur in any order desired. They are

- ***Set font...*** allows the user to set the font to any available font from this point onwards.
- ***Set color...*** allows the user to set the color of the printed text to any one of a number of choices. The printer must be able to print in color to see the best results.
- ***Set word wrap...*** allows the user to choose whether the word wrap is on or off. If the word wrap is off, then lines which are long will be cut off at the right margin. If it is on, then they will be wrapped so as to fit properly.
- ***Set tab stops...*** allows the user to specify the size of the tab stops, in inches.
- ***Set margins...*** allows the user to specify the top, bottom, left, and right margins for a page, in inches.
- ***Send form feed...*** allows the user to send a form feed to the printer. This will cause the printer to print any subsequent text on a new page.
- ***Send new line...*** allows the user to force the TextPrinter to start a new line.
- ***Send text...*** allows the user to send text to the TextPrinter object.

When a TextPrinter object is created, an operating system-dependent dialog will appear. However, printing will not occur if the TextPrinter object is not sent to the printer.

To repeat, every TextPrinter job must be created and (eventually) sent to the printer. The commands to do this through the Sourcerer are

- Create a new TextPrinter object...
- Send TextPrinter object to the printer...

Window operations

This option is generally used to launch or close secondary windows. These are usually created using a Frame, Window, or Dialog Composer. Message Boxes can also be launched.

This option can also be used to modify the top-level window of a Composer. To do this, choose the first option, ‘Manipulate the current window - Open/Close/etc...’.

To launch a secondary window, choose the second option, ‘Open a new Frame/Window/Dialog’. The next panel will include a list of classes in the current project which subclass Window. The user should choose a class and enter a name to refer to it.

To close a secondary window, choose the third option, ‘Close a Frame/Window/Dialog. The next panel will ask the user for the variable name which refers to the window to close.

The other four options will launch a datarep.common.MessageBox and wait for the end-user to respond. A MessageBox contains a title bar and a message. The last option allows the user to choose a customized set of buttons for a MessageBox.

Java system operations

These options interact with the Java Virtual Machine and the underlying operating system. They include

- Query the total amount of system memory
- Query the amount of free memory
- Run the garbage collector
- Execute any pending finalization methods
- Get a string with the current date and time
- Query the value of a system property...
- Suspend the current thread for a specified number of milliseconds
- Exit the program with a termination code
- Execute an external command...
- Write text to the standard output...
- Write text to the standard error...
- Read text from the standard input

Network operations (TCP and UDP)

These options will write code to let computers communicate over a network. Java hides some of the complexities of network connections by allowing programs to write to sockets, which can either be TCP based or UDP based. (UDP sockets are also sometimes called “Datagrams”.)

TCP operations

To start a TCP connection, a Client computer must connect to a Server computer. When this is successfully set up, there will be a "socket". Once the socket is set up, both the Server and the Client can send and receive TCP data using the socket.

To set up a TCP Server using the Sourcerer, you must do the following steps

1. Start up a TCP server...
2. Send TCP data... ***or*** Receive TCP data...

(The Server may Send and Receive TCP data as many times as is desired, and in any order desired.)

3. Stop sending or receiving TCP data...
4. Shut down a TCP server...

To set up a TCP Client using the Sourcerer, you must do the following steps

1. Create a TCP client connection...
2. Send TCP data... ***or*** Receive TCP data...

(The Client may Send and Receive TCP data as many times as is desired, and in any order desired.)

3. Stop sending or receiving TCP data...

To set up the Server and Client, it will be necessary to set the port number for the TCP connection. Port numbers 1025-65535 are typically available. The exact number is not important as long as the Client and the Server are using the same port number, and the port is not being used by another application.

Be aware that Java will wait (block) when a TCP server has been set up, waiting for a client to connect. It will also wait when trying to receive TCP data. This should be taken into consideration when considering the Client/Server interaction.

UDP operations

UDP does not differentiate between Client and Server like TCP does. To set up a UDP connection, you must do the following steps

1. Start up UDP...
2. Send UDP data... ***or*** Receive UDP data... ***or*** Reply to UDP data...

(You may ***Send*** and ***Receive*** UDP data in any order, and as many times as is desired. The ***Reply*** option should only be used after UDP data has already been received.)

3. Shut down UDP...

The ***Send*** and ***Receive*** options have many options. It is important to keep track of the port number which is being used, as well as the names of the UDP packet objects which are used to send and receive data, since UDP packets contain information about the connection.

