Using the PersonalJava emulation environment

Version 3.0

January 6, 1999

January 6, 1999

Using the PersonalJava emulation environment

Table of Contents

Preface

This user guide describes how to install and use the PersonalJava emulation environment (PJEE).

Audience

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

Additional Reading

The following documents provide important related information:

- The **PersonalJava Product Page** provides the latest information about PersonalJava technology.
- The **PersonalJava Application Environment Specification** describes the API relationship between the PJAE and JDK 1.1.
- **Using the PersonalJava Compatibility Classes** describes the PJCC which allows developers to use JDK-based tools to compile and execute Java programs that use PersonalJava-specific classes.
- **Using JavaCheck** describes a developer tool for performing static analysis of Java software to determine whether it is compatible with a specific Java application environment.
- **The Java Language Specification** (Addison-Wesley, 1996) is the standard reference for the Java programming language.
- **The Java Virtual Machine Specification** (Addison-Wesley, 1996) is the standard reference for the Java virtual machine.
- The **JDK 1.1.x API reference documentation** describes the API of the Java class library.

Technical Support

For technical comments or questions, please send e-mail to:
`personaljava-comments@java.sun.com`.

Introduction

The PersonalJava emulation environment (PJEE) is a standalone software test tool. It allows developers to test their Java software 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 95/NT and Solaris.

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 95/NT and Solaris. The PJEE is not a target application environment for running Java software; it is a developer tool for testing Java software.

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

PJEE and the PersonalJava Application Environment Specification

The **PersonalJava Application Environment Specification** states that different parts of the PJAE API are **optional**. 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**.

Every optional feature of the **PersonalJava Application Environment Specification** 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 Java debugging tools like **jdb**.

What’s New

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

- This user guide.
- Truffle and the Touchable look & feel.

Software Contents

The PJEE installation program contains the software necessary to run the PJEE on either Microsoft Windows 95/NT or Solaris. **Note:** The PJEE does **not** include a web browser.

The table below describes the software contents of the PJEE.

Component	Description	
`COPYRIGHT`	Copyright notice for the PJEE.	
`LICENSE`	License agreement for the PJEE.	
`bin/pjava`	*Optimized version.*	The **PersonalJava application launcher** loads and executes Java applications.
`bin/pjava_g`	*Debug version.*	
`bin/pjavaw`	*Optimized version.*	(*Microsoft Windows 95/NT only*). A special version of the **PersonalJava application launcher** that does not create a console window.
`bin/pjavaw_g`	*Debug version.*	
`bin/pappletviewer`	*Optimized version.*	The **PersonalJava applet viewer** loads HTML pages that contain Java applets.
`bin/pappletviewer_g`	*Debug version.*	
`bin/*.dll`	(*Microsoft Windows 95/NT only*). Dynamic link libraries (DLLs) for the virtual machine and native methods of the PersonalJava class library.	
`bin/sparc`	(*Solaris 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`.	Java property 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`	*Optimized version.*	Zip archive containing the PersonalJava class library.
`lib/classes_g.zip`	*Debug version.*	
`lib/sparc`	(*Solaris only*). Shared libraries for the virtual machine and native methods of the PersonalJava class library.	

Installing the PJEE

The PJEE can be installed on either Microsoft Windows 95/NT or Solaris. The **PJEE download page** contains the following platform-specific PJEE installation programs:

Platform	Installation Program	Description
Microsoft Windows 95/NT	`pjee3-win32.exe`	A self-extracting application for installing the PJEE.
Solaris 2.5 or greater	`pjee3-solaris.sh`	A self-extracting shell script for installing the PJEE.

Microsoft Windows 95/NT

Hardware Requirements

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

Software Requirements

- **Microsoft Windows 95**
  or
  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 95/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 95/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 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 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 `appletviewer`).

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 file.**

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 the DOS shell startup file.**
5. **Quit the text editor.**
6. **Restart Microsoft Windows 95/NT.**

CLASSPATH

Many of the problems users have with getting a Java application to run properly on a specific Java 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 steps for modifying the `CLASSPATH` environment variable.

Background

Java 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 applications or class libraries. Therefore, Java 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 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 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 95/NT). For example,

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

Java packages organize classes into hierarchical namespaces. 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.

