

Using the PersonalJava™ Emulation Environment

Version 3.1

January 26, 2000



Copyright © 2000 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, CA 94303 USA
All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun Logo, Java and all Java-based marks, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Govt is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227 -19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202 -3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE

LEGALLY INVALID.

Preface

This user guide describes how to install and use the PersonalJava™ emulation environment (PJEE). It focuses on runtime issues like installation, configuration and running Java™ technology-based software.

Note: This document refers to the Java 2 platform as JDK 1.2 software.

Audience

The primary reader is a software developer who is responsible for writing or testing Java technology-based software for the PersonalJava application environment (PJAE).

Additional Reading

The following documents provide important related information:

- The **PersonalJava Product Page** at <http://java.sun.com/products/personaljava> provides the latest information about PersonalJava technology.
- The **PersonalJava Application Environment Specification** at <http://java.sun.com/products/personaljava/>, *version 1.2* describes the API relationship between the PJAE and the JDK™ 1.1 release.
- **Using the PersonalJava Compatibility Classes** (PJCC) can be found at <http://java.sun.com/products/personaljava/pj-cc.html>. It describes how developers can use JDK technology-based tools to compile and execute programs that use classes specific to the PJAE.
- **Using JavaCheck™** at <http://java.sun.com/products/personaljava/javacheck.html> describes a developer tool for performing static analysis of Java technology-based software to determine whether it is compatible with a specific Java technology-based application environment.
- The **Touchable Look & Feel Specification** describes a flexible look & feel design intended for consumer devices based on a touchscreen display.
- The **PersonalJava Porting Guide** describes how to port the PersonalJava application environment (PJAE) to a target RTOS.
- **The Java Language Specification** (Addison-Wesley, 1996) at <http://java.sun.com/docs/books/jls/index.html> is the standard reference for the Java programming language.
- **The Java Virtual Machine Specification** (Addison-Wesley, 1996) at <http://java.sun.com/docs/books/vmspec/index.html> is the standard reference for the Java virtual

machine.

- **The JDK 1.1. API reference documentation** at <http://java.sun.com/products/jdk/1.1/docs/api/packages.html> describes the API of the class library.
- Security in JDK 1.2 software is described at <http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html>. It describes the fine-grained access control security model incorporated into this release.

Technical Support

For technical comments or questions, please visit the **Java Technology Licensee Engineering** website.

Copyright © 2000 Sun Microsystems, Inc.

Introduction

The PersonalJava™ emulation environment (PJEE) is a standalone software test tool. It allows developers to test their applications against a desktop-based implementation of the PersonalJava application environment (PJAE). The PJEE is delivered in binary form as a software application for Microsoft Windows NT and for the Solaris™ Operating Environment.

Note: The PJEE serves a different purpose than the Java Runtime Environment (JRE). The JRE is a reference implementation of the Java application environment for desktop platforms like Microsoft Windows NT and the Solaris Operating Environment. The PJEE is not a target application environment; it is a developer tool for testing applications.

This user guide describes how to install and use the PJEE for testing Java technology-based software.

PJEE and the PersonalJava Application Environment Specification

The **PersonalJava Application Environment Specification, version 1.2** states that different parts of the PJAE API are **required**, **optional** or **modified**. For example, since file system support for the PJAE is optional, the part of `java.io` related to file I/O is also optional. Therefore, an implementation of the PJAE may omit this part of `java.io` and still conform to the **PersonalJava Application Environment Specification, version 1.2**.

Every optional feature of the **PersonalJava Application Environment Specification, version 1.2** has been omitted from the PJEE except for the optional parts of `java.io` and `java.util.zip`. In addition, the PJEE provides support for only a single locale: `en_US`.

Because it is a software test tool, the PJEE has not been fully optimized for performance. This allows the PJEE to be used with debugging tools such as `jdb`.

Look & Feel

The PJEE is available in two versions of look & feel: the test platform's native look & feel (Solaris platform/Motif or Microsoft Windows NT) or the new Touchable look & feel, in which the Truffle Peer set has been enhanced to support runtime properties for scaling and square buttons.

- **Scaling**
To scale all toolkit elements by the same amount, run PJEE with the following property:
-`Dsun.awt.touchable.scale=<scale factor>`, By default, scale factor = 100.
- **Square Buttons**
By default, buttons are painted with round corners. All buttons are painted square if

-Dsun.awt.touchable.squareButtons=true is set or if the scale factor is less than or equal to 50.

What's New

Here is a list of new features in this version of the PJEE:

- Fine-grained access control based on the JDK 1.2 release
- Optional code-signing mechanism based on the JDK 1.2 release

Software Contents

The PJEE installation program contains the software necessary to run the PJEE on either Microsoft Windows NT or the Solaris platform. **Note:** The PJEE does **not** include a web browser.

The table below describes the software contents of the PJEE directory after installation. Because it can be located anywhere on a given file system, this directory is referred to as *PJEE-dir* in this document.

Component	Description	
COPYRIGHT	Copyright notice for the PJEE.	
LICENSE	License agreement for the PJEE.	
bin/pjava	<i>Optimized version.</i>	The PersonalJava application launcher loads and executes Java applications.
bin/pjava_g	<i>Debug version.</i>	
bin/pjavaw	<i>Optimized version.</i>	(Microsoft Windows NT only). A special version of the PersonalJava application launcher that does not create a console window.
bin/pjavaw_g	<i>Debug version.</i>	
bin/pappletviewer	<i>Optimized version.</i>	The PersonalJava applet viewer loads HTML pages that contain Java applets.
bin/pappletviewer_g	<i>Debug version.</i>	
bin/*.dll	(Microsoft Windows NT only). Dynamic link libraries (DLLs) for the virtual machine and native methods of the PersonalJava class library.	
bin/sparc	(Solaris platform only). Binary executables for the invocation tools called by the a front-end shell scripts in bin.	
doc/*	This user guide.	

lib/appletviewer.properties	Status message strings and security policy for sun.applet.AppletViewer.	Properties files for the PJEE. See System Properties for a description of Java system properties that are available through the -D command line option for pjava.
lib/awt.properties	Key and modifier name strings used by java.awt.event.KeyEvent.	
lib/content-types.properties	MIME content type description file used by sun.net.www. Each entry maps a MIME content type to a native application that can handle it. Files are associated with a MIME content type by either the MIME content type returned by an HTTP header or their file name extension.	
lib/font.properties	Platform-dependent font description file. See Adding Fonts to the Java Runtime for more information.	
lib/jvm.hprof.txt	Header text for reports generated by the Java heap profiler. See Profiling Java Programs and pjava.	
lib/touchable.palettes	Database containing RGB values for named color palettes for the Touchable look & feel design.	
lib/classes.zip	<i>Optimized version.</i>	Zip archive containing the PersonalJava class library.
lib/classes_g.zip	<i>Debug version.</i>	
lib/sparc	(<i>Solaris platform only</i>). Shared libraries for the virtual machine and native methods of the PersonalJava class library.	
lib/java.security	File containing security related properties	
lib/java.policy	File specifying the security policy for Java technology-based applications	

Installing the PJEE

The PJEE can be installed on either Microsoft Windows NT or the Solaris platform. The **PJEE download page** contains the following platform-specific PJEE installation programs:

Platform	Installation Program	Description
Microsoft Windows NT	pjee3_1-win32.exe	A self-extracting application for installing the PJEE.
Solaris 2.6 or greater	pjee3-solaris.sh	A self-extracting shell script for installing the PJEE.