Be aware that Java will wait (block) when trying to receive UDP data.

Miscellaneous

This option includes a variety of operations. They include

- Emit an audio 'beep' sound
- Query the available fonts
- Ensure that all components have been updated
- Get an Image from a filename...
- Get an Image from a URL...
- Get an image using Image selection dialog...
- Copy text to the system clipboard...
- Get text from the system clipboard...
- Query the size of the screen
- Query the screen resolution (in dots per inch)
- Send an E-mail message...
- Convert from Strings to other types...
- Convert variables to Strings...

Java Language statements

This option lets the Sourcerer create Java statements.

Conditional Statements

Conditional statements let the user specify code which will only happen if a certain condition is met. There are two types of conditional statements,

- if...
- if else...

In both cases, the User must enter the statements which happen if the condition is true. A comment shows where this is appropriate.

Loop Statements

Loop statements let the Java program repeat certain portions of code in a controlled manner. There are several different types of loop statements available from the Sourcerer. Each one has its own advantages.

- ***do...*** this will run the statements inside the loop, then check to see if the loop condition is true. If it is true, then the loop will be run again and the loop condition tested again. This means that the loop will be run at least once.
- ***while...*** this will check to see if the loop condition is true. If it is true, then the loop will be run, and the loop condition tested again. This means that it is possible that the loop will not be run at all.
- ***counter...*** this will start a loop which is connected to a counter. The counter will start at a starting number, end at an ending number, and count in user-defined multiples. This is a quick way to make a loop run a specific number of times, or to have access to a variable which is counting up or down.

CHAPTER 12

Canvas Composer

This chapter will discuss the Canvas Composer, which allows you to create new graphical JavaBeans. The differences between the Canvas Composer and the other Composers will also be highlighted.

This chapter will cover

- Creating a new Canvas Composer
- Differences between the Canvas Composer window and other Composers
- A brief overview of the tools in the Canvas Palette

Creating a new Canvas Composer

A new Canvas Composer can be created in the same way as the other Composer files. Choose ***Canvas*** under the ***Create*** menu in the Simplicity IDE, or use the Button Bar in the IDE.

The Composer Window for a Canvas object

Canvas Property Notebooks

The ‘Canvas’ property page contains several settings which affect how the Java source code is generated. The new JavaBean can extend one of four posible choices: Component, Container, JComponent, or Canvas. The generated Java class will have its name specified by the text in ***Object Name***, and it will extend the chosen parent class.

The Canvas Composer has a special page labelled ‘Canvas Methods’. It contains two methods:

- paint(Graphics g)
- getPreferredSize()

The getPreferredSize() should return the best size for the component being created. For example, the code

```
return new Dimension(300,200);
```

would make the preferred size of the Canvas 300 pixels wide and 200 pixels high. New Canvas objects start with a preferred size of (200,200).

The paint() method contains all of the instructions for drawing the component. Each of the palette buttons write some code to this area. The user can also write code here as well. The working model will display the appearance of the component based upon the drawing instructions in the paint() method. The paint method of a Canvas object might be called very frequently, so it is a bad idea to have computationally intensive code in this area.

Canvas Working Model

The Canvas working model initially is a window with the characteristics as defined in the canvas property notebook. The user may use the items in the Palette menu to interactively draw on the working model, or may type in the appropriate code in the ***paint method*** area. Once an object has been drawn, it may be modified by changing the code in the ***paint method*** area.

Canvas Palette Pages

The Canvas Composer has a set of special palette pages. These are

- ***Graphics*** These are tools to perform common drawing tasks, such as drawing ovals, rectangles, and placing text.
- ***Effects*** These are tools which will change the display of subsequent objects drawn on the Canvas.
- ***Colors*** These let the user choose the colors of subsequent objects drawn on the Canvas.

Palette items do not take effect until the user clicks (or drags) on the working model. Items will become unselected after one use. Palette items may be unselected by clicking them a second time.

Graphics Parts

The following graphics parts all require the user to drag on the working model. The shape will be drawn interactively.

- ***Oval***
- ***Filled Oval***
- ***Rectangle***
- ***Filled Rectangle***
- ***Line***
- ***Arc*** (this requires additional interaction)
- ***Filled Arc*** (this requires additional interaction)
- ***Rounded Rectangle*** (this requires additional interaction)
- ***Filled Rounded Rectangle*** (this requires additional interaction)

In addition, the following parts require the user to click on the working model, after which some additional interaction is required.