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

Since the CLASSPATH environment variable is not required by the default PJEE installation, the DOS 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

```
CLASSPATH=PJEE-dir\lib\classes.zip;.
```

The -classpath comand line option for pjava overrides the CLASSPATH environment variable definition.

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 file.**
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 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 file.**
5. **Quit the text editor.**
6. **Restart Microsoft Windows 95/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, then 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 95/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 95/NT system.

1. **Run the removal utility.** When the PJEE installation program for Microsoft Windows 95/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.

Solaris

Hardware Requirements

- SPARC-based workstation.
- 32 MB memory.
- 20 MB free disk space.

Software Requirements

- **Solaris operating environment**, release 2.5 or greater, SPARC version.
- System locale based on the ISO 8859-1 (Latin-1) character set.

Installation

The following steps describe how to install the PJEE on a Solaris-based system.

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 Solaris from the PJEE download page.**

4. **Run the PJEE installation program.**

   ```
   % ./pjee3-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`, `CLASSPATH` and `LD_LIBRARY_PATH` environment variables. This is important in cases where a system has another version of a Java application environment, like the JDK, or additional Java 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 `appletviewer`).

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.**

CLASSPATH

Many of the problems users have with getting a Java application to run properly on a specific Java 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 steps for modifying the CLASSPATH environment variable.

Background

Java 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 applications or class libraries. Therefore, Java 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 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 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
```

Java packages organize classes into hierarchical namespaces. 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.

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

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=PJEE-dir/lib/classes.zip:.
```

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

Modifying CLASSPATH

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

1. **Use a text editor to edit `~/.cshrc`, 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 PersonalJava class library.

If additional native method shared libraries are needed, then 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 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.

Running Java Programs

Java programs can run on the PJEE on either Microsoft Windows 95/NT or Solaris. Each implementation has a set of invocation tools for running Java software:

Invocation Tool	Description	
`pjava` `pjavaw`	*Optimized version.*	The **PersonalJava application launcher** loads and excutes Java applications.
`pjava_g` `pjavaw_g`	*Debug version.*	
`pappletviewer`	*Optimized version.*	The **PersonalJava applet viewer** loads HTML pages that contain Java applets.
`pappletviewer_g`	*Debug version.*	

`pjava_g` and `pappletviewer_g` include symbol tables for debugging.

Microsoft Windows 95/NT

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

```
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.

Solaris

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

```
% 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 PersonalJava 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.

appletviewer

`appletviewer` is a test program for loading Java 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,

```
% appletviewer 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:

- Java system properties contain information about the system and environment in which a Java program is running. The following sections contain tables that describe standard Java 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.
- Java property 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 Java property files.

Java application software can also provide user-level options that are based on Java properties and Java property files.

Standard System Properties

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

Property	Type	Description	
`file.encoding`	*string*	The system locale's character encoding. See **Supported Encodings** in **JDK 1.1 Internationalization Overview**.	
`file.encoding.pkg`	*string*	The package that contains the classes for converting between the system locale's character encoding and Unicode.	
`file.separator`	*string*	Microsoft Windows 95/NT `\` Solaris `/`	Platform-dependent name separator used in path names.
`java.class.path`	*string*	Java class path in platform-dependent form.	
`java.compiler`	*string*	Specifies a JIT compiler to use.	
`java.class.version`	*string*	Java class file version number.	
`java.home`	*string*	PJEE installation directory.	
`java.vendor`	*string*	PJEE vendor-specific string.	
`java.vendor.url`	*string*	PJEE vendor URL.	
`java.version`	*string*	Version number of PJAE specification.	
`line.separator`	*string*	Microsoft Windows 95/NT `\r\n` Solaris `\n`	Platform-dependent line separator used in text files.
`os.arch`	*string*	Host OS architecture.	
`os.name`	*string*	Host OS name.	
`os.version`	*string*	Host OS version.	
`path.separator`	*string*	Microsoft Windows 95/NT `;` Solaris `:`	Platform-dependent name separator used in search paths.
`user.dir`	*string*	User's current working directory.	
`user.home`	*string*	User's home directory.	
`user.language`	*string*	ISO 639 language code of the system locale.	
`user.name`	*string*	User's account name.	
`user.region`	*string*	ISO 3166 country code of the system locale.	
`user.timezone`	*string*	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 programs.

Property	Type	Default	Description
`awt.toolkit`	*reference*	`sun.awt.touchable.TouchableToolkit`	Use an alternate implementation of the AWT toolkit.
`sun.awt.`*`platform`*`.pixelType`	*string*	`color:8`	Set the color model and depth.
`sun.awt.aw.DefaultCursor`	*reference*	`sun.awt.aw.Touchable.FingerPrint`	Specify the class used for the cursor. A value of "`blank`" disables the cursor.
`sun.awt.im.InputMethod`	*reference*	`null`	Enable the named input method. A value of "`default`" expands to `sun.awt.otk.SampleInputMethod`.
`sun.awt.im.Japanese`	*boolean*	`false`	If true, then use a Japanese Kana keyboard instead of a QWERTY keyboard.
`sun.awt.im.NoVirtualKeyboard`	*boolean*	`false`	If true, then specify that a physical keyboard is present and disable the virtual keyboard.
`sun.awt.otk.noRandomSelectionMode`	*boolean*	`true`	If true, then disable text cursor behaviour when the user makes a selection away from a text component.
`sun.awt.otk.ObjectToolkit`	*reference*	`sun.awt.otk.ObjectToolkit`	Use an alternate implementation of the Truffle Object Toolkit.
`sun.awt.otk.textWordSelectionOff`	*boolean*	`false`	If true, then disable word selection mode for `mousePressed` with text components.
`sun.awt.palette`	*string*	`Orange`	Specify the name of the selected palette in the palette database.
`sun.awt.palette.definitions`	*string*	*`PJEE-dir`*`/lib/touchable.palettes`	Specify the path name of the palette database.
`sun.awt.touchable.doNotDrawFocusRectangle`	*boolean*	`false`	If true, then disable the focus rectangle.
`sun.awt.touchable.paletteClass`	*reference*	`sun.awt.otk.ColorPalette`	Spcify the name of color palette hashtable.
`sun.awt.touchable.sound`	*boolean*	`false`	If true, then enable sound.
`sun.graphicssystem`	*reference*	`sun.awt.aw.GraphicsSystem`	Use an alternate implementation of the Truffle graphics system.
`sun.graphicssystem.height`	*integer*	480	Set the height of the window containing the graphics system.
`sun.graphicssystem.width`	*integer*	640	Set the width of the window containing the graphics system.
`sun.windowsystem`	*reference*	`sun.awt.aw.WindowSystem`	Use an alternate implementation of the Truffle window system.

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
```

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 **PersonalJava application launcher** has command line options for specifying memory limits on an application and thread basis.

- If `appletviewer` does not correctly load applets, then 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 exists and are uncorrupted.

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

  ```
  Exception in thread NULL
  Unable to initialize threads: cannot find class java/lang/Thread
  ```

  then check the `CLASSPATH` environment variable. It may contain a directory from an older release of the PJAE or a different Java application environment.

- (*Microsoft Windows 95/NT*) If the PJEE generates one of the following error messages:

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

  then 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 95/NT.

- (*Microsoft Windows 95/NT*) If you cannot close the AppletViewer copyright window because the launch bar partially covers the copyright notice window's **Accept** and **Reject** buttons, then move the Task Bar to the side of the desktop to allow access to the copyright window **Accept** and **Reject** buttons.

Debugging Java Programs

The PJEE is a developer tool for testing Java software. This includes **running** Java software to observe its behavior and **debugging** Java software 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 Java software on a PersonalJava device 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 device.

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 should support this interface.

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

Using `jdb`

The JDK 1.1.x includes the `jdb` command-line debugger which can be used to debug Java programs running on the PJEE. The JDK must be installed on either 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 a Java applet running on the PJEE.

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

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

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

2. Record the session password *identifier*:

   ```
   % Agent password: identifier
   ```

3. **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 PersonalJava application launcher in the previous step.

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

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

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

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

6. **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, `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 95/NT	`CONTROL-break`
Solaris	`CONTROL-\`

Generating Diagnostic Output

The most basic method for generating useful data from a Java 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 the `printf()` function to format data for display on a terminal window or through a serial port.

Native Method Debugging

Debugging native methods involves using a native debugger to track the execution of native code that is being called from Java classes. The integration of a native debugger into a PersonalJava development environment can be complicated by shared library and symbol table issues. The sections that follow give some hints for using native debugging tools with the PJEE.

Native Debugger Notes

In principle, a source level debugger can be used with the PJEE to debug native methods at runtime. The following guidelines describe an approach to this task that is not based on a specific native debugger.

- Use the debug versions of the invocation tools, e.g. `pjava_g`.
- (*Solaris only.*) Be sure to use the binary executable files of the invocation tools in `bin/sparc` and not one of the front-end scripts.

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 version of the PJEE was built with the GNU C Compiler and the Microsoft Windows 95/NT version was built with Microsoft Visual C++, version 5.0.

Using the PJES Build Environment

The **PersonalJava Environment Software (PJES)** includes a build environment for building binary executable versions of the PJAE. The PJEE is an example of a binary executable built with the PJES build environment.

The PJES build environment has a mechanism for including native libraries in the list of object files that are linked with the PJAE binary executable. This can be used to include native methods for applications that will be bundled with an implementation of the PJAE. See the section **Adding Object Files** in the **PersonalJava Porting Guide** for a description of how to include object files in the PJES build environment.

A `gdb`-based Example

The PJEE includes versions of the invocation tools that include symbol tables for debugging with native debuggers like `gdb` The following procedure outlines the steps involved with using a `gdb` to debug a simple Java application with a native method.

1. **Define the `CLASSPATH` and `LD_LIBRARY_PATH` environment variables.**

```
% setenv CLASSPATH PJEE-dir/pjee/lib/classes_g.zip:.
% setenv LD_LIBRARY_PATH PJEE-dir/pjee/lib/platform:.
```

2. **Launch the `gdb` debugger.**

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

   Use the binary executable versions of the invocation tools instead of the front-end shell scripts. Using one of the front-end shell scripts in *`PJEE-dir`*`/bin` will not work because `gdb` loads an invocation tool indicated by a file name from a command line argument. Therefore the `CLASSPATH` and `LD_LIBRARY_PATH` environment variables must be defined explicitly, as in the previous step.

3. **Set a break point.**

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

4. **Launch the Java application inside the `gdb` debugger.**

   ```
   (gdb) run HelloWorld
   ```

   The Java application will then execute until it reaches a break point.

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

Profiling Java 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 `pjavag` 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`.

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 application on the PJEE. It does this by starting the PersonalJava 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**. If a Java application has native methods that are contained in shared libraries, then the debug version of the PersonalJava application launcher The naming convention for identifying debug shared libraries is to append `_g` to library file name. if the library was `libhello.`*`ext`*, the debug version would be `libhello_g.`*`ext`*.

(*Microsoft Windows 95/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 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 Java 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 class files. Directories are separated by colons. Thus the general format for *path* is:

```
.:<your_path>
```

For example:

```
.:/home/xyz/classes:/usr/local/java/classes
```

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

-mx*num*

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.

-ms*num*

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.

-noasyncgc

Turns off asynchronous garbage collection. When activated no garbage collection takes place unless it is explicitly called or the program runs out of memory. Normally garbage collection runs as an asynchronous thread in parallel with other threads.

-noclassgc

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

-ss*num*

Each Java thread has two stacks: one for Java 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").

-oss*num*

Each Java thread has two stacks: one for Java code and one for C code. The `-oss` option sets the maximum stack size that can be used by Java code in a thread to *num*. Every thread that is spawned during the execution of the program passed to **pjava** has *num* as its Java 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.

-D*propertyName=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 classes. Directory names are separated by a platform-specific path separator character. Here is a Solaris example,

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

And here is a Microsoft Windows 95/NT example,

```
C:\home\xyz\classes;C:\java\classes
```

LD_LIBRARY_PATH
: (*Solaris 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 a Solaris example,

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

Microsoft Windows NT/95 uses the PATH environment variable as a search path for both binary executables and dynamic link libraries (DLLs) that contain native method implemenations.

See Also

- `appletviewer`
- **Using the PersonalJava emulation environment**

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 Java 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**