Microsoft Windows NT

Hardware Requirements

- Pentium-based PC
- 32 MB memory
- 20 MB free disk space

Software Requirements

- **Microsoft Windows NT Workstation**, version 4.0
- **WinSock 2 Library**
- System locale based on the ISO 8859-1 (Latin-1) character set

Installation

The following steps describe how to install the PJEE on a Microsoft Windows NT-based system.

1. *(Optional.)* **Remove any previous versions of the PJEE or JRE.** This step helps to avoid software conflicts with this version of the PJEE.
2. **Download the PJEE installation program for Microsoft Windows NT from the PJEE download page.**
3. **Drag the PJEE installation program's icon to the target installation directory.**
4. **Run the PJEE installation program.** The PJEE installation program will perform some tests and present a few dialogs during the installation process. These include a dialog that prompts the user to agree to a binary license before installing the PJEE and a dialog that selects a destination directory.
5. **Select a system locale based on the ISO 8859-1 (Latin-1) character set.**
 - a. Choose **Control Panel->Regional setting->Region.**
 - b. Select **English (United States).**

The PJEE can now be used. Additional steps may be necessary to correctly define the `PATH`, `JAVA_HOME` and `CLASSPATH` environment variables. This is important in cases where a system has another version of a Java application environment, like the JDK, or additional Java technology-based class files that have been installed separately from the PJEE. In these cases, the `PATH` and `CLASSPATH` environment variables may need modification to avoid conflicts.

PATH

The `PATH` environment variable defines a list of directories that the DOS shell uses as a search path for finding executable programs like the PJEE invocation tools (`pjava` and `pappletviewer`).

Here's how to add a directory to the `PATH` environment variable:

1. **Use a text editor to edit `C:\autoexec.bat`, the DOS shell startup script.**
2. **Find the `PATH` environment variable definition.** For example,

```
PATH=C:\windows;C:\tools\bin
```

3. **Add the absolute path of the `PJEE-dir\bin` directory to the list of directories in the `PATH` definition.** For example, if the PJEE has been installed in a directory named `C:\PJEE-dir`, then the absolute path of the directory containing the invocation tools is:

```
C:\PJEE-dir\bin
```

The above string would then be inserted into the list of directories in the `PATH` definition. For example,

```
PATH=C:\PJEE-dir\bin;C:\windows;C:\tools\bin
```

If the `PATH` environment variable definition contains more than one directory, a semi-colon (`;`) is used as a name separator.

4. **Save the changes to `C:\autoexec.bat`, the DOS shell startup script.**
5. **Quit the text editor.**
6. **Restart the command prompt.**

JAVA_HOME

Many of the problems users have with getting an application to run properly on a specific Java technology-based application environment can be traced to a misdefined `JAVA_HOME` environment variable. In principle, `JAVA_HOME` environment variable should either be unset or it should be set to absolute path of PJEE directory that was created when installing the PJEE. For example, `JAVA_HOME` environment variable can be unset as follows:

```
JAVA_HOME=
```

If the problem persists, `JAVA_HOME` should be set to absolute path of PJEE directory that was created when installing the PJEE as follows:

```
JAVA_HOME=C:\PJEE-dir
```

CLASSPATH

Many of the problems users have with getting an application to run properly in a specific Java technology-based application environment can be traced to a misdefined `CLASSPATH` environment variable. In principle, all that is required is that each of the application's class and resource files be stored in one of the locations in the `CLASSPATH` environment variable. But in practice, the `CLASSPATH` environment variable can have many different kinds of name conflicts.

The following two sub-sections provide background material on the `CLASSPATH` environment variable and a set of procedures for modifying the `CLASSPATH` environment variable.

CLASSPATH Fundamentals

Java technology-based applications are collections of class and resource files that are built on one system and then installed on potentially many different target platforms. The file system on a target platform may be very different from the development platform. For example, target platforms may organize class files in different ways or they may have multiple Java technology-based applications. Therefore, application environments like the PJEE use the `CLASSPATH` environment variable as a flexible mechanism for balancing the needs of platform-independence and the realities of file systems on different target platforms.

The `CLASSPATH` environment variable allows the Java virtual machine to load classes that depend on other classes which are not part of the default platform class library. If the virtual machine finds a file with the correct file name, it attempts to load it. Otherwise, it generates a `NoClassDefFoundError` error.

The `CLASSPATH` environment variable defines a list of locations that the Java virtual machine uses as a search path for finding class and resource files. A location can be either a directory in a file system, a jar archive file or a Zip archive file that contains class or resource files. Locations in the `CLASSPATH` environment variable are delimited by a platform-dependent name separator (e.g., a semi-colon ";" on Microsoft Windows NT). For example,

```
CLASSPATH=C:\java\MyClasses;\java\MoreClasses.zip
```

The Java programming language uses packages to organize classes into a hierarchical namespace. For example, `java.awt.image` is a package that contains a number of image-related classes. When the virtual machine searches the locations in the `CLASSPATH` environment variable, it uses each location as a root for performing a search. In the case of `java.awt.image.BufferedImageOp` the virtual machine would start with each location in the `CLASSPATH` environment variable and then try to find a subdirectory named `java\awt\image` that contains a class file named `BufferedImageOp.class`. This search method is applied to both file system directories and virtual directories in Zip and jar files.

Since the `CLASSPATH` environment variable is not required by the default PJEE installation, the DOS shell startup script may not have a definition statement for it. The `CLASSPATH` environment variable can be removed if no extra directories are needed beyond the default set described above. By default, the `CLASSPATH` definition used internally by `pjava` is

```
CLASSPATH=.
```

The `-classpath` command line option for `pjava` overrides the current `CLASSPATH` environment variable

definition.

The `-classpath` command line option and the `CLASSPATH` environment variable are used for specifying application classes. System classes (also known as bootstrap classes), such as those typically found in `classes.zip`, are specified using the `-bootclasspath` command line option. If no `-bootclasspath` option is given, `pjava` searches for system classes in `PJEE-dir\lib\classes.zip`. For more information regarding `classpath` and `bootclasspath`, see the security section.

Modifying CLASSPATH

Here's how to modify the `CLASSPATH` environment variable definition:

1. **Use a text editor to edit `C:\autoexec.bat`, the DOS shell startup script.**
2. **Find the `CLASSPATH` environment variable definition.** For example,

```
CLASSPATH=C:\java\MyClasses
```

3. **Modify the `CLASSPATH` definition to add or remove directories.** The example directory below contains class files for a Java technology-based application.

```
C:\java\OtherClasses
```

The above string would then be inserted into the list of directories in the `CLASSPATH` definition. For example,

```
CLASSPATH=C:\java\MyClasses;C:\java\OtherClasses
```

If the `CLASSPATH` environment variable definition contains more than one location, a semi-colon (;) is used as a name separator.

4. **Save the changes to the DOS shell startup script.**
5. **Quit the text editor.**
6. **Restart Microsoft Windows NT.**

Native Method Search Path

The PJEE installation program manages the task of installing DLLs that contain implementations of native methods used by the PersonalJava class library.