- ***Text***
- ***Image***

Oval

This will draw an oval. Since the java.awt.Graphics definition of an oval describes an oval inscribed in a rectangle, the corner of the oval will not be exactly at the cursor. The color of the oval will be that of the current foreground color.

If the control key is held down while drawing, the oval will be constrained to be a circle.

If the shift key is held down while creating the oval, its size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the oval.

The control and shift keys may be used simultaneously.

Filled Oval

This will draw a filled oval. The oval will be filled with the current foreground color.

If the control key is held down while drawing, the oval will be constrained to be a circle.

If the shift key is held down while creating the filled oval, its size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the filled oval.

The control and shift keys may be used simultaneously.

Arc

The Arc draws a portion of an oval. First the underlying oval is drawn. Then the user must click and drag to set the first endpoint of the arc. A line is drawn from the center of the oval to the position of the cursor for reference. Next the other endpoint of the arc is chosen by clicking and dragging until the desired arc is drawn. The underlying oval is shown greyed out for reference. When the arc is finished, the reference lines will disappear, leaving only the arc. The arc will be drawn in the current foreground color.

If the control key is held down while drawing, the oval will be constrained to be a circle.

If the shift key is held down while creating the underlying oval, the Arc's size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the arc.

The control and shift keys may be used simultaneously.

Filled Arc

The Filled Arc is created identically to the ***arc***, but the interior of the arc is filled with the current foreground color.

If the control key is held down while drawing, the oval will be constrained to be a circle.

If the shift key is held down while creating the underlying oval, the filled arc's size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the filled arc.

The control and shift keys may be used simultaneously.

Rectangle

Drag to draw a rectangle in the current foreground color. The point where the user presses initially will be one corner, and the point where the user releases will be the opposite corner.

If the control key is held down while creating the rectangle, it will be constrained to be a square.

If the shift key is held down while creating the rectangle, its size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the rectangle.

The control and shift keys may be used simultaneously.

Filled Rectangle

Drag to draw a rectangle filled with the current foreground color. The point where the user presses initially will be one corner, and the point where the user releases will be the opposite corner.

If the control key is held down while creating the filled rectangle, it will be constrained to be a square.

If the shift key is held down while creating the filled rectangle, its size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the filled rectangle.

The control and shift keys may be used simultaneously.

Line

Draws a line in the current foreground color. The point where the user presses will be the beginning of the line, and the point where the user releases will be the end.

If the shift key is held down while creating the line, its size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the line.

Round Rectangle

First a ***rectangle*** will be created. The user must then choose the roundedness of the rectangle. Press the mouse a second time, and drag until the rounded rectangle looks as desired. The amount of rounding is relative to the upper left hand corner of the rectangle. The further the mouse is dragged down and to the right, the more rounding will appear.

If the control key is held down while creating the underlying rectangle, it will be constrained to be a square.

If the shift key is held down while creating the underlying rectangle, the rounded rectangle's size and position will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the size and position of the rounded rectangle.

The control and shift keys may be used simultaneously.

Filled Round Rectangle

The filled round rectangle behaves exactly as the ***round rectangle***, except the interior is filled with the current foreground color.

Text

This lets you draw a line of text in the current font and color on the canvas. To do this, you should

1. Click the mouse anywhere on the working model
2. Type the desired text in the text area
3. Hit return or press the mouse in the working model (outside of the text area)
4. Choose the location of the text. The text will be placed when the mouse button is released.

If the shift key is held down while the location of the text is being chosen, then the position of the text will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the position (but ***not*** size) of the text.

Image

This lets you place an image on the canvas. To do this, you should

1. Click the mouse anywhere on the working model
2. An image selector dialog will appear. Choose the image that you wish to display and press ***Ok*** to continue.
3. Choose the location of the image. The image will be placed when the mouse button is pressed and then released.

If the shift key is held down while the location of the image is being chosen, then the position of the image will be relative to the current size of the Canvas. Later resizing of the Canvas will result in the shifting of the position (but ***not*** size) of the image.

Effects Parts

Set Clipping

This lets you set a clipping rectangle on the canvas. Once the clipping is created, painting only occurs inside the clipping rectangle. This means that it is possible to draw a shape which is not displayed at all because it is not inside the clipping.

Translate

This resets the origin to a new location chosen by clicking the mouse. Use this tool with caution, because all subsequent graphics which are drawn will be affected by the translation, and may be difficult to draw on the working model.

