CodeWarrior®

Targeting the Java VM

Windows® | Mac® | Solaris™

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 990506 dcb

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`cw_support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	go `Metrowerks`

Table of Contents

1 Introduction

This chapter is your introduction to targeting the Java Virtual Machine (VM) with CodeWarrior. It includes the following topics:

Read the Release Notes

Before using CodeWarrior Java, read the release notes. They contain important information about any late-breaking changes.

If you are new to Java in CodeWarrior, or to CodeWarrior in general, we strongly recommend that you read this chapter carefully. You will get a high-level overview of Java in the CodeWarrior environment.

About This Manual

This manual assumes you have a working knowledge of your operating system and its conventions, including how to use a mouse and standard menus and commands, and how to open, save, and close file. For help with any of these techniques, see the documentation that came with your system.

The following sections describe the different conventions used in this manual:

Typographical Conventions

This manual uses some style conventions to make it easier to read and find specific information:

Notes, warnings, tips, and beginner's hints

An advisory statement or **NOTE** may restate an important fact, or call your attention to a fact which may not be obvious.

A **WARNING** given in the text may call attention to something such as an operation that, if performed, could be irreversible, or flag a possible error that may occur.

A **TIP** can help you become more productive with the CodeWarrior IDE. Impress your friends with your knowledge of little-known facts that can only be learned by actually reading the fabulous manual!

A ***For Beginners*** note may help you better understand the terminology or concepts if you are new to programming.

Typeface conventions

If you see some text that appears in a `different` typeface (as the word `different` does in this sentence), you are reading file or folder names, source code, keyboard input, or programming items.

Text **formatted like this** means that the text refers to an item on the screen, such as a **menu command** or **control** in a dialog box.

If you are using an on-line viewing application that supports hypertext navigation, such as Adobe Acrobat, you can click on underlined and colored text to view another topic or related information. For example, clicking the text "What is in This Book" in Adobe Acrobat takes you to a section that gives you an overview of the entire *Targeting the Java VM* manual.

Host Conventions

CodeWarrior runs on the host platforms and operating systems listed below. Throughout this manual, a generic platform identifier is used to identify the host platform, regardless of operating system.

The specific versions of the operating system that host CodeWarrior are:

- **Windows**—desktop versions of the WIndows operating system that are Win32 compliant, such as Windows 95, Windows 98, or Windows NT.
- **Mac OS**—desktop versions of Mac OS, System 7.1 or later.
- **Solaris**—Solaris version 2.5.1 or later.

Figure Conventions

The visual interface of the hosts listed in "Host Conventions" is nearly identical in all significant respects. When discussing a particular interface element such as a dialog box or window, the screenshot may come from any of these hosts. You should have no difficulty understanding the picture, even if you are using CodeWarrior on a different host than the one shown.

However, there are occasions when dialog boxes or windows are unique to a particular host. For example, a particular dialog box may appear dramatically different on a Windows host and on a Mac OS host. In that case, a screenshot from each unique host will appear and be clearly identified so that you can see how CodeWarrior works on your preferred host.

Keyboard Conventions

The default keyboard shortcuts for CodeWarrior on some platforms are very similar. However, keyboards and shortcuts do vary across host platforms. For example, a typical keyboard for a Windows machine has an Alt key, but that same key is called the Option key on a typical keyboard for a Mac OS computer.

To handle these kinds of situations, CodeWarrior documentation identifies and uses the following paired terms in the text:

- Enter/Return—the "carriage return" or "end of line" key. This is not the numeric keypad Enter key, although in almost all cases that works the same way.
- Backspace/Delete—the Windows Backspace key and the Mac OS Delete key. In most cases, CodeWarrior maps these keys the same way. This is the key that (in text editing) causes the character before the insertion point to be erased. (This is not the Delete/Del, the "forward delete" key.)
- Ctrl/Command—the Windows Ctrl (control) key and the Mac OS Command key (⌘). In most cases, CodeWarrior maps these keys the same way.
- Alt/Option—the Windows Alt key and the Mac OS Option key. In most cases, CodeWarrior maps these keys the same way.

For example, you may encounter instructions such as "Press Enter/Return to proceed," or "Alt/Option click the Function pop-up menu to see the functions in alphabetical order." Use the appropriate key as it is labeled on your keyboard.

Some combinations of key strokes require multiple modifier keys. In those cases, key combinations are shown connected with hyphens. For example, if you read "Shift-Alt/Option-Enter/Return,"

you would press the Shift, Alt, and Enter keys on a Windows host and the Shift, Option, and Return keys on a Mac OS host.

Sometimes the cross-platform variation in keyboard shortcuts is more complex. In those cases, you will see more detailed instructions on how to use a keyboard shortcut for your host platform. In all cases the host and shortcut will be clearly identified.

Special Note for Solaris Users

The Solaris-hosted CodeWarrior IDE uses the same modifier key names as used for Mac OS (Shift, Command, Option, and Control). Likewise, the Key Bindings preference panel uses Mac OS symbols to represent modifier keys. The Default CodeWarrior Key Bindings appendix in the CodeWarrior IDE User Guide shows the default modifier key mappings and the symbols used to represent them. On Solaris systems, modifier keys can be mapped to any key on the keyboard. See the Keyboard Preferences Dialog Box appendix in the CodeWarrior IDE User Guide for the default key mappings. When reading this manual, you will need to keep in mind these modifier key mappings.

New Features in This Release

Following are the changes in this version of Java for CodeWarrior:

Windows

- On Windows, the Java linker requires Sun's JDK.
- The FTP Post Linker now requires that Sun's JDK or JRE 1.1.6 (or higher) is installed. These are found in the "Extras" folder at the base of the CodeWarrior Windows NT/95 Tools CD.

Mac OS

- Metrowerks Java VM has been retired, and is no longer supported. Java applets and applications now use JBindery / Apple's MRJ VM.
- The default MRJ URL authentication dialog is used, rather than a custom dialog.

- Due to problems with MRJ 2.0 on 68k, the Mac OS version of the Java linker is PPC only.
- This version of CodeWarrior uses the new Sun Java Debugger plugin as the default plugin for debugging Java on Mac OS. For more information about this change, see "Debugger Features and Limitations" on page 72.

Solaris

- This version of CodeWarrior adds support for Sun's JDK 1.2 VM, as well as future versions of the JDK VM. For more information, see "Virtual Machine (Solaris)" on page 99.

What is in This Book

The table below gives a general description of this manual's contents.

Table 1.1 Contents of chapters

Introductory Topic	Description
Introduction	Contents of this manual; general description of the CodeWarrior development tools; where to go next
Getting Started	Installation and setup for Java
Programming Tutorial for Java	Tutorial on Java programming
Creating Java Projects	Creating Java programs
JavaDoc	How to set up and use JavaDoc
Target Settings for Java	Project settings specific to Java programming
Debugging Java Projects	Debugging Java code with the CodeWarrior Debugger
Class Wrangler for Mac OS	General purpose ZIP and JAR file management utility for the Mac OS
Metrowerks Java for Mac OS	Mac OS runtime interpreter for Java bytecodes and additional utilities
Troubleshooting	Troubleshooting information specific to Java
Standalone Applets for Mac OS	Appendix describing how to create double-clickable applets on the Mac OS

Where to Go from Here

The information in this manual is, for the most part, specific to either the Java language or the Java virtual machine as a target. This manual does not cover basic features of the CodeWarrior IDE, but only Java-specific information.

You do not have to read the chapters in this manual sequentially. Use this manual as a reference to learn about Java in CodeWarrior, or to answer questions you encounter as you develop Java code.

You will find all the manuals mentioned in this section on the CodeWarrior CD.

For everyone:

- See the *IDE User Guide* for complete information about the CodeWarrior Integrated Development Environment and debugger.

If you are new to CodeWarrior:

- Look for the CodeWarrior tutorials. You will find them on your CodeWarrior CD.

For general information on Java programming:

This manual does not teach Java syntax, nor does it introduce you to the classes and methods in the Java API—the calls you use to program the Java virtual machine.

- To learn Java, you may consult the *Java Language Tutorial*. This HTML document is available directly from Sun Microsystems at:

  `http://java.sun.com/doc.html`

- To learn more about the Java API, consult the *Java API Documentation*,. This document is in HTML format and is found on Sun Microsystems website:

  `http://java.sun.com/docs/`

- The CodeWarrior Reference CD contains an electronic copy of *Learn Java on the Macintosh* (Addison Wesley) by Barry Boone. *Learn Java on the Macintosh* starts off with some object

programming basics, then introduces you to programming in Java. CodeWarrior also comes with some Java-related Apple Guide files.

- *Discover Programming* from Metrowerks can teach you how to program in Java. You can learn more about Discover Programming at:

  `http://www.metrowerks.com/discover/`

There are also many third party books on Java programming. Check your local bookstore or favorite online bookstore for a wide selection.

CodeWarrior Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit:

`http://www.metrowerks.com/about/y2k.html.`

2

Getting Started

This chapter gives you a brief overview of Java in CodeWarrior, installation, system requirements and tools available to you.

This chapter includes the following topics:

System Requirements

If you can run CodeWarrior, you can target the Java virtual machine.

Windows Requirements

The Windows-hosted version of CodeWarrior requires the following:

- a 486DX processor or higher
- at least 32 megabytes of RAM
- approximately 90 megabytes of free hard disk space for a minimal installation
- approximately 450 megabytes free hard disk space for a full installation
- Microsoft Windows operating system (Windows 95, Windows 98, or Windows NT 4.0 service pack 3)
- a CD-ROM drive to install the software

Mac OS Requirements

The Mac OS-hosted version of CodeWarrior requires the following:

- a Motorola 68040 processor, or a PowerPC 601 processor or higher
- at least 32 megabytes of RAM
- approximately 120 megabytes of free hard disk space for a minimal installation
- approximately 400 megabytes of free hard disk space for a full installation
- Mac OS 7.6.1 or later
- a CD-ROM drive to install the software

Solaris Requirements

The Solaris-hosted version of CodeWarrior requires the following:

- a Sun SparcStation or Sparc-based machine
- at least 64 megabytes of RAM
- approximately 80 MB of free hard disk space
- Solaris 2.5.1 or later, and X11-R5/Motif 1.2 (CDE recommended)
- a CD-ROM drive to install the software

Installing CodeWarrior for Java

Use the CodeWarrior installer application to install the CodeWarrior IDE. Be sure to follow the instructions in the installer application. The installer ensures that everything is installed in the proper locations.

See the QuickStart manual for instructions on using the installer to install CodeWarrior on your particular platform.

Overview of Java in CodeWarrior

Java is both a programming language and, in a virtual sense, a computer platform—a target, in CodeWarrior terminology. As such, Java is unique among the languages and targets supported in the CodeWarrior IDE. You can think of Java as not only a language, but as an instruction set for the Java *virtual machine*.

The virtual machine, commonly called a VM, is an abstract computer microprocessor. Like any silicon computer chip, the virtual machine has an instruction set. The instructions for the Java virtual machine are called *bytecodes*.

A bytecode is a stream of formatted bytes that has a precisely defined impact on the virtual machine. Bytecodes tell the virtual machine to do things like push and pop values on a stack, branch, load and store values, and so forth. Just like traditional assembly language instructions affect a silicon-based processor, the Java bytecodes are a real instruction set for the Java virtual machine.

The reason for the abstract nature of the virtual machine and its bytecode instruction set is to allow you, the programmer, to create platform-independent code. The runtime interpreter and just-in-time compilers replicate in a real operating system the effect that a bytecode has on the virtual machine.

What you do in the Java environment is a little different from what you have done in the past using C, C++, or Pascal.

The development process for Java software in CodeWarrior has the following basic steps:

1. **Write source code.**

   In CodeWarrior, you write Java source code in exactly the same way you do for C, C++, and Pascal. You use the same project manager and the same source code editor. You get all the power and features of CodeWarrior. To target the Java virtual machine, you make the appropriate selection in the proper settings panel.

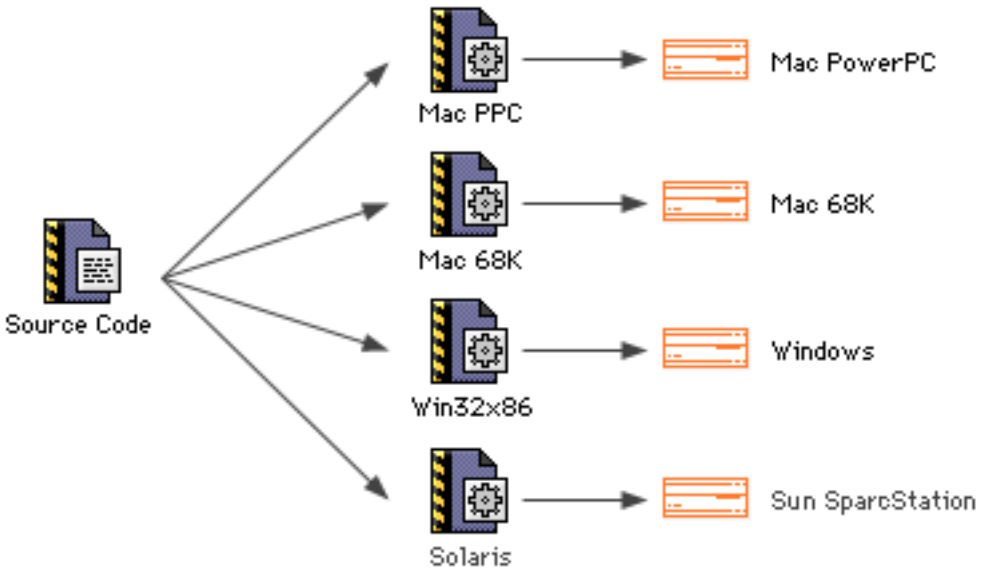
2. **Compile the source code.**

   Again, this process is exactly like it is for any language and target supported in CodeWarrior. You issue the appropriate **Compile** or **Make** command from the CodeWarrior Project menu. In response, the CodeWarrior Java compiler generates Java bytecodes based on your source code. CodeWarrior saves the bytecodes in a Java class file.

   The bytecodes in the Java class file are analogous to the object code generated by a C/C++ or Pascal compiler. The difference between traditional object code and Java bytecodes is in how they run.

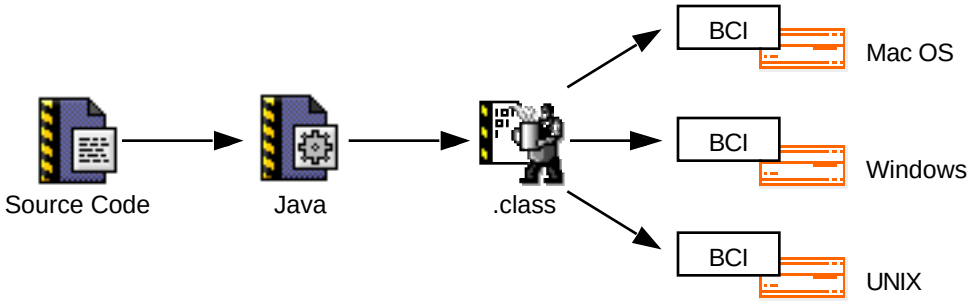
   Object code can run directly on the platform for which it was compiled, but cannot run on any other platform. You must compile it over and over again for each platform. There are also serious cross-platform problems you can encounter trying to address multiple operating systems and computer microprocessors.

Figure 2.1 Compiling and running object code



Java bytecodes cannot run directly on any real machine. On the other hand, the Java bytecodes created by the CodeWarrior Java compiler can be interpreted to run on *any* platform that has a bytecode interpreter (BCI). You write the code once, compile it once, and then you are done.

Figure 2.2 Compiling and running Java bytecodes



3. **Run the code.**

   Choose **Project > Run** and CodeWarrior launches the applet with the applet viewer chosen in Target Settings. To learn how to specify an applet viewer, see "Target Settings" on page 92. Remember, the bytecodes in the class file cannot run directly on any real platform. The class file must be run under the control of a bytecode interpreter (BCI). The BCI is responsible for translating the bytecodes into machine-native instructions.

4. **Debug the code with the CodeWarrior Debugger.**

   The final step in the development process is to test and debug your code. You debug Java code using the same CodeWarrior debugger you use for C/C++ and Pascal code. The CodeWarrior debugger understands Java. The debugger does everything you would expect the debugger to do. It has a stack crawl, displays local variables, and allows you to set breakpoints.

   To debug Java code, your applet or application must be built to include debugging information. First choose **Project > Enable Debugger** to enable the debugger. Then click the debug column in the project window next to the file you want to debug to tell the IDE to include debugging information when it compiles the file. Choose **Project > Make** to compile and link your code. You can also choose **Project > Debug** menu. CodeWarrior compiles, links, and launches your program under debugger control.

   From here you can set break points, and run your program.

Summary

That is all there is to it. As you can see, the big difference between Java and compiled languages is that Java code is interpreted at runtime.

If you have used CodeWarrior in the past, developing Java code in CodeWarrior is going to look very familiar to you. If you are new to CodeWarrior, you will find developing Java code to be a pleasure once you master the tools.

Development Tools for Java

These are the tools CodeWarrior provides for developing Java software:

- CodeWarrior IDE
- CodeWarrior RAD Tools
- Java Linker
- CodeWarrior Debugger
- Java API Headers
- JavaDoc

CodeWarrior IDE

The CodeWarrior Integrated Development Environment (IDE) provides a complete set of tools for developing application programs for a variety of target platforms, including 68K- and PowerPC-based Mac OS systems, Win32/x86 systems, Solaris, and Java. You use the same IDE when developing code for all target platforms; you designate the platform and code generator of your choice when you create your project.

For information on the CodeWarrior IDE, see the *IDE User Guide.*

CodeWarrior RAD Tools

CodeWarrior RAD tools let you visually construct an application. The tools included with the IDE are for use with C++ and Java. They extend the graphical capabilities of the IDE for use with RAD.

For more information on CodeWarrior RAD tools, see the *IDE User Guide*.

Java Linker

Like the compilers, the Java Linker is a plug-in tool integrated into the CodeWarrior IDE.

CodeWarrior Debugger

The CodeWarrior debugger controls your program's execution and allows you to see what is happening internally as your program runs. You use the debugger to find problems in your program's execution.

The debugger can execute your program one statement at a time, and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the contents of the processor's registers.

For general information about the debugger, including all of its common features and its visual interface, you should read the *IDE User Guide*.

For more information about debugging software for Java, see "Debugging Java Projects" on page 71.

Java API Headers

The Java Application Programming Interface (API) is a set of data structures and functions used to interface to the Java operating system.

JavaDoc

JavaDoc is a batch compiler that processes Java source code files, and uses the comments preceding classes, methods, etc. to generate HTML based documentation for the code. See "JavaDoc" on page 85 for more information.

3

Programming Tutorial for Java

This tutorial takes you very quickly through the CodeWarrior Java environment. It does not teach you Java programming. It is designed to teach you how to use the CodeWarrior Integrated Development Environment (IDE) to write and debug Java code. The tutorial takes you step-by-step through the entire process. The applet you create is designed to be immediately reusable with little to no change, depending on your needs.

NOTE: This tutorial requires a Java-enabled browser or Java applet viewer. If you do not already have a Java enabled browser installed, the latest version of Microsoft Internet Explorer is available on the CodeWarrior CD.

The topics discussed in this tutorial are:

Applet Description

The applet you will create in this tutorial is an animation with sound, that you can add to any web page. This particular implementation shows a nifty about box animation with sound. But when we are done with this tutorial, you will be able to use this applet for anything you want simply by changing the parameters in the `applet` HTML tag.

Theory of Operation

How the applet works is fairly simple. It loads the `AboutBox.gif` image into memory. The image contains 49 sections, from top to bottom, each containing a different slide of the animation. The applet uses an array (`sequence[]`) to describe the sequence in which the slides are displayed. The image display sequence is played and repeated indefinitely.

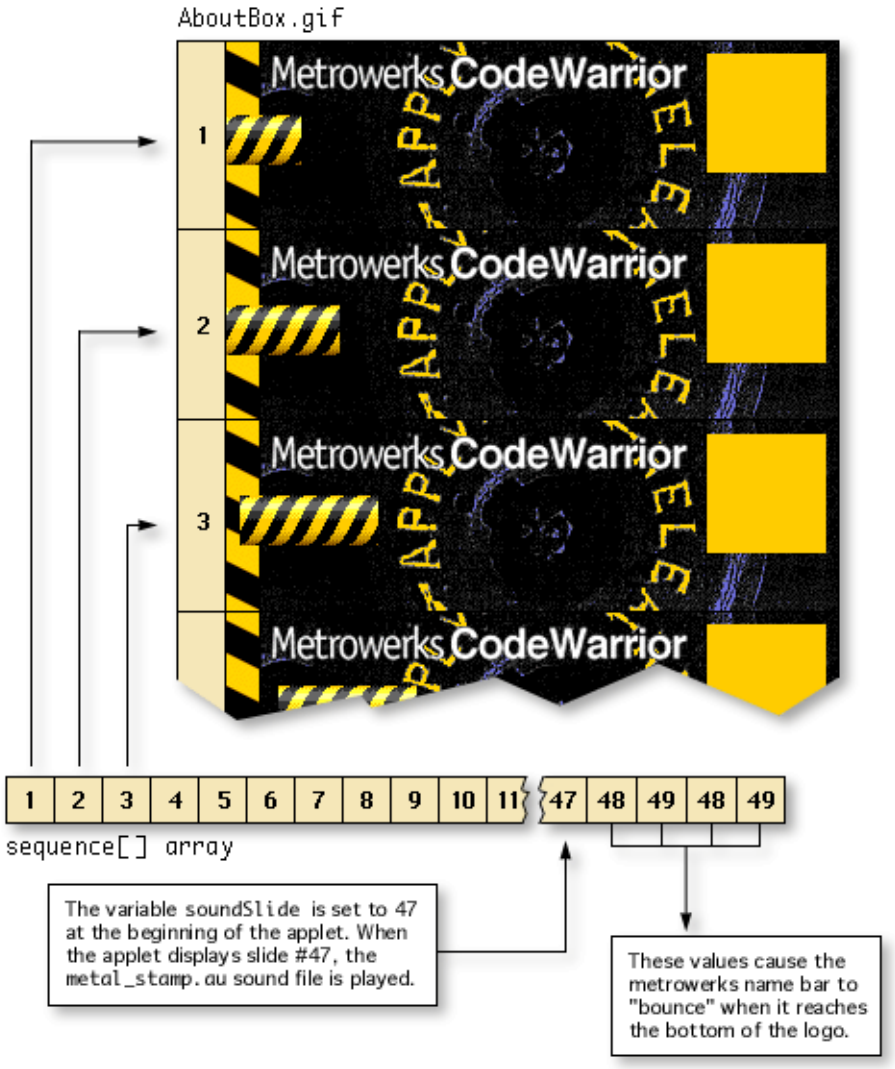
Each time the applet gets an update event, it calls a `Paint` method that draws the appropriate slide based on the slide number contained in the current element of the sequence array.

When the applet gets to the slide indicated by the `soundSlide` variable, it plays the sound file specified in the `sound` parameter of the `applet` HTML tag. In this tutorial, the `metal_stamp.au` sound file is played.

The last four elements of the sequence array cause the last two slides to be repeated. This coincides with the playing of the sound file. It creates the illusion that the Metrowerks name bar is "stamped" into place.

Figure 3.1 shows how the applet works. The first element of the sequence array holds the number one. The applet displays the section of the image corresponding to slide one. The second element of the array holds the number two. The applet displays slide two. The third element of the array contains the number three. The applet displays slide three. This continues until the end of the array is reached, and the sequence is repeated.

Figure 3.1 Theory of Operation



Before You Begin

Before you begin, locate the `AboutBox Tutorial` folder. It is in one of the locations outlined below. If the `ImageMap Tutorial` folder is not already installed on your hard drive, copy it from the CodeWarrior Reference CD to your hard drive now. The files you create and modify during this tutorial are located in this folder.

AboutBox Tutorial folder locations

Windows `CodeWarrior Examples\CodeWarrior Java\Java Tutorial\AboutBox Tutorial\`

Mac OS `CodeWarrior Examples:CodeWarrior Java:Java Tutorial:AboutBox Tutorial:`

Solaris `CodeWarrior_Examples/Java_Tutorial/AboutBox_Tutorial/`

All of the files for the completed tutorial are located in the `AboutBox Solution` folder inside of the `Java Tutorial` folder. You can compare your work against these files if you run into problems.

WARNING! Be careful to use the exact names specified in each step of this tutorial. Some features of Java are very dependent on names, and the applet may not work if names do not match exactly. Java is a case-sensitive language; so make sure you match case as well as spelling.

Creating the Project

In this section we will show you how to create a project using the CodeWarrior IDE, and how to set the project up to make a Java applet. The steps required to do this are as follows:

- Creating a New Project
- Changing Target Settings

Creating a New Project

We will use the CodeWarrior IDE to create a new project to use for our tutorial.

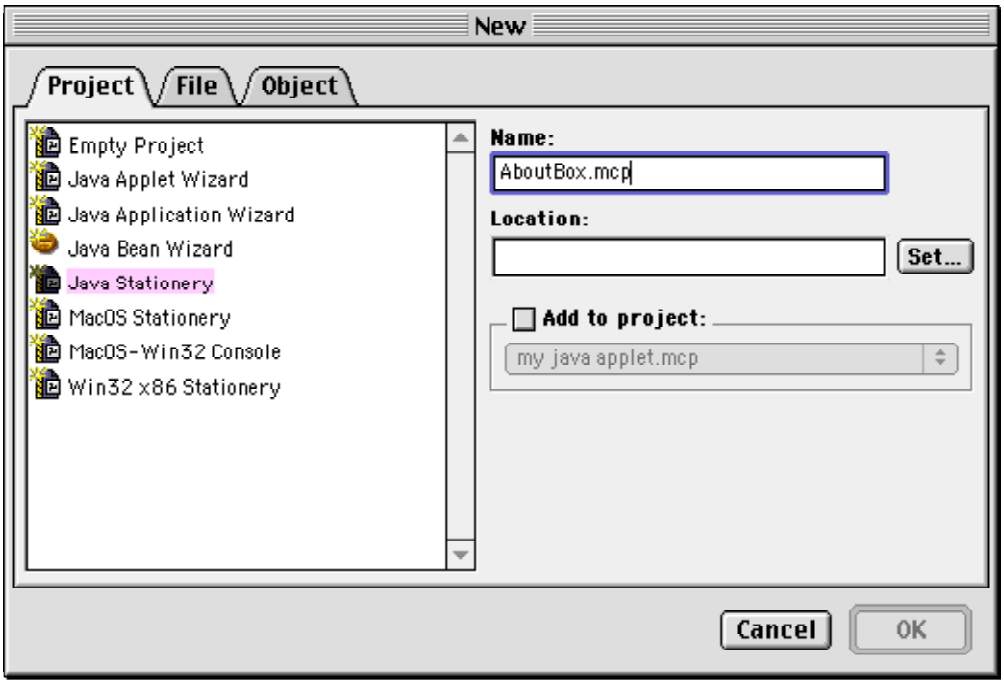
1. **Locate the CodeWarrior IDE application and launch it.**

   The IDE launches and awaits your command.

2. **Choose File > New.**

   The **New** dialog box appears (Figure 3.2).

Figure 3.2 The New dialog box



3. **Select the stationery.**

   Click the **Java Stationery** in the list so that it is highlighted.

4. **Name the project.**

   Name the project `AboutBox.mcp`.

> **NOTE:** The "`.mcp`" suffix is a CodeWarrior naming convention for project files. If you use this naming convention, you will be able to use your project files on any host that CodeWarrior runs on.