If additional native method DLLs are needed, they should either be placed in the `PJEE-dir\bin` directory or in one of the directories in the `PATH` environment variable which is used by the Microsoft Windows NT runtime as a search path for finding DLLs.

Removing the PJEE

The following steps describe how to remove the PJEE from a Microsoft Windows NT system.

1. **Run the removal utility.** When the PJEE installation program for Microsoft Windows NT installs the PJEE, it adds a removal utility with the Windows Registry.

- a. **Open the Control Panel folder by choosing Settings->Control Panel from the Start menu in the Task Bar**
 - b. **Launch the Add/Remove Programs utility.**
 - c. **Select the PJEE from the software list.**
 - d. **Press the Add/Remove button.**
2. **Restore the PATH or CLASSPATH environment variables to their original values.** If these environment variables have been modified during the PJEE installation procedure, their original definitions should be restored.

The Solaris Operating Environment

Hardware Requirements

- SPARC™ platform-based workstation
- 32 MB memory
- 20 MB free disk space

Software Requirements

- **Solaris Operating Environment**, release 2.6 or greater, SPARC platform version
- System locale based on the ISO 8859-1 (Latin-1) character set

Installation

The following steps describe how to install the PJEE on the Solaris platform.

Note: The procedures below describe how to set various environment variables. There are many different mechanisms for defining environment variables because there are several different UNIX shell programs (e.g., `csh(1)`, `sh(1)` and `ksh(1)`), each with one or more mechanisms for defining environment variables. The actual method used in the procedures below is based on the `setenv` command for the `csh(1)` shell.

1. *(Optional.)* **Remove any previous versions of the PJEE or JRE.** This step helps to avoid software conflicts with this version of the PJEE.
2. **Change the shell's current directory to the target installation directory.**

```
% cd install-directory
```

The PJEE installation program will unpack the PJEE software into the shell's current directory, even if the PJEE installation program is in a **different** directory.

3. **Download the PJEE installation program for the Solaris platform from the PJEE download page.**
4. **Run the PJEE installation program.**

```
% ./pjee3_1-solaris.sh
```

The PJEE installation program will prompt the user to agree to a binary license before installing the PJEE.

5. **Select a system locale based on the ISO 8859-1 (Latin-1) character set.** This is controlled by the `LANG` environment variable. For example,

```
% setenv LANG en_US
```

The PJEE can now be used. Additional steps may be necessary to correctly define the `PATH`, `JAVA_HOME`, `CLASSPATH` and `LD_LIBRARY_PATH` environment variables. This is important in cases where a system has another version of a Java technology-based application environment, or additional class files that have been installed separately from the PJEE. In these cases, the `PATH` and `CLASSPATH` environment variables may need modification to avoid conflicts.

PATH

The `PATH` environment variable defines a list of directories that the UNIX shell uses as a search path for finding executable programs like the PJEE invocation tools (`pjava` and `pappletviewer`).

Here's how to add a directory to the `PATH` environment variable:

1. **Use a text editor to edit `~/.cshrc` the `csh(1)` shell startup file.**
2. **Find the `PATH` environment variable definition.** For example,

```
setenv PATH ./bin:/usr/bin:/usr/sbin
```

3. **Add the absolute path of the `PJEE-dir\bin` directory to the list of directories in the `PATH` definition.** For example, if the PJEE has been installed in a directory named `/java/PJEE-dir` then the absolute path of the directory containing the invocation tools is:

```
/java/PJEE-dir/bin
```

The above string would then be inserted into the list of directories in the `PATH` definition. For example,

```
setenv PATH ./java/PJEE-dir/bin:/bin:/usr/bin:/usr/sbin
```

If the `PATH` environment variable definition contains more than one directory, a colon (`:`) is used as a name separator.

4. **Save the changes to the shell startup file.**
5. **Quit the text editor.**
6. **Restart the shell.**

JAVA_HOME

Many of the problems users have with getting an application to run properly on a specific Java technology-based application environment can be traced to a misdefined `JAVA_HOME` environment variable. In principle, `JAVA_HOME` environment variable should either be unset or it should be set to absolute path of PJEE directory that was created when installing the PJEE. For example, `JAVA_HOME` environment variable can be unset as follows:

```
unsetenv JAVA_HOME
```

If the problem persists, `JAVA_HOME` should be set to absolute path of PJEE directory that was created when installing the PJEE as follows:

```
setenv JAVA_HOME /java/PJEE-dir
```

CLASSPATH

Many of the problems users have with getting an application to run properly in a specific Java technology-based application environment can be traced to a misdefined `CLASSPATH` environment variable. In principle, all that is required is that each of the application's class and resource files be stored in one of the locations in the `CLASSPATH` environment variable. But in practice, the `CLASSPATH` environment variable can have many different kinds of name conflicts.

The following two sub-sections provide background material on the `CLASSPATH` environment variable and a set of procedures for modifying the `CLASSPATH` environment variable.

CLASSPATH Fundamentals

Java technology-based applications are collections of class and resource files that are built on one system and then installed on potentially many different target platforms. The file system on a target platform may be very different than the development platform. For example, target platforms may organize class files in different ways or they may have multiple Java technology-based applications. Therefore, application environments like the PJEE use the `CLASSPATH` environment variable as a flexible mechanism for balancing the needs of platform-independence and the realities of file systems on different target platforms.

The `CLASSPATH` environment variable allows the Java virtual machine to load classes that depend on other classes which are not part of the default platform class library. If the virtual machine finds a file with the correct file name, it attempts to load it. Otherwise, it generates a `NoClassDefFoundError` error.

The `CLASSPATH` environment variable defines a list of locations that the Java virtual machine uses as a search path for finding class and resource files. A location can be either a directory in a file system, a jar archive file or a Zip archive file that contains class or resource files. Locations in the `CLASSPATH` environment variable are delimited by a platform-dependent name separator (e.g., a colon ":" on Solaris). For example,

```
setenv CLASSPATH /java/MyClasses:/java/MoreClasses.zip
```

The Java programming language uses packages to organize classes into a hierarchical namespace. For example, `java.awt.image` is a package that contains a number of image-related classes. When the virtual machine searches the locations in the `CLASSPATH` environment variable, it uses each location as a root for performing a search. In the case of `java.awt.image.BufferedImageOp` the virtual machine would start with each location in the `CLASSPATH` environment variable and then try to find a subdirectory named `java/awt/image` that contains a class file named `BufferedImageOp.class`. This search method is applied to both file system directories and virtual directories in Zip files.

Since the `CLASSPATH` environment variable is not required by the default PJEE installation, the shell startup file may not have a definition statement for it. The `CLASSPATH` environment variable can be

removed if no extra directories are needed beyond the default set described above. By default, the CLASSPATH definition used internally by pjava is:

```
setenv CLASSPATH .
```

The `-classpath` command line option for pjava overrides the current CLASSPATH environment variable definition.

The `-classpath` command line option and the CLASSPATH environment variable are used for specifying application classes. System classes (also known as bootstrap classes), such as those typically found in `classes.zip`, are specified using the `-bootclasspath` command line option. If no `-bootclasspath` option is given, pjava searches for system classes in `PJEE-dir/lib/classes.zip`. For more information regarding the classpath and bootclasspath, see the security section.