Choose Font

This sets the font, size, and style for all subsequent text placement.

Color Parts

These parts let you set the foreground color of the canvas. This color will be used by all subsequent operations.

Choose a Color

This brings up the standard Swing color chooser. From it, it is possible to choose a color using several different methods.

Black, Blue, Cyan, etc.

These parts each change the foreground color to a standard Java color, defined in Color.

CHAPTER 13

Java Command Window

The Java Command Window allows the user to experiment with Java language statements, execute methods dynamically for testing and debugging, and view the local symbol tables while working in a Composer.

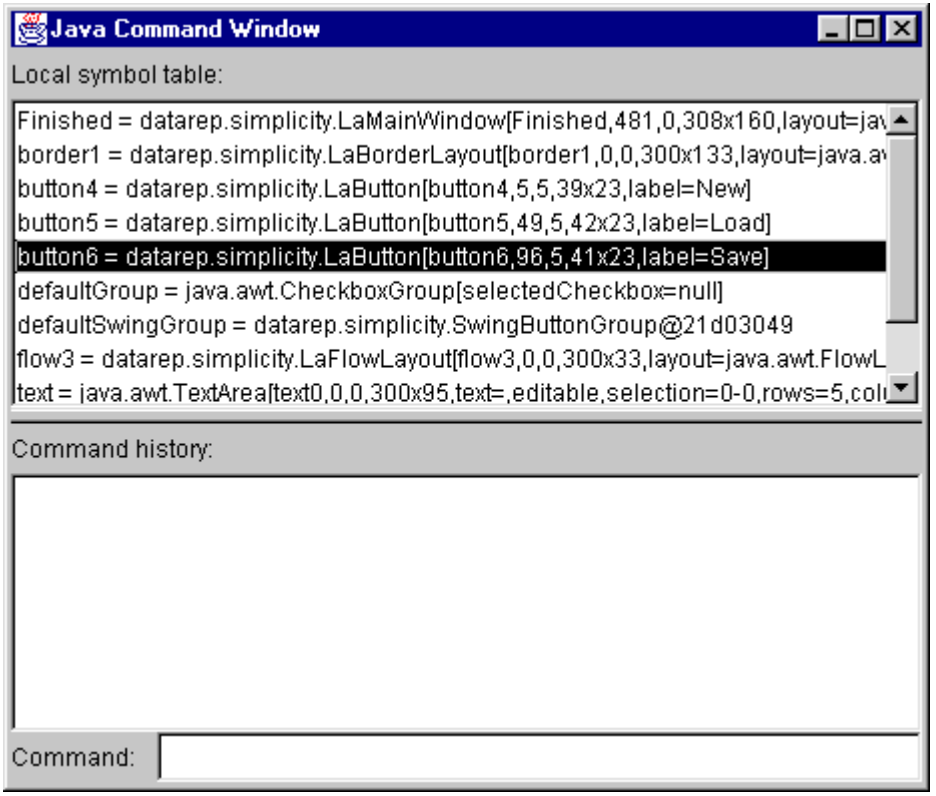
This chapter will discuss:

- Inputting Java statements
- Reviewing the command history
- Manipulating the local symbol table
- Using the Java Command Window from the IDE
- Using the Java Command Window from the Composer
- Using the Java Command Window from the Debugger

Using The Java Command Window

The Java Command Window allows the user to execute individual Java statements to test code, manipulate running applications, and discover information about running applications.

The Window itself three major parts: The Command Input line, The Local Symbol Table, and the Command History.



Command Input

The Command Input, which appears at the bottom of the Command Window, allows the user to enter one or more Java language statements. These will be executed when the user presses the Enter key.

It is not necessary to terminate a single statement with a semicolon, as one will be added implicitly. Multiple statements may be separated by semicolons. For example, the following line may be entered into the Command Input:

```
import java.util.*;
Vector v = new Vector();
int i=0; while(i<8) v.addElement(String.valueOf(i++));
System.out.println(v);
```

Local Symbol Table

The Local Symbol Table contains a list of all of the local identifier names as well as the String value of the object being referenced.

Double clicking a symbol will add the name of the symbol to the Command Input field and then launch the Sourcerer's Apprentice (page 73) to allow the user to select a method or field name from the class.

Command History

The Command History lists all of the commands which you have previously entered. Selecting any command will copy the command to the Command Input field for further editing or execution. Double-clicking a command will immediately re-execute the command.

The Three Java Command Window Contexts

The Java Command Window is used in different ways depending on the context from where it has been launched.

IDE

A Command Window may be launched from the IDE, using the "New Command Window..." item on the "Project" menu. This Command Window will inherit the classpath as set in the dynamic classpath in the IDE. Its symbol table will initially be empty.

This Command Window may be used for general purpose experimentation with Java statements and for execution and testing of classes.

Composer

A Command Window may be launched from any Composer, using the “Command Window” item on the “Program” menu. This Command Window will inherit the classpath as set in the dynamic classpath in the IDE. Its symbol table will contain all of the symbols in the class scope of the Composer. For example, a Command Window opened from within a Frame Composer will contain references to the Frame itself, all of the graphical components that have been added to the Frame, and any declarations added to the Declarations code page.

This Command Window may be used for testing the class being constructed in the Composer. Properties of the components in the Composer may be changed programatically for testing and experimentation purposes. Any method in the Methods code area may be executed for testing.

Debugger

A Command Window is launched whenever the Debugger is started. This Command Window allows the user to manipulate the application being debugged directly from within its own Java Virtual Machine. The Symbol table is initially empty.

This Command Window has two additional buttons at the top that are specific to the debugger. The first, ***Run program...***, allows the user to specify a class with a `public static void main(String[])` method for execution. The second, ***Load classes...***, lets the user specify a set of classes to load into memory.

This Command Window may be used in many ways depending on the type of debugging being done. It may be used to create specialized invocations of Java class methods for testing and debugging, without the need to have a main application previously created to invoke them. It also may be used to query and modify running applications through static method calls.