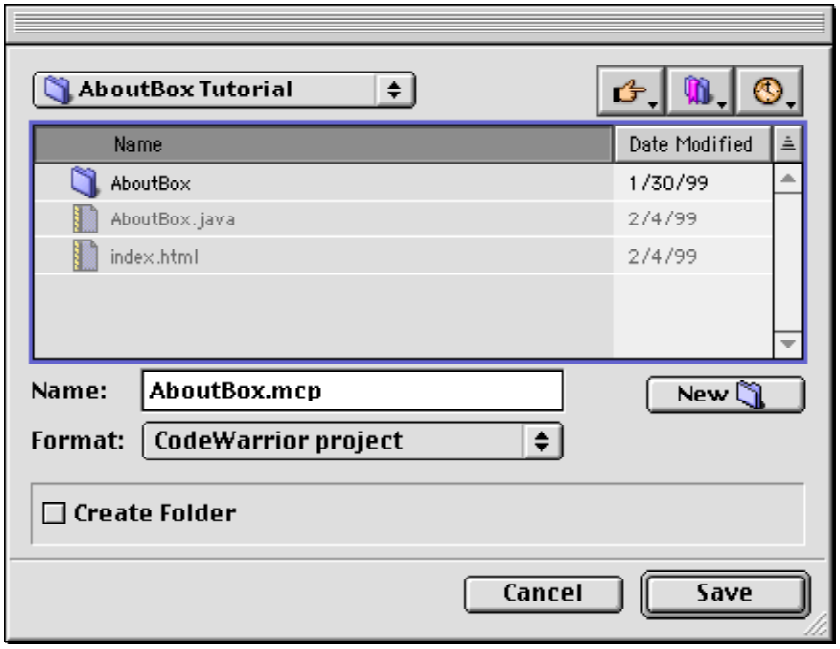
5. **Click Set….**

   CodeWarrior displays a save file dialog.

6. **Uncheck the Create Folder option.**

   This option is located at the bottom of the dialog box. Since we are providing you with a tutorial folder to work with, there is no need to create a enclosing folder for this project.

Figure 3.3 The Save Project As box



7. **Choose the location of the new project.**

   Navigate to the `AboutBox Tutorial` folder on your hard drive. The dialog box should look like Figure 3.3. Click the **Save** button to return to the **New** dialog box.

8. **Click the OK button.**

   CodeWarrior displays the **New Project Stationery** window, listing all of the available Java stationery types.

9. **Choose Java Applet from the list of stationery types.**

   Click **Java Applet** in the list to select it. Then click **OK**. CodeWarrior creates your new project.

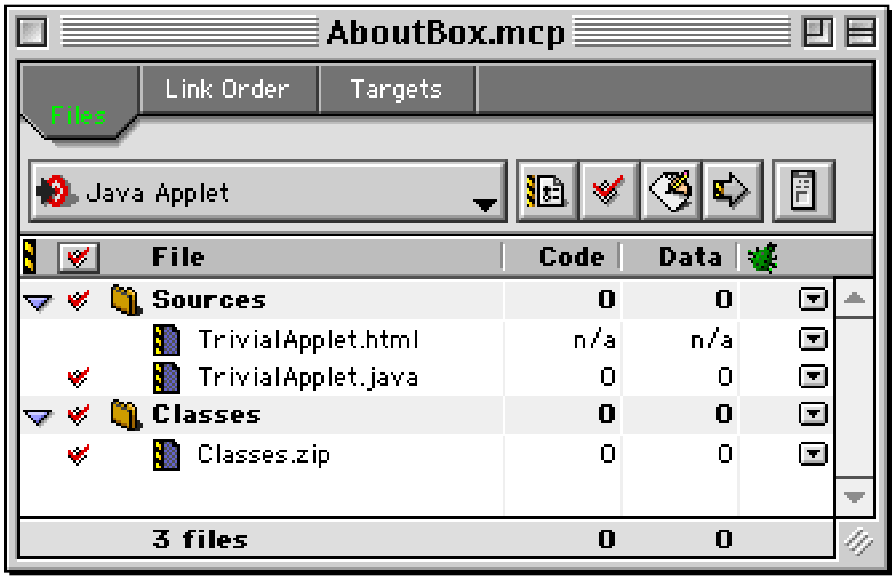
   When saving the new project, CodeWarrior creates three files in the tutorial folder:

   - `AboutBox.mcp` — the project file itself
   - `TrivialApplet.html` — an HTML file
   - `TrivialApplet.java` — a Java source code file
   - `AboutBox Data` — the project data folder

WARNING! The project data folder contains files with information about your project file, target settings, object code, and browser information. Do not change the contents of this folder.

Code Warrior opens the project and displays a window similar to that shown in Figure 3.4. We are now ready to proceed to the next section.

Figure 3.4 The project window



Changing Target Settings

In this section, we will change the settings in the project to instruct CodeWarrior to build a Java class folder for the applet.

About Build Targets

A project file can have multiple build targets. For example, it is common to have a *debug* build target and a *release* build target in the same project. Each build target has its own unique collection of settings that control how CodeWarrior creates the final executable. These settings are called the *build target settings*. These settings can be accessed by choosing **Edit > *Target* Settings** (where ***Target*** is the name of the current build target displayed in the project window). Choosing this menu item open the Target Settings window. When the settings for a build target are changed, the change affects only the current build target.

NOTE: CodeWarrior also has “global” preferences that affect all projects. For more information on preferences and build target settings, see the *IDE User Guide*.

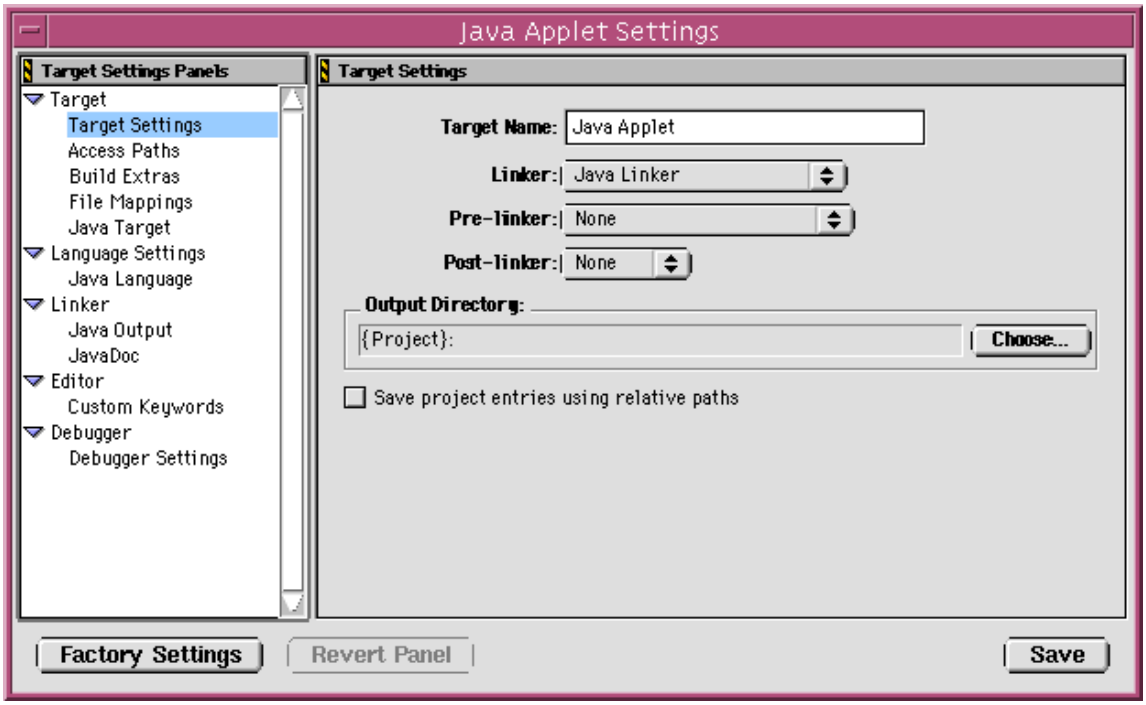
When you create a new project using stationery, the default build target settings are already set. However, for most projects, you will need to change a few of these settings in order to get the results you want.

The steps required to make appropriate changes to the project for this tutorial are:

1. **Open the Target Settings window**

   Choose **Edit > *Java Applet* Settings**. CodeWarrior displays the **Target Settings** dialog box (Figure 3.5). Note that ***Java Applet*** is the name of the build target in the new project. This menu item always reflects the name of the build target currently being displayed in the project window.

Figure 3.5 Target settings dialog box



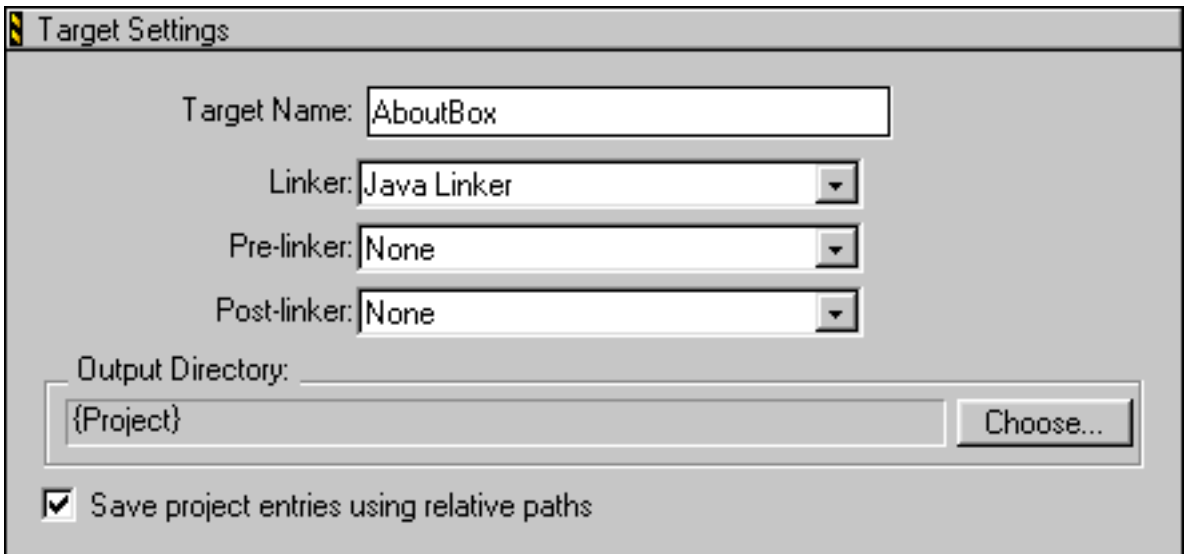
The Target Settings window contains a list of *settings panels* on the left. The right side of the dialog box shows the options for the currently selected panel.

For a more detailed description of each settings panel, see "Target Settings for Java" on page 91.

2. **Change the target name**

   Select **Target Settings** under the **Target** section of the list of settings panels on the left. Change the **Target Name** field to "`AboutBox`" (Figure 3.6). Notice how changing the name of the target changes what is displayed in the build target display area in the project window. For more information on items in the project, see the *IDE User Guide*. The **Edit > Target Settings** menu item also changes to reflect the new build target name.

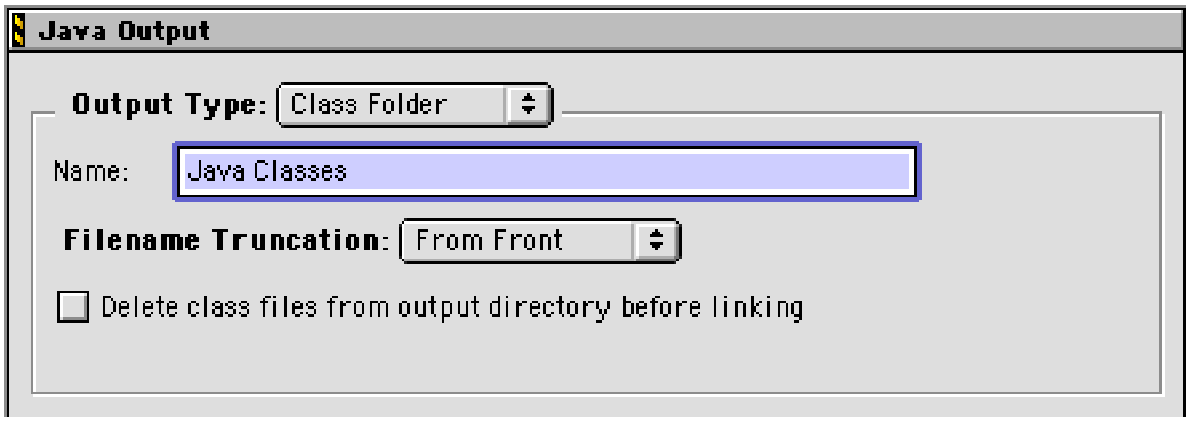
Figure 3.6 Target Name set to "AboutBox"



3. **Change the Output Type to Class Folder**

   Select **Java Output** under the **Linker** section in the list of settings panels. Change the **Output Type** to **Class Folder** as in Figure 3.7.

Figure 3.7 Output Type set to Class Folder

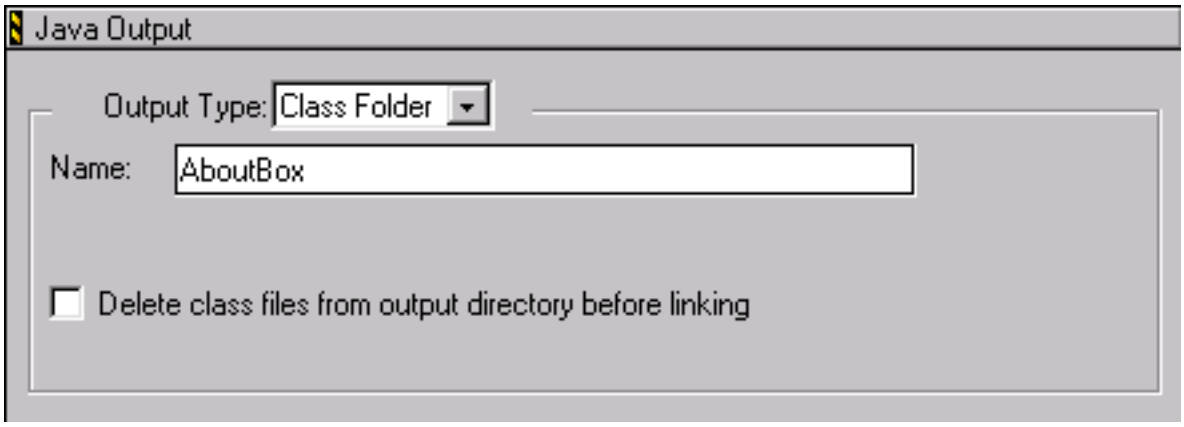


NOTE: The **Filename Truncation** option in Figure 3.7 only applies to the Macintosh version of the CodeWarrior IDE, and will only appear in that version of the IDE.

4. **Change the Name of the Class Folder**

   Change the **Name** field to `AboutBox` as in Figure 3.8. These settings instruct CodeWarrior to create a new class folder named `AboutBox` in the same folder as your project. For this tutorial, we do not need to modify any other settings in this panel.

Figure 3.8 Java Output panel



5. **Save the settings**

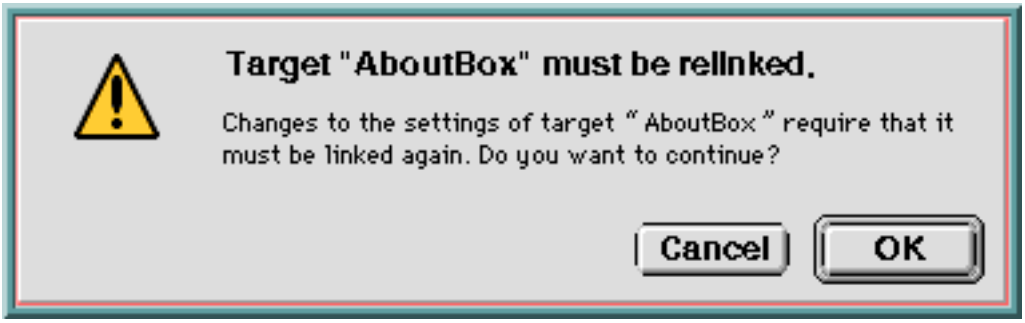
   Click the **Save** button at the bottom of the Target Settings dialog box.

6. **Close the Target Settings dialog box.**

   Click the close box on the Target Settings window now to close it.

NOTE: Depending on which options were changed, you may see the dialog in Figure 3.9 upon closing the Target Settings dialog. Click the **Cancel** button to cancel the closing of the dialog allowing you to make more changes before continuing. Click the **OK** button to close the dialog.

Figure 3.9 Relink Target caution dialog



Now that all the settings for the project have been changed, CodeWarrior will build the type of applet we want. You are now ready to proceed to the next section, where you will start writing the code for the applet.

Writing the Applet

In this section, you write the actual code for the applet. Topics discussed in this section of the tutorial are:

- Adding the Java File
- Editing the HTML File

Adding the Java File

The code for the AboutBox applet is fairly extensive at almost 75 lines. To make things easier, the code is provided for you in the tutorial folder. All you need to do is add it to the project, and remove the place-holder file that is there now.

1. **Tell the IDE where to add the file**

   We want the `AboutBox.java` file to be added to the appropriate group in the project window (in this case, the **Sources** group). To do this, highlight any file in the Sources group, or highlight the group title itself.

2. **Add `AboutBox.java` to the project**

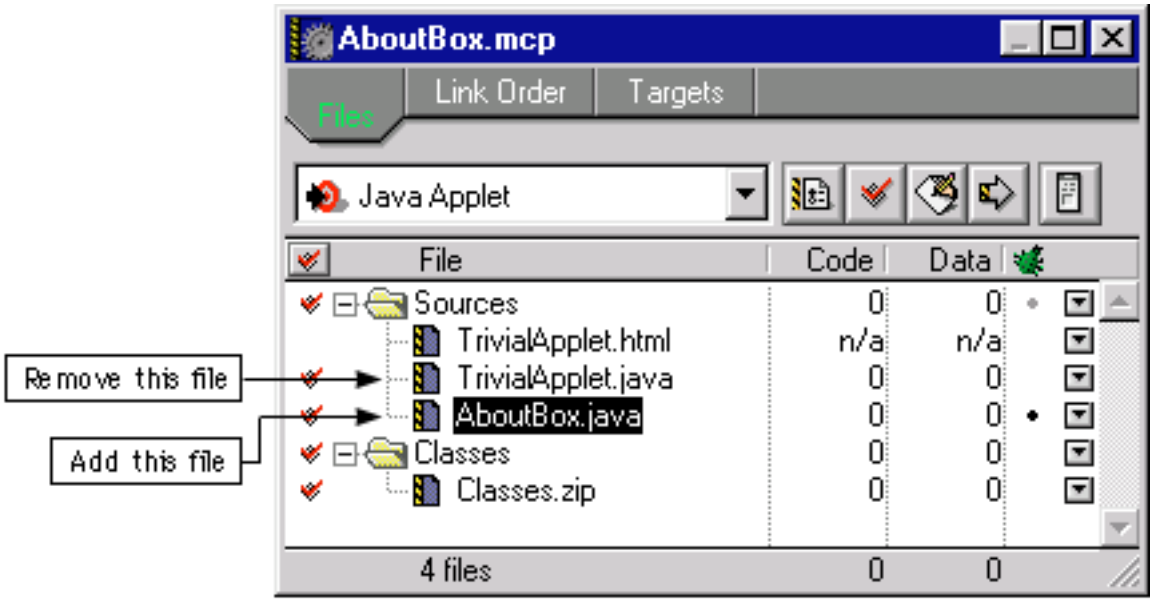
   Choose **Project > Add Files**. An add files file dialog will appear. Navigate to the `AboutBox Tutorial` folder and add the `AboutBox.java` file to the project.

3. **Remove the `TrivialApplet.java` file**

   Select the `TrivialApplet.java` file and choose **Project > Remove Selected Items**. The resulting project window should look similar to the one in Figure 3.10.

NOTE: The `TrivialApplet.java` file is still in the AboutBox `Tutorial` folder. You can delete it if you want. But leaving it there will not affect your project.

Figure 3.10 Adding Java file to the project



Java Naming Conventions

The name of the Java file is important. The name of the file should be the same as the name of the class created in the file, and end with the `.java` extension. This extension is a Java programming convention for naming source files. The Metrowerks Java compiler uses this extension to recognize Java source files more easily.

NOTE: Actually, the Metrowerks Java compiler allows Java filenames to be different from the classes contain in them. This feature is useful on the Mac OS, which limits file names to 32 characters, which is too short for some Java class names. However, you should name a Java file after its class whenever possible, so your code is compatible with all Java compilers. If you must give a file a different name, choose a name that is as close as possible to the name of the class defined in that file.

Now when you compile the project, CodeWarrior will use the new code you have supplied in the `AboutBox.java` file. We are now ready to advance to the next section.

Examining the code

Open the `AboutBox.java` file and look at the source code.

NOTE: Do not make any changes to the source code yet.

Reading the comments in the code will help you understand the functionality of the applet. If you are not familiar with Java syntax, you may not understand all the code. That is OK. The purpose of this tutorial is not to teach you Java, but to introduce you to CodeWarrior. For a list of recommended Java documentation, see "Where to Go from Here" on page 14.

Editing the HTML File

CodeWarrior allows you to add files to a project which do not contain source code. This allows you to keep track of all files related to a project, even those that are not compiled.

While developing an applet, programmers normally use a small HTML file that embeds the applet code, which is used to view it as the applet is being developed. The project stationery for an applet project includes a sample file, `TrivialApplet.html` for this purpose.

When you choose **Project > Run**, the CodeWarrior IDE passes the first HTML file in the **Link Order** view of the project window to the applet viewer. The applet viewer uses the HTML file to pass parameters to the Java applet.

NOTE: For more information on either the File view or Link Order view of the CodeWarrior project window, see the *IDE User Guide*.

In this section, you edit the `TrivialApplet.html` file so that it contains what we need for our applet to function.

1. **Open the `TrivialApplet.html` file**

   Double-click the `TrivialApplet.html` file in the project window. The CodeWarrior IDE opens a new window titled "TriviaApplet.html" and displays the html.

2. **Edit the HTML.**

   Choose **Edit > Select All**. CodeWarrior selects all of the HTML. Press Delete on the keyboard to remove the HTML. Replace it with the HTML in Listing 3.1.

Listing 3.1 New HTML for the AboutBox applet

```
<html>
  <head><title>Java Tutorial</title></head>
  <body bgcolor="#F5DE93">
    <center>
      <applet codebase="AboutBox" code="AboutBox.class"
        width="319" height="99"
        image="AboutBox.gif"
        sound="metalstamp.au">
      </applet>
      <p>
        <font size="5"><b>metro</b>werks - </font>
        <font size="4"><i>Software at Work &#153;</i></font>
      </p>
    </center>
  </body>
</html>
```

The applet tag in the HTML code tells the web browser that the applet code can be found in a folder named `AboutBox`. It also sets the viewing size of the applet, and specifies the image file to use and the sound file to play.

3. **Save the new HTML under a different filename.**

   Choose **File > Save As…**. The editor displays a save file dialog. Rename the file "index.html". Navigate to the `AboutBox Tutorial` folder and click the **Save** button to save the file.

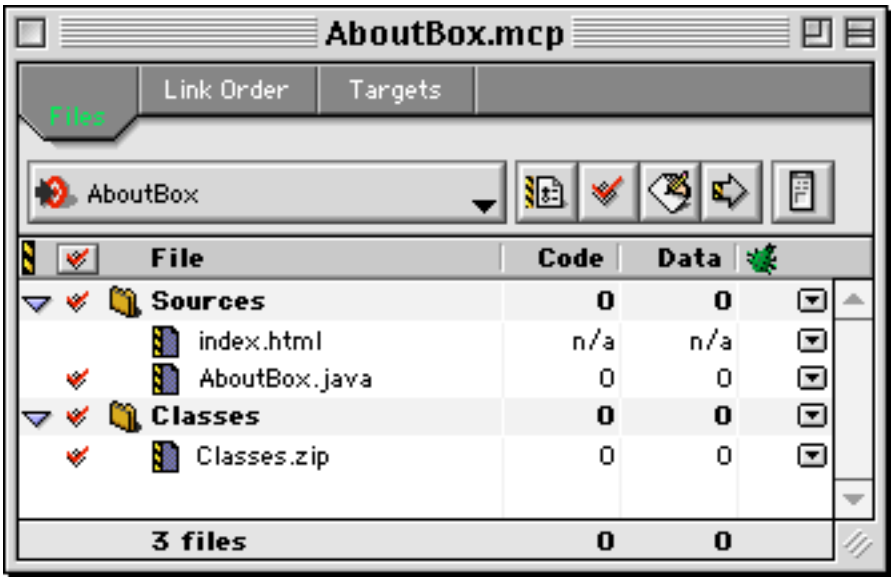
The file `TrivialApplet.java` is still in the AboutBox `Tutorial` folder. You can delete it if you want. But leaving it there will not affect your project.

4. **Remove the TrivialApplet.html file from the project**

   Select the TrivialApplet.html file in the project window, and choose **Project > Remove Selected Items** to remove it from the project.

   Your project window should now look similar to the one in Figure 3.11.

Figure 3.11 Project window with new HTML file



Now that all code and needed files have been included in the project, we are ready to proceed to the next section.

Compile and Run

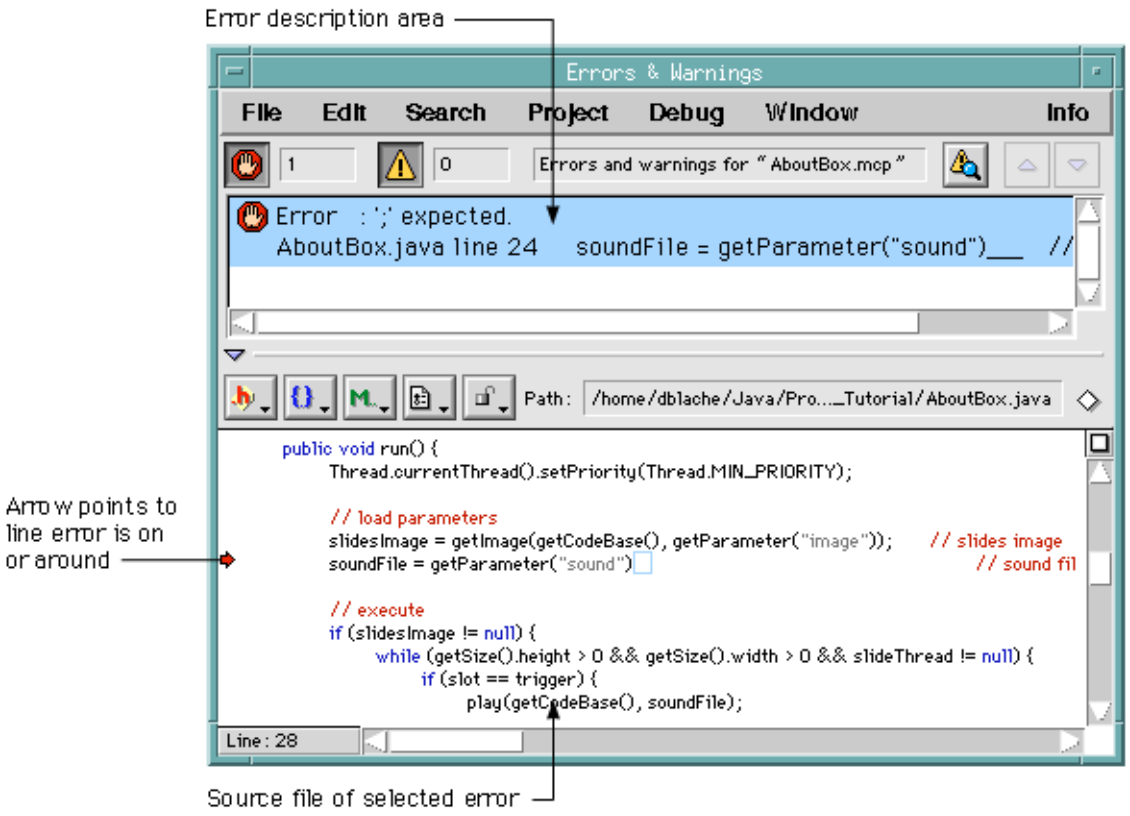
You are now near the final stages of your first Java applet, and this tutorial. All that is left to do is compile and test the applet to make sure it runs smoothly.