Modifying CLASSPATH

Here's how to modify the CLASSPATH environment variable definition:

1. **Use a text editor to edit `~/.cshrc` or the `csh(1)` shell startup file.**
2. **Find the CLASSPATH environment variable definition.** For example,

```
setenv CLASSPATH /java/MyClasses
```

3. **Modify the CLASSPATH definition to add or remove directories.** The example directory below contains class files for a Java application.

```
/java/OtherClasses
```

The above string would then be inserted into the list of directories in the CLASSPATH definition. For example,

```
setenv CLASSPATH /java/MyClasses:/java/OtherClasses
```

If the CLASSPATH environment variable definition contains more than one location, a colon (:) is used as a name separator.

4. **Save the changes to the shell startup file.**
5. **Quit the text editor.**
6. **Restart the shell.**

Native Method Search Path

The PJEE installation program manages the task of installing shared libraries that contain implementations of native methods used by the PJAE class library.

If additional native method shared libraries are needed, they should either be placed in the `PJEE-dir/lib/sparc` directory or in one of the directories in the `LD_LIBRARY_PATH` environment variable, which is used by the Solaris runtime as a search path for finding shared libraries.

Removing the PJEE

The following steps describe how to remove the PJEE from a Solaris Operating Environment system:

1. **Remove the *PJEE-dir* directory and all its contents.**

```
% rm -rf PJEE-dir
```

2. **Restore the `PATH` or `CLASSPATH` environment variables to their original values.** If these environment variables have been modified during the PJEE installation procedure, their original definitions should be restored.

Copyright © 2000 Sun Microsystems, Inc.

Running Java Technology-Based Programs

Java technology-based programs can run on the PJEE on either Microsoft Windows NT or the Solaris Operating Environment. Each implementation has a set of invocation tools for running Java technology-based software:

Invocation Tool	Description	
pjava pjavaw	<i>Optimized version.</i>	The PersonalJava application launcher loads and executes applications.
pjava_g pjavaw_g	<i>Debug version.</i>	
pappletviewer	<i>Optimized version.</i>	The PersonalJava applet viewer loads HTML pages that contain applets.
pappletviewer_g	<i>Debug version.</i>	

pjava_g and pappletviewer_g include symbol tables for debugging.

Microsoft Windows NT

The PJEE includes a set of invocation tools for launching applications and loading HTML files that contain applets. Running Java technology-based software on the PJEE is based on using one of these invocation tools with command line options that identify the Java technology-based software and control various runtime options.

pjava

Here is an example of how to use pjava to launch a Java application and run it on the PJEE:

```
C:\> pjava HelloWorld
```

Note: The `PJEE-dir\bin` directory must be in the `PATH` environment variable. See `PATH` for a description of how to modify the `PATH` environment variable.

This example loads a class file named `HelloWorld.class` in the current directory. Note that the `.class` suffix is omitted from the command line. The PJAE application launcher locates and executes the `main` method in the `HelloWorld` class which then runs the application. If `HelloWorld` has a GUI, the PJEE creates a window for displaying it.

pjavaw

The `pjavaw` command is identical to `pjava`, except that `pjavaw` does not create a console window for displaying a standard output stream. Here's how to launch an application with `pjavaw`:

1. Choose the `Run` command from the `Start` menu.

2. Enter the full path name of the `pjavaw` executable or use the `Browse` button to find the executable.
3. Enter the file name for the main application class and any other command line arguments.
4. Press the `Ok` button.

The PJEE will then launch the application. The behavior is identical to the `pjava` command with the exception that it does not create or use a console window.

pappletviewer

`pappletviewer` is a test program for loading applets. It reads an HTML file, parses the first `<APPLET>` tag while ignoring all other HTML tags, and then loads and executes the corresponding applet.

Here is an example,

```
C:\> pappletviewer HelloWorldApplet.html
```

Note: The `PJEE-dir\bin` directory must be in the `PATH` environment variable. See **PATH** for a description of how to modify the `PATH` environment variable.

The Solaris Operating Environment

The PJEE includes a set of invocation tools for launching applications and loading HTML files that contain applets. Running Java technology-based software on the PJEE is based on using one of these invocation tools with command line options that identify the Java technology-based software and control various runtime options.

pjava

Here is an example of how to use `pjava` to launch an application and run it on the PJEE:

```
% pjava HelloWorld
```

Note: The `PJEE-dir/bin` directory must be in the `PATH` environment variable. See **PATH** for a description of how to modify the `PATH` environment variable.

This example loads a class file in the current directory named `HelloWorld.class`. Note that the `.class` suffix is omitted from the command line. The PJAE application launcher locates and executes the `main` method in the `HelloWorld` class which then runs the application. If `HelloWorld` has a GUI, the PJEE creates a window for displaying it.

pappletviewer

`pappletviewer` is a test program for loading applets. It read an HTML file, parses the first `<APPLET>` tag while ignoring all other HTML tags, and then loads and executes the corresponding applet.

Here is an example,

```
% pappletviewer HelloWorldApplet.html
```

Note: The `PJEE-dir/bin` directory must be in the `PATH` environment variable. See `PATH` for a description of how to modify the `PATH` environment variable.

System Properties

The PJEE uses two standard mechanisms for specifying system options:

- JDK API system properties contain information about the system and environment in which a Java technology-based program is running. The following sections contain tables that describe standard JDK API system properties and PJEE-specific system properties. See **System Property Example** for an example of how to use `pjava` with a command line option that specifies a system property value.
- JDK API properties files are text files located in `PJEE-dir/lib` that control platform-specific features like fonts and MIME content-type handlers. See **Software Contents** for a description of these JDK API properties files.

Java technology-based applications can also provide user-level options that are based on JDK API properties.

Standard System Properties

The table below describes standard JDK API system properties that are part of the **JDK 1.1.x API**.

Property	Type	Description					
<code>file.encoding</code>	<i>string</i>	The system locale's character encoding. See Supported Encodings in JDK 1.1 Internationalization Overview .					
<code>file.encoding.pkg</code>	<i>string</i>	The package that contains the classes for converting between the system locale's character encoding and Unicode.					
<code>file.separator</code>	<i>string</i>	<table border="0" style="width: 100%;"> <tr> <td style="text-align: center; width: 50%;">Microsoft Windows NT</td> <td style="text-align: center; width: 5%;">\</td> <td rowspan="2" style="vertical-align: middle;">Platform-dependent name separator used in path names.</td> </tr> <tr> <td style="text-align: center;">Solaris Operating Environment</td> <td style="text-align: center;">/</td> </tr> </table>	Microsoft Windows NT	\	Platform-dependent name separator used in path names.	Solaris Operating Environment	/
Microsoft Windows NT	\	Platform-dependent name separator used in path names.					
Solaris Operating Environment	/						
<code>java.class.path</code>	<i>string</i>	Class path in platform-dependent form.					
<code>java.compiler</code>	<i>string</i>	Specifies a JIT compiler to use.					
<code>java.class.version</code>	<i>string</i>	Class file version number.					
<code>java.home</code>	<i>string</i>	PJEE installation directory.					
<code>java.vendor</code>	<i>string</i>	PJEE vendor-specific string.					
<code>java.vendor.url</code>	<i>string</i>	PJEE vendor URL.					
<code>java.version</code>	<i>string</i>	Version number of PJAE specification.					

<code>line.separator</code>	<i>string</i>	Microsoft Windows NT \r\n Solaris Operating Environment \n	Platform-dependent line separator used in text files.
<code>os.arch</code>	<i>string</i>	Host OS architecture.	
<code>os.name</code>	<i>string</i>	Host OS name.	
<code>os.version</code>	<i>string</i>	Host OS version.	
<code>path.separator</code>	<i>string</i>	Microsoft Windows NT ; Solaris Operating Environment :	Platform-dependent name separator used in search paths.
<code>user.dir</code>	<i>string</i>	User's current working directory.	
<code>user.home</code>	<i>string</i>	User's home directory.	
<code>user.language</code>	<i>string</i>	ISO 639 language code of the system locale.	
<code>user.name</code>	<i>string</i>	User's account name.	
<code>user.region</code>	<i>string</i>	ISO 3166 country code of the system locale.	
<code>user.timezone</code>	<i>string</i>	The POSIX.1 time zone name.	

PJEE-Specific System Properties

The table below describes PJEE-specific system properties.

Note: These PJEE-specific system properties are for diagnostic purposes only. They should not be used in production versions of Java technology-based programs.

Property	Type	Default
<code>awt.toolkit</code>	<i>reference</i>	<code>sun.awt.touchable.TouchableToolk:</code>
<code>sun.awt.platform.pixelType</code>	<i>string</i>	<code>color:8</code>
<code>sun.awt.aw.DefaultCursor</code>	<i>reference</i>	<code>sun.awt.aw.Touchable.FingerPrint</code>
<code>sun.awt.im.InputMethod</code>	<i>reference</i>	<code>null</code>

sun.awt.im.Japanese	<i>boolean</i>	false
sun.awt.im.NoVirtualKeyboard	<i>boolean</i>	false
sun.awt.otk.noRandomSelectionMode	<i>boolean</i>	true
sun.awt.otk.ObjectToolkit	<i>reference</i>	sun.awt.otk.ObjectToolkit
sun.awt.otk.textWordSelectionOff	<i>boolean</i>	false
sun.awt.palette	<i>string</i>	Orange
sun.awt.palette.definitions	<i>string</i>	<i>PJEE-dir/lib/touchable.palettes</i>
sun.awt.touchable.doNotDrawFocusRectangle	<i>boolean</i>	false
sun.awt.touchable.paletteClass	<i>reference</i>	sun.awt.otk.ColorPalette
sun.awt.touchable.sound	<i>boolean</i>	false
sun.graphicssystem	<i>reference</i>	sun.awt.aw.GraphicsSystem
sun.graphicssystem.height	<i>integer</i>	480
sun.graphicssystem.width	<i>integer</i>	640
sun.windowssystem	<i>reference</i>	sun.awt.aw.WindowSystem

sun.boot.class.path	<i>string</i>	PJEE-dir/lib/classes.zip
---------------------	---------------	--------------------------

System Property Example

The `pjava` command uses the `-D` command line option to specify the value of a system property. For example,

```
% pjava -Dsun.awt.palette=Sand HelloWorld
```

Security in PJAE

The security system provided in this release is based on the policy-based, easily configurable, fine-grained access control security present in the *Java 2 SDK, Standard Edition, v 1.2* (referred to as "JDK 1.2" in this document). This section provides a high-level overview of the security model in the JDK 1.2 release. For more detailed information regarding the security system in the JDK 1.2 release, see <http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html>.

When code is loaded, it is assigned permissions based on the security policy currently in effect. Each permission specifies a permitted access to a given host, port, etc. For example, read or write access can be specified for files and directories and connect access can be specified for a given host and port. The policy that specifies which permissions are available for code from various locations can be initialized from an external, configurable policy file. Code that is not granted permission may not access the resource guarded by that permission. This fine-grained access control can be specified and extended for applets, applications, JavaBeans™ components, servlets, etc. Support for fine-grained access control is required.

An implementation of the PJAE may also optionally include code signing support. Code signing support includes support for the Java Cryptography Extension (JCE) API from the JDK 1.2 release, including APIs for digital signatures and message digests, and certificate interfaces for parsing and managing certificates. An X.509 v3 implementation of the certificate interfaces is provided. The implementation also supports downloading and verifying signed jar files (for applets only).

PJAE 1.2 is based on the JDK 1.1.8 release. To support the security model from the JDK 1.2 release, several classes from that release were added to PJAE. These are listed in the PJAE specification. In addition to these classes, other changes not necessarily reflected by the public specification were made to adopt semantics from the JDK 1.2 release if required to support the security model. Some of these changes include:

- The addition of "bootclasspath"
- Inclusion of property files related to security, such as `java.security` and `java.policy`
- The classloader delegation model was not adopted.

These are described in more detail below.

Security Policy File

The core PJAЕ security system provides a policy-controlled, permission-based, fine-grained, extensible, access control model. This is based on the security system currently present in the JDK 1.2 release. A policy file is used to initialize the policy for the security system in the PJAЕ. For more information regarding the default policy implementation and policy file syntax, see [Default Policy Implementation and Policy File Syntax](#).

In implementations which do not support code signing, the `signedBy` keyword is not supported by the policy file parsing code. If this keyword is encountered, an `UnsupportedOperationException` will be thrown and the policy file will not be parsed. The following example shows how the security model is policy-controlled, permission-based, and fine-grained.

```
grant {
    permission java.io.FilePermission ".tmp","read";
}

grant codeBase "http://mysecureserver.mydomain.com/mailapp" {
    permission java.io.SocketPermission "mydomain.com", "connect";
}
```

In the above example, all code is given permission to read files in the `.tmp` directory. Code downloaded from `http://mysecureserver.mydomain.com/mailapp` is allowed to make IP network connections to any server in the domain `mydomain.com`. Note that the permission granted to a code base includes an action; it is not a binary "Yes/No" permission. In this manner, permissions may be granted or denied based on the URL from which the code was downloaded and/or the signers of that code.

Extensible Security Model

To extend this model, subclass the `Permission` class and implement the `implies()` method. The policy may now include permissions that are checked using this new `Permission` class. For more information regarding the `Permissions` class, see [Permissions in the Java 2 SDK](#).

The class `TVPermission` extends `Permission` and implements the `implies()` method to enable permission checking. This class is part of the jar file that comprises the Java TV API tuner implementation. The policy file is now enhanced with the following permission:

```
grant codeBase "http://trustedserver.mydomain.com/tuningapp" {
    com.tci.TVPermission "Channel 2-12", "tune";
}
```

While checking permissions, the Java TV API implementation makes the call:

```
AccessController.checkPermission(new com.tci.TVPermission("channel5", "tune");
```

This call would succeed if the appropriate policy were in place. In this manner, applications can extend the security model.

Bootstrap Classes versus Application Classes

In the JDK 1.1 software-based security model, applications are given the same privileges and permissions as Java platform classes (i.e., classes found in `classes.zip`). The user has some control over which applications execute on the platform through the use of code signing. However, once the application is executing, there was no easy way to limit access to various resources. The JDK 1.2 software-based security model provides the capability to limit an application's access to system resources. However, to do so, it is necessary to distinguish between bootstrap classes (also known as system classes) and application classes. Bootstrap classes are the classes that comprise the Java platform (for example, the classes contained in `classes.zip`). Bootstrap classes require a wider range of access to system resources that are normally not accessible to application code.

To distinguish between bootstrap classes and application classes, the concept of a *bootclasspath* has been added. Classes loaded from the bootclasspath are assumed to be Java platform classes and are given a wider range of security privileges when they are loaded. Classes loaded from the standard classpath are given narrower permissions as defined by the `java.policy` file. When loading a class, the bootclasspath is always searched first.

The addition of the bootclasspath concept required changes to PJAEE VM command line options, as well as changes in the semantics of the `-classpath` command line option and the `CLASSPATH` environment variable.

The `-bootclasspath` option has been added as a new command line option for the `pjava` command line. This option has syntax identical to the `-classpath` option. When loading classes, the classpath components specified by the `-bootclasspath` option are always searched first. Classes loaded from the bootclasspath are then given a wider range of access to system resources. If the `-bootclasspath` option is not specified, a default search path is installed. This default search path includes `classes.zip`.

With the addition of the `-bootclasspath` command line option, the semantics of the `-classpath` command line option and the `CLASSPATH` environment variable change slightly from previous releases. Classes loaded from classpath components specified by either the `-classpath` command line option or the `CLASSPATH` environment variable are given privileges as defined by the platform security policy. The security policy by default is derived from the `java.policy` file. A user may also specify an alternate policy file by setting the `java.security.policy` property. The `-classpath` command line option and `CLASSPATH` environment variable may no longer be used to specify the location of the bootstrap classes (e.g., `classes.zip`). The default setting for the application classpath also no longer contains references to bootstrap classes (e.g., `classes.zip`). Other than this, the behavior and purpose of `-classpath` and `CLASSPATH` remain the same.

If the system is built without support for `java.io.File` or the "file" URL protocol handler, the implementation will not assign default values to `classpath` or `bootclasspath`. Loading classes from `classpath` and `bootclasspath` implies that classes may be loaded locally from a file system. If the

implementation does support a file system, the implementation assumes no classes can be loaded from classpath or bootclasspath and, therefore, does not assign a default value for either. The security system creates code bases for the elements of classpath and bootclasspath in the form of file URLs. Hence the requirement for `file` URL protocol handler support.

The Java Native Interface (JNI) Invocation Interface

The changes to classpath discussed above also apply to native applications that invoke a VM using the JNI invocation interface. These applications must ensure that the bootclasspath and classpath are set correctly before invoking the VM. The current implementation uses the JDK 1.1 software version of the JNI invocation interface. The classpath is set using the classpath element of the `JDK1_1InitArgs` structure. This structure is passed as an argument to the `JNI_CreateJavaVM()` function. The bootclasspath may be set by passing the `sun.boot.class.path` property set to the desired bootclasspath. This is set using the properties element of the `JDK1_1InitArgs` structure.

Default SUN™ Security Provider

In PJAЕ version 1.2, the `java.security.Provider` class has become a required class. PJAЕ version 1.2 supports the Provider class that is compliant with the JDK 1.1.8 API and provides the default SUN™ security provider. However, its behavior differs depending on whether optional code signing is included.

If PJAЕ is built with the code signing option, the `sun.security.provider` package includes all the functions that JDK 1.1's `sun.security.provider` supports:

- An implementation of the Digital Signature Algorithm (NIST FIPS 186).
- An implementation of the MD5 (RFC 1321) and SHA-1 (NIST FIPS 180-1) message digest algorithms.

If PJAЕ is not built with the code signing option, the SUN provider package only supports:

- An implementation of the SHA-1 (NIST FIPS 180-1) message digest algorithms.

For more information on the JDK 1.1.8 compliant Provider class, see the following documentation:

- Java Platform 1.1 API Specification -- Class `java.security.Provider`
- Java Cryptography Architecture API Specification & Reference (of JDK 1.1.8)

`java.security.manager` Property

By default, when an applet is executed, a security manager is installed for the applet. The same is not true for applications. By default, applications run without a security manager. When a security manager is not specified, the system does not enforce many of the security policies related to what a program is allowed to do. The class `java.lang.SecurityManager` specifies the checks a security manager normally performs. To enable a security manager for applications, set the `java.security.manager`

property. This property is set on the command line using the `-D` command line option; for example:

```
pjava -Djava.security.manager Foo
```

This installs the default security manager. The user can specify an alternate security manager as follows:

```
pjava -Djava.security.manager=MySecurityManager Foo
```

where `MySecurityManager` specifies a class that represents the alternate security manager.

When the security manager is enabled, the security policy is enforced by the system.

Security Tools

Tools exist for creating and modifying security policy files, managing a keystore, and creating and verifying signed jar files. These tools are available in JDK 1.2 software. For more information, regarding these tools, see Summary of JDK 1.2 Security Tools. To download the JDK 1.2 software, see the JDK 1.2 software product page.

Troubleshooting

Here are some troubleshooting tips for running the PJEE.

- The PJEE operates with memory limits that are set at runtime. If an application requests more memory than the PJEE has available, the PJEE will throw an exception and the application will exit. The **application launcher** has command-line options for specifying memory limits on an application and thread basis.
- If `pappletviewer` does not correctly load applets, try using the `pjava` command directly:

```
% pjava -verbose sun.applet.AppletViewer URL
```

This generates a list of classes the AppletViewer tries to load and where it's trying to load them from. Check to make sure that the class files exist and are uncorrupted.

- If the PJEE generates one of the following fatal error messages:

```
StringParameterAction: Exception: java.net.MalformedURLException: unknown protocol  
MakeApplicationProtectionDomain: failed
```

verify the implementation was built with `java.io.File` support and support for the File URL protocol handler. If the system was explicitly built without these options, verify that the `CLASSPATH` environment variable is not set and the `-classpath` command line option is not being used. Loading classes from the application classpath is dependent upon an underlying file system and support for the File URL protocol handler in the class libraries.

- (*Microsoft Windows NT*) If the PJEE generates one of the following error messages:

```
net.socketException: errno = 10047  
Unsupported version of Windows Socket API
```

check which TCP/IP drivers are installed. Third-party TCP/IP drivers may not work correctly because the PJEE supports only the Microsoft TCP/IP drivers included with Microsoft Windows NT.

- (*Microsoft Windows NT*) If you cannot close the AppletViewer copyright window because the launch bar partially covers the copyright notice window's **Accept** and **Reject** buttons, move the Task Bar to the side of the desktop to allow access to the copyright window **Accept** and **Reject** buttons.

Copyright © 2000 Sun Microsystems, Inc.

Debugging Java Technology-Based Programs

The PJEE is a developer tool for testing Java technology-based software. This includes **running** applications to observe their behavior and **debugging** applications to explore the relationships between the source code's structure, the compiled code's behavior and the PJEE's capabilities.

Some of the debugging techniques described here can be used to debug applications on a PersonalJava platform as well as on the PJEE. But for most debugging tasks, the PJEE will be more convenient and have more debugging resources (e.g., symbol tables) than a PersonalJava platform.

Note: The debugging support in the PJEE is based on the **Java Virtual Machine Debugger Interface** (JVMDI). To support compatibility with the PJEE, third-party developer tools must support this interface.

The following sections describe the debugging resources available for PersonalJava technology-based software development and introduce their basic usage.

Using `jdb`

JDK 1.2 software includes the `jdb` command line debugger which can be used to debug programs running on the PJEE. JDK software must be installed either on the same system as the PJEE, or on a system connected over an IP network.

The following steps describe how to use `jdb` to debug an applet running on the PJEE.

1. **Run the *debug* version of the application launcher.** Use the `pjava_g` version with the `-debug` option to enable full debugging support.

```
% pjava_g -ss1024k -debug sun.tools.agent.EmptyApp
```

`EmptyApp` is a placeholder application which is used by `jdb` when it is launched without an initial class.

Note: The default stack size may be too small for debugging purposes. So it may be necessary to increase the stack size with the `-ssnum` option.

Record the session password *identifier*:

```
Agent password: identifier
```

2. **Run `jdb` with the session password identifier string and an optional remote host address.**

```
% jdb -host pjava_host -password identifier
```

pjava_host is the host name or IP address of the system running the PJEE. *identifier* is the session

password identifier displayed by the application launcher in the previous step.

3. **Load the `AppletViewer` class:**

```
> load sun.applet.AppletViewer
```

4. **Set a break point in the applet:**

```
> stop in HelloWorldApplet.paint
```

5. **Run the `AppletViewer` class with an additional *URL* argument that indicates an HTML page that contains the applet:**

```
> run sun.applet.AppletViewer HelloWorldApplet.html
```

At this point, execution should be stopped at the first line in the `paint` method, and `jdb` should be connected to the PJEE for debugging. See `jdb` for a list of debugging commands or type `help` at the `jdb` command line prompt.

Dumping a Thread Stack

During a `jdb` session, the currently executing thread stack can be dumped with a platform-specific key sequence:

Platform	Key Sequence
Microsoft Windows NT	CONTROL-break
Solaris Operating Environment	CONTROL-\

Generating Diagnostic Output

The most basic method for generating useful data from an application at runtime is to use the `println` method. This technique displays a text stream on the standard output. If the PJAE implementation is running on a device, then the device must be attached to a development system with a serial cable so that the standard output stream can be captured with a communications terminal program.

Similarly, the best way to get runtime information about a native method is to use `printf()` to format data for display on a terminal window or through a serial port.

Debugging Native Methods

Debugging native methods requires techniques that are different from those used for methods written in the Java programming language. There are two basic approaches:

- Use `printf()` to format data for display in a terminal window or through a serial port. This is sometimes the only way to debug native code on a device that does not yet have native debugger support.

- Use a native debugger to track program execution and examine data. This technique is described below in an example for `gdb` in the Solaris Operating Environment.

Native Debugger Notes

A native debugger used with the PJEE should be compatible with the symbol tables generated by the compiler used to build the PJEE. The Solaris Operating Environment version of the PJEE was built with the GNU C Compiler, version 2.7.3. The Microsoft Windows NT version was built with Microsoft Visual C++, version 5.0.

The **PersonalJava Environment Software (PJES)** includes a build environment for building binary executable versions of the PJAE like the PJEE. The PJES build environment has a mechanism for including native libraries in the list of object files that are statically linked with the PJAE binary executable. This procedure can be used to include native methods for applications that will be bundled with an implementation of the PJAE on a device. Debugging statically linked native code is much easier than debugging native code that is included in a shared library.

See the **PersonalJava Application Environment Porting Guide** for a description of how to include object files in the PJES build environment.

A `gdb`-based Example for the Solaris Operating Environment

The PJEE includes versions of the PersonalJava invocation tools that include symbol tables for debugging applications with native debuggers like `gdb`(1). The following procedure outlines the steps involved in using `gdb` to debug a simple Java technology-based application containing a native method. The emphasis here is on the mechanics of debugging rather than special debugging techniques for different programming problems.

1. **Explicitly define the `LD_LIBRARY_PATH` environment variable.** `gdb` uses the debugging executable version instead of front-end shell script, which usually sets this environment variable.

```
% setenv LD_LIBRARY_PATH .:PJEE-dir/lib/sparc
```

2. **Launch `gdb` with `pjava_g`.** Be sure to use the debugging executable version of the application launcher instead of the front-end shell script.

```
% gdb PJEE-dir/bin/sparc/pjava_g
```

3. **Set a break point in `main()`.**

```
(gdb) tbreak main
```

4. **Launch `pjava_g` with the `HelloWorld` class.** This loads the symbol tables for the `pjava_g` application launcher.

```
(gdb) run HelloWorld
```

5. **Set a break point in `sysAddDLsegment()`.**

```
(gdb) break sysAddDLSegment
```

Note: `sysAddDLSegment()` is an internal VM function that dynamically links shared libraries for native methods. It is used here solely as a convenient reference for setting a break point **after** a shared library has been linked. This technique may not apply to other VM implementations.

6. **Allow execution to proceed to the point where the shared library that contains the native method implementation has been linked.** This procedure involves one or more cycles of the following steps:

- a. *Continue execution to the break point at the beginning of `sysAddDLSegment()`.*

```
(gdb) continue
```

- b. *Continue again until the function returns.*

```
(gdb) finish
```

- c. *Determine whether the shared library has been loaded.*

```
(gdb) share
```

- If not, repeat the steps above.
- If so, set a break point in the shared library, according to the instructions below.

7. **Clear the break point at the beginning of `sysAddDLSegment()`.**

```
(gdb) clear sysAddDLSegment
```

8. **Set a break point in the shared library.**

```
(gdb) break HelloWorldImpl.c:Java_HelloWorld_greet
```

9. **Continue execution to the break point in the shared library.**

```
(gdb) continue
```

10. **Step through the native code.**

```
(gdb) step
```

`gdb` has several other commands that display source code, show a stack trace and examine data. These are described in **Debugging with GDB**.

Profiling Java Technology-Based Programs

Profiling is the measurement of runtime data for a specific application on a specific runtime system. The PJEE includes profiling support as a command-line option for the `pjava` application launcher.

Note: The profiling support in the PJEE is based on the **Java Virtual Machine Profiler Interface** (JVMPi). To support compatibility with the PJEE, third-party developer tools should support this interface.

The mechanics for using the profiler in the PJEE are very simple:

1. **Select an application for profiling.**
2. **Choose a set of `-Xhprof` command line options for the profiler.** See the `pjava` manual page for a list of command line options for the profiler.
3. **Run the application with `pjava_g` and the profiler options.** For example,

```
% pjava_g -Xhprof:monitor=y HelloWorld
```

4. **Examine the data in the profile report file.** By default, the profiler generates an ASCII report file named `java.hprof.txt`.

Copyright © 2000 Sun Microsystems, Inc.

pjava - The PersonalJava™ Application Launcher

Synopsis

```
pjava [ options ] class_name [ argument ... ]
pjavaw [ options ] class_name [ argument ... ]
pjava_g [ options ] class_name [ argument ... ]
pjavaw_g [ options ] class_name [ argument ... ]
```

Description

The **pjava** invocation tool launches a Java technology-based application on the PJEE. It does this by starting the PJAE virtual machine, loading *class_name*, and invoking that class's **main** method which must have the following signature:

```
public static void main(String[])
```

By default, the first non-option argument is the name of the class to be loaded. A fully-qualified class name should be used. The PersonalJava virtual machine searches for the startup class, and other classes used, in three sets of locations: the bootstrap class path, the installed extensions, and the user class path.

Non-option arguments after the class name are passed to the **main** function.

pjava_g is a non-optimized version of **pjava** suitable for use with debuggers like **jdb**. The naming convention for identifying debug shared libraries is to append *_g* to the library file name. If the library was *libhello.ext*, the debug version would be *libhello_g.ext*.

(*Microsoft Windows NT only*). The **pjavaw** command is identical to **pjava**, except that **pjavaw** does not create a console window for displaying a standard output stream. To launch a Java technology-based application with **pjavaw**, use the **Run** command on the **Start** menu and give it the full path name of the **pjavaw** executable along with the main application class and any command line arguments.

Example

```
% pjava com.yournamehere.HelloWorld
```

Options

-debug

Allows the debugger, **jdb**, to attach itself to a **pjava** session. When **-debug** is specified

on the command line **pjava** displays a password which must be used when starting the debugging session.

-classpath *path*

Specifies the search path the virtual machine uses to look up application class files. Directories are separated by colons. Thus, the general format for *path* is:

```
.:<your_path>
```

For example:

```
./:/home/xyz/classes
```

This command line option overrides the `CLASSPATH` environment variable if it is set.

-bootclasspath *path*

Specifies the search path the virtual machine uses to look up system (also known as bootstrap) class files. Directories are separated by colons. Thus, the general format for *path* is:

```
<your_path>:<next_path>
```

For example:

```
/pjava/lib/classes.zip:/usr/local/java/classes
```

For more information regarding the distinction between the **-classpath** and **-bootclasspath** options, see the security section.

-mxnum

Sets the maximum size of the memory allocation pool (the garbage collected heap) to *num*. The default is 16 megabytes of memory. *num* must be greater than or equal to 1000 bytes. The maximum memory size must be greater than or equal to the startup memory size (specified with the `-ms` option, default 16 megabytes).

By default, *num* is measured in bytes. The meaning of *num* can be modified by appending either the letter "k" for kilobytes or the letter "m" for megabytes.

-msnum

Sets the startup size of the memory allocation pool (the garbage collected heap) to *num*. The default is 1 megabyte of memory. *num* must be > 1000 bytes. The startup memory size must be less than or equal to the maximum memory size (specified with the `-mx` option, default 16 megabytes).

By default, *num* is measured in bytes. The meaning of *num* can be modified by appending either the letter "k" for kilobytes or the letter "m" for megabytes.

-noclassgc

Turns off garbage collection of classes. By default, the interpreter reclaims space for unused classes during garbage collection.

-Xhprof[:keyword=value]

Enables heap profiling. Optional parameters can be specified with one or more keyword/value pairs, separated with commas. Valid parameters are:

Option	Range	Default	Description
help			prints a help message for the profiler
heap	dump sites all	all	heap profiling
cpu	samples times old	off	CPU usage
monitor	y n	n	monitor contention
format	a b	a	ASCII or binary output
file	<file>	java.hprof[.txt]	write data to file
net	<host>:<port>	write to file	send data over a socket
cutoff	<value>	0.0001	output cutoff point
lineno	y n	y	line number in traces
thread	y n	n	thread in traces
doe	y n	y	dump on exit

-version

Print the build version information.

-help

Print a usage message.

-ssnum

Each thread has two stacks: one for Java programming language code and one for C code. The `-ss` option sets the maximum stack size that can be used by C code in a thread to *num*. Every thread that is spawned during the execution of the program passed to **pjava** has *x* as its C stack size. The default units for *num* are bytes. The value of *num* must be greater than or equal to 1000 bytes.

By default, *num* is measured in bytes. The meaning of *num* can be modified by appending either the letter "k" for kilobytes or the letter "m" for megabytes. The default stack size is 128 kilobytes ("`-ss128k`").

-ossnum

Each Java thread has two stacks: one for Java programming language code and one for C code. The `-oss` option sets the maximum stack size that can be used by Java programming language code in a thread to *num*. Every thread that is spawned during the execution of the program passed to **pjava** has *num* as its stack size. The default units for *num* are bytes. The value of *num* must be greater than or equal to 1000 bytes.

By default, *num* is measured in bytes. The meaning of *num* can be modified by appending either the letter "k" for kilobytes or the letter "m" for megabytes. The default stack size is 400 kilobytes ("`-oss400k`").

-t

Prints a trace of the instructions executed (**pjava_g** and **pjavaw_g** only).

-v, -verbose

Causes **pjava** to print a message to `stdout` each time a class file is loaded.

-verify

Performs bytecode verification on the class file. Beware however, that **pjava -verify** does not perform a full verification in all situations. Any code path that is not actually executed by the interpreter is not verified. Therefore, **pjava -verify** cannot be relied upon to verify class files unless all code paths in the class file are actually run.

-verifyremote

Runs the verifier on all code that is loaded into the system via a classloader. *verifyremote* is the default for the interpreter.

-noverify

Turns verification off.

-verbosegc

Causes the garbage collector to print out messages whenever it frees memory.

-DpropertyName=newValue

Redefines a property value. *propertyName* is the name of the property whose value will be changed to *newValue*. For example, this command line

```
% pjava -Dawt.button.color=green ...
```

sets the value of the property `awt.button.color` to "green". **pjava** accepts any number of `-D` options on the command line.

-version

Version number of the PJES used to build the PJEE.

-fullversion

String describing the PJES build parameters.

Environment Variables

CLASSPATH

Provides a search path for finding user-defined application classes. Directory names are separated by a platform-specific path separator character. Here is an example for the Solaris Operating Environment:

```
./home/xyz/classes
```

And here is a Microsoft Windows NT example:

```
C:\home\xyz\classes
```

LD_LIBRARY_PATH

(*Solaris Operating Environment only*). Provides a search path for finding shared libraries that contain native method implementations. Directory names are separated by a platform-specific path separator character. Here is an example for the Solaris Operating Environment:

```
./usr/local/lib
```

Microsoft Windows NT uses the `PATH` environment variable as a search path for both binary executables and dynamic link libraries (DLLs) that contain native method implementations.

See Also

- `pappletviewer`
- **Using the PersonalJava Emulation Environment**

Copyright © 2000 Sun Microsystems, Inc.

pappletviewer - The PersonalJava™ Applet Viewer

Synopsis

```
pappletviewer [ options ] URL ...  
pappletviewer_g [ options ] URL ...
```

Description

pappletviewer is a test program for loading applets. It reads the input HTML file *URL*, parses the first `<APPLET>` tag while ignoring all other HTML tags, and then loads and executes the corresponding applet.

Options

-debug

Starts the applet viewer in the debugger, **jdb**, for debugging applets.

-encoding *encoding_name*

Specify the character encoding to be used for parsing the HTML file. The applet itself will use the character encoding of the locale of the environment.

See Also

- `pjava`
- **Using the PersonalJava Emulation Environment**