CHAPTER 14

Debugger

This chapter will discuss Simplicity's Debugger. It will cover

- Starting the Debugger
- The Debugger window
- The Command Window

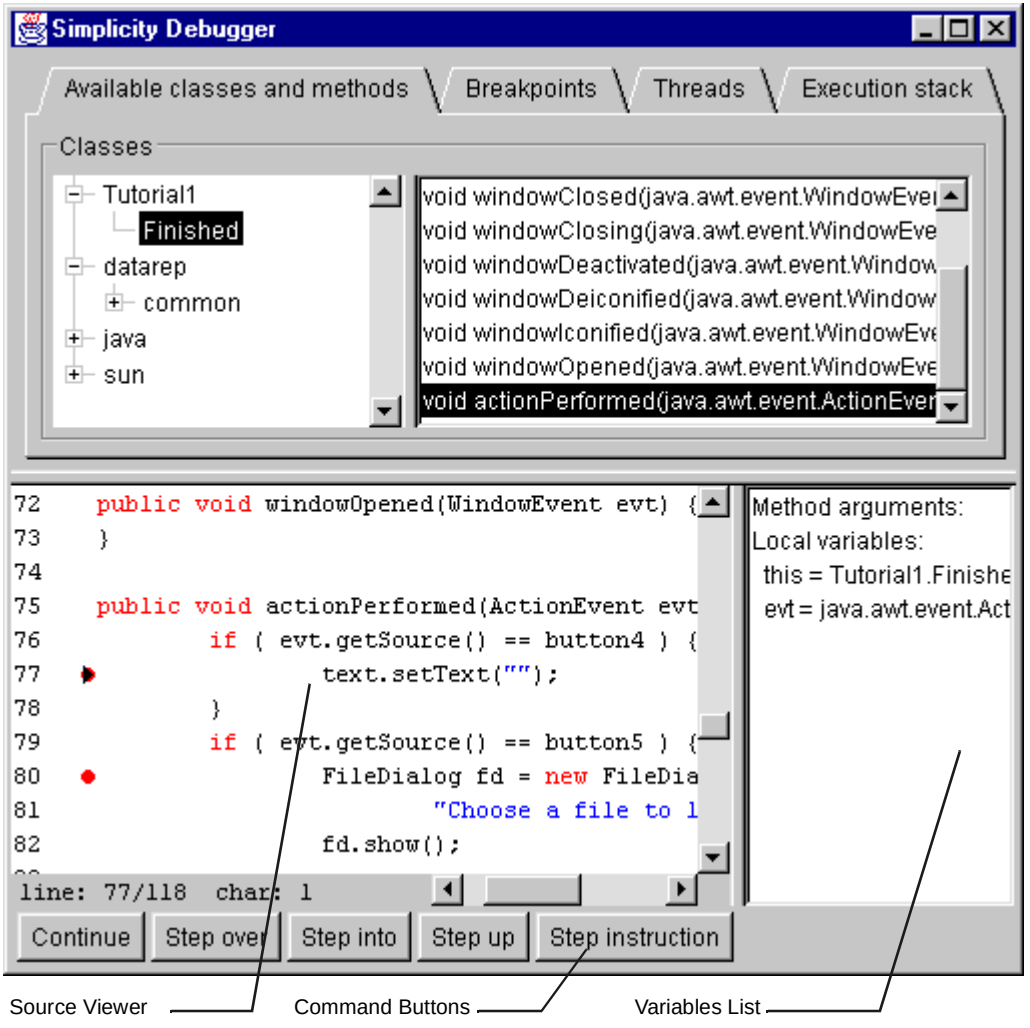
Starting the Debugger

The debugger is started from the Simplicity IDE. When you right-click (or Ctrl-click for single button mice) on any Java Source or Java Class icon, the pop-up menu that appears will include options to launch the debugger. Initially, the Java class which is selected in the IDE will be loaded into the debugger.

Once the Debugger has started, three windows will appear. The Debugger Window, the Debugger Command Window, and the Run Class dialog. The Run Class dialog will allow you to start an application. This dialog will let you specify a Java class which contains a `public static void main(String[])` method, and command line parameters for the application.

The Debugger Window

The debugger window is the central location for controlling and debugging your application. From the Debugger window, you can view all of your classes and their methods, set breakpoints, observe and modify thread states, observe the execution stack, view the source code being debugged (along with markers indicating breakpoints and current location), and view all local variables.



The top portion of the Debugger window contains a tabbed pane with four tabs labeled “Available classes and methods”, “Breakpoints”, “Threads”, and “Execution Stack”. The information displayed on each page is updated in response to program events. An update may be forced by clicking on a tab at the top of a page. (This may be used to refresh the current page, or any other page.)

Available classes and methods

This page contains a tree showing all of the classes that have been loaded into the debugger’s Virtual Machine.

Selecting any class in the tree will display a list of the class’ methods in the list to the right. If available, it will also display the class’ source code and local variables in the Source Viewer and Variables List.

Double clicking any method in the methods list will set a breakpoint at the entrypoint to that method.

Breakpoints

The Breakpoints tab displays a list of all of the current breakpoints. The name of the class and the line number are listed.

Selecting any of the breakpoints in the list will display that source file in the Source Viewer, and jump to that line.

Double-clicking a breakpoint will clear the breakpoint.

Threads

The Threads tab displays all of the threads that the application being debugged contains, as well as information about the current state of each thread.

The four buttons at the bottom of this page can be used to individually suspend or resume a particular thread, or to suspend or resume all threads simultaneously. Note that these buttons have no effect while an application is stopped at a breakpoint. You must press the ***Continue*** button to continue execution from a breakpoint.

Selecting a thread makes that thread the active thread being debugged. The Execution Stack, Source Viewer, and Variables list will display information in the context of the active thread.

Execution Stack

The Execution Stack page displays the execution stack and current location within the current thread's stack frame. This only has meaning when the thread is suspended or is stopped at a breakpoint.

Selecting a method call within the execution stack will cause the Source Viewer and Variables list to display their contents within the context of that method.

Source Viewer

The Source Viewer displays the source code (if available) for the class being debugged. It is context sensitive, and will jump to the source file or method or line number being chosen in the class list, breakpoints list or execution stack.

Double clicking a line will set a breakpoint at that line or clear a breakpoint if one already exists. A breakpoint can only be set at a line containing an executable statement.

Variables List

The Variables List contains a list of all local variables and method arguments in the current Source Viewer context. It also contains a reference to the "this" pointer in the current class context.

Double clicking any item in the Variables List will launch a window containing an additional Variables List showing an expanded view of that item. Double-clicking an item in the new list will again launch an additional Variables List. Class fields may be viewed to an arbitrary depth in this manner. The contents of any given Variables List window may be unavailable depending on the current active thread and stack depth, but will be refreshed when the appropriate context becomes active.

Command Buttons

At the bottom of the Debugger Window are five Command Buttons, which are used to control execution when an application is stopped at a breakpoint. The buttons are:

- ***Continue*** This button will resume normal execution of an application after it has been stopped at a breakpoint.
- ***Step over*** This button will execute the line of code at the cursor, and attempt to remain in the same stack location.

- ***Step into*** This button will begin execution of any method call in the line of code at the cursor, and will leave the cursor at the top of that method call for further debugging.
- ***Step up*** This button will instruct the debugger to finish executing the method in the current stack location, and leave the cursor at the next line in the stack location above the current stack location.
- ***Step instruction*** This button will execute a single Java byte code instruction within the current line of code.

Java Command Window

A Java Command Window is launched whenever the Debugger is started. The Command Window allows the user to manipulate the application(s) being debugged directly from within its own Java Virtual Machine. The Command Window may be used in many ways depending on the type of debugging being done. It may be used to create specialized invocations of Java class methods for testing and debugging, without the need to have a main application previously created to invoke them. It also may be used to query and modify running applications through static method calls.

This Command Window has two buttons at the top that are specific to the debugger. The first, ***Run program...***, allows the user to specify a class with a `public static void main(String[])` method for execution. The second, ***Load classes...***, lets the user specify a set of classes to load into memory.

Run program

The ***Run program*** dialog lets the user specify a class with a `public static void main(String[])` method for execution. The user may also enter command line parameters (one per line) into the dialog. The specified class's `main` method will be executed when the ***Run*** button is pressed. All applications started through the ***Run program*** dialog execute within the same Virtual Machine and are capable of interacting with each other through static methods.

Load classes

The ***Load classes*** dialog allows the user to instruct the Java Virtual Machine to load particular classes into memory. It is primarily used to force the debugger to load a class so that the class can be viewed in the Debugger Window or so that breakpoints can be set in the class.

The dialog works in a similar fashion to the “Import Java Bean” dialog. A classpath tree is on the left panel. The middle panel shows a list of classes in the selected package. Selecting and pressing the ***Add*** button will add the classes to the panel on the left. The classes are loaded when the ***Ok*** button is pressed.

CHAPTER 15

Advanced Features - Extending the IDE

This chapter will discuss how to extend Simplicity's Integrated Design Environment (IDE) and how it can be customized.

It will cover

- How to extend the IDE
- Some Samples of the extended IDE

Extending the IDE

Simplicity version 1.2 introduced the possibility to extend the IDE. The menus in the IDE can all be customized to meet the user's current needs and to allow for easy access to programming utilities.

The IDEmenu.config file

When Simplicity starts, it looks for a file called IDEmenu.config in the Personal Settings Directory (See page 6). If this file does not exist, Simplicity will use the default factory settings for the menus.

A copy of IDEmenu.config is included in the install, but it will not be used unless it is moved to the Personal Settings Directory.

The IDEmenu.config file is a plain text file, which uses a simple markup language similar to HTML to format the menus. Capitalization of the tags and commands are also not important. Spaces and line breaks between tags are not important. Like HTML, tags must be contained within angled brackets <>.

The only allowed tags are

- <menubar> </menubar>
- <menu> </menu>
- <menuitem> </menuitem>
- <separator>
- <action> </action>
- A # sign signifies a comment, which causes the rest of the line to be ignored.

MenuBar

The IDEmenu.config file must have both the <menubar> and </menubar> tags. The text between these will be used to build the menus.

Menu

The Menu tag starts a menu. The ***label=*** command gives the name. Quotation marks are not necessary if the name is only one word; names longer than a word should be enclosed in straight quotes. Menus can be nested (to give sub-menus). Each <menu> tag must be paired with a </menu>. The following example will create one menu, named File, which only contains a sub-menu named “Open This...”.

```
<menubar>
<menu label=File>
<menu label="Open This..."> </menu>
</menu> </menubar>
```

MenuItem

The MenuItem tag defines a menu item. A MenuItem should be given a name using the ***label=*** command. Quotation marks are not necessary if the name is only one

word; names longer than a word should be enclosed in straight quotes. If a MenuItem should be available even when there is no open Project, then the command ***alwaysEnabled=true*** should be included in the MenuItem tag. If this is not included, the default of ***alwaysEnabled=false*** is used. For a MenuItem to have any function, it must contain at least one Action tag. The </MenuItem> tag is optional, since MenuItems cannot nest. The following sequence will be correctly interpreted by Simplicity as being two MenuItems in a menu: `<Menu label=a> <MenuItem label=one> <MenuItem label=two> </Menu>`

Separator

The Separator tag creates a separator between MenuItems. In most operating systems, the separator appears as a line.

Action

The Action tag tells Simplicity what to do when a menu item is chosen. It should be enclosed within MenuItem tags. There are two different ways to tell Simplicity what should be done. The ***command=*** command or the ***class=*** command can be used.

The ***command=*** command lets the user specify a command just as would normally be done from the command line of the operating system being used. Tabs can be used to separate parts of the command. The following command might be used to open Netscape on a machine running Windows:

```
command=C:/Netscape.exe
```

Windows normally uses the "\" character to separate directories. However, this character is used to denote escape sequences in Java. To use \s in the above example, the command would have to be `command=C:\\Netscape.exe`.

Even though the MacOS does not have a command line, ***command=*** will work to open programs with the MacOS. See the next section for an example.

The ***class=*** command can be used to have Simplicity open a class. The class to be opened must extend datarep.ide.config.Action. The actionPerformed(ActionEvent) method will be called when the menu item is chosen. There are two possible ways to use this command. One would be to create a new class which extends Action; the other would be to extend one of Simplicity's existing menu commands by extending that class. Examples of both will be given in the next section.

Any number of Action tags can be associated with a menuItem; they will be executed in sequential order. The </Action> tag is optional, because Action tags do not nest. Simplicity interprets the following sequence correctly as being two Actions associated with the menuItem “test”: `<menuItem label=test> <action command=command1> <action command=command2> </menuItem>`

Samples of the extended IDE

The following samples have been shortened, but will demonstrate the proper use of the tags and commands available to extend the IDE. Package names have been omitted from the Java code, but may be used if desired.

Adding a command to the help menu

This example shows how to add a new command to the help menu, using ***command*=** . The command being added opens up Netscape to the Simplicity home page. The path given is for an iMac; you will have to change this to reflect your computer

```
<Menubar>
# all text from the ‘#’ onwards is a comment.

# ... other menus should be defined here...
<Menu label=Help>
<MenuItem label="User Guide..." alwaysEnabled=true>
# the user guide should always available
<action class=datarep.ide.config.LaunchUserGuideAction>
# the above action launches the user guide
</menuitem> # this tag is optional
<Separator>
<MenuItem label="Go to data representation’s home page"
alwaysEnabled=true>
<action command="/myiMac/Internet/Netscape Navigator™
   http://www.datarepresentations.com">
# note that there is a tab separating the path name from
# the web site.
</menuItem>
</menu> # this tag is not optional
```

```
</menubar>
```

Adding a new action

This brief example adds a new menuItem called “beep”, which beeps, using the ***class=*** command. First a new class is created, then the addition to the IDEmenu.config file is described.

First a new BeepAction class must be created and compiled:

```
import java.awt.event.*;
import datarep.ide.config.Action;
public class BeepAction extends Action {
     public void actionPerformed(ActionEvent event) {
         getToolkit().beep();
    }
}
```

Next the following addition needs to be made to the IDEmenu.config file. Note that the class which is used in the action tag is the new BeepAction class.

```
<menuItem label=beep> <action class=BeepAction> </menuItem>
```

Modifying existing actions

In addition to creating new classes extending the Action class, users can easily change existing classes by extending them. In this example, the Refresh menu item (under the Edit menu) will be modified. First a new class will be created which extends RefreshAction, then the modifications to the IDEmenu.config file will be shown.

First the BeepAndRefreshAction class must be created and compiled:

```
import java.awt.event*;
import datarep.ide.config.RefreshAction;
public class BeepAndRefreshAction extends RefreshAction {
    public void actionPerformed(ActionEvent event) {
       getToolkit().beep();
       super.actionPerformed(event);
    }
}
```

Next the following change must be made to the IDEmenu.config file. The tag which reads

```
<action class=datarep.ide.config.RefreshAction>
```

should be replaced with

```
<action class=BeepAndRefreshAction>
```

Now whenever the Refresh item is chosen from the Edit menu, the user will hear a beep before the refresh occurs. Although this example was simple, more complex classes could be created.

A Complex Action

This last example will show how multiple action tags can be associated with a MenuItem. It will also involve making the Refresh menu item beep, both before and after the Refresh occurs. To do this,

1. Make sure that the BeepAction class (page 141) is compiled.
2. replace the code for the Refresh MenuItem with the following code:

```
<MenuItem label=Refresh> <action class=BeepAction>
<action class=datarep.ide.config.RefreshAction>
<action class=BeepAction>
```

More Action tags could be chained together if desired. Also ***command=*** and ***class=*** Action tags can be associated with the same MenuItem.

Index

D

E

Z