Compile

Choose **Project > Make**. CodeWarrior compiles and links your code into the finished applet, all in one step.

CodeWarrior is very smart about picking up syntax errors at compile time. Usually these errors are the result of mistyped or unused variable names, incorrect class information, or other related errors. The tutorial code has one deliberate error. CodeWarrior will display it now as shown in Figure 3.12.

Figure 3.12 The Error & Warnings window



The **Errors & Warnings** window displays a list of errors the compiler found when it tried to make the project. In this case there is only one error. The bottom part of this window displays the source file that contains the selected error. The red arrow points to the area

of code where the error was encountered. Sometimes the error is on the same line, but it can also be found a line or two before depending on the type of error.

Fix the Error

The source view of Errors & Warnings window is completely editable. You can edit and save your code directly from this window. There is no need to open up the source file and try to find the error manually. This is a great time saver if you have many errors.

In this case, the error is a missing semicolon (;) at the end of the line pointed to by the red arrow. We will fix this line and recompile the project.

1. **Correct the problem line**

   Click at the end of the problem line and type a semicolon (;).

2. **Save the corrected code**

   Then choose **File > Save** to save the file.

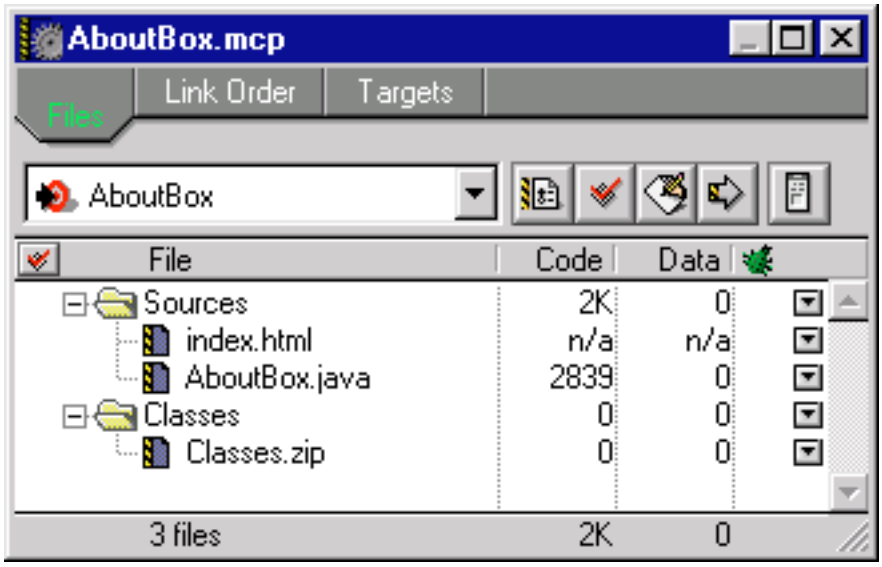
3. **Compile the project again**

   We are done with the Errors & Warnings window now. Close it. Then choose **Project > Make** to compile the project again.

   Note that while CodeWarrior is building your project, the check mark to the left of the files in the project window is erased, indicating that the files no longer need to be compiled. In the Code column, the number of bytes of code is updated. The number is zero for the `classes.zip` file because that is a shared library to which the applet links at runtime. This file contains the Java classes and methods called from the `AboutBox.java` source file.

   When the project is completely compiled and ready to run, the project window should look like Figure 3.13.

Figure 3.13 Compiled java project



Examine the Output

Take a minute to examine the class folder produced by CodeWarrior. It is in the project folder, and is named `AboutBox`. It contains three files:

AboutBox.class — the class file generated by the Make command

AboutBox.gif — the supplied image which is used by the applet

metalstamp.au — the supplied sound file used by the applet

NOTE: The applet parameters in the HTML we are using for this tutorial look for the images in the AboutBox folder, where the class file resides. For your convenience, we have placed the image file and sound file inside of this folder so that after a successful compile, you will be able to use the class folder without any additional work. Normally, however, you will need to ensure that you place any additional files such as images in the appropriate place so that your applet will run correctly.

Run the Applet

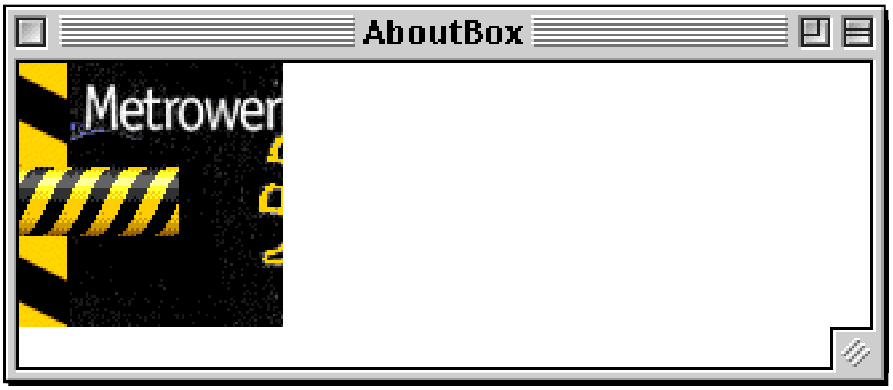
The last thing to do at this point is test whether the applet works as expected. Choose **Project > Run**. CodeWarrior launches the applet viewer chosen in the Java Target settings panel, passing it the HTML file in your project. The applet viewer then reads the HTML file, which tells it where to find the class file, image file, and sound file used in the applet.

There is a bug in the applet, which we will discuss in the next section.

Debugging the Applet

As you have probably already noticed, the applet appears to animate, and the metalstamp.au sound is played. But only part of the applet is being displayed, as shown in Figure 3.14. The bug that is causing this problem is probably in the method that does the drawing. So we will look at that part of the file now to see if we can spot the bad code.

Figure 3.14 Applet bug



Using the CodeWarrior Debugger

The CodeWarrior Debugger is integrated into the CodeWarrior IDE to help you debug your code. The same debugger is used for Java, C, C++, Pascal, and assembly language.

The debugger is called a "source level debugger" because it displays the source code so you can see exactly which line will be executed next. This lets you get a clearer understanding of your code, without the need to learn assembly language.

For more information on the debugger, see the *Debugger User Guide*.

1. **Enable the Debugger**

   Before you can debug the Java code, you must first tell CodeWarrior to use the debugger when running the applet. Choose **Project > Enable Debugger**. CodeWarrior prepares the project for debugging.

2. **Turn On Debugging for the Java File**

   In order for CodeWarrior to debug a file, the file must be marked for debugging in the project window. We want to mark the `AboutBox.java` file for debugging. To do this, click in the debug column next to `AboutBox.java`.

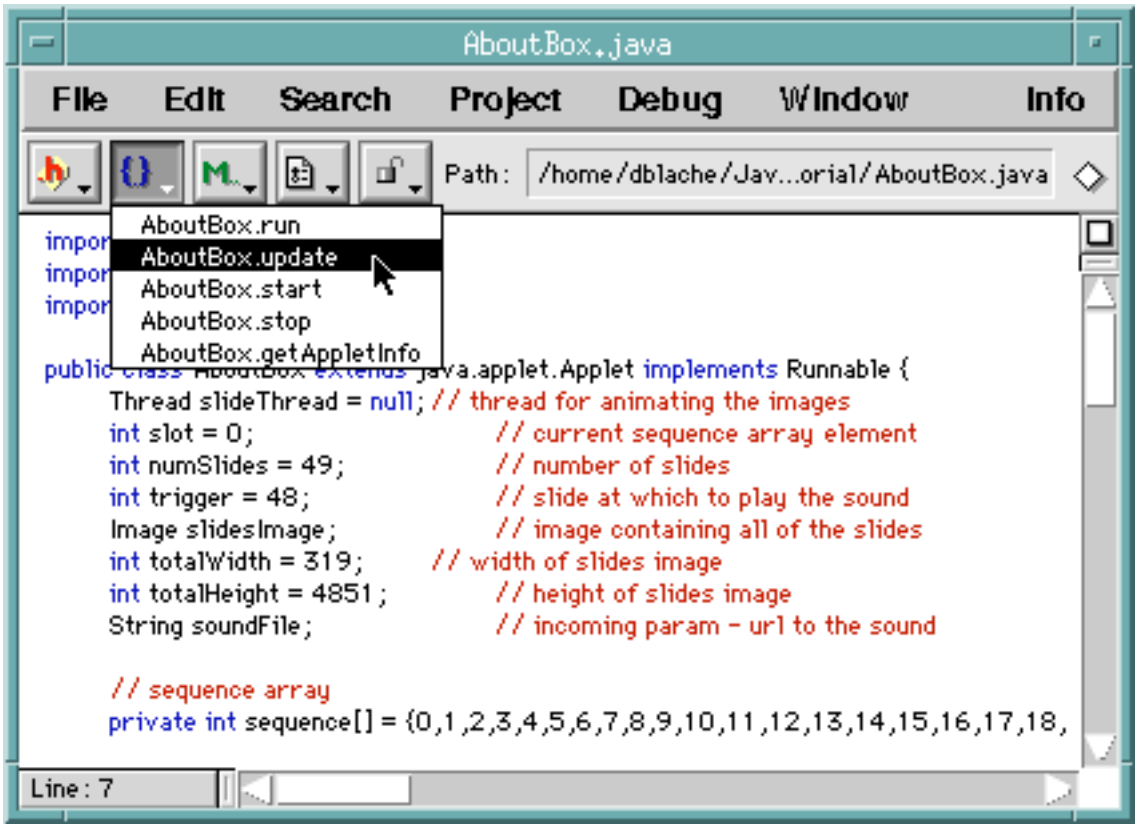
3. **Open the Java source code file**

   We know roughly where we can find the problem code in the Java source code file. Open the `AboutBox.java` file by double-clicking it in the project window.

4. **Locate the `update()` method**

   Find the `update()` method. An easy way to scroll a method into view in the CodeWarrior editor is to select the method name in the function pop-up menu as shown in Figure 3.15.

Figure 3.15 Using the function pop-up menu



5. Set a Breakpoint

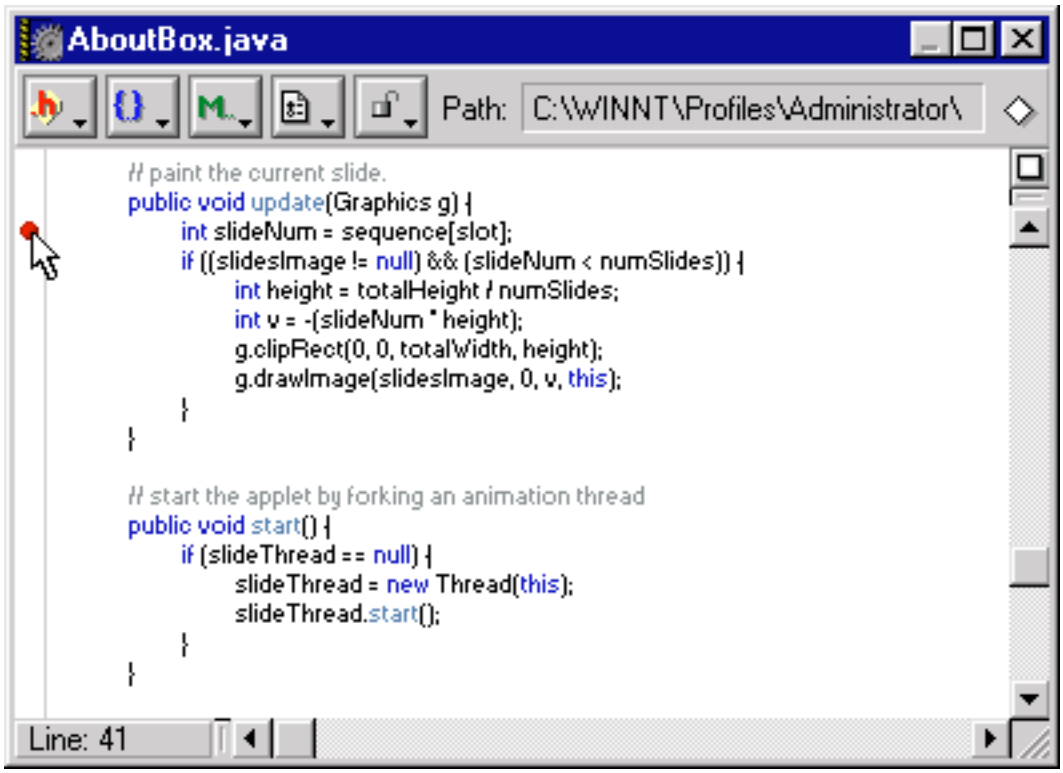
To set a breakpoint, click the breakpoint column on the left-hand side of the window, next to the line of source code you want execution to stop on. A red stop sign will appear in the column where you click.

In this case we want to set the breakpoint on the first line of code in the `update()` method:

```
int slideNum = sequence[slot];
```

Figure 3.16 shows how the editor window should look once the breakpoint has been set.

Figure 3.16 Setting a Breakpoint



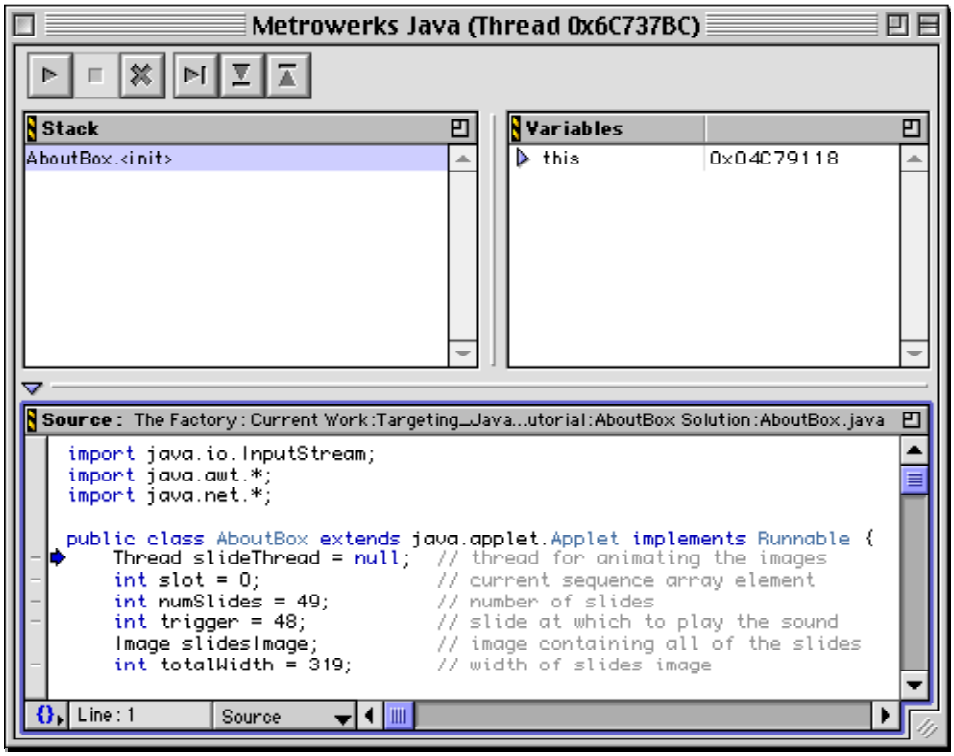
6. Start the Debugging Session

To start the debugging session, choose **Project > Debug**. CodeWarrior passes the HTML file to the debugger. The debugger launches the applet viewer chosen in the Java Target settings panel. The applet viewer then loads the applet and creates a window for it to run in. Finally, the debugger comes to the front, displaying the program window for the applet, with the current-statement arrow at the first line of code in the main class of applet, as shown in Figure 3.17.

The debugger allows you to examine variables, step through the program, set breakpoints, and perform other debugger functions. The top left pane is the **Stack** pane. This pane shows the order in which each method has been executed, called a *call chain*. The pane at the top right of the window is the **Variables** pane. It lists the variables in use by the current function, along with the current values of those variables. The bottom pane is the **Source** pane. It displays the

source code currently being executed. The arrow in the Source pane is called the *current statement arrow*. This arrow points to the line of code that will be executed next. See the *Debugger User Guide* for a more detailed description of the debugger windows.

Figure 3.17 Starting the debug session

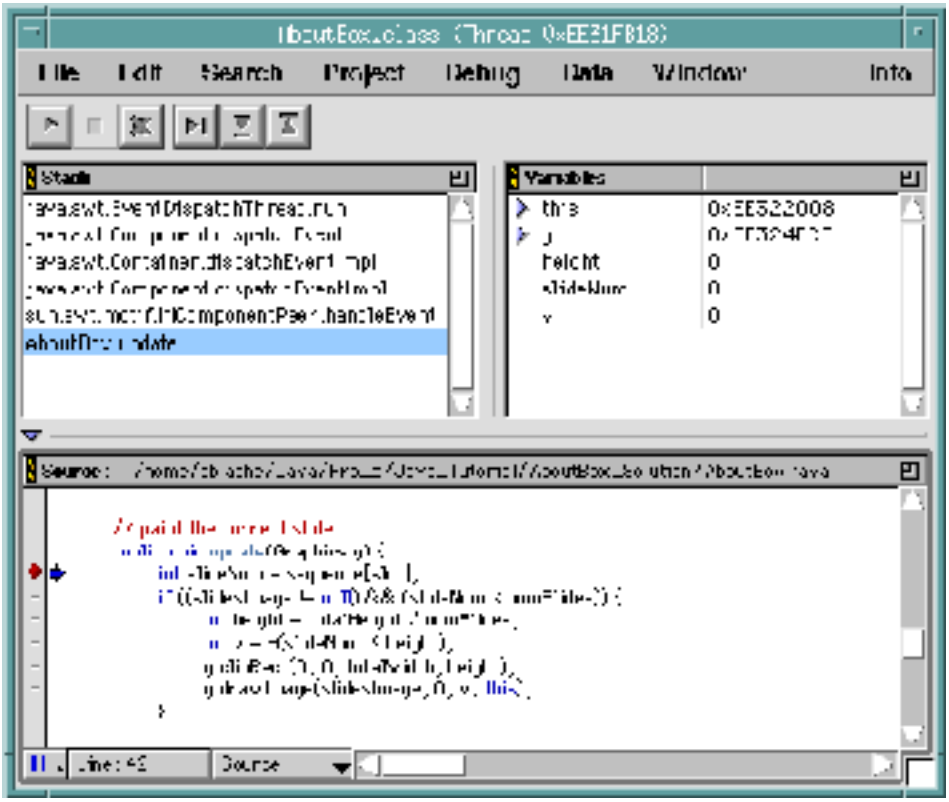


7. Jump to the Breakpoint

To get to the breakpoint we set earlier, we need to tell the debugger to run the applet. Do this by pressing the Run button (shown at the left).

CodeWarrior hides the debugger window, and the applet continues execution until the breakpoint is reached. Then the debugger window is displayed again, this time with the current-statement arrow at the line with the breakpoint, as shown in Figure 3.18.

Figure 3.18 Jumping to the breakpoint



8. **Step Through the code**

The debugger lets you step through the source code one line at a time. You do this by clicking the Step button (shown at the left).

Step through the code until you get to the following line:

```
g.clipRect(0, 0, height, totalWidth);
```

The Solution

Remember that the applet seems to run correctly except that the right side of the animation is not being displayed. If you open the image you will see that the image is the correct size. But only part of the image is being displayed in the applet. It is as if the right side of the animation is being clipped out of the display.

If we look up the `clipRect()` method in Sun's JDK 1.1.6 API documentation, we see the following description:

```
public abstract void clipRect(int x,
                              int y,
                              int width,
                              int height)
```

There is the problem! The order of the width and height parameters in *our* code is reversed in our code! We are supplying `height` where the width should go, and `totalWidth` where the height should go.

The image being used for this tutorial is 319 pixels wide, and 99 pixels *high*. Our `clipRect()` parameters are instructing the applet to clip to only 99 pixels *wide*.

Now we will stop the debugger and correct the problem line in the Java source code.

1. **Stop the debugger**

   To stop the debugger, click the Stop button (shown at the left). The applet viewer will terminate the applet, and will return control to the debugger. The debugger will then close it's window and return control to the CodeWarrior IDE.

2. **Open the Java source code file**

   If the `AboutBox.java` file was open prior to the debugging session, the CodeWarrior IDE displays it once again. If the `AboutBox.java` file is not open, open it now by double-clicking it in the project window.

3. **Edit the source code**

   Edit this line:

```
g.clipRect(0, 0, height, totalWidth);
```

   Change it so that it reads:

```
g.clipRect(0, 0, totalWidth, height);
```

4. **Save the changes**

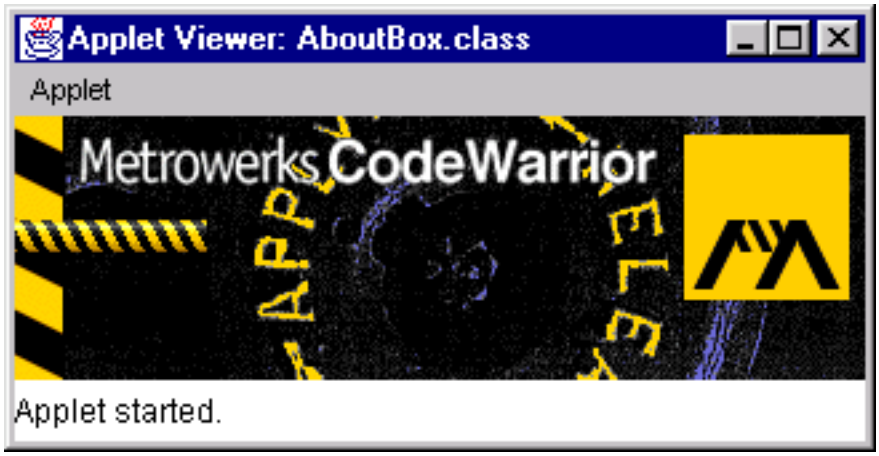
   Choose **File > Save** to save the correction.

5. **Run the applet**

   Choose **Project > Run** to run the applet again.

   The applet will now display properly as in Figure 3.19.

Figure 3.19 Applet running properly



You have just successfully completed your first Java applet using the CodeWarrior IDE!

Exercise

There is still more that can be done with this applet. This section discusses some improvements that can be made to the applet. As we progress in this section, you can check your progress against the files in the `AboutBox Excercise` folder if you are unsure of anything.

Currently, the number of slides, the sequence in which to play the slides, the slide to play the sound on, and the dimensions of the image file are all "hard coded" into the applet. If we want to change any of these things, we have to recompile the applet.

It would be much better if these things were supplied as parameters in the HTML file, just as the sound file and image file are now. To make it so, we need to change quite a few things in the Java source file.

Number of Slides Parameter

Rather than having a predetermined number of slides, the user will specify how many slides are in the slide show via a "`slides`" parameter in the `applet` tag in the HTML file.

The following line of code in the Java file sets the value of the `numSlides` variable:

```
int numSlides = 49;
```

Change this to:

```
int numSlides = 0;
```

This makes the default number of slides zero. To get the `slides` parameter, add the following line to the end of the section that loads the parameters:

```
numSlides = Integer.parseInt(getParameter("slides"));
```

Slides Image Size and Orientation Parameter

The current applet has no provisions for a slides image with a different width or height. It also has no provisions for using a different orientation in the slides image - it assumes that the slides will be stacked vertically in the slides image. We want the applet to be more flexible in this respect.

First, we will make provisions for a new parameter called `orientation`. Add the following line to the top of the file where variables are initialized:

```
String orientation;
```

Then add the following line to the section that loads parameters:

```
orientation = getParameter("orientation");
```

The `totalWidth` and `totalHeight` variables are currently set to `319` and `4851` respectively when the applet loads. This is done at the top of the file with the following lines:

```
//int totalWidth = 319;
//int totalHeight = 4851;
```

Change these lines to:

```
  int totalWidth = 0;
  int totalHeight = 0;
```

Just under the section of code that loads parameters, add the following code. This code calculates the values for the `totalWidth` and `totalHeight` variables based on the orientation, the size of the applet, and the number of slides in the slide show.

```
    // calc total height & width
    if ((orientation.equals("v")) |
(orientation.equals("vertical"))) {
      totalHeight = getSize().height * numSlides;
    } else {
```

```
      totalHeight = getSize().height;
    }
    if ((orientation.equals("h")) |
(orientation.equals("horizontal"))) {
      totalWidth = getSize().width * numSlides;
    } else {
      totalWidth = getSize().width;
    }
```

Now the applet will calculate the image width and height based on the number and orientation of the slides and the size of the applet.

Sound Slide Trigger Parameter

The sound file is currently played when slide number 47 is displayed. And the only way to change that is to recompile the applet. It would be a lot better to be able to specify this value in the HTML file that calls the applet.

The following line at the top of the applet sets the value of the `trigger` variable to 47 when the applet loads:

```
int trigger = 47;
```

Change this line to:

```
int trigger = 0;
```

Add the following line to the section of the code that loads the parameters:

```
trigger = Integer.parseInt(getParameter("trigger"));
```

Now the user may specify the sound trigger via a "`trigger`" parameter in the `applet` tag in the HTML file.

Slide Sequence Parameter

The last thing we want to do is allow the user to specify the slide sequence via a sequence tag in the HTML file.

The following line at the top of the Java file defines and initializes the sequence array:

```
//private int sequence[] =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,
47,48,49,48,49};
```

Change this line to:

```
private int[] sequence;
```

Add the following code just after the section of code that calculates the total height and width of the image:

```
    // generate the sequence array
    StringTokenizer st = new
StringTokenizer(getParameter("sequence"),",");
    sequence = new int[st.countTokens() + 1];
    int x = 1;
    while(st.hasMoreTokens()) {
      sequence[x] = Integer.parseInt(st.nextToken());
      x = x + 1;
    }
```

The code above uses the StringTokenizer class to parse the sequence string and extract the elements of the array. StringTokenizer is part of the java.util package, which is not currently imported in the Java file. Add the following line to the very top of the file:

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

Now the sequence can be specified via a `sequence` tag in the `applet` tag in the HTML file.

Using HTML Tags to Supply the New Parameters

To supply the new parameters we just made provisions for, you will add new parameters to the applet tag in the HTML file. Four new parameters are required: "`slides`", "`orientation`", "`trigger`", and "`sequence`". Open the index.html file. This is the current applet tag:

```
<applet codebase="AboutBox" code="AboutBox.class" width="319"
height="99" image="AboutBox.gif" sound="metalstamp.au">
</applet>
```

Change it to:

```
<applet codebase="AboutBox" code="AboutBox.class" width="319"
height="99" image="AboutBox.gif" sound="metalstamp.au" slides="49"
orientation="vertical" trigger="47"
sequence="0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,
44,45,46,47,48,49,48,49">
</applet>
```

That is it. No more modifications are required.

Choose **Project > Run** to make the project and run the new applet.

There are still things that can be done to this applet to make it better. For instance, little checking is done to verify that the incoming parameters are valid.

4

Creating Java Projects

This chapter explains how to create and manage any Java project, and how to set the options in the Java Project settings panels.

This chapter contains these sections:

- Types of Java Projects describes the types of Java projects you can create with CodeWarrior.
- Using Project Stationery describes each project stationery option in the New Projects dialog box.
- Working with Java in CodeWarrior describes how to create and manage any Java project from start to finish.
- Using the classes.zip Library describes what is in classes.zip, the library file that is in most every Java project, and how to add it to a project.

Types of Java Projects

This section discusses the various types of code objects that you can create for Java. They include the following:

- Applets
- Applications
- Libraries

Applets

An Applet is a small Java application that runs over the web, usually through a web browser. Applets can consist of multiple class files or a single Jar file.

Applications

Java Applications are executables that usually run through a Java Interpreter. Java Applications can consist of multiple class files or a single Jar file.

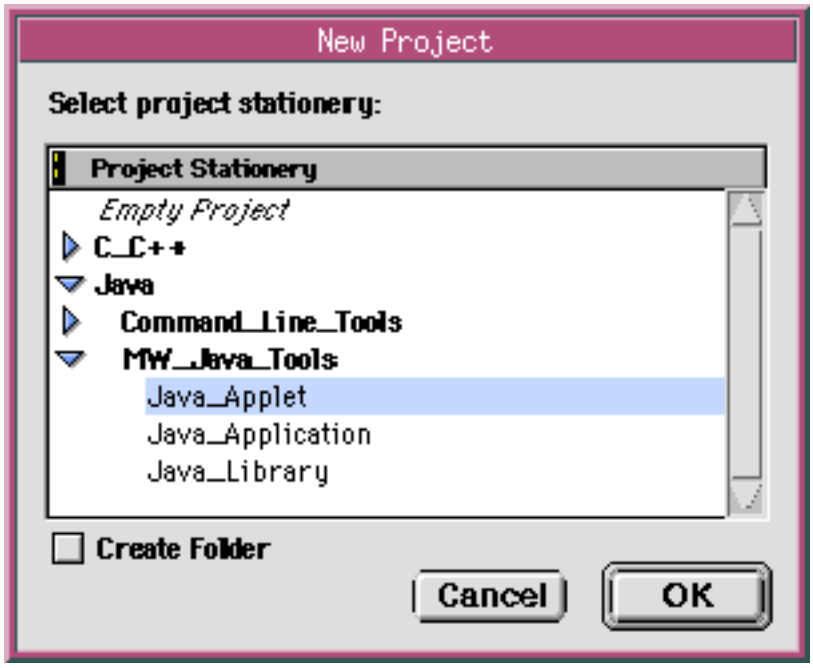
Libraries

A Java Library is a collection of Java classes that can be used in many projects. Libraries are usually compiled into zip files (like `classes.zip`) or Jar files.

Using Project Stationery

The **New Project** dialog box (Figure 4.1) displays a list of several different pre-configured projects and their associated files, called *Project Stationery*.

Figure 4.1 New Project dialog box



Each set of project stationery is pre-configured for different types of Java projects. This saves you time when creating your own project files as you only need to change a few options instead of setting up a project from scratch.

Table Table 4.1 lists each project stationery option.

Table 4.1 Project stationery options

Stationery	Description
Java Applet	Pre-configured settings for a Java applet. Output is a class folder.
Java Application	Pre-configured settings for a Java application. Output is a Jar file.
Java App-Applet	A multi-target project (Applet & Application) that demonstrates how to use `AppletFrame.java` to turn your applet into an application.
Java Library	Pre-configured settings for a Java library. Output is a Jar file.

NOTE: Project stationery is configurable by the user. The supplied stationery may vary between releases of CodeWarrior. Because of this, what you see in the New Project window may be slightly different from the pictures in this manual.

Working with Java in CodeWarrior

This section describes how to create, run, and debug any kind of Java project, including applets, applications, and libraries. The topics in this section are:

- Creating a New Java Project
- Creating Java Code
- Changing Settings

- Running a Java Project
- Debugging a Java Project
- Kinds of Application Projects

Creating a New Java Project

The following procedure describes how to create a new Java project. The project can be either an applet, an application, or a library of Java code.

1. **Create a new project.**

   Choose **File > New Project**. CodeWarrior displays the New Project dialog box.

2. **Select project stationery.**

   In the New Project dialog box, select the proper stationery for the kind of project you want to create: applet, application, or library. To learn more about various kinds of application projects, see "Kinds of Application Projects" on page 67.

   You may also decide whether to have CodeWarrior create a new folder for you in which to place all your files. Typically you want to create a new folder.

   When you click the **OK** button, CodeWarrior displays a file dialog.

3. **Name the project.**

   In the file naming dialog, enter a name for the project, and click **OK**.

   If you use a suffix on the Windows hosted CodeWarrior, it must be `.mcp`. If you do not use a suffix, CodeWarrior adds it for you.

TIP: If you want your project to be recognized by all versions of CodeWarrior, the project suffix *must* be `.mcp`.

If the **Create Folder** check box was enabled in the New Project dialog box, CodeWarrior creates a new folder. Inside that folder, CodeWarrior puts a new project file (with all project settings identical to the stationery project you selected), and duplicates any files included in that stationery project.

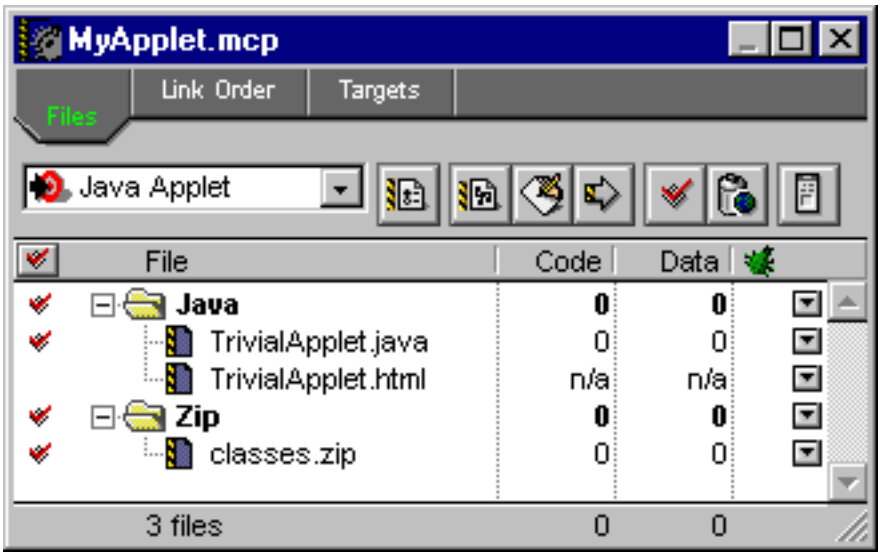
For example, for an applet, you get a new project file, an HTML file, named `TrivialApplet.html`, and a Java source file named `TrivialApplet.java`.

The project file has the name you entered in the dialog box. The new folder has the same name as the project file without any naming extension.

WARNING! The project manager also creates a directory named *MyProject* Data, where *MyProject* is the name of your project. CodeWarrior uses this directory to store internal data. Do not modify or move any file in this directory.

After creating the files and folder, CodeWarrior displays a new project window, like the one in Figure 4.1.

Figure 4.2 The Java Applet project



The stationery projects hold example files. In typical use you modify these files and save them with a new name, or replace them with your own files.

The `classes.zip` file is a shared library of standard Java classes your applet can use. For more information, see "Using the classes.zip Library" on page 68.

Creating Java Code

To create Java code, use the CodeWarrior tools: the editor, the browser, and the project manager. For details on these tools, see the *IDE User Guide*.

In typical practice, you start with a project based on stationery, which includes example or "placeholder" files. To learn how to create a project in this way, see "Creating a New Java Project" on page 62.

For example, if you are creating an applet based on the Java applet stationery, there are two such files: `TrivialApplet.java` and `TrivialApplet.html`.

The `TrivialApplet.java` file contains a sample Java applet. You can open the file, replace its contents with your own code, and save the file under a new name. You can add any other source file as well.

NOTE: Each Java file must contain only one public class. The file name must be the same name as the public class with the suffix `.java`. For example, if the name of your class is MyClass, the Java file must be named `MyClass.java`. If you do not follow this convention, CodeWarrior may compile your project anyway, but other Java compilers may not.

Similarly, the `TrivialApplet.html` file contains sample HTML code. You can modify this file so that it displays your applet. You typically specify both the location of the applet (a folder name) and the name of the class containing the main function. You can set the name of your classes folder by opening the **Target Settings** dialog box, and modifying the name specified in the **Output** panel. See "Target Settings" on page 92 and "Java Output" on page 112 for more information.

Changing Settings

If you use project stationery as a starting point, you generally will not need to change many of the settings. The default settings are perfect for most projects. However, this is not the case for all settings or all projects. This is especially true if you have more than one target in a project. You can modify a variety of settings for Java projects. Each panel is explained in detail in "Target Settings for Java" on page 91.

Common settings you might change are the main class (for Java applications) and the output name (the name of your final program). For applications you may also choose whether to output your files in a class folder, or in a zip file.

Specifying the main class

To tell CodeWarrior the name of your main class for a Java application, go to the **Java Target** panel, make sure that you are creating an application, and set the name in the **Main Class** field in the panel. For more information, see "Main Class" on page 98.

Specifying output name

To tell CodeWarrior what to name your final output file (be it an applet or an application), go to the **Java Output** panel and enter the name in the **Name** field. For more information, see "Java Output" on page 112.

Specifying output type

To tell CodeWarrior whether to create a class folder or a Jar file, go to the **Java Target** panel and set the Output Type option. For more information, see "Java Target" on page 95.

Running a Java Project

You may run applets and applications. You cannot run libraries. To run a project, choose **Project > Run**. At that time, CodeWarrior compiles and links any changed files, and launches your program. Precisely how your program launches depends upon whether it is an applet or an application.

Running a Java applet

When developing an applet, you typically have a small HTML file that embeds the applet code. CodeWarrior uses this file to run your program from within the IDE. When you choose **Project > Run**, CodeWarrior opens the first HTML file in the Link Order view with an HTML browser or applet viewer. In other words, the order of HTML files in the File view in the Project Window does not matter. It is the order as shown in the Link Order view in the Project Window.

NOTE: For applet projects, if there is no HTML file in your project, the **Run** command is disabled.

You can add as many HTML files as you want to a project. If your applet is complex, you may want several HTML files that test different aspects of your applet. To change the HTML file that CodeWarrior uses, simply rearrange the files in the Link Order view of your project window.

NOTE: You can also run an applet by opening one of its HTML files with any Java-enabled browser.

Running a Java application

To run an application, choose **Project > Run** in the CodeWarrior IDE. The Virtual Machine chosen in the Java Target settings panel loads the application's code and runs the `main` method of the application's main class. If you entered a value in the **Main Class** field in the Java Target settings panel, it will use that class. Otherwise, it will assume the main class is the first class in the first file in the project window. For more information on how to specify the main class, see "Main Class" on page 98.

NOTE: On the Mac OS, if you run a droplet by dragging and dropping files onto it, the full pathnames of those files are passed as arguments to the main class's `main` method. Note that if you

run a droplet by double-clicking it or choosing **Project > Run**, you cannot specify arguments for it.

Debugging a Java Project

You debug an applet project the same way you debug any other project. For more information on the CodeWarrior debugger, consult the *IDE User Guide*. For more information on Java-specific features in the debugger, see "Debugging Java Projects" on page 71.

To debug a Java project, follow these steps:

1. **Turn on debugging.**

   This step can be skipped if debugging is already enabled.

   Choose **Project > Enable Debugger**. You may see a dialog box telling you that certain settings must be modified for debugging. Click the **OK** button, and CodeWarrior sets up debugging automatically. In the project window, black dots appear in the debugging column next to the source files.

2. **Run the project under the debugger.**

   Choose **Project > Debug**. CodeWarrior compiles any changed files and generates debugging information for all files that have a black dot in the debugging column of the project window. CodeWarrior launches your project using the appropriate mechanism for your project. For a discussion of this process, see "Running a Java Project" on page 65.

   Once the project is running, control returns to the debugger. You can now debug the applet like you would any other program. For more information on the debugger, see the *Debugger User Guide*. For more information on Java-specific features in the debugger, see "Debugging Java Projects" on page 71.

Kinds of Application Projects

CodeWarrior lets you create Java applications that you can run without any applet viewer or browser.

These are the other types of applications you might create for specific purposes:

- A *Jar file* is an archive of all your classes. It is a convenient way to transfer and organize all the class files and any associated other files in one package.
- A *class folder* is a folder that contains all your classes. Choose it if you need to run the application on computer platforms that cannot use zip files.

All these files types contain Java bytecodes that will run on any computer platform that supports Java. However, standalone application files use special utilities to launch them without the help of java-enabled browsers.

Use an application when you will use the file only on a Mac OS computer. Use a jar file or class folder when you will use the file on other computer platforms as well.

Using the classes.zip Library

The `classes.zip` file is a library of Java classes that all Java programs share. It includes all the packages in the Java API, such as the Abstract Windowing Toolkit (AWT), I/O, and Applet packages.

The location depends on where JDK is installed on your machine. Following is a list of the locations where the classes.zip file can normally be found:

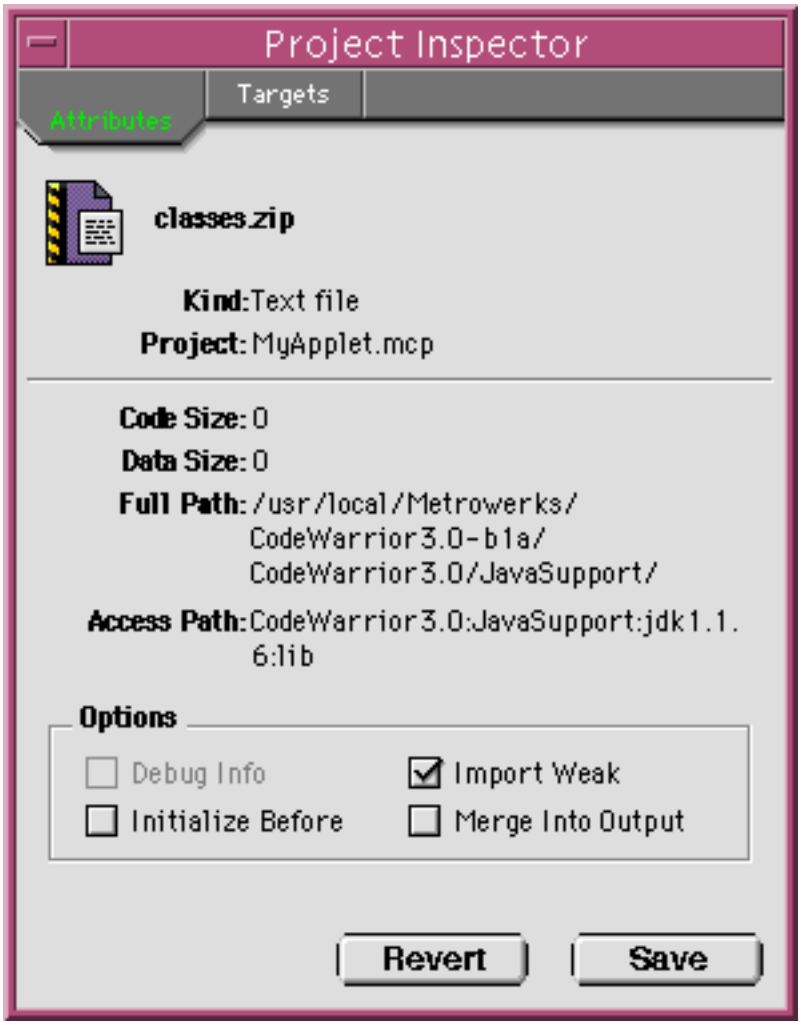
Windows inside the `Java` folder of the System folder

Mac OS in a Classes folder in a subfolder of your Extensions folder

Solaris in the lib directory of the JDK directory

Almost every Java program that you create uses `classes.zip`. However, `classes.zip` is a large file (about 1.5 megabytes). So you probably do not want to copy it into every one of your Java applications and applets.

Figure 4.3 The Import Weak Classes Setting



To instruct CodeWarrior not to merge the `Classes.zip` file into the final output, first select the `Classes.zip` file in the project window. Then choose **Window > Project Inspector** to open the **Project Inspector**. Make sure that the **Merge Into Output** checkbox is not checked. CodeWarrior do not copy the library into your finished program.

To enable your project to compile without the `Classes.zip` library, make sure that the **Import Weak** checkbox is checked. With

these settings, your program will automatically look for the library on the host system when it is executed.

The **Project Inspector** window should now appear as in Figure 4.3.

`classes.zip` is automatically set up this way for you whenever you create a project with Metrowerks-supplied project stationery,

---

NOTE: The **Initialize Before** option is ignored by the CodeWarrior Java compiler. **Merge Into Output** will add `Classes.zip` to the final binary.

5

Debugging Java Projects

The CodeWarrior debugger lets you debug Java code as easily as C/C++ or Pascal code. This chapter assumes you are already familiar with the CodeWarrior debugger. If you need more information about the debugger, see the *IDE User Guide*. If you want a quick tutorial showing you how to use the debugger with Java, see "Programming Tutorial for Java" on page 25. If you want step-by-step information on running a Java project under the debugger, see "Debugging a Java Project" on page 67.

This chapter contains the following sections:

- Debugger Features and Limitations describes some of the features in the debugger that help you debug Java code and some limitations of the debugger.
- Special Debugger Features for Java goes into more detail about some special debugger features for Java.

NOTE: To debug Java source on Windows, you must install Microsoft Internet Explorer or the Sun JDK. Both are included on your CodeWarrior CD.

Debugger Features and Limitations

The CodeWarrior Debugger contains many features that are specifically for debugging Java projects. It lets you:

- Debug Jar files, class files and applications.
- View a disassembly of the Java Virtual Machine instructions for your class, described in "Viewing the Java VM Disassembly" on page 75.
- Debug multi-threaded programs, as described in "Debugging Threads" on page 74.
- Break on Java exceptions, described in "Breaking on Java Exceptions" on page 73.

The debugger has a few limitations:

- Due to a limitation in the Sun 1.1.6 VM, Jar file debugging is not supported when using version 1.1.6 of Sun's VM. JDK 1.2 and later does support jar file debugging.
- On Windows and Solaris, multi-language stepping is not supported. When debugging a Java program with native C methods in it, the debugger treats it as a Java program and it will not be able to step into any native code.
- On the Macintosh, multi-language stepping is not fully supported. If you implement some native methods in C, you can open both the Java class files and the C SYM files for the native methods in the debugger at the same time. And if you set breakpoints in both the C and Java files, the debugger will automatically break at those statements. However, the stack crawl window does not show the call chain across languages and you cannot single step from C code to Java code or from Java code to C code.
- The debugger cannot debug compressed Jar or compressed zip files. This includes Java applications that are in compressed Jar or compressed zip format.
- When using the new Sun Java Debugger plugin, you will not be able to debug external stand-alone Java libraries. For instructions on how to debug Java code that does not originate within your Java project, see "Debugging External Java Sessions (Windows Only)" on page 78.

Special Debugger Features for Java

This section explains some of the special CodeWarrior Debugger features for Java. It contains the following:

Breaking on Java Exceptions

The CodeWarrior debugger can automatically break when your class throws an exception. Just choose one of these options from the **Control > Break on Java exceptions** hierarchical menu.

Option	Debugger breaks on exceptions from…
All exceptions	Any class, including the Java API classes.
No exceptions	No classes.
Exceptions in targeted classes	Only those classes in the file that you opened. *(This option is available only on Mac OS.)*

Opening Multiple Class Files in One Browser

If you are debugging a project that creates multiple class files, you can view all the class files for a folder in one Browser window. Just turn on the **Debug all class files in directory hierarchy** option in the **Global Settings** IDE Preferences panel. When you open a class file, the Browser window displays the classes for that file as well as for all the files in the same folder and its subfolders. If this option is off, the Browser window displays the classes for one class file only.

Choosing a Java Applet Viewer for Debugging

When debugging a Java applet, the applet runs within an applet viewer or a Java-aware HTML browser like such as Internet Explorer. To learn how to specify the applet viewer, see "Applet" on page 95.

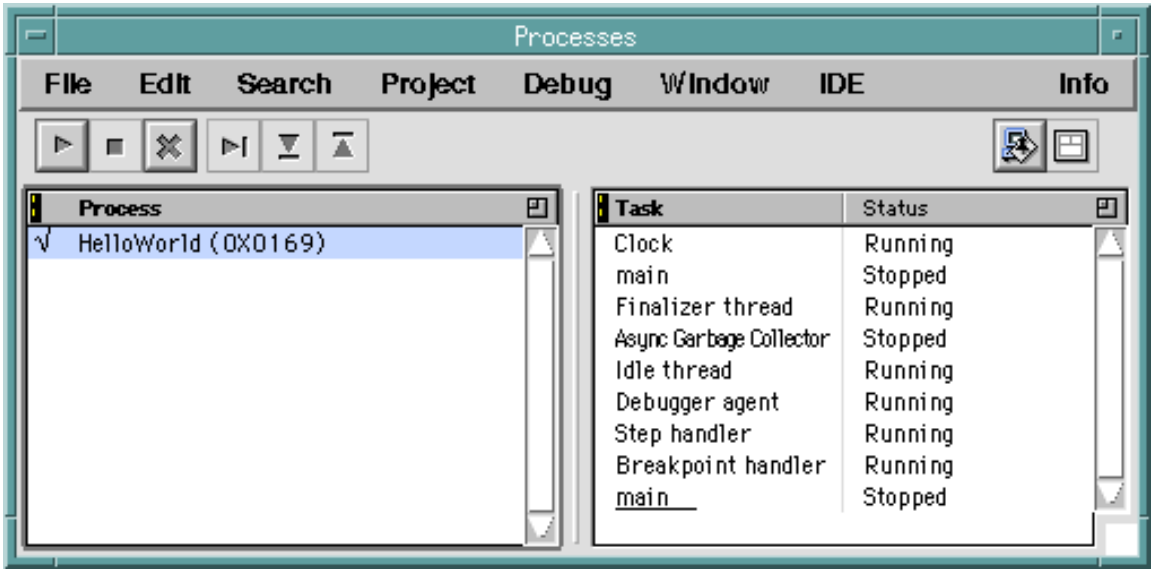
NOTE: Although many applications can run an applet on Mac OS, you can only use Apple MRJ to run an applet under the Metrowerks debugger. This is because the debugger requires the application to support Metrowerks's Java debugging API, and this is the only Virtual Machine that does so at this time.

Debugging Threads

When you are debugging a Java program, the running process is either the specified applet viewer, or the Java interpreter.

You can view all the threads in a Java program in the **Processes** window. Choose **Window > Show Processes** and select the specified applet viewer, or the Java interpreter process in the left pane. The Java threads appears in the pane to the right.

Figure 5.1 The Processes Window



To display a debugger Program window for a thread, double-click it. To pause, continue, or kill a thread, select it and use the buttons in the Processes window.

NOTE: Many of the threads listed in the processes window are spawned by the applet viewer or interpreter running the Java program, or threads for the CodeWarrior Debugger. Pausing or killing one of them will have unforeseeable (and possibly quite unfortunate) consequences.

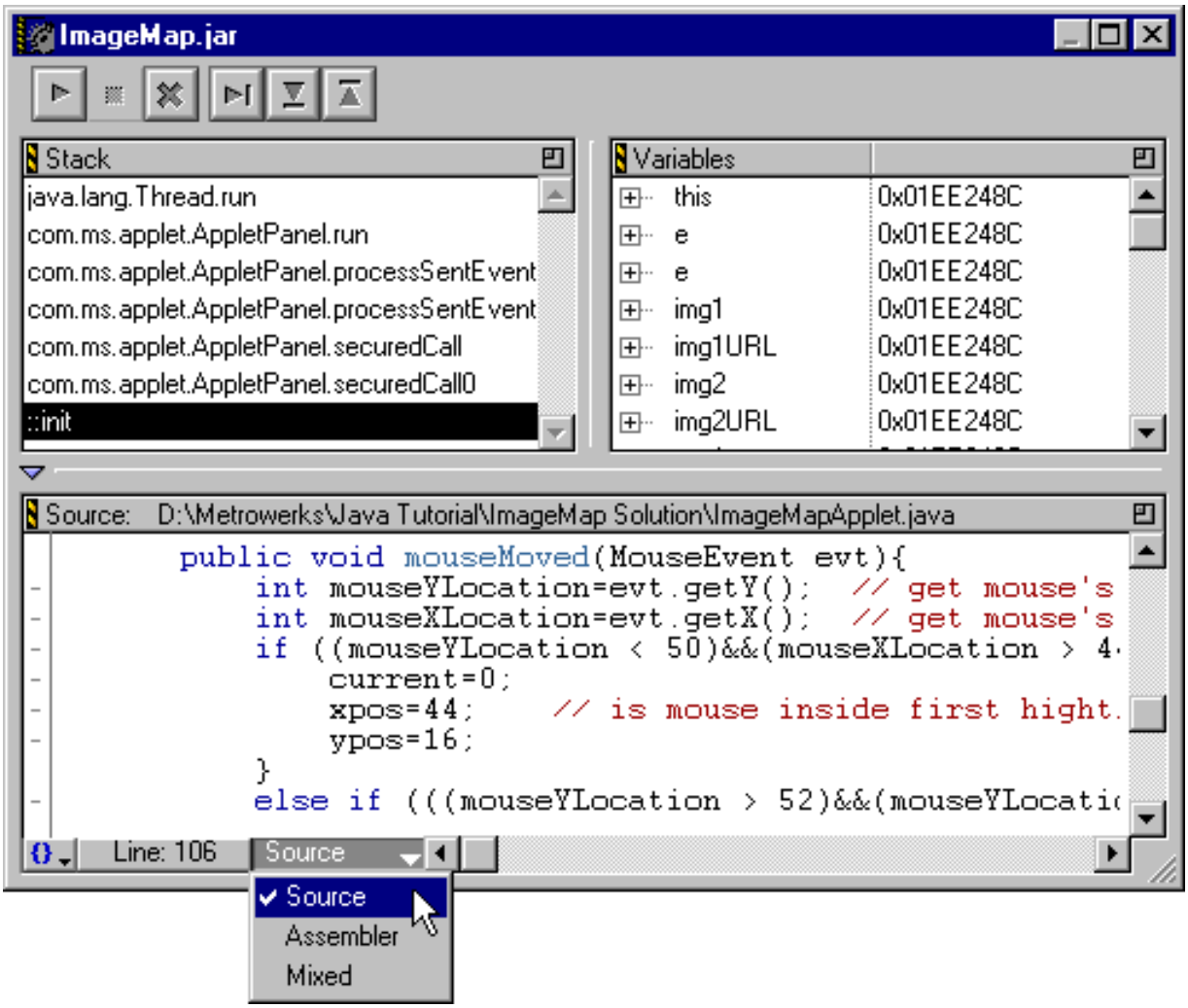
Viewing the Java VM Disassembly

You can view a listing of the Java Virtual Machine instructions that implement the class. Choose **Assembler** or **Mixed** from the **Source** pop-up menu at the bottom of any Browser or Program window (Figure 5.2).

The mixed view shows your Java source first, then the assembly language instructions that make up the Java code immediately after.

See also "Disassembling Classes" on page 151.

Figure 5.2 Viewing Java Assembly

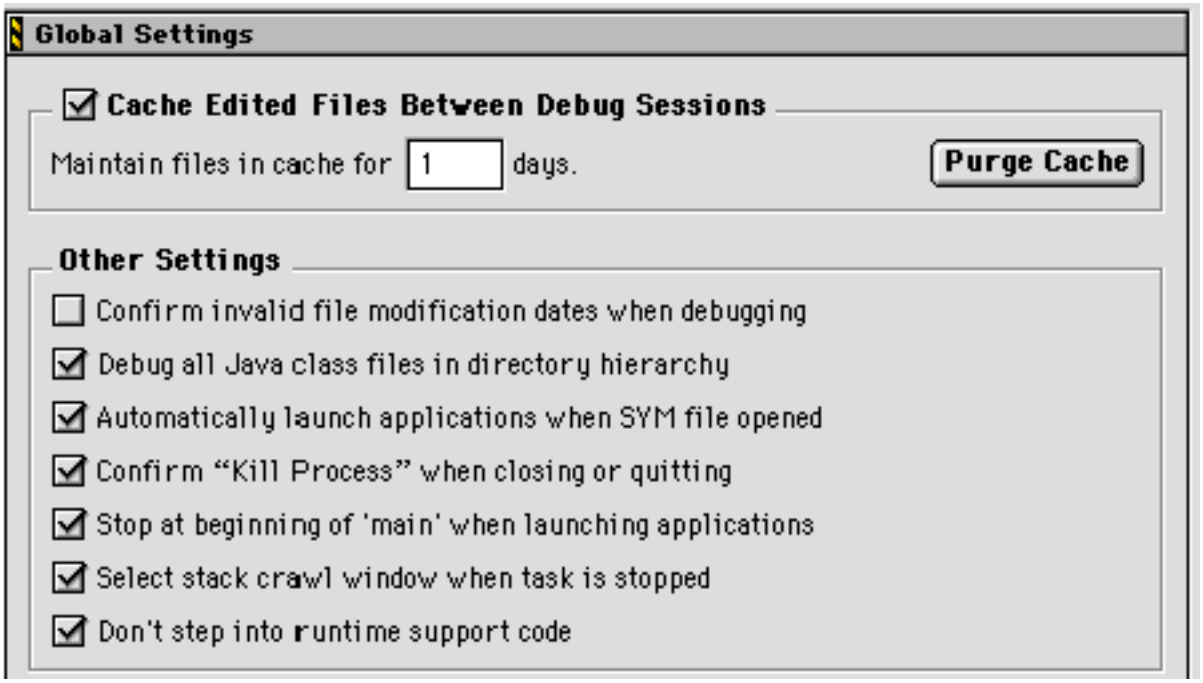


When viewing the assembly, you can still set breakpoints, step through code, and view variables. You can also view the assembly for a Java file in CodeWarrior by choosing **Project > Disassemble**.

Specifying Java Debugger Settings

There is one setting in Global Settings panel of the debugger group in the main IDE Preferences dialog that applies to Java: **Debug all Java class files in directory hierarchy**.

Figure 5.3 Global Settings Panel



The **Debug all class files in directory hierarchy** option opens all the class files in the directory and all contained directories, and merges them all together in the same Browser window.

Debugging External Java Sessions (Windows Only)

This section discusses how to launch and debug a Java application that runs within a native Windows application.

In the text that follows, we make several references to a folder named `VM Launcher Example` on the CodeWarrior CD. It is in the following location.

```
CodeWarrior Examples/CodeWarrior Java/
```

About Sun VM Debugging

To better understand what is needed to debug an external Java session with CodeWarrior using the Sun Java VM, it is important that you know the following:

- Sun VM debugging sessions are done over TCP/IP sockets.
- In order to debug a Java VM, the VM must be launched with special switches that make it responsive to a debugger. These switches can be stored in an environment variable. This environment variable must be set before the VM is launched.
- There is no way to detect whether there are any VM sessions running in debug mode at any particular time, or which of them will accept a connection from a debugger.

Because of these things, attaching to a running VM requires that the following be true:

1. The VM must be launched to be debuggable on a particular TCP/IP port.
2. The debugger must attach to the known TCP/IP port once the VM has been launched.

So, you can see that attaching to a running Java VM is not as seamless as attaching to a native Windows executable.

Registering TCP/IP Debugger Ports

The first thing you must do is configure CodeWarrior so that it will allow you to attach the Java debugger to a TCP/IP port. You must define one or more ports that CodeWarrior will use for debugging.

The way to define the port(s) is via the registry. The `VM Launcher Example` folder contains a file named `DebugPorts.reg` for this purpose. Examine the file in **Notepad** by right-clicking the file and choosing **Edit** from the resulting pop-up menu. The file should appear similar to Listing 5.1. Edit the port numbers to suit your needs. Then save and close the file.

Listing 5.1 DebugPorts.reg

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Metrowerks\CodeWarrior\4.0\Java VMs]
“JDWP debug ports”=”8000,8001,8002,8003”
“11XWP debug ports”=”8004,8005,8006,8007”
```

JDK versions 1.1.x, 1.2, and 1.2.x support 11XWP debugging ports. Only JDK version 1.2.2 with JBug installed supports JDWP debugging ports. The value data must consist of one or more port numbers, separated by commas. For more information on debugging with JDWP, see the Sun Java Debugger plugin release notes.

To add the contents of `DebugPorts.reg` to the registry, simply double-click the file. Windows creates the appropriate key in the registry, and adds the values to it.

WARNING! TCP/IP port numbers are limited to the range of 0-65535. Ports below 8000 are reserved and should not be used. So, you can use any port above 8000, assuming no other application on your system is using that port, too.

Instructing the VM to Run in Debug Mode

The next step is to ensure that when the native application launches a VM session, it does so in debug mode. Modifying a native application itself to launch the VM in debug mode is not practical. Instead, we use environment variable `_JAVA_OPTIONS`, which is new in JDK 1.2. Options defined via this environment variable will be used any time a VM session is instantiated.

Because you may not want to always run Java sessions in debug mode (for performance and security reasons), you will usually want to specify the `_JAVA_OPTIONS` on a per session basis. The best way to do this is to use a command (`.cmd`) file.

The `VM Launcher Example` folder contains two example command files. One is to be used with a standard non-debug JDK 1.2; the other is for a JDK 1.2.2 installation that has the JBug binaries installed (and hence supports the new Sun debugger specification).

NOTE: The command files require a minor modification to work correctly. The path to JDK should be changed to the location of JDK on your particular system. Examine the command files for more information.

Instead of launching `VMLaunch.exe` directly, you will launch these command files. The command files will set the `_JAVA_OPTIONS` environment variable, and then launch `VMLaunch.exe` for you.

Initiating a Debug Session

To show you how to attach CodeWarrior's debugger to a VM that is embedded in a native application, we have provided a short tutorial in the `VM Launcher Example` folder. Follow the steps below.

1. **Import `DebugPorts.reg` into your registry.**

   In the Windows Explorer, double-click the file named `DebugPorts.reg`. Windows creates a matching registry key.

2. **Open the `VMLaunch.mcp`** project.

3. **Verify the path to your JDK.**

   Open `VMLaunch.cpp`. Make sure `USE_1_2_JDK` is defined, and that the `LoadLibrary()` call is pointing to your particular JDK installation, as only 1.2 VMs and later can be seemlessly attached to.

4. **Build the project.**

   Choose **Project > Make** to build the project.

5. **Verify the PATH environment variable in the command file.**

   Open the file named `LaunchMe.cmd` by right-clicking on it and choosing **Edit** from the resulting pop-up menu. Verify that the extension to the `JDKPATH` environment variable matches the location of JDK on your particular system. Close the file when you are done.

   The `LaunchMe_Jbug.cmd` file is provided for attaching to a JBug/JDWP enabled VM. You may use this file in place of `LaunchMe.cmd`.

6. **Launch the command file.**

   From Windows Explorer, launch `LaunchMe.cmd`. Ensure that the Java application runs without problems.

7. **Open TrivialApplication.class**

   Select **File > Open**. The file selection dialog box is displayed. Change the **Files of Type** pop-up menu to **Java Class Files**. The dialog box displays the `TrivialApplication.class` file. Double-click `TrivialApplication.class` to open it. The browser window displays the class information for this class.

8. **Open the Process Window.**

   Select **Window > Process Window** to open the Processes window.

9. **Open the debugger port.**

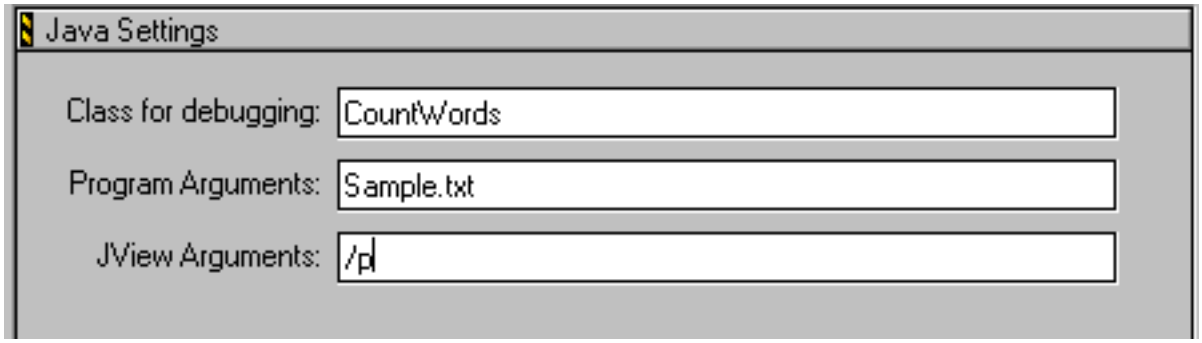
   Double-click on "11XWP debug port 80004 (inactive)" in the list. The debugger attaches to the VM.

   The CW Java Output Window displays the output of the Java application. Set a break point in `TrivialApplication` by selecting it in the browser window. The debugger takes control and lets you step through the program.

Java Settings Panel (Windows Only)

The Java Settings panel (Figure 5.4) controls how the debugger interacts with JView when debugging Java applets or applications.

Figure 5.4 The Debugger's Java Settings preferences panel



The items in this panel are:

Class for Debugging	Program Arguments
JView Arguments	

Class for Debugging

This field specifies the particular class you wish to debug.

Program Arguments

This field specifies command-line arguments to be passed to your Java program when the debugger launches the program.

JView Arguments

This field specifies arguments to be passed to the jview interpreter by the debugger when it launches the Java application. Argument options are shown in Table 5.1.

Table 5.1 Jview argument options

Option	Description
`/?`	displays usage text
`/cp <classpath>`	set class path
`/cp:p <path>`	prepend path to class path
`/cp:a <path>`	append path to class path
`/n <namespace>`	namespace in which to run
`/p`	pause before terminating if an error occurs
`/v`	verify all classes
`/d:<name>=<value>`	define system property
`/a`	execute AppletViewer

6

JavaDoc

JavaDoc is a batch compiler written in Java by JavaSoft™. It processes Java source code files, and uses the comments preceding classes, methods, etc. to generate HTML based documentation for the code. The Java API documentation on your CodeWarrior CD was generated by this compiler.

There is one primary topic discussed in this section:

CodeWarrior JavaDoc Implementation

CodeWarrior implements JavaDoc as a Pre-Linker because of the batch nature of the JavaSoft compiler. The JavaSoft version of JavaDoc must know about all the compiled files simultaneously. In the current CodeWarrior plugin architecture, a compiler works on one file at a time, but a linker knows about all compiled files. So JavaDoc in CodeWarrior is implemented as a pre-linker.

Another added benefit of implementing JavaDoc as a pre-linker is speed. Creating JavaDoc files should be the last thing you do in with your Java project. Assuming this is true, your code will have already been compiled and tested. At this point, when you execute JavaDoc, the process is very fast as everything the pre-linker needs to know was already done at compile time. No need to recompile, just relink.

The JavaDoc plugin uses the Sun JDK or JRE 1.1.6. Therefore, Sun JDK or JRE 1.1.6 must be installed in the system folder in order for JavaDoc to work.

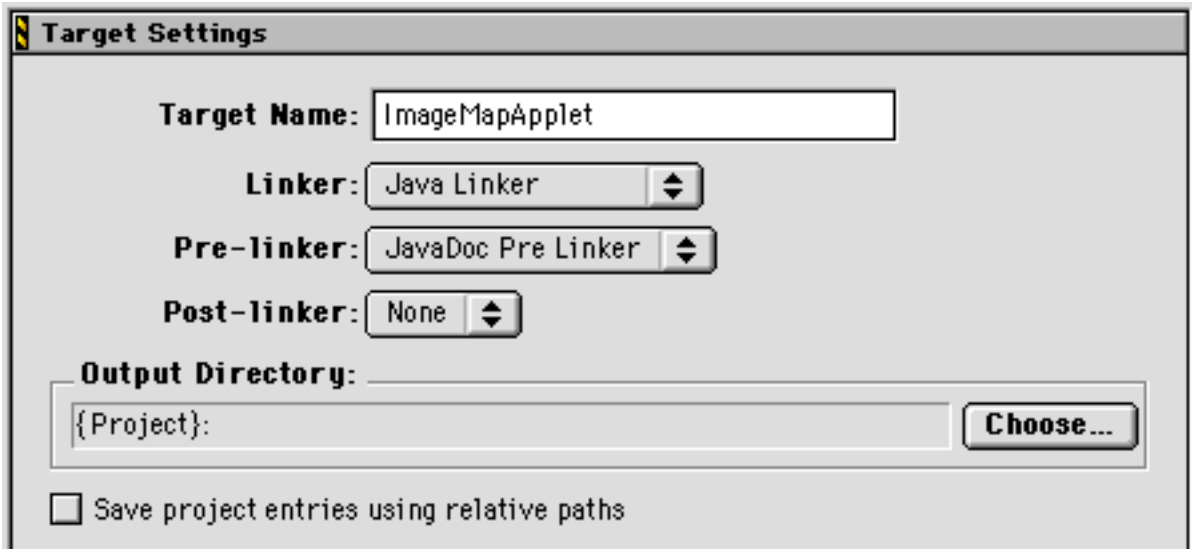
Using JavaDoc

Using JavaDoc is as simple as choosing the JavaDoc pre-linker, and compiling the files.

1. **Choose pre-linker for JavaDoc target.**

   Choose **Edit > *Target* Settings**, where ***Target*** is the name of your current Java target. Choose **Target Settings** from the list of panels on the left so that the Target Settings panel is displayed. Choose **JavaDoc Pre Linker** from the **Pre-Linker** pop-up menu in the Target Settings panel (Figure 6.1). Click Save to save changes. Close the Target Settings dialog box. Click **OK** for subsequent warning dialogs.

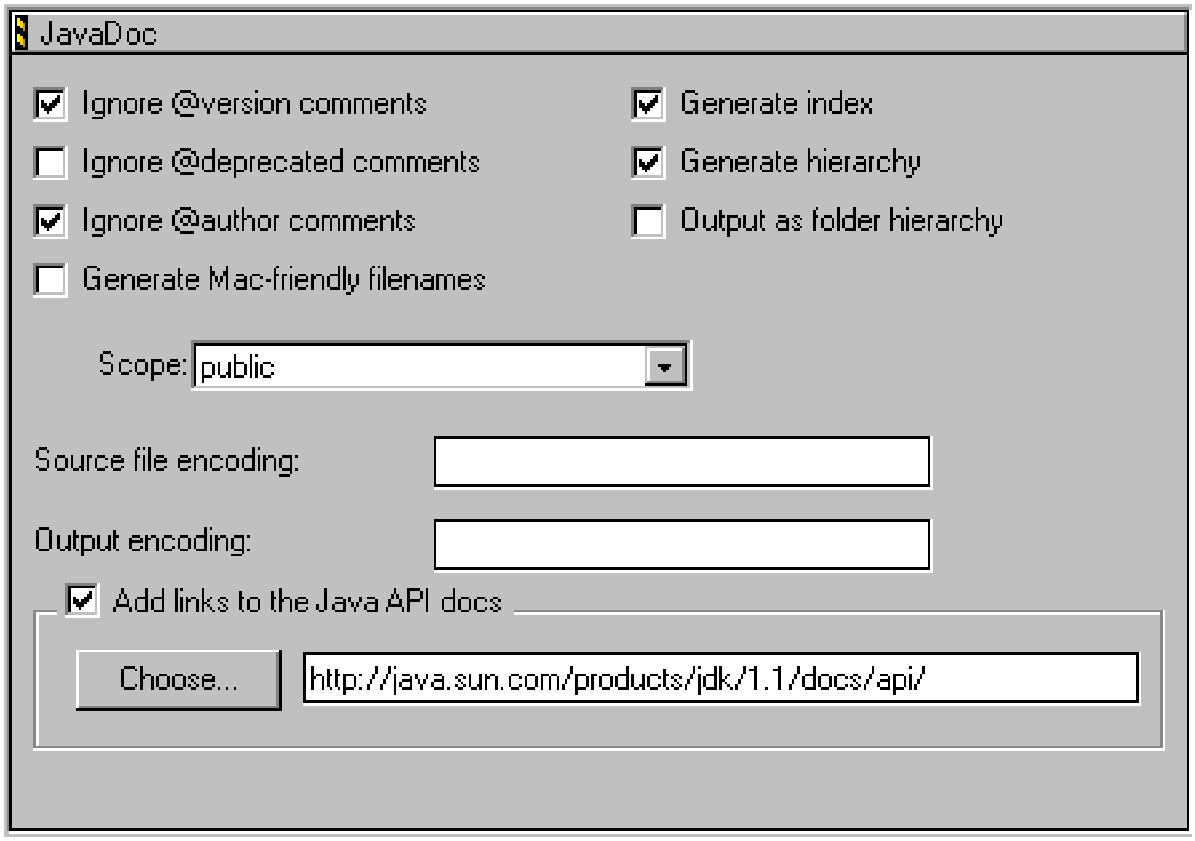
Figure 6.1 Choosing JavaDoc linker



2. **Set JavaDoc Project options.**

   Choose the settings you wish to use for JavaDoc from the **JavaDoc** panel (Figure 6.2). For a full description of each setting in this panel, see "JavaDoc" on page 115.

Figure 6.2 JavaDoc panel



Choose **Project > Make**. JavaDoc creates a new folder in the same folder as your project called "Docs".

The last step is to open the HTML files in your preferred HTML Browser (Figure 6.3). A separate HTML file is generated for every public Java class included in the target. There are also some default files generated depending on the structure of your Java project.

You may find some links do not work. This is because your project does not include this information. For example, if your project does not include packages, the link to `Packages.html` fails because the file was not generated by JavaDoc.

The images used in the HTML documents are not automatically generated by JavaDoc, though the HTML source that references these images is generated. You may create your own images to use,

use the images from the Java API documentation, or use the images supplied with your CodeWarrior installation. The images provided with the CodeWarrior installation are located in the following places:

Windows `Metrowerks CodeWarrior\Java Support\Metrowerks\JavaDoc\images`

Macintosh `Metrowerks CodeWarrior:JavaSupport:Metrowerks:JavaDoc:images`

Solaris `Metrowerks CodeWarrior/Java Support/Metrowerks/JavaDoc/images`

Figure 6.3 ImageMap Doc Class in Browser

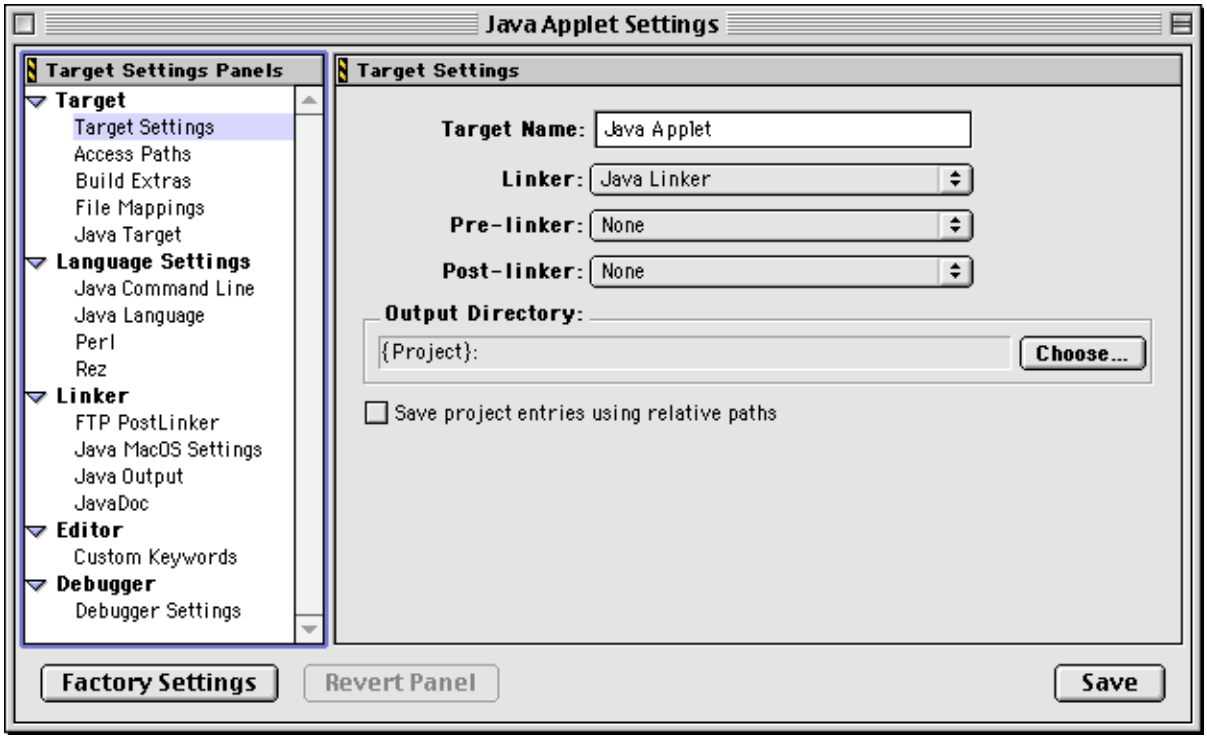
7

Target Settings for Java

CodeWarrior *target settings* are options you can specify to determine various aspects of your project's behavior, such as how it compiles and links. Choosing the right settings can significantly improve the size and speed of your final code.

Target settings are organized into *panels* that you can display in CodeWarrior's target settings window (Figure 7.1). Different settings panels control various properties of the project.

Figure 7.1 Target Settings Window



This chapter discusses only those project settings panels that relate specifically to Java programming:

- Target Settings
- Java Target
- Java Command Line
- Java Language
- FTP Post Linker
- Java Mac OS Post Linker
- Java Output
- JavaDoc

NOTE: Some of these settings panels may not be available in your particular version of CodeWarrior.

See the *IDE User Guide*, *C Compilers Reference*, and *Pascal Compilers Reference* for information about other settings panels available in the Target Settings Dialog.

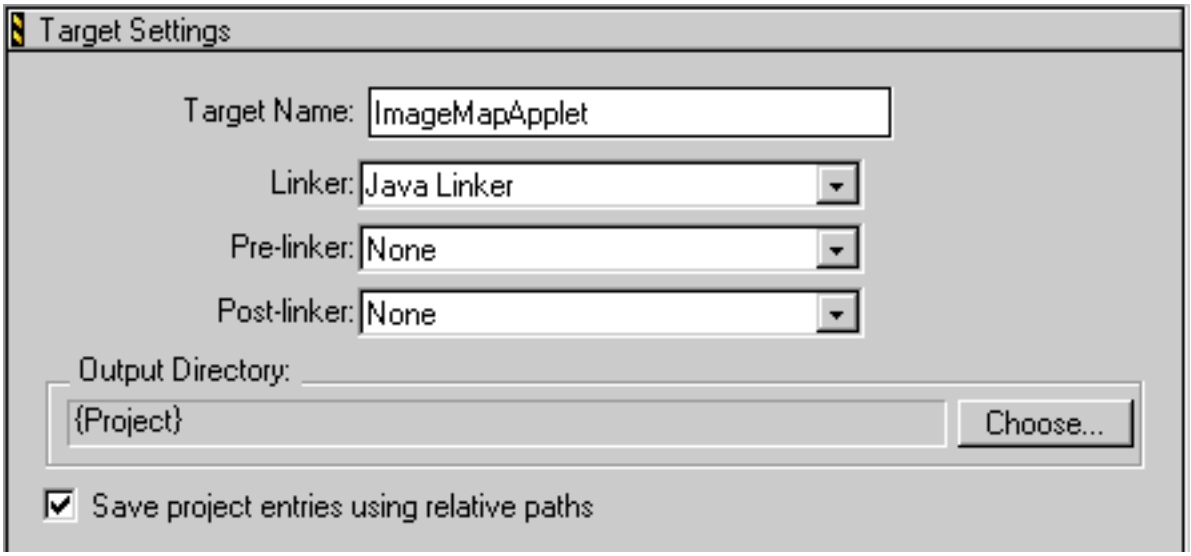
Target Settings

The **Target Settings** panel is the most critical panel in CodeWarrior. This is the panel where you pick the operating system and/or microprocessor your project is to run on.

The **Target Settings** panel, shown in Figure 7.2, allows you to set the name of your target, as well as which linker and post linker plugins to use for the target. When you select a linker, you are specifying the target operating system and/or chip. The other panels available in this dialog box will change to reflect your choice.

Because the linker choice affects the visibility of other related panels, you must set which linkers your project will use before you can specify other target-specific options like compiler and linker settings.

Figure 7.2 Target Settings Panel



The items in this panel are:

Target Name	Post-linker
Linker	Output Directory
Pre-linker	Save project entries using relative paths

Target Name

Use the **Target Name** field to set or change the name of a build target. When you use the Targets view in the Project window, you will be able to see this name.

This is not the name of your final output file. It is the name you assign to the build target for your project file to use. The name of the final output file is set in the Java Output panel.

Linker

Choose a linker from the items listed in the **Linker** pop-up menu.

Pre-linker

Some targets have pre-linkers that perform additional work (such as a data format conversion) before linking. There is one pre-linker for Java: the **JavaDoc Pre-Linker**.

See also "JavaDoc" on page 85.

Post-linker

Some targets have post linkers that perform additional work (such as a data format conversion) on the final executable. The post linkers available for your use are described in Table 7.1:

Table 7.1 Post Linker Documentation

FTP Post Linker	"FTP Post Linker" on page 106
Java Mac OS Post Linker	"Java Mac OS Post Linker" on page 108
JCommand Line	"Java Command Line" on page 100

Output Directory

This is the directory where your final linked output file will be placed. The default location is the directory that contains your project file. Click the **Choose** button to specify another directory.

Save project entries using relative paths

When enabled, this setting allows the IDE to distinguish between files with the same names but in different directories. If you do not have files with the same name in your project, you can leave this option off.

Java Target

The settings you can specify on the **Java Target** panel depend on whether you are creating an applet, a library, or jar file.

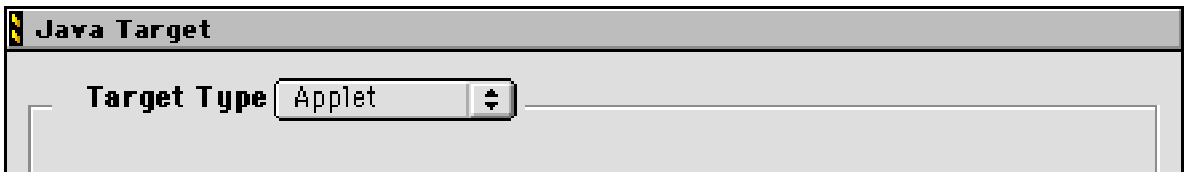
The settings in this panel change based on which option is selected in the **Target Type** pop-up menu. This section contains the following sections:

- Applet
- Application
- Library

Applet

If the **Target Type** pop-up menu is set to **Applet** as shown in Figure 7.3, CodeWarrior will create a Java applet from your project when you build it.

Figure 7.3 Target Type set to Applet



Applet Viewer (Windows)

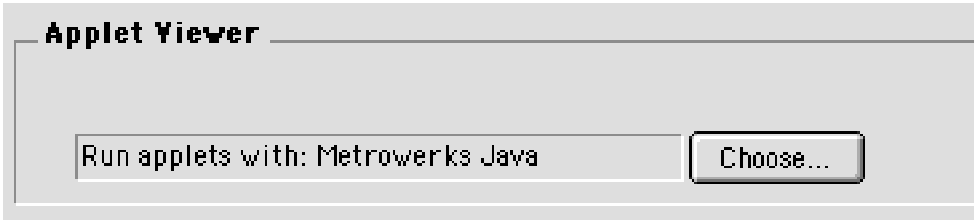
On Windows, this area of the panel contains a pop-up menu as shown in Figure 7.4. The menu contains an item for each applet viewer found on your computer, and an **Other** item, which enables the **Choose** button.

Figure 7.4 Windows Applet Viewer area

Applet Viewer (Mac OS and Solaris)

On Mac OS and Solaris, this area of the panel conatins a text field which displays the currently-selected applet viewer (Figure 7.5). Currently, Sun's `appletviewer` program is the only supported applet viewer on Solaris.

Figure 7.5 Mac OS and Solaris Applet Viewer area



Choose

To use an applet viewer other than those listed by default, click the **Choose** button. CodeWarrior displays an open file dialog box, allowing you to locate and select another applet viewer.

Virtual Machine (Solaris)

CodeWarrior uses the JDK 1.1.6 Virtual Machine by default. The items in this pop-up menu reflect the contents of the `(jdk_options)` folder, which resides in the `JavaSupport` folder of CodeWarrior. The IDE scans this folder when building this pop-up menu and adds to the menu any files it finds in the folder. CodeWarrior includes **jdk1.1.6** and **jdk1.2** by default.

The files in the (jdk_options) folder are expected to be soft links to the root level of the JDK package in question. Items can be added to this menu to allow you to run your applets with newer versions of JDKs as they become available. The JDK package itself must reside in the JavaSupport folder. And a soft link to that package must be placed into the (jdk_options) folder.

For example, to add a menu item for JDK 1.3, you would issue the following commands in a terminal:

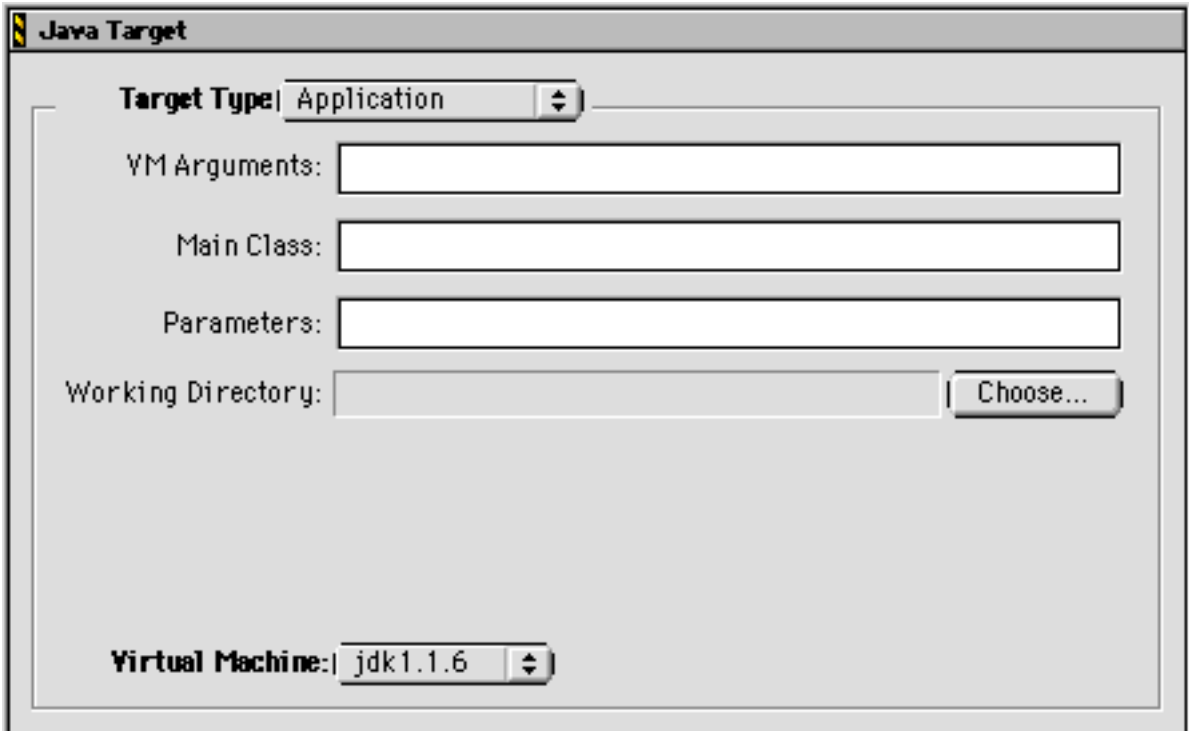
```
cd /usr/local/Metrowerks/CodeWarrior3.3/JavaSupport/
ln -s “(jdk1.3)” “/(jdk_options)/jdk1.3
```

This creates a soft link in the (jdk_options) folder which points to the (jdk1.3) folder which resides in the JavaSupport folder. The next time the menu is built, it will contain the new link.

Application

If the **Target Type** pop-up menu is set to **Application** as shown in Figure 7.6, CodeWarrior will create a Java application from your project when you build it.

Figure 7.6 Target Type Set to Application



VM Arguments (Solaris only)

The **VM Arguments** field is only available on Solaris. It allows you to specify the options you would normally specify in the `java` command line. When you launch your Java application from within the CodeWarrior IDE, these arguments will be bpassed to the VM. Multiple arguments must be delimited by spaces.

Main Class

The **Main Class** field identifies the name of the class containing the `main()` method in a Java application. Enter only the name of the

class. Do not use the `.class` extension. You are specifying the class itself, not the file.

CodeWarrior passes this name to the VM running the application so it can begin execution. This field must have a value in order to run a Java application.

NOTE: the `main()` method must be declared `public static void main(String args[])`, and must be in a public class. Also, the capitalization of the name in the **Main Class** field must match the capitalization in the Java file. In Java, `myclass` and `MyClass` are *different* classes.

Parameters

Enter any arguments you wish to pass to the `main()` method when the application launches. A series of arguments must be delimited by spaces.

Virtual Machine (Windows)

The Windows-hosted CodeWarrior IDE uses the Windows VM by default. You may also use Suns JDK VM.

Virtual Machine (Solaris)

The Solaris-hosted CodeWarrior IDE uses the JDK 1.1.6 Virtual Machine by default. The items in this pop-up menu reflect the contents of the `(jdk_options)` folder, which resides in the `JavaSupport` folder of CodeWarrior. The IDE scans this folder when building this pop-up menu and adds to the menu any files it finds in the folder. CodeWarrior includes **jdk1.1.6** and **jdk1.2** by default.

The files in the `(jdk_options)` folder are expected to be soft links to the root level of the JDK package in question. Items can be added to this menu to allow you to run your applets with newer versions of JDKs as they become available. The JDK package itself must reside in the `JavaSupport` folder. And a soft link to that package must be placed into the `(jdk_options)` folder.

For example, to add a menu item for JDK 1.3, you would issue the following commands in a terminal:

```
cd /usr/local/Metrowerks/CodeWarrior3.3/JavaSupport/
ln -s “(jdk1.3)” “/(jdk_options)/jdk1.3
```

This creates a soft link in the `(jdk_options)` folder which points to the `(jdk1.3)` folder which resides in the `JavaSupport` folder. The next time the menu is built, it will contain the new link.

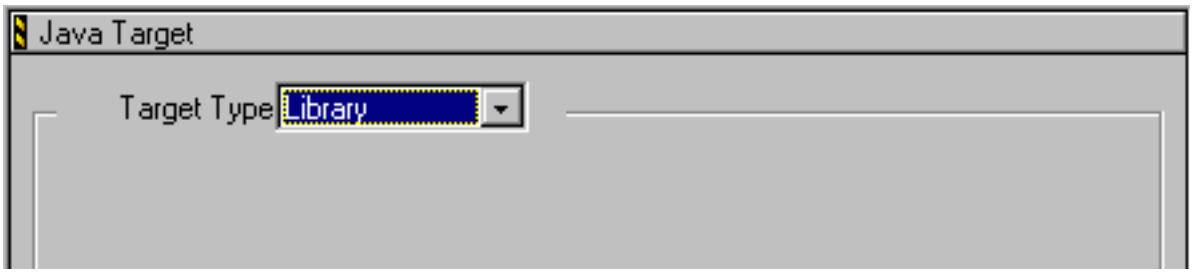
Working Directory (Mac OS and Solaris only)

The **Working Directory** field is optional.The default working directory is the directory where the VM resides. Set this field if you require a different working directory.

Library

If the **Target Type** pop-up menu is set to **Library** as in Figure 7.7, CodeWarrior creates a Java library from your project.

Figure 7.7 Target Type Set to Library



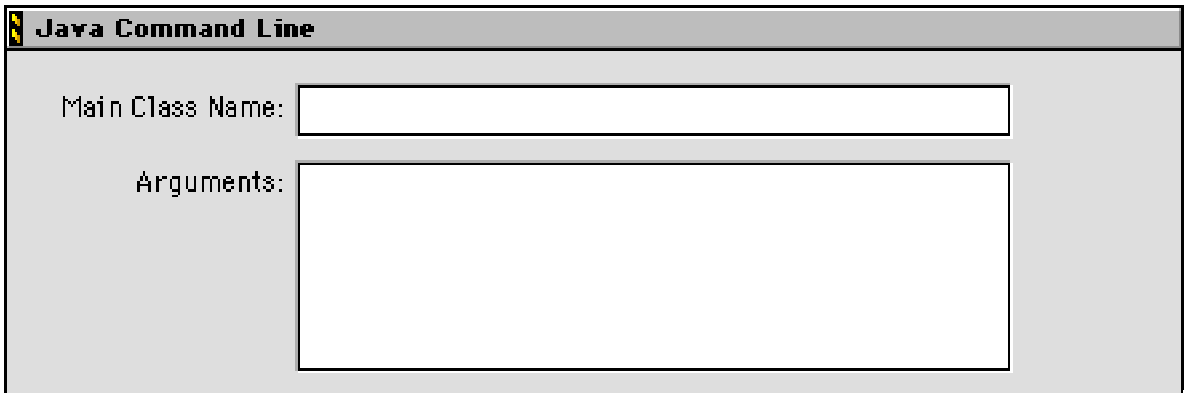
Java Command Line

The **Java Command Line** panel (Figure 7.8) allows you to launch a Java application after your project is successfully built to do further processing of your project output.

For instance, you may use it to launch RMIC, a Java utility that takes .class files and generates stub classes. The java linker would gener-

ate the .class files. Then the RMI compiler (RMIC) would take those class files and generate stub class files for use with RMI. Similar utilities can be found in the `Classes.zip` Java library file.

Figure 7.8 Java Command Line Panel



This panel contains the following items:

Main Class Name

Put the name of the main class of the application you want to be invoked into this field. Once your project is successfully built, the linker will run the application.

Arguments

Put any arguments to be sent to the application into this field.

This linker adds paths to all zips and jars in the project to the VM classpath. It also adds the project output to the classpath.

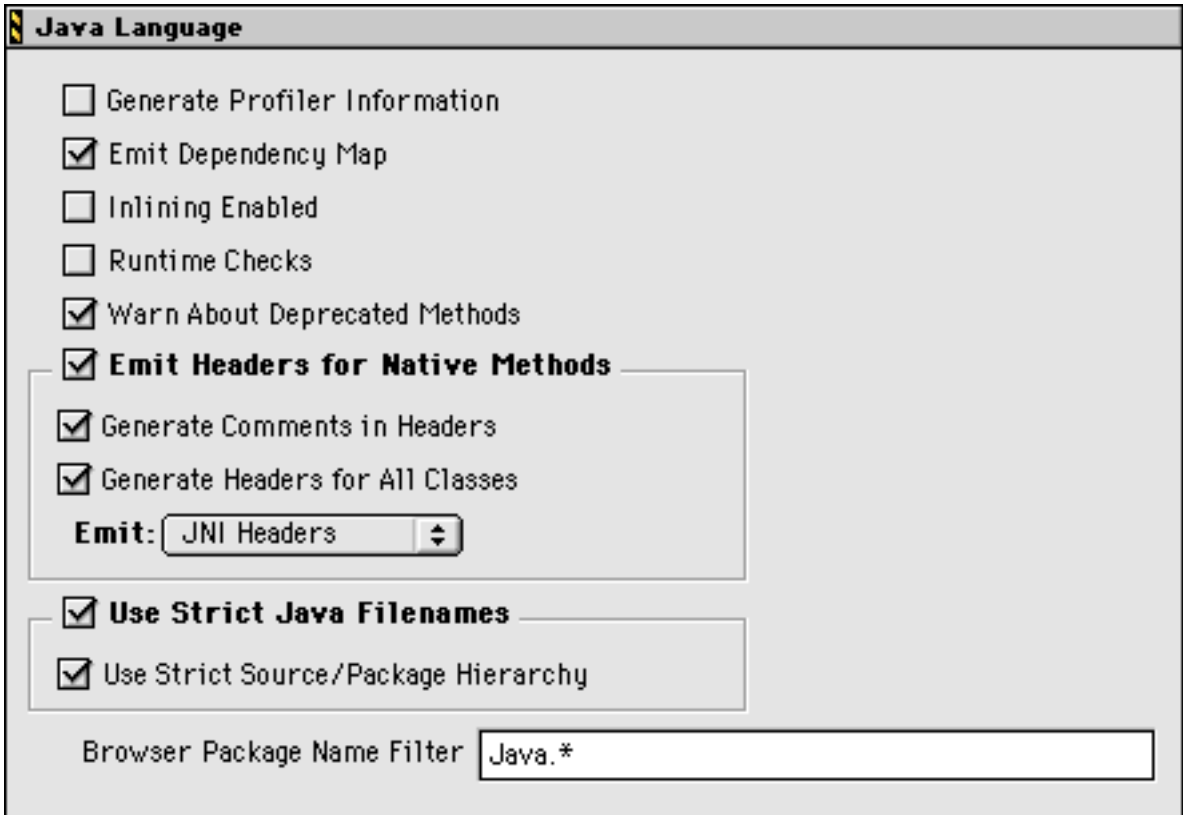
NOTE: Although this linker can execute any Java application specified in the command line settings, its intended use is to invoke java command line post linker type tools (such as RMIC, obfuscators, etc.). Therefore, there is no support for AWT based apps nor apps which make use of System.in.

WARNING! Since it is impossible for the post linker to know what type of application it will be invoking, it is your responsibility to make sure that only the appropriate type of command line tools are run using it.

Java Language

The **Java Language** panel (Figure 7.9) contains project settings related to code generation for the Java platform.

Figure 7.9 Java Language panel



The options in this panel are:

Generate Profiler Information	Emit Headers for Native Methods
Emit Dependency Map	Use Strict Java Filenames
Inlining Enabled	Use Strict Source/Package Hierarchy
Runtime Checks	Browser Package Name Filter
Warn About Deprecated Methods	

Generate Profiler Information

This option adds timing information to your Java methods. You can only profile Java applications. To use this option, you must include `Profiler.zip` in your project. When your Java code finishes executing, a new file is created with the timing data. You can view this file with the MW Profiler application on Mac OS.

NOTE: The profiler is not currently available on Solaris.

Emit Dependency Map

This option is similar to generating a link map in other languages such as C and C++. When enabled, a new file is created in the same directory as your project file called *`sourcefile`*`.JMAP`, where *`sourcefile`* is the name of your `.java` source file. The JMAP file lists all the class dependencies for each class in the file.

For example, the statement `import java.*` is common in Java source code. The dependency map tells you the *exact* class dependencies. So the statement `import java.*` may become:

```
import java.applet.Applet;
import java.applet.AppletContext;
```

You can then use these statements in your Java source files instead of importing everything. This makes things cleaner, faster and more portable to other Java compilers.

Inlining Enabled

Allows inlining of smaller methods where appropriate.

Runtime Checks

This option only effects the J2N (Java To Native) compiler, which is currently in pre-release stages.

When this option is enabled, the J2N compiler generates runtime checks required by Java, such as null pointers or array lengths. This is useful for debugging; but is probably not neccessary in the final application.

When this option is disabled, more optimized and efficient code is generated; but no checks are performed.

Warn About Deprecated Methods

Gives warning messages if you are using any depreciated methods. This allows you to find references to depreciated methods in your code so that you can update your code to the newer methods.

Emit Headers for Native Methods

Emits headers for any native methods for *every* class in your project. When this option is enabled, the **Emit** pop-up menu appears allowing you to choose between **JNI Headers** and **Sun VM Headers**. JNI is the newest and preferred method. However, for compatibility reasons, you may wish to choose the **Sun VM Headers** option.

The other options that can be used in conjunction with Emit Headers for Native Methods are **Generate Comments in Headers** and **Generate Headers for All Classes**.

Generate Comments in Headers moves comments from the java sources files to the native headers.

Generate Headers for All Classes generates the headers in a tree (like sun/tools/java/Identifier.h) instead of mangled with the package (like sun_tools_java_Identifier.h)

Use Strict Java Filenames

Forces strict class names based on the Java source file names. For example, if you have a java source file called FooBar.Java, the compiler expects to find a class called FooBar.class in this file. No other class can be in this file.

NOTE: This option should be used in conjunction with Use Strict Source/Package Hierarchy.

Use Strict Source/Package Hierarchy

Forces strict adherence to source and package hierarchy paths. For example, if you have the following package:

```
Foo\bar\seam
```

The compiler expects a to find file `seam.java` in a directory called "`bar`" which is in a directory called "`Foo`."

This option is only available when Use Strict Java Filenames is enabled.

Browser Package Name Filter

Accepts a semicolon delimited list of names that you do not want to show up in the class browser. If you type in: `java.io; java.lang`, the browser will only display `File`, `Object` and `reflect.Method` for classes like `java.io.File`, `java.lang.Object`, and `java.lang.reflect.Method`.

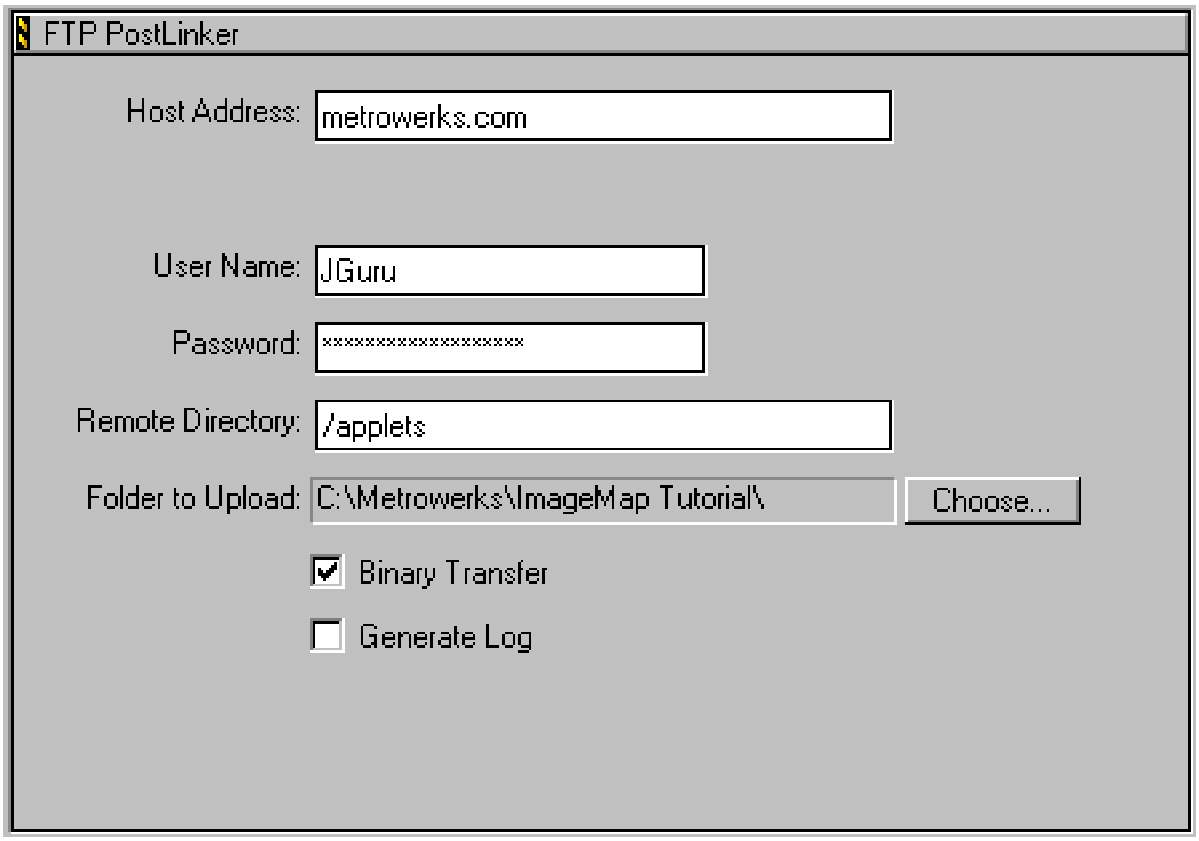
You can also use wildcards like `java.*`, in which case the browser will strip off the entire package for any class starting with "java."

Finally, you can also type in "`*`" (no quotes), in which case the browser strips all packages off everything.

FTP Post Linker

The **FTP Post Linker** panel allows you to move a folder containing your completed Java binary and any associated files to a server to be accessed through the World Wide Web.

Figure 7.10 FTP Post Linker panel



The items in this panel are:

Host Address	Folder to Upload
User Name	Binary Transfer
Password	Generate Log
Remote Directory	

Host Address

The host address of the server you want to upload your Java files to.

User Name

Your user ID on the host server.

Password

Your password on the host server.

Remote Directory

Directory you want to upload the Java files to on the host system. You must have access privileges for this directory.

Folder to Upload

The full path to the folder on your local hard drive containing the files to upload. There is no way to specify that individual files be uploaded, so the linker will transfer the entire contents of the folder specified here. Make sure you do not have any files that you do not want transferred in this folder.

Binary Transfer

Transfer the files to the host system using Binary mode instead of ASCII. It is recommended this option always be turned on to avoid any transfer problems.

Generate Log

Generates a text log file of the transfer. Any errors or problems are recorded in this file.

Java Mac OS Post Linker

The settings in the **Java Mac OS Settings** panel control the operation of the Java Mac OS Post Linker. This post linker determines how Mac OS Java applications are packaged.

The **Mac OS Java Output Type** pop-up menu can be set to one of the following settings:

- JBindery
- Mac OS Zip

This section describes the settings for each option in detail.

JBindery

When the **JBindery** output type is selected, the panel appears as in Figure 7.11. JBindery is an application that you use to package or execute Java™ applications on the Mac OS platform.The settings contained in this panel allow you to control how JBindery builds your application.

The Java Mac OS Post Linker will launch JBindery using the follwing settings.

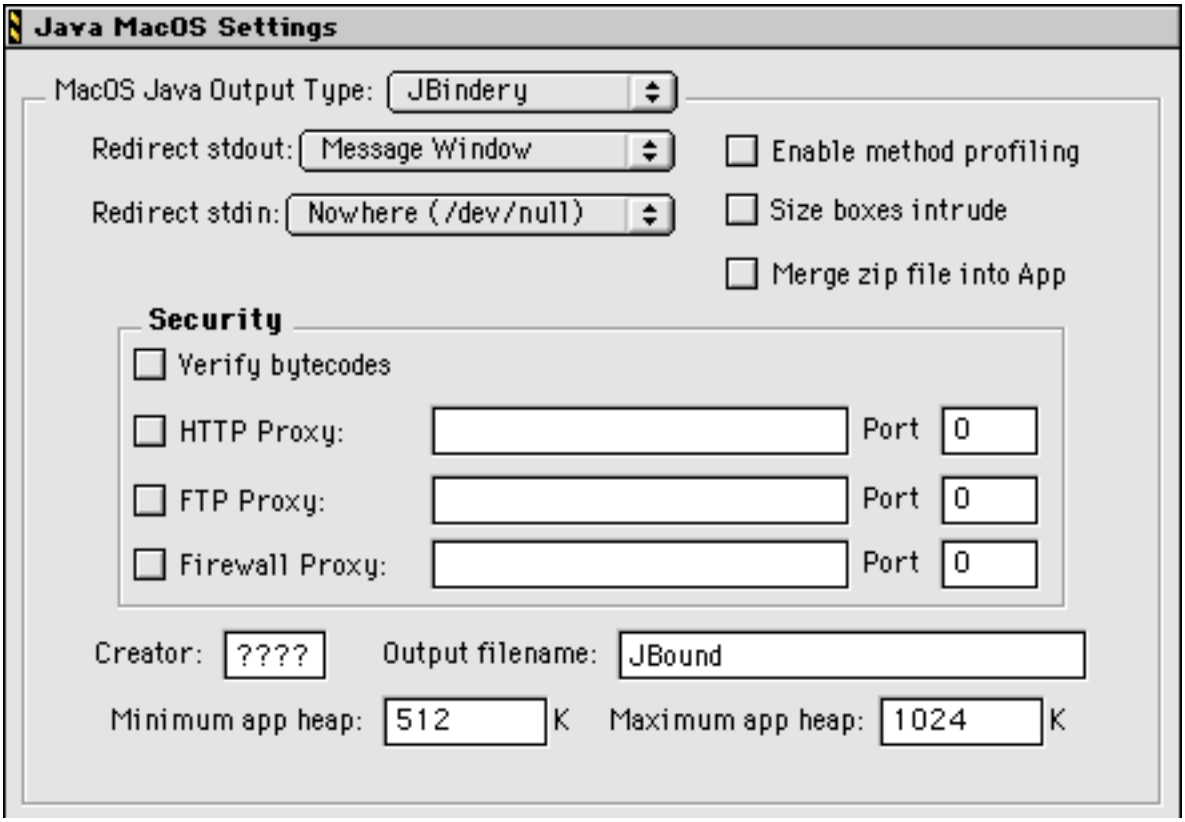
Redirect stdout

This pop-up menu allows you to redirect stdout. You can either redirect stdout to a message window generated by JBindery, or you can instruct JBindery to ignore stdout altogether.

Redirect stdin

This pop-up menu allows you to redirect stdin. You can either redirect stdin to a message window generated by JBindery, or you can instruct JBindery to ignore stdin altogether.

Figure 7.11 Output as JBindery



Enable method profiling

Check this checkbox to enable method profiling.

Size boxes intrude

This checkbox determines whether the size box should appear within the corner of the actual window. If the size boxes intrude, any Abstract Window Toolkit components that would normally appear under the size box are not drawn. If you do not check this box, an extra strip of empty space is added to the bottom of the window to accomodate the size box.

Merge zip file into app

If this checkbox is checked, the zip file will be merged into the final application when you build your project.

Verify Bytecodes

Check this checkbox if you want the code verifier to check local Java bytecodes before execution. If this checkbox is not selected, JBindery will still automatically check any bytecodes obtained from a remote source (such as over a network).

HTTP Proxy

If you would like your application to use an HTTP proxy server, check this checkbox, and supply the name and port ID of the proxy server in the corresponding edit fields.

FTP Proxy

Check this checkbox if you would like your application to use an FTP proxy server. Supply the name and port ID of the proxy server in the corresponding edit fields.

Firewall Proxy

This option specifies whether you want to use a firewall proxy. When this checkbox is checked, the Java application uses the firewall proxy server specified in the name and port edit fields when accessing servers outside the security firewall.

Creator

This edit field lets you specify the creator of your final Java application.

Output Filename

This edit field lets you specify a creator for your Java application. This unique string identifies the application and any documents that the application may create. If you plan to publically distribute your application, you must register its creator name with Apple through Developer Technical Support to avoid collisions between names used by different developers.

You can register a creator online or view currently registered creators at the following Web site:

`http://devworld.apple.com/dev/cftype/main.html`

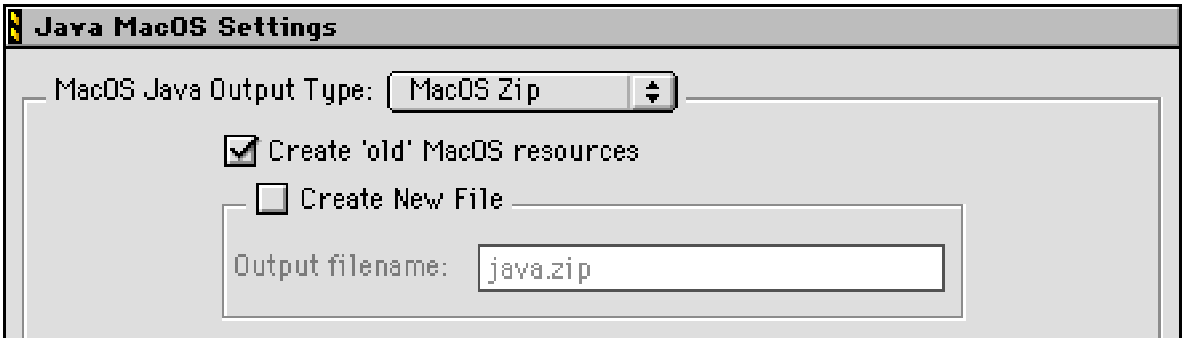
Minnimum and Maximum app heap

These two edit fields let you specify the amount of memory to use when executing this Java program. Mac OS Runtime for Java uses temporary memory for most allocations (the Java virtual machine and so on) so an application heap of 512K is usually sufficient.

Mac OS Zip

When the **Mac OS Zip** menu item is selected, the panel appears as in Figure 7.12. Previous versions of the Java Linker copied Mac OS resources into the output zip file. This was handy for adding 'vers' resources to your zip file, for example. This option allows you to do the same thing.

Figure 7.12 Output as Mac OS Zip



Create 'old' Mac OS resources — The Java Linker used to create certain resources in the output zip file that were used to assist in building standalone applications. These resources are no longer needed by CodeWarrior, and are no longer written. However, with JBindery in MRJ SDK 2.0.1, if you drop jar/zip files containing these resources onto JBindery, it will still parse out the main class name and the main arguments and use them. This option is available for this reason.

Create New File — If this checkbox is checked, resources will be added to a copy of the Java Linker output file, rather than the original output file. Otherwise, resources will be merged into the Java

Linker output file. Supply the name of the new file in the **Output filename** edit field.

Java Output

The settings on the **Java Output** panel control the operation of the CodeWarrior linker for Java. The **Output Type** can be:

- Class Folder
- Jar File
- Application

This section describes the settings for each option in detail.

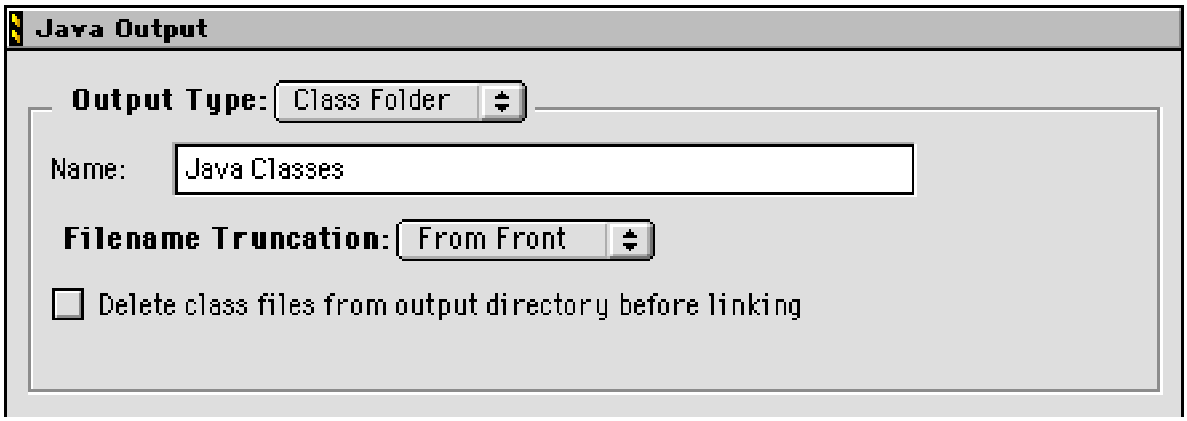
Class Folder

If you choose **Class Folder** from the **Output Type** pop-up menu, the IDE will create a new folder with the name you specify in the **Name** field. Every class in your project will be placed in this folder.

The Mac OS has a 31 character maximum for filenames. The **Filename Truncation** menu controls where the IDE will remove characters of the final class file name if it is too large. You can choose **From Front**, **From Middle**, or **From End**.

Delete class files from output directory before linking simply removes all class files in the target directory before linking your code. This is useful if you recompile your source often as it makes sure no old code exists in the target folder.

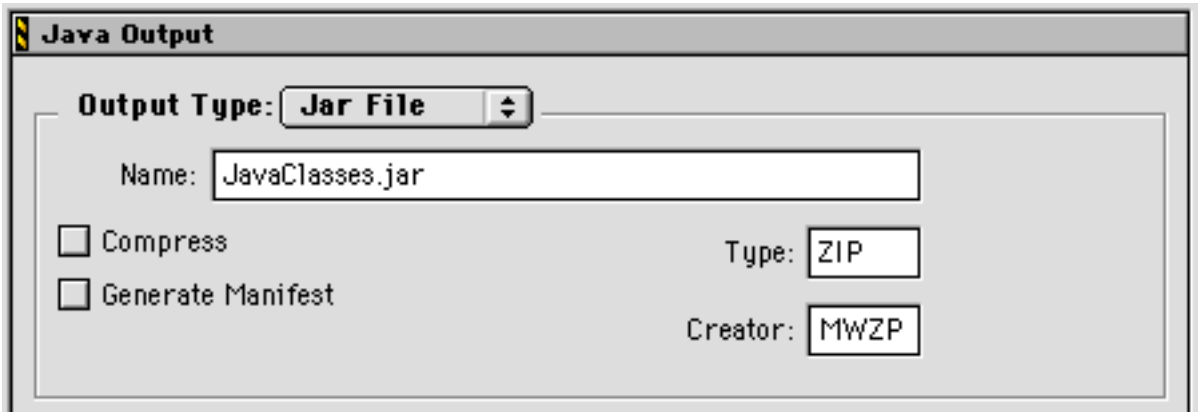
Figure 7.13 Output as class folder



Jar File

A Jar file is a Java Archive file. The **Jar File** option (Figure 7.14) allows you to control how the Jar file is created. Choose **Compress** to create a compressed Jar file. Choose **Generate Manifest** to have manifest information added to the Jar file.

Figure 7.14 Output as Jar file (Mac OS)



The **Type** and **Creator** fields only appear in the Mac OS hosted version of CodeWarrior. These fields allow you to change the default application used to open them if you double-click on the file on the Mac OS. The default settings open the Jar file in Class Wrangler.

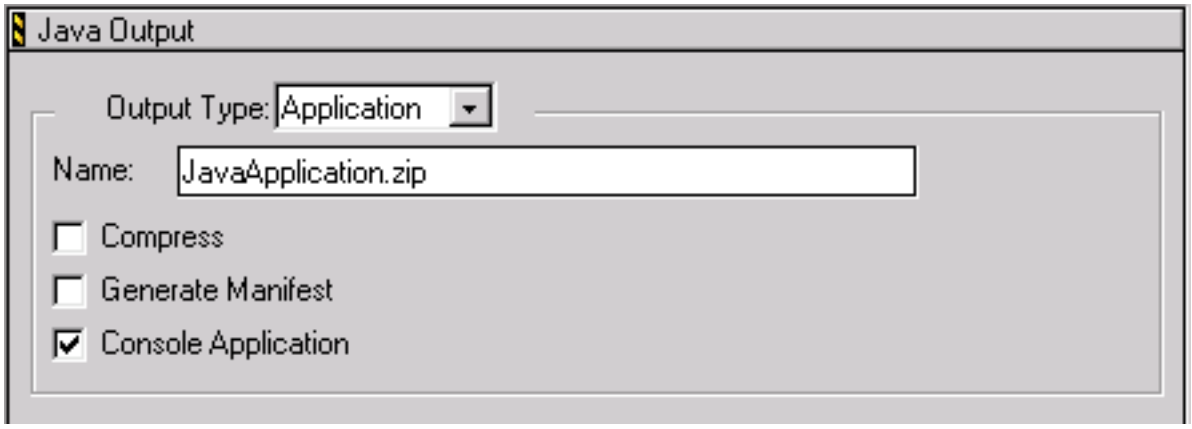
See also "Class Wrangler for Mac OS" on page 119.

Application

Choose **Application** from the **Output Type** popup menu to create a Java application (Figure 7.15).

NOTE: This option is only available on the Windows and Solaris hosted versions of CodeWarrior. For information on how to create Mac OS standalone applications, see "Standalone Applets for Mac OS" on page 137.

Figure 7.15 Application output type



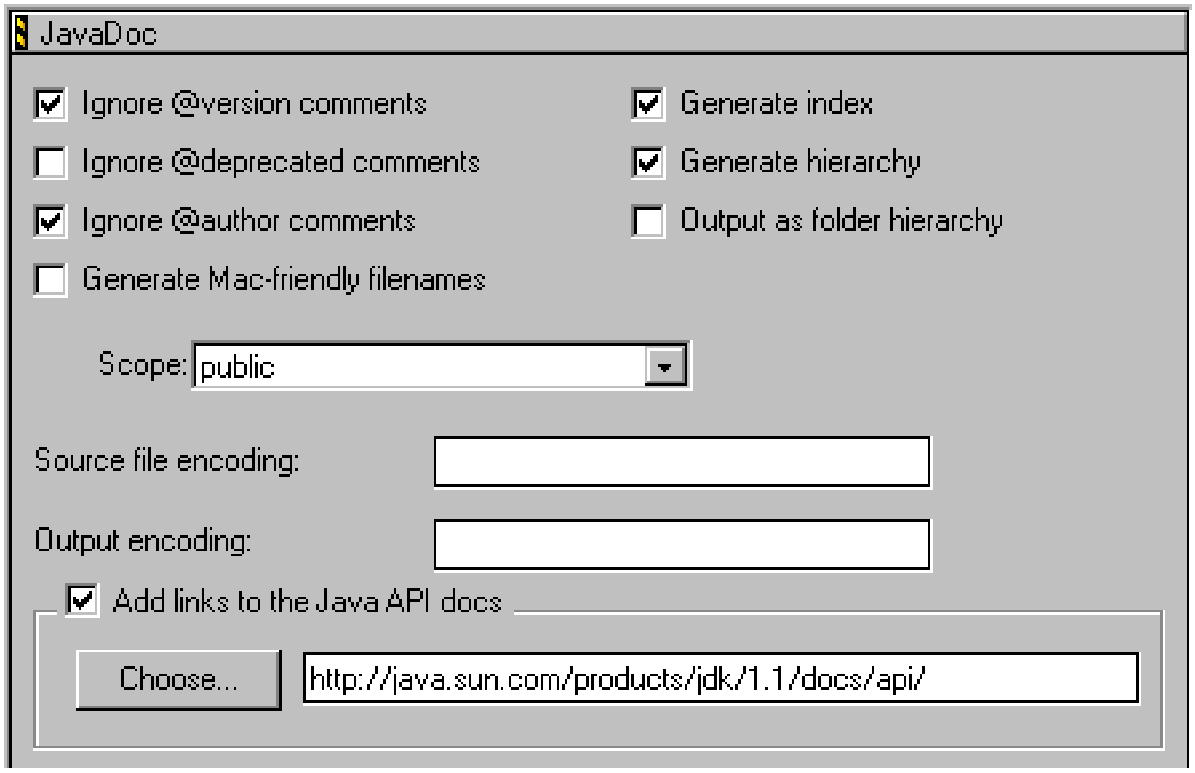
Choose **Compress** to create a compressed Jar file. Choose **Generate Manifest** to create manifest information in the file.

When the **Console Application** option is enabled, the DOS prompt window will display when you launch the application. System.out, System.err and System.in will use the console's io streams for standard input/output. In other words, the application is a CUI Windows application. When this option is disabled, the console window does not display because the application is built as a GUI Windows application. The user will not be able to see standard I/O unless they pipe it somewhere else.

JavaDoc

The **JavaDoc** panel (Figure 7.16) controls how JavaDoc creates documentation for your Java source.

Figure 7.16 JavaDoc panel



Ignore options

Three ignore options: **Ignore @version comments**, **Ignore @deprecated comments**, and **Ignore @author comments** tell JavaDoc to omit comments with these "tags" from the final output when enabled.

Generate Mac-friendly filenames

Tells JavaDoc to generate filenames shortened to 31 characters to create cross-platform friendly HTML documents. This option is not available on the Mac OS version of CodeWarrior.

Generate index

Generate index tells JavaDoc to create index information in the final output. Sometimes, the index can be very large. Turn this option off if you do not want index information generated for your project.

Generate hierarchy

Generate hierarchy tells JavaDoc to generate class hierarchy information. Turn this option off if you do not want hierarchy information created.

Output as folder hierarchy

Instead of only outputting the HTML files for a project to a flat folder hierarchy with extremely long filenames, the JavaDoc Pre-Linker has the option of putting the files into a package-based folder hierarchy. The actual file names are only the class name.

HTML file creator code

Use this option to set the creator code for HTML files to your favorite HTML browser. This option is only available on the Mac OS hosted version of CodeWarrior.

Scope

The **Scope** pop-up menu controls which classes will be included in the documentation. Choices include:

- public
- protected and public
- protected, public and package
- all

NOTE: If you do not specifically declare a class to be public, private, or protected, it will automatically be declared as package.

Encoding options

The two encoding options: **Source file encoding** and **Output encoding** are for internationalization of your JavaDoc files. For example, the Java source files can be in Arabic and output in English.

Encoding numbers are used in each field. JavaDoc converts the number in the source field to Unicode, and then from Unicode to the value specified in the output field. Leave these fields blank to use the default ISO Latin (9859-1) encoding.

For more information on encoding, see:

`http://java.sun.com/products/jdk/1.1/intl/html/intlspec.doc7.html`

Add links to the Java API docs

This option adds links to the main Java API docs where appropriate in your code. The default link is on the Web. If you have a dial-up connection, you might prefer to have the API docs somewhere on your local machine. This way, you can specify the URL to be something like `file://my_drive/jdk_docs/`. You can also specify a relative URL.

NOTE: If you move the docs around, the links may break.

8

Class Wrangler for Mac OS

Class Wrangler is Metrowerks' utility program for working with Java Archive (jar) files on the Mac OS. It allows you to add or remove files, as well as copy files from one zip/Jar file to another. Class Wrangler has other useful features as well.

NOTE: Class Wrangler *is not* a general purpose utility for compressed zip files. It will only handle zip archives that are all contained in one file (no disk-spanning files). If the zip archive is compressed, the deflate algorithm must be the one used in JDK 1.1 JAR files. Class Wrangler cannot handle encrypted files.

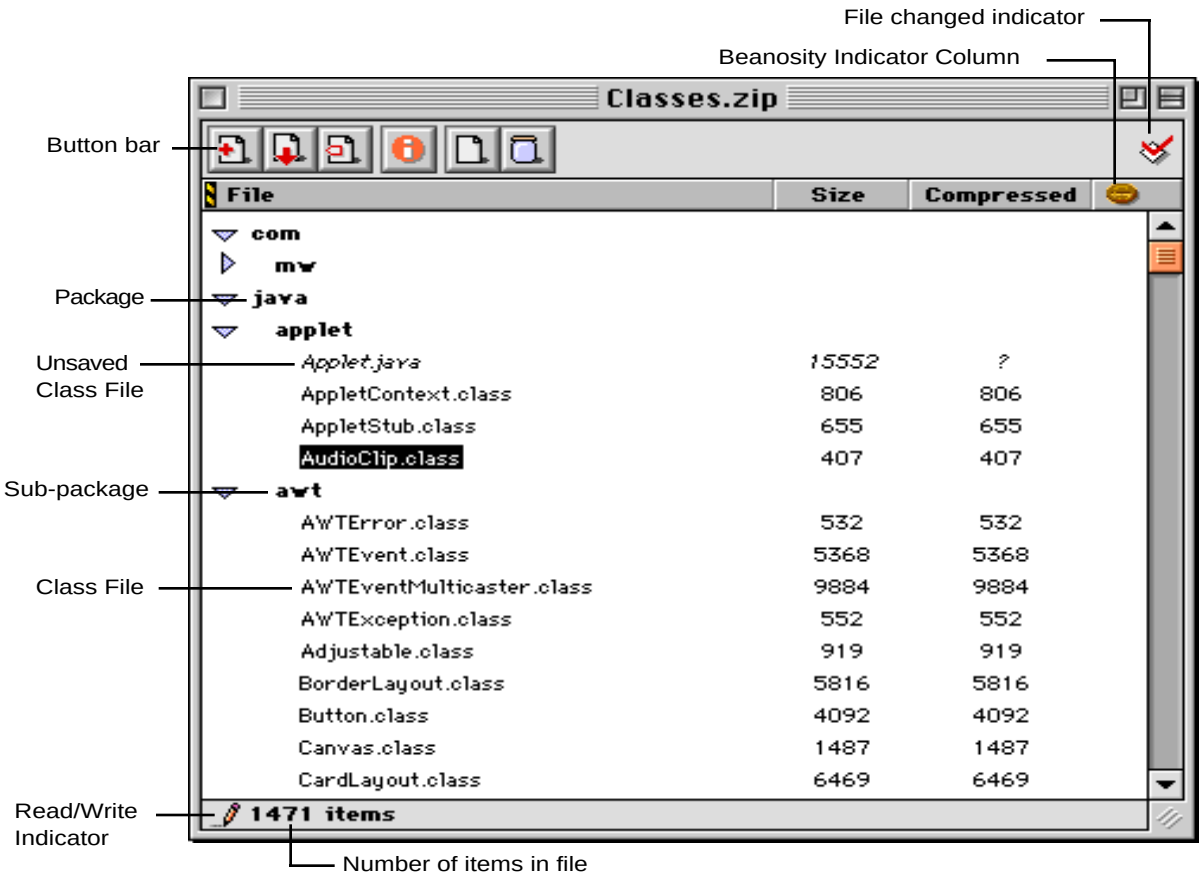
This chapter explains the Class Wrangler interface, and how you can use it to get your work done. The sections in this chapter are:

- Class Wrangler Window—what you see in the Class Wrangler interface
- Working with Files and Archives—discusses how to work with files and zip archives
- Editing Manifest Files—describes how to use the **Edit Manifest** command
- Class Wrangler Preferences—shows you how to control the way Class Wrangler works for you
- Comparing Archives—describes how to compare two Jar files

Class Wrangler Window

Class Wrangler uses a single window to display each package and class file stored in a zip archive or Jar file, as shown in Figure 8.1.

Figure 8.1 The Class Wrangler Window



You can use the button bar to add, extract, delete, and get information on files. Table 8.1 describes each button.

Table 8.1 Class Wrangler button bar

Button	Description
	Add File to Jar file.
	Extract Item and save item to disk.
	Delete item from Jar file.
	Get Info on class file. If a package or sub-package is selected, will get info on every file in the package and display the information separate windows, one for each class.
	Compression indicator. Clicking this button does not compress the file, rather it marks the file for compression (or uncompression) the next time the file is saved. A compressed file will occupy less disk space, but may require more time to load and process.
	Generate Manifest. This button indicates whether a Jar file has a manifest file or if one will be generated the next time the file is saved. You can toggle the bean status of an item in the Jar file by clicking in the JavaBean column. If the Jar file has a manifest, and a particular item is a bean, Class Wrangler draws a dot in the JavaBean column. Save the file to retain the changed state.

The diamond icon at the top right is empty if the zip archive has not changed since the last time it was saved, and contains a red check mark if the file has been changed.

Each package lists all the files and sub-packages it contains. To show or hide the contents of a package, click the disclosure triangle.

To the right of each file is its size in bytes. If a file is in italics, it has recently been added to the archive and has not been saved to disk yet. To save it, choose **File > Save**.

You can choose **Edit > Copy** to copy the names of the selected files and packages to the Clipboard. When you copy a file, its fully qualified name is placed in the Clipboard. For example, if you copy `AudioClip.class`, this text is placed in the Clipboard:

```
java.applet.AudioClip
```

If you hold down the Option key while copying a class, an `import` statement for the selected class is placed in the clipboard:

```
import java.applet.AudioClip;
```

If you copy a package, its name, the names of its subpackages, and the names of its class files are placed in the Clipboard. If you hold down the Option key while copying, an `import` statement for the subpackages, and the class files are placed in the Clipboard.

Typing a full or partial file or package name selects the closest matching item in the Class Wrangler window. This is called "type ahead selection" and allows you to move around the archive quickly when you know the file or package you are looking for.

See also "Use full package name for type ahead" on page 132.

Working with Files and Archives

This section explores how you can get work done using Class Wrangler. To avoid confusion, these topics refer to class *files*, zip *archives*, and Jar files (Java Archive files) consistently. Keep in mind that each is really just a kind of file.

Zip archives and Jar files are used interchangeably. The main difference is a Jar file can have manifest files, whereas Zip archives do not.

The topics in this section are:

Opening a Zip Archive

There are two ways to open a zip archive:

- Drag and drop a zip archive onto the Class Wrangler icon.
- Choose **File > Open** and choose the zip archive.

Class Wrangler opens the file and displays its contents in a window.

At the bottom of the Open dialog, there are three check boxes that let you choose which files the dialog displays. When no option is checked, the dialog displays only files whose Macintosh file types are for zip or class files. When the **Filter using file extension** option is on, the dialog also shows any file whose name ends in `.zip` or `.class`. When the **Filter using file contents** option is on, Class Wrangler examines the internal contents of each file and displays any file that it determines to be a zip or class file. This option is the most accurate, but takes the most amount of time. When the **Show all files** option is checked, Class Wrangler displays all files in a directory. This option overrides the other two, even if the other two are checked.

You can set these options generally for the program as well. See "File Filtering Settings" on page 132.

Creating a Zip Archive

There are three ways to create a zip archive with Class Wrangler. To create an empty zip archive, choose **File > New**. To create a zip ar-

chive that contains one class file, choose **File > Open**, and select the class file. Class Wrangler displays a new window that contains that file.

To create a zip archive for several class files, follow these steps:

1. Make sure that Class Wrangler is not running or that there are no open windows in Class Wrangler.
2. Drag and drop the class files onto the Class Wrangler icon.

Class Wrangler creates a window that contains those class files.

NOTE: If you drag class files onto the Class Wrangler icon while there is an open archive, the class files are added to the archive displayed in the frontmost Class Wrangler window.

Adding Files

Class Wrangler gives you three ways to add files to an archive.

- Choose **Package > Add Files** or click the **Add Files** button in the Class Wrangler window. Class Wrangler displays a dialog box that lets you choose a group of files to add at once. For more information, see "Using the Add Files Dialog" on page 125.
- Drag and drop class files onto the Class Wrangler icon. Class Wrangler adds them to the frontmost window.
- Drag and drop class files onto a Class Wrangler window. Class Wrangler adds them to the window.

If you try to add a class file that is already in the zip archive, Class Wrangler displays an alert asking you whether you want to overwrite the existing file.

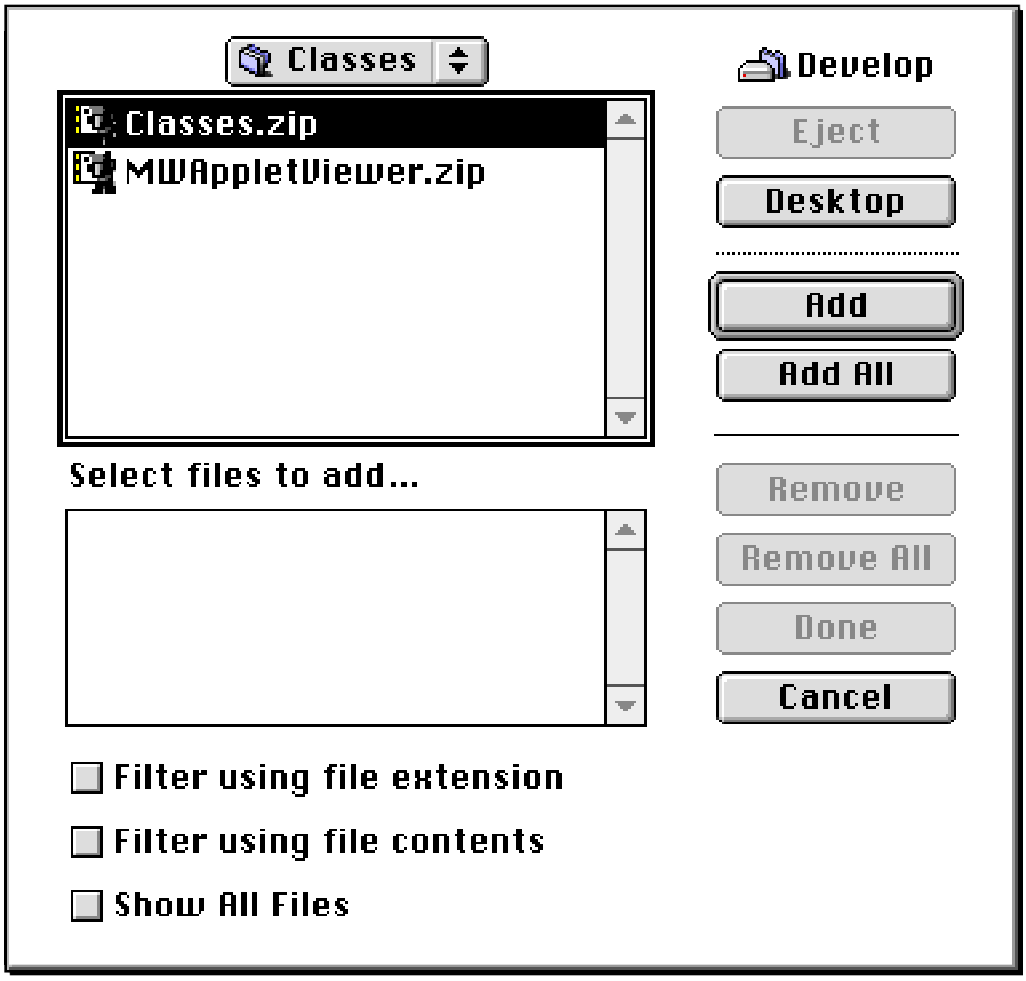
After you add class files, their names appear in italics in the window to show the new files have not been saved to the zip archive on disk. To save the new files to disk, choose **File > Save**.

Using the Add Files Dialog

If you use the **Add Files** command or button, Class Wrangler displays a dialog like the one shown in Figure 8.2.

The top list is part of a standard file dialog and displays the contents of a folder. The bottom list contains all the files that will be added to the zip archive when you click the **Done** button. To choose a file to add, select it in the top list and click **Add**. To add all the files in the top list, click **Add All**. To remove a file from the bottom list, select it and click **Remove**. To remove all the files from the bottom list, click **Remove All**.

Figure 8.2 The Add Files dialog



The three check boxes at the bottom of the dialog let you choose which files the dialog displays. When no option is checked, the dialog displays only files whose Macintosh file types are for zip or class files. When the **Filter using file extension** option is on, the dialog also shows any file whose name ends in `.zip`, `.jar`, or `.class`. When the **Filter using file contents** option is on, Class Wrangler examines the internal contents of each file and displays any file that it determines to be a zip or class file. This option is the most accurate, but takes the most amount of time. When the **Show all files** option is checked, Class Wrangler displays all files in a directory. This option overrides the other two, even if the other two are checked.

You can set these options generally for the program as well. See "File Filtering Settings" on page 132.

Add Directory

The **Package > Add Directory** command allows you to add a package to the frontmost archive as shown in Figure 8.3. You must type the full path name. The directory is created in the archive.

Figure 8.3 Add Directory dialog

Extracting Files

Class Wrangler gives you two ways to extract class files from a zip archive. The class files remain in the zip archive after they are extracted. The process of extraction does *not* remove the class file from the zip archive.

- Select the files and choose **Package > Extract Files** or click the **Extract Files** button in the Class Wrangler window. Class Wrangler displays a file dialog asking you where to place the files. Select a location and click OK. Class Wrangler places the class files in that location.
- Drag the files from the Class Wrangler window onto the Finder desktop. Class Wrangler places the files in that location.

TIP: If you do not want the folder hierarchy, hold down the Option key while dragging.

When Class Wrangler extracts a file, it places the file in a folder hierarchy that matches the file's package hierarchy. For example, if the file is in the hierarchy `java.util`, Class Wrangler creates the folder `java` at the location you selected, creates the folder `util` inside `java`, and places the file in `util`.

Deleting Files

To delete a class file from a zip archive, select the file and do one of the following:

- Choose **Package > Delete File**
- Click the **Delete File** button in the Class Wrangler window
- Press Option-Delete

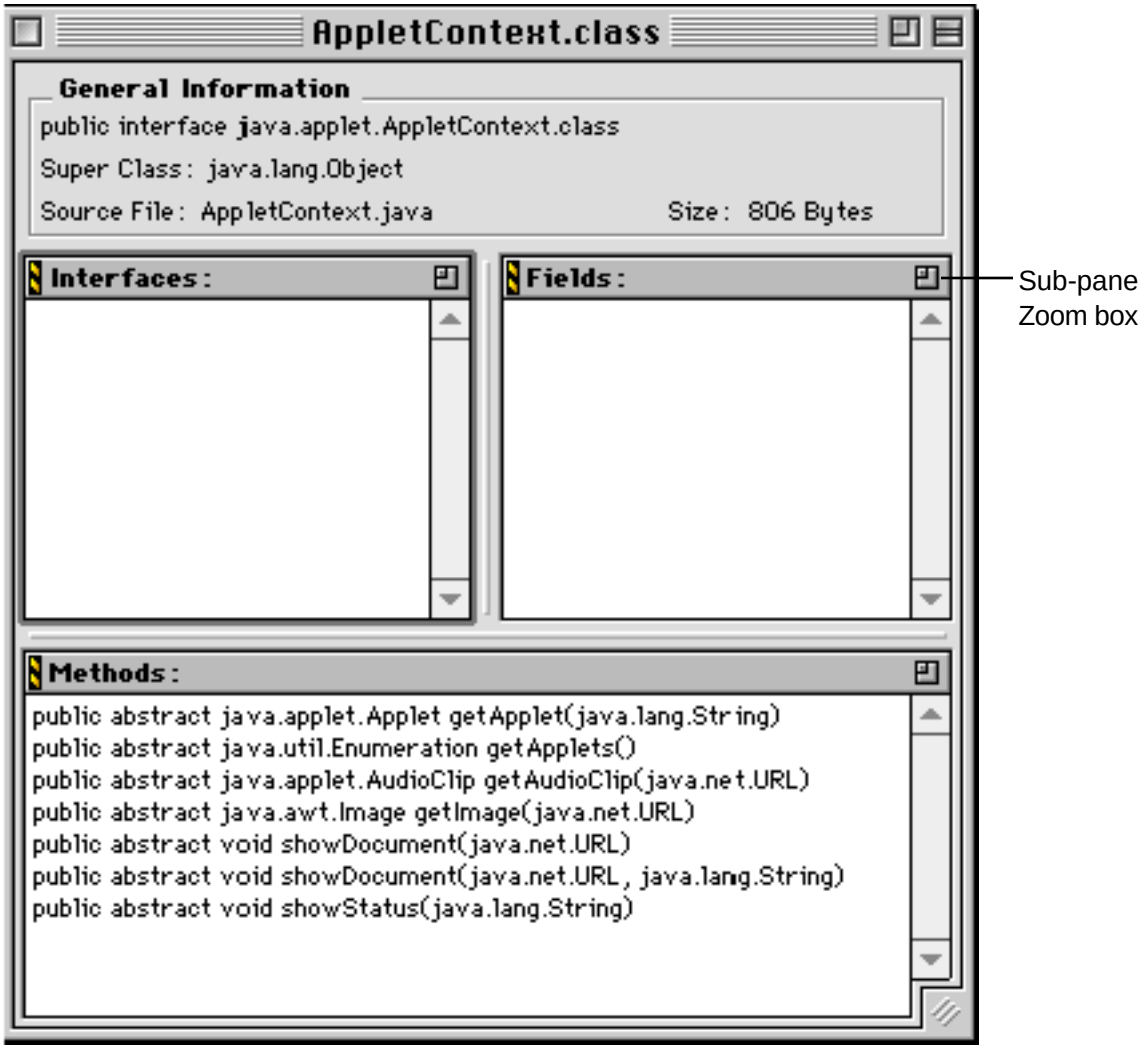
Class Wrangler indicates the file is flagged for deletion by displaying the file name in italics. You must choose **File > Save** to completely remove the files.

Getting Information on Files

You can see information on any class file in a zip archive. Just select the class file and choose **Package > Get Info** or click the Information button in the Class Wrangler window. Class Wrangler displays a window of information on the public class in the class file, including information on its fields, methods, interfaces, and superclass, as shown Figure 8.4.

Information windows are associated with an open archive document. If the associated archive document is closed, all of its information windows are closed.

Figure 8.4 The Class Wrangler Get Information window



Click the sub-pane zoom box control to expand the pane to fill the entire information window, or shrink the pane to see all sub-panes.

Moving Files Between Archives

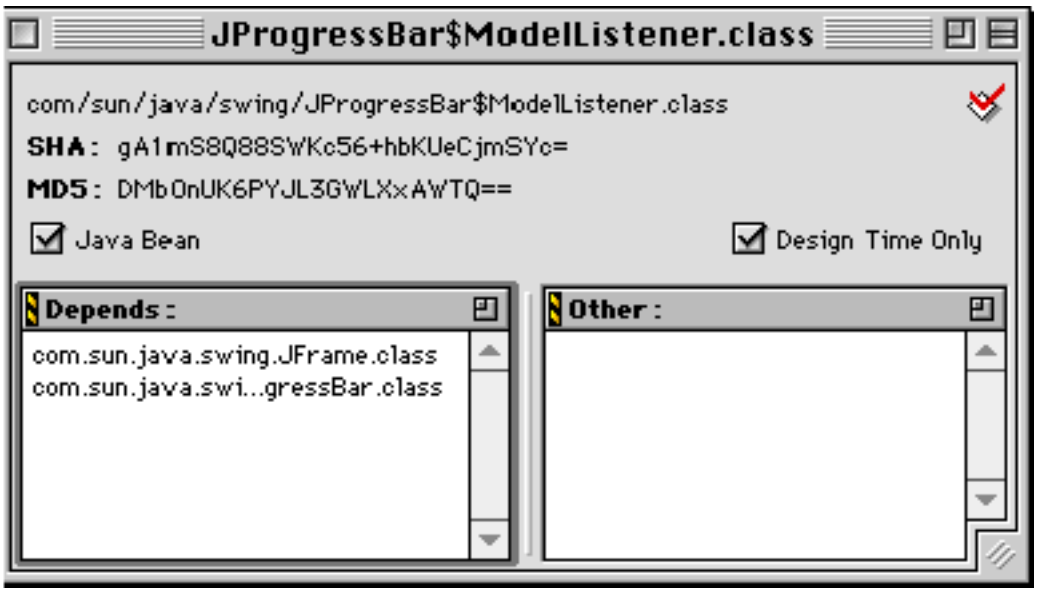
You can move a file from one archive to another. Open both archives in Class Wrangler, and drag the files from one window to another. Class Wrangler copies the files from the source archive into

the destination archive. The files are not removed from the source archive.

Editing Manifest Files

Class Wrangler also allows you to edit some manifest data for the currently selected item in the active archive. Choose **Package > Edit Manifest** to edit the manifest data for the currently selected item.

Figure 8.5 Edit Manifest dialog



Manifest data is displayed for the item (Figure 8.5) for you to verify. This data reflects the state of the item in memory, not on disk. So the JavaBean item will display the state of that item as it would currently be saved to disk, not as it was originally.

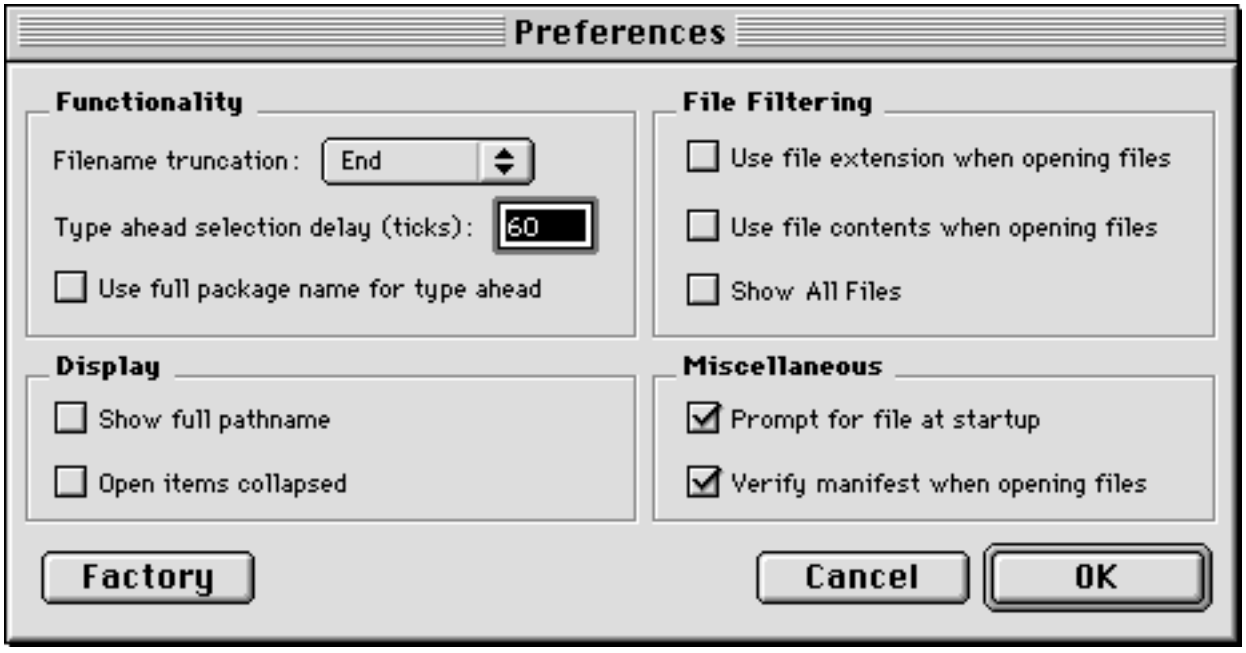
Certain items, such as the name, **SHA**, and **MD5** hash codes, cannot be modified. The **Depends** list is a list of items. Any text can be dragged to this pane in order to add it to the list. The **Other** list holds all manifest tags that are not yet specifically parsed by Class Wrangler. These could include new tags from Sun or special user-defined tags.

See also Sun's documentation for the complete explanation of manifest tags and the manifest file format.

Class Wrangler Preferences

This section discusses how to control the way Class Wrangler works for you. As with most programs, Class Wrangler has preferences you can set to control its operation. When you choose **Edit > Preferences**, Class Wrangler displays a Preferences dialog box as shown in Figure 8.6.

Figure 8.6 The Class Wrangler Preferences dialog



Click the **Factory** button to restore the settings to program defaults.

The effect of these options is described in the following topics:

- Functionality Settings
- File Filtering Settings
- Display Settings

- Miscellaneous Settings

Functionality Settings

The first group of preference settings control how you interact with Class Wrangler.

Filename truncation

The **Filename truncation** pop-up menu operates similar to the same menu in the **Java Target** panel in the CodeWarrior IDE. On the Mac OS, file names cannot be greater than 31 characters long. If you extract a class file that is greater than 31 characters, Class Wrangler uses this setting to determine where the extra characters will be eliminated from. Options are: **Front**, **Middle**, and **End**.

If you choose **End**, Class Wrangler preserves any file name extension in the final file name.

Type ahead selection delay (ticks)

The amount of time, in ticks, Class Wrangler will wait before selecting the file represented by typing on the keyboard.

Use full package name for type ahead

Enable this option to force type ahead selection based on the full package name of the class. For example:

```
java/awt/Adjustable.class
```

File Filtering Settings

When you add or open a file using the standard file dialog box, Class Wrangler displays only zip, Jar, and class files in the file list. You control how Class Wrangler identifies the proper files with this group of options in the Preferences dialog.

If you use files created on Mac OS computers, turn off all options. When these options are off, Class Wrangler looks at a file's Mac OS file type to determine whether a file is a zip archive, Jar file, or a

class file. Class Wrangler assumes that class files have a file type of `'Clss'` or `'COåk'`. Zip archives have a file type of `'ZIP '` or `'ZipF'`. Jar files have a file type of `'ZIP '`.

If you use files created on other operating systems (such as Windows or UNIX), you will need to turn on one of the options. When you transfer such a file to a Mac OS computer, the file does not have the proper Mac OS file type. Class Wrangler needs some other mechanism it can use to identify that the file is appropriate.

NOTE: These options do not affect drag and drop operations. To successfully drop a file, it must have the proper Mac OS file type.

Use File Extension when opening files

Will only show files with a `.zip`, `.jar`, or `.class` extension.

TIP: Use Internet Config to map common Java extensions, such as .zip and .jar to the appropriate Mac OS file types. See "Use Internet Config for File Mappings" on page 161 for more information.

Use file contents when opening files

Will only show files based on their contents. This option is useful if a zip or Jar file does not have the appropriate extension.

Show All Files

Displays all files in the list regardless of file extension or content. This option overrides the other options, even if they are set.

TIP: Each of these options are also available to you in the Open dialog box.

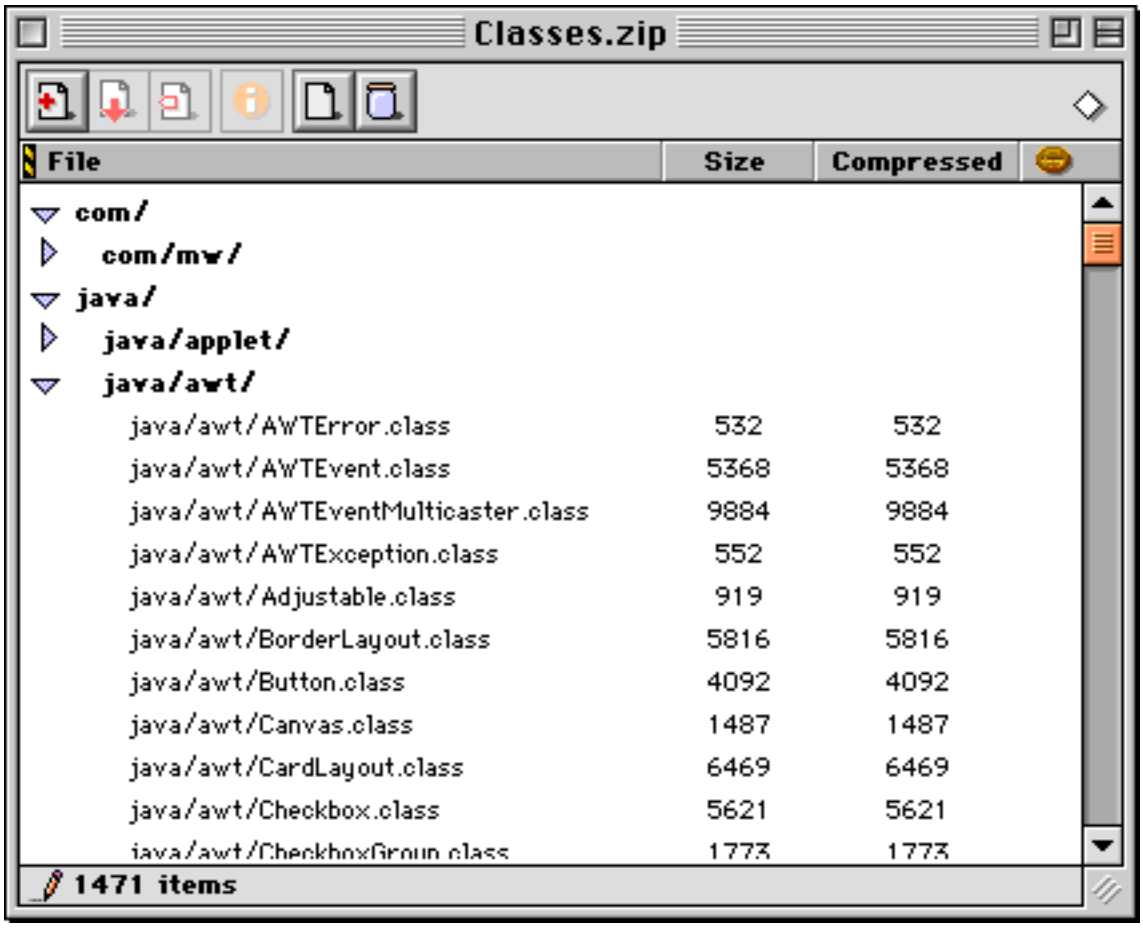
Display Settings

The display settings control how Class Wrangler displays a zip/Jar file.

Show full pathname

If the **Show full pathname** option is on in the Preferences dialog box, Class Wrangler shows the fully qualified name for every entry in its window, as shown Figure 8.7.

Figure 8.7 The Class Wrangler window with full pathnames



If this option is off, only the name of that entry's package or class file is shown.

Open items collapsed

If the **Open items collapsed** option is on in the Preferences dialog box, the program will always open archives with all hierarchical entries collapsed. Click the disclosure triangle to display the contents of a package.

Miscellaneous Settings

The last two settings tell Class Wrangler what to do at startup and when opening files.

Prompt for file at startup

If the **Prompt for file at startup** option is on, Class Wrangler will ask you to specify a file to open when you launch the program.

Verify manifest when opening files

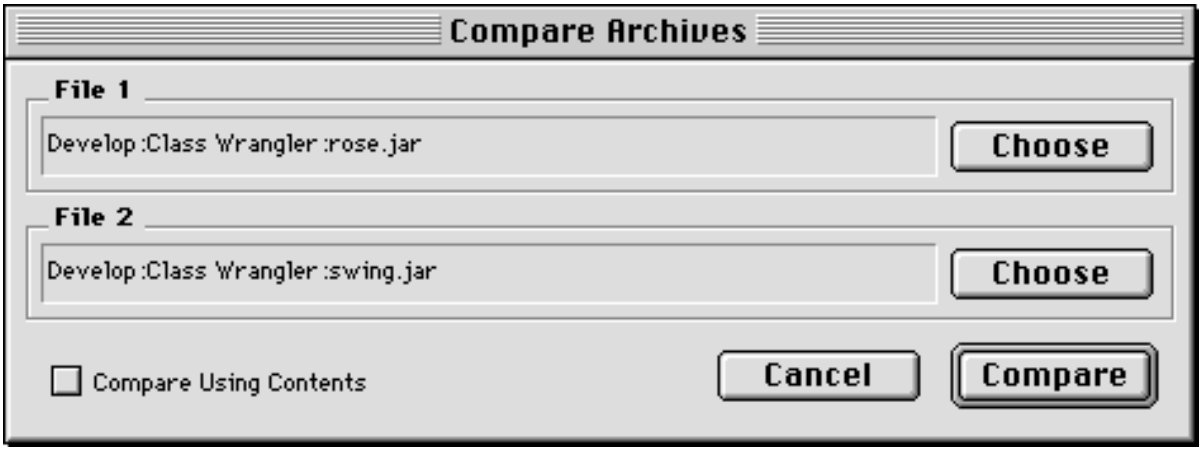
Verifies the manifest information, if present, of a Jar file as Class Wrangler opens the file. Jar files will take longer to open with this option enabled. If a particular entry that has a manifest entry does not verify, Class Wrangler draws it in red.

Comparing Archives

Class Wrangler allows you to compare two Jar files to determine their differences. You can compare archives on two levels. At the first level, comparison is based on the names and positions of items in the Jar file. The second level compares the Jar files by differences in the contents of each item.

To compare two Jar files, choose **File > Compare Files**. The Compare Archives dialog, shown in Figure 8.8, is straight forward to use.

Figure 8.8 Compare archives dialog



Click the **Choose** button to pick the files you wish to compare. If you want to compare the files based on differences in the content of each item, click the **Compare Using Contents** checkbox. Click **Compare** to start the process.

NOTE: Comparing files requires a lot of RAM. This is especially true if you are comparing large Jar files. It may be necessary to increase the memory partition of Class Wrangler in some cases.

Once the compare process is completed, each archive is displayed in a separate window. Items that are unique to an archive are drawn in green. Items that are in both archives, but are different, are drawn in blue.

TIP: You will only see items drawn in blue with the **Compare Using Contents** option enabled.

A

Standalone Applets for Mac OS

You can use CodeWarrior to create standalone Mac OS Java applications. This release of CodeWarrior uses a method for creating standalone Java applications which makes use of JBindery. This chapter will walk you through creating standalone applications.

Using JBindery, CodeWarrior "binds" the neccessary code to your Java code to make standalone applications.

Topics discussed in this section are:

About the JBindery Application

JBindery is an application that binds the VM interface to Java code to create standalone Java applications. It is supplied with the Mac OS Runtime for Java (MRJ) software development kit from Apple Computer Inc. MRJ can be found on the CodeWarrior CD in the **Cool Demos, SDKs, & Tools** folder. For more information on JBindery and other parts of the MRJ SDK, see the documentation provided with MRJ.

Creating a Standalone Application

Following is a step-by-step guide to creating a standalone Mac OS Java application using JBindery.

1. **Create a new Java project.**

   Choose **File > New Project**. Select `Java Application` from the Java section and click **OK**. CodeWarrior will ask you to name and save the new project to your hard disk via a standard save file dialog box.

2. **Add your Java files.**

   Choose **Project > Add Files** to add your Java source files to the project. Make sure you add your files to the correct build target if you have more than one build target in your project.

3. **Open the Target Settings dialog box.**

   Choose **Edit > Java Application Settings…** to open the Target Settings dialog box.

NOTE: The wording of this menu item changes to reflect the name of the build target being displayed in the project window. If you changed the name of your build target, then look for that name in this menu.

---

4. **Activate the Java Mac OS Post Linker.**

   Click **Target Settings** in the panel list on the left-hand side of the dialog box. The Target Settings panel appears. Choose **Java Mac OS Post Linker** from the **Post-Linker** pop-up menu This activates the Java Mac OS Post Linker and tells CodeWarrior to use the settings in the **Java Mac OS Settings** panel when building your project

5. **Change the main class setting.**

   Click **Java Target** in the panel list on the left-hand side of the dialog box. The **Java Application Settings** dialog box appears. Click on **Target Settings** in the panel list to the left of the dialog box to display the Target Settings panel. Change the **Main Class** field to the same name as your application's main class.

6. **Set the Output Type.**

   Select **Java Mac OS Settings** from the panel list on the left-hand side of the dialog box. This displays the Java Mac OS Settings panel.

   Choose **JBindery** from the **Mac OS Java Output Type** pop-up menu.

7. **Redirect stdin and stdout.**

   If you want `stdin` and `stdout` to be redirected or ignored, choose the appropriate menu items in the two **Redirect** pop-up menus.

8. **Set the Creator and Output Filename.**

   Change the **Creator** and **Output Filename** fields to the creator and filename you want your application to have.

9. **Set the Minimum and Maximum heap sizes.**

   Set the **Minimum app heap** to the minimum number of bytes your application will be allowed to run in. Set the **Maximum app heap** to the prefered number of bytes for your applications heap.

10. **Save your settings.**

    Click the **Save** button at the bottom of the dialog box to save all of your settings. Then close the dialog box.

11. **Compile and run.**

    Choose **Project > Make** to create your application. Once it has been successfully built, double-click it's icon to run it.

B

Troubleshooting

This section is designed to give you a quick reference point for common problems with their solutions when targeting the Java VM with CodeWarrior. This should be the first place you look before contacting technical support.

Troubleshooting questions and answers cover the following main categories:

Programming Problems

This section deals with problems encountered writing Java code using the CodeWarrior development environment.

Cannot Find Main Class in Java Application

Problem

The following errors are encountered when building Java applications (depending on which host CodeWarrior is running on, and which VM is being used):

- `Could not execute class name: The system cannot find the file specified.`
- `Can't find class class name`
- `NoClassDefFound`

Solution

The common cause of this error is the incorrect entry of the **Main Class** name in the Java Target panel. Open this panel by choosing **Edit > *Target* Settings** (where ***Target*** is the name of the build target

in question), then select **Java Target** from the settings panels list on the left side of the window. Make sure that the **Main Class** field matches the name of the main class in your code. Remember that class names are case sensitive.

If this field does match your main class exactly, make sure that you have a virtual machine installed on your computer. CodeWarrior can use the Apple MRJ, Sun JDK 1.1.6 (or higher), and Microsoft Internet Explorer 4.0 as a virtual machine. We include all of these on the CodeWarrior CD(s). You can specify which one you want to use via the **Virtual Machine** pop-up menu in the Java Target panel. You can also download the latest versions from Apple, Sun or Microsoft.

Invalid Class Name in Applet Tag

Problem

This error may be worded in one of the following ways, depending on which host CodeWarrior is running on, and which VM is being used:

- `File not found when looking for: class name`
- `java.lang.NullPointerException`
- `load: class class name not found`

Solution

This is usually the result of the class name specified in the `applet` HTML tag not matching the actual class name in the Java code. Make sure that the class name in the `applet` tag matches the name of the main class in your code. Also be sure that the name of the class file matches the name of the main class. Remember that class names are case sensitive.

Debugging Classes.zip

Problem

When stepping through Java code, you cannot step into the `classes.zip` library by default.

Solution

To debug classes.zip, requires that you switch to the debug version of the library. Locate the supplied `Classes_debug.zip` library. It can be found in the following location:

Windows `Metrowerks CodeWarrior\Java Support\Libraries\Classes_debug.zip`

Mac OS `Metrowerks CodeWarrior:Java Support:Libraries:Classes_debug.zip`

Drag this file into the same group as classes.zip in the project window. And remove the classes.zip file from your project. Now you can debug any class in classes.zip.

Additional Problems

If you find you are having problems in this area, please send a bug report to Metrowerks support, `support@metrowerks.com`, and that information will be added to this manual in the next release.

Conversion Problems

This section deals mainly with problems encountered converting older CodeWarrior Java projects to the latest release.

Cannot Convert Older Droplet Projects

Problem

The Droplet project type was used in previous versions of CodeWarrior to make a “pseudo” stand alone applet. Droplet is no longer supported by CodeWarrior.

Solution

While the Droplet type is no longer supported, you can still make a Stand Alone Java application with CodeWarrior. The CodeWarrior project converter does not convert your project to the new type for you. The best method is to use the StandAlone Applet stationery and create a new project. Then add your Java source files to this project.

See “Standalone Applets for Mac OS” on page 137 for more information.

Index

D

E

F

G

Z

CodeWarrior

Targeting the Java VM

Credits

writing lead: David Blache

other writers: L. Frank Turovich, Marc Paquette, Chris Magnuson, Derek Saldana, Jim Trudeau

engineering: Greg Bolsinga, Kevin Buettner, John Cortell, Bernie Estavillo, Michael Farrar, Tim Freehill, Michael Stricklin

frontline warriors: Greg Bolsinga, Bar

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting Java	How to use CodeWarrior to program for Java
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows