# *100% Pure Java™ Cookbook*

*Guidelines for achieving the 100% Pure Java Standard*

Revision 4.0


THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, California 94303 USA

**Revision History**

| | |
|---|---|
| 4/97 - 1.0 | First release |
| 5/97 - 1.1 | Minor updates |
| 4/98 - 2.0 | Added certification instructions for beans |
| | Introduced new feature based dynamic testing requirements |
| | Dropped code coverage requirements |
| | Added instructions for recertification |
| 4/99 - 2.1 | Added certification instructions for servlets |
| | Added guidelines for certification under 1.2 runtime environments |
| | Updated recertification requirements |
| | Added guidelines for certifying hybrid programs |
| | Added FAQ |
| 10/99 - 3.0 | Reorganized and updated |
| 3/00 - 3.1 | Reorganized and updated |
| 10/00 - 4.0 | Formerly "100% Pure Java Certification Guide", rewritten to remove the certification process and branding program. |

# *Table of Contents*

## Guidelines for Developing Pure Programs



## Explanations of Purity Problems and Variances

## Available Java Platforms



## Frequently Asked Questions for Purity

# *Introduction 1*

## *The 100% Pure Java™ Standard*

The "100% Pure Java™ Standard" is part of Sun Microsystems initiative to promote the development of portable applications, applets, beans, class libraries, and servlets written using the Java™ Programming language. Compliance to the standard consists of code analysis and testing the program on multiple Java Application Environment.

This cookbook will assist you in the process of understanding the guidelines for writing portable code. The steps described in this guide are designed to ensure that your code is portable and that it will meet your customer's expectations of a 100% Pure Java program.

## *About this Guide*

This guide covers all of the basic things that you should know about the 100% Pure Java Standard.

This document serves as:

- **Developers Style Guide**: for developers using the Java programming language who want to maximize the portability of their programs. It explains the important difference between merely writing programs using the Java programming language, and writing effective and portable programs that will indeed run on Java Compatible platform or device.
- **Programming Manual for Purity**: for programmers. It presents the rules for compliance with 100% Pure Java standard. It also provides practical hints and

advice for maximizing portability of programs, and offers workarounds for the most common portability pitfalls encountered when writing software.

This document can help you:

- **Gain purity expertise**: developers can benefit from the collective experience of Sun's development team. This guide contains information and ideas to help developers achieve the maximum portability with their programs. This guide describes the principles for purity, along with hints for portability.

## *Contents*

This guide has two purposes: it sets out the rules for achieving the 100% Pure Java standard, and it presents advice and examples to aid the development of fully portable software. It defines the steps you must take to certify a program. It also defines what criteria applies to 100% Pure Java programs and, equally important, what does not.

### Chapter 1, "Introduction"

gives basic definitions that describe the vocabulary of purity. This chapter is important for everyone.

### Chapter 2, "Understanding Portability & Purity"

describes the virtues of purity. The 100% Pure Java standard is not designed for all programs; this chapter will help determine if following the purity standard is appropriate for you.

### Chapter 3,"Guidelines for Developing Pure Programs"

describes the rules and guidelines that a programmer must know in order to develop a 100% Pure Java program.

### Chapter 4, "Explanations of Purity Problems and Variances"

describes the pitfalls that programmers frequently encounter while trying to modify their programs for purity. This chapter will give Sun's perscribed solutions to these pitfalls. Some pitfalls will require "bending" purity rules to some degree.

The Appendices also give helpful information describing:

### Appendix A, "Available Java Platforms"

describes the supported architectures, OS types, JAE versions that a pure java program should run on.

**Appendix B, "Frequently Asked Questions for Purity"**
describes common questions that people have relating to the rules of purity, how to interpret results from JavaPureCheck™ software, etc..

## *100% Pure Java Definitions*

**Java Compatible :** A brand that is associated with the JAE. This brand describes that a given version of the Java language compiler, Virtual Machine (VM), and Java Class Libraries (Application Program Interfaces, or API's) have passed Sun's conformance tests.

**100% Pure Java :** A term used to describe programs that follow Sun's purity standard. This term describes that a program that runs on top of a Java Compatible platform has passed the requirements set forth in this document..

**Java Application Environment (or 'JAE'):** A specific version of the Java language compiler, Virtual Machine (VM), and Java Class Libraries (Application Program Interfaces, or API's) define the JAE. The Java 2 Platform, Standard Edition is an example of a JAE.

**Java Compatible Platform (or 'Platform'):** A specific combination of computer hardware (or architecture), operating system, and version of a supported JAE.

**Program:** In the context of this document, a program is a self-contained set of classes. A program can be a application, an applet, a class library, a servlet, a bean, or more than one of these.

## *Online Information*

For online information about Sun Microsystems and some of their programs, go to these sites:

**Sun Microsystems Main Web Site**
`http://www.sun.com`

**Java Developer News & Information**
`http://industry.java.sun.com/`

# *What is the 100% Pure Java Standard?*

The 100% Pure Java standard is the set of guidelines that a programmer can follow to assure a reasonable sense of portability. Customers can be assured that the program relies only on the documented and specified Java platform, so that it will run on any computer hosting a Java Compatible application environment. The end result is that your program delivers on the promise of "Write Once, Run Anywhere™."

Programmers can inspect the program for compliance to the standards defined in this guide, using the procedures and test tools provided by Sun.

The steps described in this guide are necessary to ensure that your program is portable and that it will meet your customer's expectations of a product meeting the 100% Pure Java standard.

## *Purpose*

The purpose of the 100% Pure Java standard is to provide reasonable and economic assurance that a program will run on any Java Compatible platform. Sun addresses these goals by defining the domain where purity is analyzed:

- **API Constraints** the parts of the Java Class Library that are defined as usable in a pure manner.
- **Common Behavior** The behavior that your program exhibits on different Java Compatible platforms.

### API CONSTRAINTS

API Constraints describe the rules that Sun has chosen regarding how you call methods defined in the Java Class Library. Only Java Compatible platform API's can be referenced, and must be used in a portable manner.

For more information about API constraints , see "Rules of Purity" in Chapter 2, and Chapter 3, "Guidelines for Developing Pure Programs".

### COMMON BEHAVIOR

An important part of the 100% Pure Java standard is to observe that a program behaves with functional commonality independent of the platform upon which it is run. This means (in principle) that each product feature is demonstrable on any Java Compatible platform. While platforms may exhibit cosmetic user interface and "look and feel" differences or have limitations in underlying hardware, operating system, or browser functionality, all product features

should still function in every supported Java environment. “Common behavior” does not mean identical behavior, but functional behavior conformant with the underlying platform.

For more information about portability, see Chapter 2, "Understanding Portability & Purity".

# *Should I Comply With The Purity Standard?*

Before deciding whether to modify your program for purity, read the rest of this document, and the JavaPureCheck users guide.

## *The “Suggested Compliance Evaluation Criteria?” Checklist*

Depending on the design details of your program it might be more or less advisable to try to comply with the standard. Review the Table 1 - 1 to see how your program stacks up to the recommended criteria.

Evaluate each Program Characteristic question and the evaluation statement that follows it, then check ‘Yes’, for Advisable, ‘No’ for Not Advisable, or ‘Maybe’ if you are not sure.

**Table 1-1: Suggested Compliance Evaluation Criteria**

| Program Characteristic | Yes | No | Maybe |
|---|---|---|---|
| Is platform portability an important differentiator for my program?<br>A single code base can significantly reduce maintenance costs if you plan to support your program on several hardware platforms. If your product is an applet, servlet, or development library/bean, then you might want to market your code as usable in many/all Java environments. | | | |
| Can my program comply with the purity standard?<br>Use the JavaPureCheck tool to see if there are existing portability problems. It's free, and takes only a few minutes to run. If problems exist, determine whether they can be fixed or worked around. See Chapter 3**,** "Guidelines for Developing Pure Programs" for more information. | | | |
| What if my program can't comply with the purity standard?<br>Impure programs are not necessarily *bad* programs. In some cases, it isn't possible to achieve certain functionality without making direct calls to the operating system or using native code (e.g. using platform-specific devices). Identifying portability problems in your program will, however, make it much easier to port to other hardware platforms. The JavaPureCheck tool provides an excellent means to find potential portability issues in your program. | | | |
| Is complying with the standard worth my effort?<br>A competent engineer should need 3-4 full days (without distractions) to prepare a typical program for for purity compliance. Your time may vary, depending on the nature of your program.<br>Another valuable benefit you might realize is that the process of testing and checking your program with the JavaPureCheck tool might expose bugs or problems in your code. Naturally, it is always best to find these problems — and fix them — before your customers find them. | | | |

# *Understanding Portability & Purity* 2

## *Standards for Portability*

The ideal of the 100% Pure Java standard is to assure that the end user has a seamless and painless experience running a given Java program on any conforming Java platform. Identical behavior is not the standard; a portable program might behave differently on different platforms, but still exhibit the same feature set.

For some programs, portability is not a property of the program, but of the program's input. As an example, consider a program that parses and interprets Java programming language statements. This program will necessarily behave unportably when given an unportable program to interpret. As long as the portability problem is not intrinsic to the program, the program cannot be considered compliant with the 100% Pure Java standard. The software's developer must be the final authority on the program's correct behavior.

Similarly, the appearance of the program (if it has a GUI) is the responsibility of the developer. The standard for purity is that the program display essentially the same elements on all platforms, not that those elements have the same appearance.

## *The Purpose of the Purity Standard*

The purpose of the purity standard is to provide reasonable and economic assurance that the 100% Pure Java program design guidelines have been followed, and that the program will run on any Java Compatible platform. The process is not intended to be immune to subversion, but is intended to be a chance for you to provide evidence of the portability of your product in a uniform and consistent way.

## *Which Java™ API?*

Sun Microsystems recommends writing to the 1.1 or later versions of the Java API.

The Java technology web site has helpful information about making the transition between v1.0 and v1.1.

For information, go to:
`http://java.sun.com/products/jdk/1.1/compatible`

Unless your circumstances prohibit you from moving away from version 1.0, it is a good idea to do so. You will gain portability benefits along with extra features. In particular, the conformance requirements for 1.1 and later Java platform versions are much more stringent than those that were in place for the 1.0 version; you can thus expect a greater uniformity between platforms implementing 1.1 and later versions than among 1.0 platforms.

By writing to the 1.0 interfaces, you achieve backward compatibility with old installations. By writing to the 1.1 or later interfaces, you take advantage of the recent improvements and prepare for the future. The choice depends on your circumstances. Programs written to any version are eligible for certification.

The enhancements made to the Java 2 Platform, Standard Edition, v 1.2 API are quite compelling for many developers. Choosing this version of the Java platform (or later) is recommended for all developers who need the state of the art API support found in this version.

# *Understanding 'Purity'*

The Java platform promises streamlined software development and delivery. It saves developers time and it saves their companies the expense of multiple ports and traditional distribution methods. Unlike programming systems that tie developers to a single hardware platform, the Java platform bridges many varieties of hardware and system software with a common language, opening up new markets rather than limiting market opportunities.

This portability brings new freedom to developers, who can now deliver solutions for a wide variety of hardware, without the expense and delay of porting and qualification. It also brings new freedom to users, who can choose to change hardware platforms to best meet their needs, while preserving their investment ins software.

Yet the inherent portability of the Java platform alone does not ensure seamless operation for every program. A number of pitfalls can adversely affect the portability of programs, and developers need to know what they are and how to work around them. That is why Sun has developed guidelines for writing 100% Pure Java code.

A pure program is defined as one that relies only on the documented and specified Java platform. By focusing on this design criterion now, developers can reduce their exposure to portability risks in the future.

## *Purity is Not Goodness*

Not all good programs are pure; not all pure programs are good.

It is entirely possible to use the Java programming language to write a program that depends on platform-specific capabilities, defines native methods, and violates all the rules for purity. Just because a program is not pure does not mean it is poor quality or bad in any way; the program may in fact meet the needs of its users in ways not possible to achieve in a pure manner. An program can be good without being pure.

Likewise, the measurements made to ascertain the purity of a program are intentionally blind to the user's requirements. A program can be completely pure and still be quite unsuited to the user's requirements — even be quite buggy. The purity process attempts to find out if the program will behave the same on all Java Compatible platforms, not if it will behave well on all platforms. An program can be pure without being good.

Purity is intended to be a measure of only one of the many characteristics required of a program. Although it is not a perfect measure of portability, it is nonetheless a useful measure. Experience has shown that performing purity assessment will detect some common portability problems. In that way, checking the 'purity' of programs does result in better portability of programs.

## *Portability vs. Purity*

There is a critical distinction—and connection—between portability and purity.

Most people think of a portable program as one that produces the same results on any platform. This is actually a very imprecise definition.

For example, consider this program:

```
class ShowOS {
     public static void main (String[] args) {
          try {
            String osName = System.getProperty("os.name");
            System.out.println(osName);
          } catch (RuntimeException re) {
            System.err.println(“Problem: " + re);
          }
      }
}
```

**Figure 2 - 1: Example Code — Pure Example (class ShowOS)**

Most people would say that this is a portable program, even though it produces different results on different platforms. Compare it to this program:

```
class BadOS
     public static void main(String[] args) {
         try {
            String osName = System.getProperty("os.name");
            if (osName.equals("Solaris")) {
                 throw new RuntimeException();
             }
            System.out.println(osName);
         } catch (RuntimeException re) {
             System.err.println("Problem: "+ re);
          }
      }
}
```

**Figure 2 - 2: Example Code —Impure Example (class BadOS)**

The difference between these two programs is not in the use of the Java platform; formally, they are very nearly identical. The difference is in the functionality they implement. The second example contains an OS platform dependency that is implemented into the program in a portable way.

# *Rules of Purity*

A 100% Pure Java program is one that depends only on the Java platform, as defined in this documentation. It is relatively easy to determine if a program conforms to the design criteria detailed in this manual. If it does, it is defined as ‘pure.’

Purity is measured at the bottom (platform) edge of the program, rather than at the top (user) edge. Experience has shown us that if a program has been designed according to these standards, in other words, is 'pure,' it is a good predictor of portability; a pure program should not be accidentally unportable.

The platform edge is defined by the calls to the Java API (Application Program Interfaces) that the program makes. In this discussion, we will often refer to the "Java core API". The core API is the set of documented classes (typically under the `java` and `javax` packages) that Sun designates as public classes.

This document sets out the rules and principles for purity, along with hints and tips for the more general goal of portability.

## *The Rules*

A pure program should:

**1. Use no native methods.**

**2. Depend only on the Java core Application Programmer Interfaces .**

**3. Rely on no hardwired platform-specific constants.**

**4. Follow any applicable API protocols.**

### RULE 1: USE NO NATIVE METHODS.

The first and foremost rule of purity is to avoid the usage of native methods. Attempting to introduce native code into a program results in the sacrifice of most of the benefits of the Java programming language: security, platform independence, garbage collection, and easy class loading over the network.

For users, the security issues of software that includes native methods are substantial. There is no assurance that the code is virus-free; moreover, if a native method has a pointer overrun or attempts to access protected memory, it can crash the Java virtual machine, possibly corrupting and certainly interrupting the user s work.

The only exception to this rule is in alternative implementations of standard interfaces. Several of the Java core APIs function as standardized interfaces to specific external functionality; examples are the JDBC™ database access interface and the JSA cryptography interface. These are circumscribed interfaces, well-defined and with precise specifications, designed to facilitate free substitution of the user s chosen implementation. Due to this substitutability, a program can include native code implementations of these

interfaces, as long as it also includes at least one pure implementation. The other implementations may be made available to users as an installation or runtime option.

## RULE 2: DEPEND ONLY ON THE JAVA CORE APPLICATION PROGRAMMER INTERFACES .

The Java core APIs form a standard foundation for components, applets and applications; it is the essential framework for program development. 100% Pure Java programs must depend only on classes and interfaces documented in the Java core API specification.

This means, in detail, that a 100% Pure Java program must be a complete program, without dependencies on external libraries or interfaces that are not part of the Java core API specification.

The Java core APIs provide the basic language, utility, I/O, network, GUI, and applet services; vendors who have licensed this Java technology from Sun have contracted to include them in any Java platform they deploy.

The specifications for all Java APIs are freely accessible on the World Wide Web at http://java.sun.com.

### INCOMPLETE PROGRAMS ARE IMPURE

To be certified as a program meeting the 100% Pure Java certification standard, must be self-contained; it must include all required classes aside from the core API. An incomplete program is unportable, because it will not run on a Java platform implementation that lacks the special classes required by the program.

### DON'T DEPEND ON THE INTERNALS OF ANY PARTICULAR IMPLEMENTATION

Any implementation of the Java core APIs will include classes and packages that are not part of the documented API interface. Portable programs must not depend on these undocumented classes, because they might vary among different Java platform implementations. This is true even if the classes in question are undocumented parts of the reference Java platform implementation from Sun Microsystems. Those interfaces are not part of the Java platform definition, and they are not checked by the Java tests for compatibility, so they might be absent or might behave in subtly and dangerously different ways on different Java platform implementations. They are not documented because they are not intended for client use.

One subtle way that a program might depend on a class is by defining classes into the packages that are part of the core APIs or a specific implementation. Defining a class within an existing core package is called “injecting a class into a core package”. This breaks protection boundaries that the core implementors are entitled to count on.

Another subtle dependency on implementation details is direct use of the AWT component peer interfaces defined in classes in the `java.awt.peer` package. These interfaces are documented as being for use by AWT implementors; a portable program uses the AWT rather than implementing it.

### USING RUNTIME.EXEC

Using `Runtime.exec` is generally not acceptable as pure. Certain features of the Java programming language definition give the programmer access to hardware-specific code; native method definition (see Rule 3, “No Hardwired Platform-Specific Constants”), and some methods in the class `java.lang.Runtime`. This hardware access is very useful for writing programs that interface to legacy systems, but such interface programs are by definition not compliant to the 100% Pure Java standard.

The use of the `Runtime.exec` method is only allowed if it’s at the request of the user of the program. For example, a command interpreter that executed programs named in the user’s input could be pure. Another example is the invocation of an external program with a specific function, such as a Web browser, as long as the user has control of which browser is invoked.

The exact criteria for use of the `Runtime.exec` method are:

- The invocation must be a direct result of a specific user action; the user must know they’re executing a separate program.
- The user must be able to choose, by configuration or as part of the invocation action, which program is executed.
- A failure of the `Runtime.exec` method, especially one caused by the absence of the requested program, must be handled cleanly.
- Use of the `Runtime.exec` method is platform neutral. No platform specific functionality documented by your product can be made exclusively available to users of any particular hardware/OS platform.

Examples of the correct use of `Runtime.exec` are:

- Invocation of a Java compiler, with the name of the compiler specified as a user-settable Property.
- Execution of a command the user typed in (a “shell”).
- Invocation of a browser, configured as part of the installation of the program, when the user presses a “Help” button.

see Chapter 4, "Explanations of Purity Problems and Variances" for additional information.

### DUMMY CLASSES

The use of “stubs” or “dummy classes” to hide references to external classes is prohibited in pure programs. For example, it is considered cheating if you include a class in your program that can be substituted (e.g., via classpath manipulation) with another more functional — and possibly impure — class.

## RULE 3: RELY ON NO HARDWIRED PLATFORM-SPECIFIC CONSTANTS.

This rule prevents programs from using platform-specific constants (such as os.name, os.arch, or os.version) to determine function that your program provides. It is not pure to for a program to provide one set of services for one platform, and a different set of services on another platform.

This does not mean that all uses of platform-specific constants are prohibited. For example, testing for and working around platform bugs is encouraged, as long as the solution to the bugs does not provide extra features for a given platform.

The `java.io.File` class can be used in an unportable way, by constructing Files using a platform-specific path constant. Similarly, input and output streams can be used unportably, with hard-coded and hardware-specific line termination characters. Fortunately, it is easy to avoid these sources of unportability, because the Java core APIs provide portable alternatives.

## RULE 4: FOLLOW ANY APPLICABLE API PROTOCOLS

There are certain methods in the core API that must be called or implemented in a certain pattern. By failing to follow these patterns, it is possible to write a program that is syntactically correct, but which will be highly non-portable.

For example:

- An AWT implementation is allowed to invalidate a Graphics object after a call to a Component’s update method returns. A Component that retains a reference to the Graphics object may happen to work on one implementation, but is not portable.
- The JavaBeans™ component protocol and patterns, which must be followed in order to make a portable bean, are described in the JavaBeans documentation.
- A class that overrides the java.lang.Object equals method must also override the `java.lang.Object.hashCode` method, to preserve the invariant that equal Objects have the same hashCode.
- The 1.1 JDK introduced a new AWT event model; a program should not mix the

1.0 and 1.1 (or later) AWT event models.

*Note:* **This rule is not completely detectable, because not all protocol-specified behavior is predictable. This means that the compliance testing will not catch all violations of this rule. Certain applications of this rule, however, do lead to practical measures, and those will be incorporated into the compliance detection.**

# *Guidelines for Developing Pure Programs* 3

## *How to Develop a Program*

This section identifies common problems encountered in program programming for portability, and offers solutions or workarounds.

This is information about portability, not specifically about purity; some of these portability problems are detected in the purity checking process and some are not. Similarly, some of these pitfalls are specific to given methods of classes, while other pitfalls are stated with principles instead of remedies.

### *1. Pitfall: Thread Scheduling*

Explanation: Thread scheduling may differ on different platforms. If you rely on priorities or luck to prevent two threads from accessing the same object at the same time, your program is not portable.

For example, this program is not portable:

```
class Counter implements Runnable {
   static long val = 0;

   public void run() {
      val += 1;
   }

   public static void main(String[] args){
      try {
         Thread t1 = new Thread(new Counter());
         t1.setPriority(1);
         Thread t2 = new Thread(new Counter());
         t2.setPriority(2);

         t1.start();
         t2.start();

         t1.join();
         t2.join();

         System.out.println(val);

      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

**Figure 3 - 1: Example of an Unportable Program**

This program might not print “2” on all platforms, even barring errors, because the two threads are not synchronized.

Unfortunately, this is a deep problem, and there is no quick check for its presence nor easy fix to prevent it occurring.

**Workaround:** One simple, if drastic, answer is to make all methods synchronized. This may make some synchronization errors show up as obvious deadlocks rather than as silent data corruption. Unfortunately, there are examples of thread contention that will not be detected. For example, the JavaPureCheck Tool will not detect the problem in the example `Counter` class, because the contention in the example is in field access rather than method access.

**Solution:** Adopt a discipline of multithreaded application programming, as described in several textbooks. One that is specifically written for developers using the Java programming language is *Concurrent Application Programming in Java™, Second Edition,* by Doug Lea, Addison-Wesley 1999, ISBN 0-201-69581-2.

## *2. Pitfall: Use of Native Code*

**Explanation:** Calling native functions is inherently not portable. In rare instances however, it may seem necessary to include native methods to deliver access to a system resource that is not supported by the Java technology APIs or to boost performance of a particular program.

**Solutions:**

- Define a simple protocol to give that service, then write your 100% Pure Java program as a client of that protocol.
- Rewrite the native method using the Java programming language.

It may seem that you could confine the native methods to one class and provide an implementation of that class for every Java platform. However, this "solution" in fact only complicates matters, because the number of Java platforms is not fixed, but is ever-increasing; and some Java platforms, such as the JavaStation™ workstation, have no ability to execute native program code.

There is no 100% Pure Java program workaround for native code. Even though it is possible that a program with native methods can be made portable (by writing a class or method with the Java programming language to serve as a fallback for the native code), such a program cannot be certified as pure because we know no principled way to show that the native-code and Java language versions of such a program actually implement the same functionally.

If the two versions do, in fact, implement the same functionality, the program without the native code is an equally capable 100% Pure Java program. That program is the one you can measure purity against, and classify as a 100% Pure Java program.

*Note:* **The Java Native Method Interface is not a way to make native code platform-independent; it is a way to make it easy to port native code. The native code still must be recompiled for each different hardware, and that recompilation will be difficult or impossible if the target hardware does not provide the library or the capabilities required by the native method.**

## *3. Pitfall: Use of* `exec`

**Explanation:** The `java.lang.Runtime.exec` method is not generally portable; not all platforms have programs that can be run, and not all platforms have the notion of "standard input" or "standard output." A hard-coded program name will not be portable; there is no program that has the same name on all Java platforms.

**Solution**: Don't use this method except under the restrictions detailed in "Using Runtime.exec" in Chapter 2.

## *4. Pitfall: Failure to Use the Portability Features of Java Core API*

**Explanation:** It is unportable to hard-code text display sizes, colors, layout management details, etc.

**Solution:** The system properties in general are very useful for portability; use them whenever applicable. The AWT abstracts the details of coping with the platform's window system. Use that abstraction to the fullest. For example:

- Use a `LayoutManager` rather than hard-coding component sizes or positions
- Use the various `getSize` methods
- Use the desktop colors available in `java.awt.SystemColor`.

## *5. Pitfall: Reflection*

**Explanation:** The reflection facilities of JDK 1.1 and later versions of the Java platform are an immensely powerful addition to the language; they also make it harder to predict what a program will do. `Method.invoke` can be used to invoke any method, including ones that present portability problems.

**Solution:** Be careful. Any use of `Method.invoke` to start another program must follow the same requirements as `Runtime.exec` detailed in "Using Runtime.exe" in Chapter 2.

## *6. Pitfall: Direct Use of AWT Peer Classes*

**Explanation:** Portable programs, that use rather than implement the AWT interfaces, should not use the AWT peer classes directly. The protocol for interaction with the peer classes is platform specific.

**Solution:** Stay on the client side of the AWT.

## *7. Pitfall: Misuse of System.exit*

**Explanation:** The `System.exit` method forces termination of all threads in the Java virtual machine. This is drastic. It might, for example, destroy all windows created by the interpreter without giving the user a chance to record or even read their contents.

**Solution:** Programs should usually terminate by stopping all non-daemon threads; in the simplest case of a command-line program, this is as easy as returning from the main method. `System.exit` should be reserved for a catastrophic error exit, or for cases when a program is intended for use as a utility in a command script that may depend on the program's exit code.

## *8. Pitfall: Use of Hard-Coded File Paths*

**Explanation:** Hard-coded filenames may present portability problems. Hard-coded paths (with directory names joined to filenames) certainly do.

**Solution:** The most portable way to construct a `File` for a file in a directory is to use the `File(File,String)` constructor to build up the path. Other portable solutions are to use the system properties to get the local file separator and starting directory, or to use a file dialog to ask the user for a filename.

*Note:* **The concept of an "absolute path" is somewhat system dependent; for example, UNIX® absolute paths all start with "/", while Windows absolute paths may start with any letter. For this reason, the use of an absolute path that is not derived from user input or from a system property is unportable.**

Here is the code for a utility class that might make it more convenient to construct portable pathnames:

```
package util;

import java.io.File;
import java.util.StringTokenizer;

/** A utility class to make it easier to use
 *  java.io.File in a portable way.
*/

public class FileUtil {
 /** Create a new pathname by gluing together
   * a series of names.
   * If initial base is null, works from
   * current directory.
   */
   public static File fromDir(File bse,
```

```
                            String[] path) {
   File val = bse;
   int i = 0;

   if (val == null && path.length > 0) {
      val = new File(path[i++]);
   }

   for ( ; i < path.length; i++) {
      val = new File(val, path[i]);
   }

   return val;
}

public static File fromHere(String[] path) {
   return fromDir(null, path);
}

private static File fromProp(String propName) {
   String pd = System.getProperty(propName);
   return new File(pd);
}

/** A File for system property "user.dir".
 */
public static File userDir() {
   return fromProp("user.dir");
}

/** A File for system property "java.home".
 */
public static File javaHome() {
   return fromProp("java.home");
}

/** A File for system property "user.home".
 */
public static File userHome() {
   return fromProp("user.home");
}

/** Split first argument, using second arg
 *  as separator char.
 *  Convenient for creating a portable pathname.
*/
public static String[] split(String p,
                             String sep) {
   StringTokenizer st =
         new StringTokenizer(p, sep);
   String[] val = new String[st.countTokens()];
```

```
            for (int i = 0; i < val.length; i++) {
               val[i] = st.nextToken();
            }

            return val;
         }
      }
```

## 9. *Pitfall: JDBC™ Driver Loading*

**Explanation:** The JDBC interface, defined by the `java.sql` package, provides for flexibility in loading the actual JDBC driver code. This flexibility allows for substitution of different JDBC drivers without changes to the programs code.

This flexibility is provided by the `DriverManager` class, which selects among the available JDBC drivers during connection establishment. Drivers can be made available to the DriverManager in two ways:

- They can be named in the `jdbc.drivers` system property
- They can be explicitly loaded by use of the `java.lang.Class.forName` method.

This becomes a portability issue because JDBC drivers, particularly those that include native code, might be less portable than the programs that uses them. If you code a specific JDBC driver name into your program, your program is only as portable as that driver. To be pure, your program must be certified with a portable driver.

**Solution:** In order to maximize the portability of your JDBC program, we recommend that you make the exact JDBC driver name configurable. You can do this either by relying on the `jdbc.drivers` system property, or by using a property file (or some other configuration mechanism) to provide the name of the JDBC driver class that will be loaded with the `Class.forName` method.

It is also possible to load a selection of drivers, relying on the `DriverManager` selection mechanism to find an appropriate driver when the database connection is made.

If you chose to use this approach, consider the following guidelines:

- Drivers are tried in the order they were registered, so earlier drivers will have priority over later ones, with drivers listed in `jdbc.drivers` having highest priority.
- A driver that includes native code will fail to load on any platform other than the one for which it was written; therefore, your program must cope gracefully with the ensuing `ClassNotFoundException`,
- A driver with native code must not register itself with the `DriverManager` until it knows that it has successfully loaded.
- A driver with native code is not governed by the security sandbox, and presents a potential security hazard.

In addition, relying on `Class.forName` to run the static section of the driver (which typically creates a driver instance and registers it) causes portability problems. The workaround is to explicitly create and register an instance of the driver class, like this:

```
DriverManager.register(
            Class.forName("MyDriver").newInstance());
```

Since the static section may run, and may register the class, this can result in duplicate instances and duplicate registrations; however, according to the JDBC team this should not create a problem.

## *10. Pitfall: Line Termination*

**Explanation:** Different platforms have different conventions for line termination in a text file. This is a common instance of the general problem of representing text.

Different machines have different internal representations of text. The Java platform uses Unicode internally, which is an international standards-based solution to the problem; but we still need to get text to and from files.

The Java 1.1 (and later) API has the `java.io.Reader` and `java.io.Writer` classes to handle that kind of character-set conversion. However, the problem can arise even when reading and writing plain ASCII files, because the ASCII standard isn't specific about the line termination character. Some machines use "`\n`", some use "`\r`", some use the sequence "`\r\n`". The portable way to deal with this, so you'll write files that the user can read, is to use the "`println`" methods to write a line of text, or to put the line marker at the end of a line. You may also use the "`line.separator`" system property (using the System properties may be problematic for applets that can not access the System for security reasons).

**Solution:** For input, rather than writing the code yourself to look for the various line-termination sequences, it's best and easiest to use the `readLine` method from the `java.io.BufferedReader` class to fetch a complete line of

text. The other `readLine` methods are also useful, but the one in `BufferedReader` gives you code set translation as well. Likewise, for output, use `writeLine or println` function to output lines of text.

## *11. Pitfall: Unportable Command-Line Programs*

**Explanation:** Command-line programs that use `System.in, System.out`, or `System.err` might be less than perfectly portable, because not all Java platforms have the concept of standard input or output streams.

**Solution:** Consider using a GUI, at least as an alternative — some platforms don't have a command line.

## *12. Pitfall: Command Line Processing*

**Explanation:** The Java platform leaves command line processing up to the programmer. However, the syntax and conventions are quite different on the different platforms.

**Solution:** The most portable answer is not to use the command line, but that's no good for batch programs that need to be driven from a script. Use the widely understood POSIX convention (options indicated with a dash) when processing command line options. Provide, at least as an alternative, a GUI, or read options from properties files (and document the properties).

## *13. Pitfall: Internationalization*

**Explanation:** Computing is a worldwide phenomenon. Particularly if your program is an applet, it is highly likely to be run in an environment with a different native language than yours.

**Advice:** Use the internationalization and localization features of JDK™ 1.1 software and later Java platform versions. Complete documentation is included in the JDK software, and is also available at:

`http://java.sun.com/products/jdk/1.1/docs/guide/intl`

and

`http://java.sun.com/products/jdk/1.2/docs/guide/intl`

## *14. Pitfall: Unicode Rendering*

**Explanation:** Not all platforms can render all Unicode characters.

**Workaround:** For the default text of messages, buttons, labels, and menus, use only ASCII. Of course, it is all right to use non-ASCII in localization resources and in text obtained from the user.

## *15. Pitfall: File I/O*

**Explanation:** The JDK 1.0 input and output classes are not portable to hardware architectures with non-ASCII native file formats.

**Solution:** Use the reader & writer classes found in the JDK 1.1 software and later versions of the Java platform.

## *16. Pitfall: GUI Element Size*

**Explanation:** The exact size of the AWT elements will differ from platform to platform, as will the size of the screen and the default and maximum size of a window. Any hard-coded positions or sizes will run afoul of these variations.

**Solution:** Use a layout manager.

## *17. Pitfall: GUI Fonts*

**Explanation:** The size and availability of fonts varies from display to display (even on the same hardware platform, depending on installation).

**Solution:** Don't hard-code text sizes; let text elements assume their natural size in a layout, and use the FontMetrics methods to find the actual displayed size of a string on a Canvas. When setting a nondefault font, be sure to implement a fallback in the "catch" block. When creating a font menu, get the font names from the `java.awt.Toolkit.getFontList` method rather than using a hardwired font list.

When updating a program from 1.0 to 1.1 or later versions of the Java platform, be sure to update the font names as described in the documentation for the `java.awt.Toolkit.getFontList` method:

- `Times Roman` becomes `Serif`
- `Helvetica` becomes `Sans-serif`
- `Courier` becomes `Monospaced`

## *18. Pitfall: GUI Appearance*

**Explanation:** The size of the screen and the number of available colors may change from platform to platform, or even user-to-user or day-to-day. This can make a display illegible (for example, black text on a dark blue background), or can hide buttons if they display off the screen.

**Solution:** It may be necessary to adjust font size or default window size according to screen resolution, which may be obtained by:

```
Toolkit tk = java.awt.Toolkit.getDefaultToolkit();
int dpi = tk.getScreenResolution();
```

Applets are probably safest to use the default font, as it may have been customized by the user according to their personal preferences.

For an application, you may wish to give the user an option or control to select among several appearances, so they can choose one that suits their display and mood.

If writing to the JDK 1.1 software or later interface, you can make your colors harmonize with the user's desktop by using the colors from the `java.awt.SystemColor` class.

If you choose to use your own color scheme instead, beware that displays vary greatly in the color displayed for a particular RGB triple. Your program's appearance will be more portable if you use named color fields from `java.awt.Color` rather than numeric colors.

## *19. Pitfall: The Paint Protocol*

**Explanation:** The AWT `Component.paint` and `Component.update` methods take a Graphics object as a parameter. This object should not be retained, because it might be a transient object valid only during the paint; an AWT implementation is free to destroy that Graphics object after the paint method returns, which makes it pointless (and dangerous) to retain the object.

**Solution:** Do not retain the paint Graphics object. In general, it is not safe to paint outside a call to `update()` (or `paint()`, which is called by the default `Component.update()`). If you do need a long-lived Graphics object, you should create a new one from the argument value, but be warned that this might fail on some platforms:

```
Graphics myGraphics = null; // retained

void paint(Graphics g) {
   if (myGraphics == null) {
      myGraphics = g.create();
   }
   // painting code goes here
}
```

## *20. Pitfall: Mixed Event Models*

**Explanation:** The JDK 1.1 software (and later) AWT uses a different event model from the previous AWT. Programs written to the 1.0.2 event model will work on a 1.1 or later platform version, but mixing the two event models in one program will almost certainly not work.

**Solution:** Stick to one event model per program. Don't try to convert a program gradually from the old event model to the new; do it all at once.

## *21. Pitfall: Use of Deprecated Methods*

**Explanation:** Certain methods from the Java core APIs have been marked as deprecated. While these methods work in the current release, they are slated for removal at some point.

**Advice:** When you work on code (class, method, or field) that uses deprecated methods, it is probably a good idea to update the code for the replacement as suggested in the API documentation. This will also give you a head start on future work.

## *22. Pitfall: The `Object.hashCode` and `Object.equals` Methods*

**Explanation:** The `Object.hashCode` method returns a number which is effectively random. The number is also implementation- and instance-dependent. This has several consequences:

- The order of iteration for elements in a `Hashtable` will be unpredictable, and will generally differ from one program invocation to the next.
- A class that overrides `Object.equals` must also override `Object.hashCode`. It is safe, if inefficient, if the `hashCode` method returns the same value for unequal objects. It is unsafe if the `hashCode` method returns different values for two objects that are, in fact, equal.

**Solution:** If a repeatable (although unpredictable) order of enumeration over the elements in a `Hashtable` is important to your program, then any object used as a key in the `Hashtable` should have a class-specific `hashCode` method which returns a value computed from data fields of the object.

If any of your classes defines an equals method, it must also define a `hashCode` method, such that for any two objects `a,b` of that class, `a.equals(b)` implies that `a.hashCode() == b.hashCode()`. Note that hashCode should not depend on any mutable property. If an object's `hashCode` value changes, that object is very likely to become unfindable.

## *23. Pitfall: Installation Issues*

**Explanation:** There are a series of problems that can arise when installing on various platforms, due to limitations or restrictions on filenames. This might interfere with the convention that the Java virtual machine uses to locate required class files, which depends on a simple mapping from class name to filename.

The problems might occur when installing, or when attempting to run the installed program.

The problematic restrictions are:

- **File name length:** This is particularly a problem with inner classes, which are represented as class files with concatenated names.
- **Case distinctions:** Some platforms ignore case when comparing filenames. If you have two classes — or a class and a package — whose names differ only by case, your program will not be portable to those platforms.
- **Imperfect Unicode Support:** Class names may contain Unicode characters that are not usable as filenames on all platforms.
- **Special Filenames:** Some platforms assign special meaning to certain filenames, such as "`LPT`" or "`con`". These filenames cannot be used as part of a package name for classes that are to be installed as files on those systems.

**Workarounds:** Package your classes into a `.jar` archive, which is part of the JDK, starting with version 1.1. This will work around the problems of long class names and of case distinctions. It might not work around the problem of non-ASCII class names.

If you are writing for the JDK 1.0 platform, a `.zip` file is a possible alternative.

**Solution:** Either rename your packages and classes, or use a `.jar` or `.zip` file.

## *24. Pitfall: Hostname Format*

**Explanation:** The format of the string returned by the `java.net.InetAddress.getHostName` method depends on the hardware platform. In some cases, it will be a fully-qualified domain name; in others, it will only be the host part of that name.

**Workaround:** In most cases, this problem, which is a case of under-specification, will pose no practical problem, because the unqualified name and the fully-qualified name can be used for identification and connection within a domain. In cases where the name must be exported to distant hosts, it may be best to give the IP number in addition to the host name. The IP number is available from the `getAddress` method.

## *25. Pitfall: Pluggable Look and Feel*

**Explanation:** The Pluggable Look and Feel (PLAF) architecture built into the Swing classes of the Java 2 Standard Edition and the Java Foundation Classes standard extension for JDK 1.1 software allows windows, dialogs and other GUI components to take on a distinctive visual identity called a LookAndFeel. Not all PLAFs are available on every platform and some may only be supported on the operating system the PLAF emulates. Portable programs should ensure that a PLAF is both supported and available to the underlying Java platform before it is set.

*Note:* **The Java platform Look and Feel is the default LookAndFeel and is always available. It will be used if the requested PLAF is not available or unsupported.**

**Advice:** For method:

```
javax.swing.UIManager.setLookAndFeel(javax.swing.LookAnd
Feel)
```

Make sure that the specified PLAF class is supported on the active Java platform by using `LookAndFeel.isNativeLookAndFeel()` or use the current PLAF class from method `UIManager.getLookAndFeel()`.

For method:

```
javax.swing.UIManager.setLookAndFeel(java.lang.String)
```

Use `UIManager.getSystemLookAndFeelClassName()` or `UIManager.getCrossPlatformLookAndFeelClassName()` to ensure that the PLAF class is supported.

Alternatively, you can call `UIManager.getInstalledLookAndFeels()`

This returns a `LookAndFeelInfo` object. This object can be queried to determine which PLAFs are available to the currently active Java platform.

You must be careful when designing a user interface based upon a particular PLAF since your user interface might be compromised if your program is run on a platform where that PLAF is unavailable. To be safe, check your user interface with the Java platform Look and Feel unless you are certain the targeted PLAF will always be available.

Obviously, the most portable PLAF is the default Java Platform Look and Feel.

## *26. Pitfall: Mixing Classes Compiled on Different Versions of the Java Platform*

**Explanation:** Classes are compiled against a particular API set defined by the Java platform version. A program compiled on one major version (i.e., 1.0, 1.1, 1.2, etc.) of the Java platform might not run on earlier versions because the program might target APIs that do not exist in earlier versions.

Mixing classes compiled against different major versions of the Java platform is generally not recommended because, in some rare situations, bug fixes or subtle incompatibilities between the versions may cause portability problems.

*Note:* **When the JavaPureCheck tool is used to check a program using the 1.2 API system model, it will generate a warning when the program references a method from JDK 1.1 software that has been removed in the 1.2 API. The 1.2 virtual machine will properly resolve this reference if the superclass contains a method with the same signature. However, the JavaPureCheck tool does not resolve these references to the superclass, and it generates a warning to flag the use of a class compiled on an earlier Java platform version.**

Sometimes you might be forced to include a library or `.jar` of class files compiled against JDK1.1 software with a program that is compiled against 1.2. In this case, the warnings are unavoidable.

**Advice:** If possible, compile all class files against the same Java platform version on which you plan to run your program. For situations where you must make use of classes that were compiled against earlier versions of the Java platform, check the relevant compatibility page for the Java development kit:

```
http://java.sun.com/products/jdk/1.1/compatibility.html
```

```
http://java.sun.com/products/jdk/1.2/compatibility.html
```

If you have difficulty with a 3rd party class library, contact the vendor and ask to have a library compiled for your target Java platform version.

*Note:* **Compiling your classes with the `-target 1.1` compiler flag on version 1.2 only affects the class file format; it does not change the version of the JDK software APIs that the compiler compiles against.**

# *Portability Hints*

This section presents general hints for writing portable code.

## *1. Hint: Writing Portable Applets*

Writing portable applets is somewhat harder than writing portable programs. An applet is, formally, a class that extends `java.applet.Applet`. In actuality, an applet's portability situation includes the Web page the applet loads from, the other classes the applet uses, the HTML that loads the applet, and the security manager and AppletContext of the user's browser.

One subtle, possible problem in the HTML `<applet>` tag is that the contents of the applet markup must follow these rules:

- `<param>` elements come before the alternate contents
- alternate contents are text elements, such as the contents of a paragraph (the `<p>` tag). This means that you cannot use `<p>` tags in the alternate contents!

Applets will almost certainly have to run under control of a security manager. However, there is no standard profile for security managers. The user can instruct their security manager to deny any combination of access. The best tactic is to make sure that your applet handles any security exception gracefully.

Similarly, the Java technology API specifications do not specify required content types or protocols. The MIME types `image/jpeg` and `image/gif` are probably safe, as are the `http:`, `file:`, and `ftp:` protocols. Again, handle any errors gracefully.

In the Java 1.0 technology API, the specification of the `AppletContext` / `Applet` protocol was not very precise. As a consequence, different browsers call the applet's enter & leave methods at different times. The protocol specification is much more precise in the Java 1.1 software and later API, so this problem will disappear in time. Meanwhile, beware that an applet that behaves politely on one browser may hog resources on another, while a different applet may function normally on one browser but stall on another.

## *2. Hint: Security Exceptions*

If your program will run under control of a `SecurityManager` — for example, if it's an applet — it might encounter a `SecurityException`. This is an unchecked exception, so it may be thrown without being declared.

Depending on the particular `SecurityManager`, the user can restrict any access to protected resources attempted by the program. Therefore, it will greatly enhance the portability of your program if you catch any `SecurityException` and report the situation to the user in a polite manner.

## *3. Hint: Coping with Bugs*

There are bugs in some Java platform implementations. To be most portable, it is courteous to work around those bugs. It is helpful to consult the known bug lists available from Sun and from some browser makers. Bugs are harder to cope with than features, because they are less well defined and less well designed.

**Workarounds:** In some cases, a program will have to include different code for different platforms. Workarounds are expensive, both to code and to test. They are, unfortunately, inescapable in some circumstances.

In general, you can activate a workaround either proactively by branching on system properties, such as `java.class.version` and `os.name`, or reactively by putting the workaround in an exception handler. The choice depends on the character of the workaround. If the structure of the program or of long-lived data structures is affected, it's probably best to be proactive. If the workaround has relatively local effect, a reactive fallback may be easiest to code and understand. When possible, the reactive fallback will probably perform better when the fallback is not taken, because the cost of establishing an exception handler is quite low.

A preemptive fallback is required if the normal code path would cause irreversible changes before the problem is detected.

Here's an example of a preemptive workaround:

```
class Preemptive {
   private boolean hasFooBug() {
      String java_name =
System.getProperty("java.maker");
      return java_name.equals("KnownBad");
   }

   void something() {
      if (hasFooBug()) {
         workaround();
      } else {
         // normal code path
```

```
        }
    }
}
```

**Figure 3 - 1: class Preemptive**

And here is an example of a reactive workaround:

```
class Reactive {
    void something() {
        try {
            // normal code path
        } catch (Exception e) {
            workaround();
        }
    }
}
```

**Figure 3 - 2: class `Reactive`**

# *Explanations of Purity Problems and Variances* 4

## *Explanations of Purity Problems*

For a detailed HTML-generated list of the purity problems, use the following command line to run the JavaPureCheck tool to generate a report:

```
java JavaPureCheck -d <model> -h
```

where `<model>` is: `jdk102`, `jdk11` or `jdk12`

In this usage, the JavaPureCheck tool creates an html description file of the system model JavaPureCheck loads for a given JDK version..

## *Report Information Categories*

The report categorizes reports into three categories:

- **Pure:** No portability problems. The class file is a pure Java class.
- **Warning:** Possible portability problem. You must provide an explanation for all warnings.
- **Error:** An unequivocal portability problem. An error must be granted a variance by Sun. The only acceptable explanation for such an error is a reference to a variance number granted by Sun.

## *Explanations*

You should provide explanations for all warnings and errors identified on the JavaPureCheck tool report. An explanation must describe why the reported problem does not, in fact, impede portability. Below is a list of typical problems reported by the JavaPureCheck tool, the reason they were flagged, and an indication of how to handle them:

*Note:* **See the documentation included with the JavaPureCheck tool and the information below for complete coverage of all warnings and errors.**

## *Warnings*

### *Warning: Possible Hardcoded Path*

The JavaPureCheck tool checks the string constants in your application, looking for any that resemble hardcoded file pathnames. The JavaPureCheck tool attempts to resolve any suspicious strings by determining how they are used. If a suspicious string (one that looks like a hardcoded path) is used in an I/O class constructor, this is reported as an error. If the JavaPureCheck tool cannot determine how a suspicious string is used, it reports a warning.

If the string identified in the JavaPureCheck tool report is not a hardcoded path, you should supply an explanation that describes how the string is used. Strings used as URLs, for example, are acceptable.

### *Warning: Mixing the 1.0 and 1.1 Event Models*

This is likely to be a application programming error that will impair portability.

Possible explanations for this warning would have to show reasonable need for mixing event models.

### *Warning: Dynamically Loaded Classes:*

- Use of `java.lang.Class.forName`
- Use of `java.lang.ClassLoader.defineClass`
- Use of `java.lang.ClassLoader.resolveClass`
- Use of `java.lang.ClassLoader.findSystemClass`

Classes that are dynamically loaded might contain portability bugs.

To explain an acceptable use of these methods, you should list all the classes that could be dynamically loaded such that you can determine that those classes are part of the systembeing checked for compliance. All classes should be checked for purity.

If your application does dynamic loading of arbitrary classes, you should provide a description of how the names of those classes are determined.

## *Warning: Object is Compiled for a Previous (Java Development Kit) release*

This warning appears when your class files are compiled on one version of the Java platform, and have dependencies on methods that no longer exist when tested on a later system model by the JavaPureCheck tool.

Even though the Java virtual machine will handle these situations by deferring execution of a missing method to its superclass, mixing classes compiled against different versions of the Java platform is discouraged because, in rare situations, bug fixes or subtle incompatibilities between the versions might cause portability problems. See “Mixing Classes Compiled on Different Versions of the Java Platform” for more information.

Explanations for these warnings should describe why all application classes are not compiled against the same Java platform version (e.g., your application might require the use of 3rd party libraries which were compiled against a previous JDK version).

## *Warning: Unsupported Pluggable Look and Feel*

The Swing classes in the Java 2 SDK and the JFC standard extension for JDK 1.1 software define a pluggable look and feel architecture that may cause portability problems when an unsupported or unavailable LookAndFeel is hard-coded into your application.

Explanations for this warning should show that the proper allowances for unavailable and/or unsupported LookAndFeel have been made in your application.

## *Errors*

### *Error: Constant Strings with OS-Specific Syntax Used in I/O Class Constructors*

Use of strings including OS-specific constants, such as hardcoded line termination characters or device names, are not portable.

### *Error: Peer Use*

The AWT peer classes are not intended for direct use by Java applications.

### *Error: Use of Native Code*

- Native method definition
- Use of `java.lang.Runtime.load`
- Use of `java.lang.Runtime.loadLibrary`

100% Pure Java programs cannot depend on native code.

### *Error: Use of `java.lang.Runtime.exec`.*

Execution of an external application is not platform independent, except under the circumstances detailed in the sections of this chapter that deal with Variance Numbers 1, 5, and 17.

### *Error: Injection into Core Library*

100% Pure Java applications may not insert classes into the Java core library (e.g. packages whose names start with "`java`", "`javax`", or "`omg`").

### *Error: Undefined Reference*

An incomplete application (i.e., an application with dependencies on missing or unavailable class files) cannot, obviously, be completely checked. It will be reported as an error, because that is the most conservative estimate of what might be in the missing class files. To remove these errors, simply include the required classes to your list of `.class` files that JavaPureCheck parses.

### *Error: Undocumented Internal `sun.*` Class*

Use of `sun.*` classes is not allowed by 100% Pure Java applications because these classes are not guaranteed to be available on all platforms.

# *Variances*

Errors generated by the JavaPureCheck tool should include a Variance number as part of the explanation in the JavaPureCheck tool report. Applications containing errors without explanations are not compliant with the purity standard.

Variances were previously granted to developers making variance requests to Sun's Variance Council.

## *Types of Variances*

Variances are classified into three types:

- **General Variances:** General variances are available to all developers, and are accepted as valid explanations for JavaPureCheck errors, and still being compliant to the purity standard.
- **Restricted Variances:** Restricted variances are available to all developers. Restricted variances typically have the potential for abuse that might compromise application portability.

## *The Variances*

The following variances have been collected into groups. All variances listed are general variances unless marked "RESTRICTED." The few specific variances that have been granted are not identified due to their limited applicability.

*Note:* **Variances are numbered according to the order they were received as variance requests from developers. Numbers that do not appear in the lists below are associated with variance requests that were either denied, dropped, or granted as Specific Variances (see above).**

### VARIANCES FOR PROGRAM INVOCATION

Invoking another program by using either `Runtime.exec()` or `Method.invoke()` is normally not portable, especially when the invoked program itself is not pure. However, in most situations, portability will not be compromised if the following criteria are followed:

- The invocation must be a direct result of a specific user action; the user must know they're executing a separate application.
- The user must be able to choose, by configuration or as part of the invocation action, which program gets invoked.

- A failure of the `Runtime.exec()` or `Method.invoke()` methods, especially one caused by the absence of the requested program, must be handled cleanly.
- Use of this invocation is platform neutral. No platform specific functionality documented by your product may be made available only to users of any particular hardware/OS platform.

See "Using Runtime.exec" in Chapter 2 for more information about these variances.

## *Variance: #1 — Invoking a Compiler*

A compiler may be invoked using `Runtime.exec()` as long as the above mentioned criteria are met. This variance also applies to the use of `Method.invoke()` when used in a manner similar to `Runtime.exec()`.

**Application of Variance:** Variance #1 may be used as an explanation for errors or warnings generated when these methods are used to start a compiler. A statement must be included in the explanation that says:

- Invocation of the compiler is triggered by a user action.
- The specific compiler invoked can be selected (or preselected at installation time) by the user.
- Any failure due to absence of the compiler is handled cleanly.
- No documented platform-specific functionality is made available.

*Note:* **This error may be reported as "Requires explanation; see Program Invocation Variance." Use Variance #1 for compiler invocation errors and warnings.**

## *Variance: #5 — Invoking a Browser*

A browser may be invoked using `Runtime.exec()` as long as the above mentioned criteria are met. This variance also applies to the use of `Method.invoke()` when used in a manner similar to `Runtime.exec()`.

**Application of Variance:** Variance #5 may be used as an explanation for errors or warnings generated when these methods are used to start a Web browser (e.g., to display help text). A statement must be included in the explanation that says:

- Invocation of the compiler is triggered by a user action.
- The specific compiler invoked can be selected (or preselected at installation time) by the user.
- Any failure due to absence of the compiler is handled cleanly.
- No documented platform-specific functionality is made available.

*Note:* **This error may be reported as "Requires explanation; see Program Invocation Variance." Use Variance #5 for browser invocation.**

### *Variance: #17 — Invoking a Configurable User Action*

A program that is specified by a user (e.g., a user-configurable menu application) can be invoked using `Runtime.exec()` as long as the above mentioned criteria are met. This variance also applies to the use of `Method.invoke()` when used in a manner similar to `Runtime.exec()`.

**Application of Variance:** Variance #17 may be used as an explanation for errors or warnings generated when these methods are used to start a program selected by a user (e.g., a program chosen from a user configurable menu). A statement must be included in the explanation that says:

- Invocation of the program is triggered by a user action.
- The specific program invoked can be selected (or preselected at installation time) by the user.
- Any failure due to absence of the program is handled cleanly.
- No documented platform specific functionality is made available.

*Note:* **This error may be reported as "Requires explanation; see Program Invocation Variance." Variance #5 is for browser invocation; use Variance #17 for application invocation errors and warnings.**

## VARIANCES FOR PROGRAMS CONTAINING NATIVE CODE

The following variances address allowed uses of native code in pure programs.

### *Variance: #45 — Use of Native Code to Enable Common Platform Functionality ***RESTRICTED****

It is often desirable to have an application "well integrated" into the host operating system. For example, a server running on Windows NT could benefit greatly by running as an NT service (i.e., so it responds to task-shutdown events). Similarly, a server running on UNIX could benefit from receiving signals like `SIGHUP` or `SIGINT`. Unfortunately, this cannot currently be done on the Java platform without using native methods. At some time in the future, Sun may provide a Java technology API for this, but in the meantime, a workaround is needed.

This variance extends the purity criteria to allow the use of native code for situations where Java technology API functionality cannot be accommodated due to a limitation (e.g., security measures) in a particular OS. Native code can be used to enable features on a specific Java platform when these features

cannot be exposed using Java APIs, and those features are exposed on other Java platforms without the use of native code. Proper application of this variance can actually improve portability.

*Note:* **It is never acceptable to add new platform-specific features in this manner or to use native code for the purpose of improving performance.**

For example, to connect to port 80 on UNIX, a process must have root privileges; but since it is generally unacceptable for an program to run with root privileges, the privileges must be reduced after the connection is made. This can be accomplished via a native call to UNIX. On Mac and other operating systems, connections to port 80 do not have this restriction. Thus, by allowing a check for various UNIX platforms, a 100% Pure Java program can call the appropriate native routines to ensure it can run on these otherwise blocked platforms.

**Application of Variance:** Variance #45 is a RESTRICTED variance.

## *Variance: #47 — Use of Class Libraries Containing Unused Native Code or Unused External References ***RESTRICTED****

If your program has dependencies on a class library that contains native or impure code — but your program never exercises this code — purity is not compromised, and you can use this variance.

The rule of thumb for these cases is that you must be able to adequately prove that a potential portability problem cannot occur.

**Application of Variance:** Variance #47 is a RESTRICTED variance.

## *Variance: #67— Use of Netscape Navigator and Internet Explorer Security Managers*

For an applet to work outside the sandbox (i.e., access security-constrained methods), it must be granted permission by the Java platform security manager. The two most widely used browsers in the industry today are Netscape Navigator and Internet Explorer. However, both of these browsers implement security manager functionality in a proprietary manner that requires calls to impure classes within the browser itself.

While these proprietary implementations are within the scope of the Java platform specifications, true platform independence might be compromised by any applet that makes use of these classes. In particular, API differences existing between platform implementations and/or versions of these products — although these are unlikely — are outside of Sun's control, so portability is not assured to the same level of certainty as with Java technology APIs.

Recognizing the need for this applet functionality and its low risk to portability, Sun has granted Variance #67 to allow applets to directly call the necessary APIs in Netscape Navigator™ and Internet Explorer to access security constrained methods.

*Note:* **It is not acceptable to add platform-specific functionality through this variance. The result of all method calls should preserve platform-neutral behavior.**

Calls to these proprietary APIs should produce “Undefined Reference” errors when tested by the JavaPureCheck tool. Use of stub or dummy classes to mask calls to these APIs is not permitted.

Use of this variance is acceptable as long as care is taken to minimize the risk of the Java virtual machine accidentally trying to load a class that does not exist (which would cause a runtime error). This is best done by isolating calls for each of these browsers into separate classes, and only loading those classes that are applicable to the active browser.

**Application of Variance:** Variance #67 can be used as an explanation for “Undefined Reference” errors or warnings generated when making calls to security methods in Netscape Navigator and Internet Explorer. Include in your explanation a statement specifying that no platform specific functionality is enabled by these calls other than to gain the necessary security permissions required. Contact Netscape or Microsoft for information about developing to their APIs.

*Note:* **Compliance is always measured using Java Compatible platforms. Specific code written to enable a browser’s ability to access secure methods will only be exercised if that browser uses a Java Compatible runtime environment. See the list of recommended platforms in Appendix A, "Available Java Platforms" for more information.**

## *Miscellaneous Variances*

The following variances address special cases that do not fall into a particular category.

### *Variance: #44— Use of Transitional Beans in 1.0.2 Applications*

Transitional beans are a set of APIs used with JDK 1.0.2 to implement bean-like functionality on the 1.0.2 Java platform. Sun discourages the use of these classes since the JavaBeans architecture available on JDK 1.1 software (and later Java platform versions) is much more robust.

A transitional bean must implement the following classes:

- `sunw.io.Serializable`
- `sunw.util.EventObject`
- `sunw.util.EventListener`

See `http://java.sun.com/beans/initial.html` for more information about transitional beans.

**Application of Variance:** Variance #44 can be used as an explanation for errors or warnings generated when adding the `sunw.*` classes necessary to implement transitional beans. In addition, you must provide an explanation stating that the classes are used for transitional beans purposes, and explain why the code cannot be implemented as a "real" bean.

## *Variance: #68— Use of Java Platform-Specific (Hybrid) Functionality*

Programs can be designed that run differently, depending upon the version of the Java platform active during runtime. These programs are called hybrid applications and can be classified as a 100% Pure Java program on each targeted Java platform version.

For example, a hybrid program could use AWT calls when the program runs on Java platform version 1.1, but use Java 2D™ calls when it runs on a 1.2 platform. The JavaPureCheck tool reports errors for the Java 2D API alls when it tests the program using the 1.1 system model because the Java 2D APIs are not defined for the 1.1 platform.

The JavaPureCheck tool will generate "Undefined Reference" errors for any API call made that does not exist in the selected system model.

Use of this variance is acceptable as long as care is taken to minimize the risk of the Java virtual machine accidentally trying to load a class that does not exist (which would cause a runtime error). This is best done by isolating differences in application logic into separate classes and only loading those classes which are applicable to a particular Java platform version.

*Note:* **Making direct calls to a method and catching `ClassNotFoundException` errors is not sufficient for this purpose.**

**Application of Variance:** Variance #68 can be used as an explanation for undefined reference errors on hybrid programs only. A description of how the undefined reference has been isolated from affected Java platform versions must be included in the explanation.

# *Available Java Platforms*

# *A*

You should execute your program on multiple Java platforms as a Dynamic Test of your program for purity. Your program should run to completion and demonstrate common behavior of all major features on at least three unique Java platforms. A Java platform is a combination of the operating system and a Java Compatible implementation of the Java application environment.

## *Platforms for Java Applications and Applets*

Tables A-1, A-3, and A-5 list the available platforms for the 1.0, 1.1, and 1.2 versions of the Java application environment used to verify the purity of Java applications, servlets, beans and class libraries.

Tables A-2, A-4, and A-6 list the available platforms and browsers for the 1.0, 1.1, and 1.2 versions of the Java application environment used to verify the purity of Java applets.

*Note:* **Servlets should be verified using the Servlet Development Kit which is available from `http://java.sun.com/products/servlet/index.html`.**

# *Available Platforms for Java Application Environment Version 1.0.2*

**Table A-1: Test Platforms for Applications and Class Libraries for Java Version 1.0.2**

| Operating System | Java Compatible Implementation Vendor |
|---|---|
| Win95/NT (32-bit Windows) | Sun Microsystems |
| Solaris Operating Environment | Sun Microsystems |
| Mac OS | Apple Computer |
| OS/2 Warp | IBM |
| AIX | IBM |
| Network Station | IBM |
| Netware 4.11 | Novell |
| HP UX | Hewlett Packard |

**Table A-2: Test Platforms and Browsers for Applets for Java Version 1.0.2**

| Operating System | Browser | Java Compatible Implementation Vendor |
|---|---|---|
| Win95/NT (32-bit Windows) | Navigator 3.x | Netscape |
| Solaris Operating Environment | Navigator 3.x | Netscape |
| Mac OS | Navigator 3.x | Netscape |
| OS/2 Warp | Navigator 3.x | Netscape |
| AIX | Navigator 3.x | Netscape |
| HP UX | Navigator 3.x | Netscape |
| Win95/98/NT (32-bit Windows) | Internet Explorer 3.x | Microsoft |
| Mac OS | Internet Explorer 3.x | Microsoft |
| Mac OS (IE option for MRJ 1.x) | Internet Explorer 3.x | Apple Computer |
| (any Java Compatible platform) | AppletViewer | (most) |
| Mac OS | AppletRunner | Apple Computer |

# *Available Platforms for Java Application Environment 1.1*

**Table A-3: Test Platforms for Applications, Beans, Class Libraries and Servlets for Java Version 1.1**

| Operating System N/C | Java Compatible Implementation Vendor |
|---|---|
| Win95/98/NT (32-bit Windows) | Sun Microsystems |
| Solaris Operating Environment | Sun Microsystems |
| JavaStation™ | Sun Microsystems |
| Mac OS | Apple Computer |
| OS/2 Warp | IBM |
| AIX | IBM |
| Network Station | IBM |
| Netware 5 | Novell |
| HP UX | Hewlett Packard |

**Table A-4: Test Platforms and Browsers for Applets for Java Version 1.1**

| Operating System N/C | Browser | Java Compatible Implementation Vendor |
|---|---|---|
| Win95/98/NT (32-bit Windows) | HotJava™ Browser 1.1 | Sun Microsystems |
| Solaris Operating Environment | HotJava Browser 1.1 | Sun Microsystems |
| JavaStation | HotJavaViews 1.0 | Sun Microsystems |
| Mac OS (IE option for MRJ 2.x) | Internet Explorer 3.x | Apple Computer |
| Mac OS (IE option for MRJ 2.x) | Internet Explorer 4.x | Apple Computer |
| Win95/98/NT (32-bit Windows) | Internet Explorer 4.x with Java Plug-In | Microsoft and Sun Microsystems |
| Win95/98/NT (32-bit Windows) | Navigator 4.x with Java Plug-In | Netscape and Sun Microsystems |
| (any Java Compatible platform) | AppletViewer | (most) |
| Mac OS | AppletRunner | Apple Computer |

# *Available Platforms for Java Application Environment 1.2*

**Table A-5: Test Platforms for Applications, Beans, Class Libraries and Servlets for Java Version 1.2**

| Operating System/NC | Vendor |
|---|---|
| Win95/98/NT | Sun Microsystems |
| Solaris Operating Environment | Sun Microsystems |
| Solaris x86 Operating Environment | Sun Microsystems |

**Table A-6: Testing Platforms and Browsers for Applets for Java Version 1.2**

| Operating System N/C | Browser | Implementation Vendor |
|---|---|---|
| Win95/98/NT (32-bit Windows) | Internet Explorer 4.x with Java Plug-In | Microsoft and Sun Microsystems |
| Win95/98/NT (32-bit Windows) | Navigator 4.x with Java Plug-In | Netscape and Sun Microsystems |
| Win95/98/NT (32-bit Windows) | AppletViewer | Sun Microsystems |
| Solaris Operating Environment | AppletViewer | Sun Microsystems |
| Solaris x86 Operating Environment | AppletViewer | Sun Microsystems |

# *Frequently Asked Questions for Purity* *B*

## *Frequently Asked Questions*

**Question 1: How do I explain all the bogus "possible hard-coded path name" warnings I get with the JavaPureCheck tool?**

Write a brief statement explaining that it is not a hard-coded path (if obvious) or a description of the string usage. The intent of your explanation is to show that the warning does not identify a potential portability problem.

**Examples**:

```
Class :MyClass1
   Warning: possible hard-coded path: big/small
   Note :Defines a bad path
   Explanation :This is not a pathname.


Class :MyClass2
   Warning: possible hard-coded path: product/big/pricing
   Note :Defines a bad path
   Explanation :This is part of a URL string.
```

**Question 2: How do I explain the Class.forName warnings?**

List each class that could be referenced by this statement. If your code allows a user to select or input the class being loaded (e.g., the class is user configurable), it is sufficient to explain how the class is chosen.

**Examples**:

```
Class :com.myorg.foo.bar
   Warning: method reference:
```

```
            java.lang.Class.forName(java.lang.String)
      Note :May load impure class
      Explanation :The following classes can be loaded by this
      statement and all are included as part of my
      verification package:
            com.myorg.Class1
            com.myorg.Class2a
            com.myorg.Class2b
            com.myorg.Class3


Class :com.abc.x123
      Warning: method reference:
            java.lang.Class.forName(java.lang.String)
      Note :May load impure class
      Explanation :This class is selected by the user from a
      menu that is preconfigured at install time.
```

**Question 3: I get Class.forName warnings in places I don't use it. What's up?**

The JavaPureCheck tool is not always able to distinguish how calls to this method were initiated. For example, the fragment of source code

```
return java.lang.System.class;
```

results in the execution of the following bytecodes (or equivalent)

```
ldc #1 <String "java.lang.System">
invokestatic #12 <Method
   java.lang.ClassforName(java.lang.String)>
areturn
```

Currently, the JavaPureCheck tool is not sophisticated enough to tell this code from the very similar

```
return Class.forName(cn);
```

which results in

```
aload_1
invokestatic #12 <Method
   java.lang.Class.forName(java.lang.String)>
areturn
```

Another example is the following code, which will also generate the warning:

```
public class test
{
```

```
    Class c = test.class;
}
```

Enter an explanation for the warning; the relevant line of source code would be a perfectly acceptable explanation. It's important that the explanation tell the class to which the `.class` operator is being applied. If the class is not part of the core set of Java technology APIs, then it is probable that the `forName` method is being used in a way which will require a much more detailed explanation.

**Example:**

```
Class :com.myorg.foo.bar
   Warning: method reference:
            java.lang.Class.forName(java.lang.String)
   Note :May load impure class
   Explanation :This is not a use of method Class.forName.
   The warning is being generated for the source statement:
            return java.lang.System.class
```

**Question 4: How do I mark warnings and errors that exist in pure 3rd party products I use in my code?**

If the product is certified, mark each warning and error as being dependent on a pure 3rd party product.

**Example:**

```
Class :sunsoft.jws.visual.rt.type.Converter
   Warning: method reference:
            java.lang.Class.forName(java.lang.String)
   Note :May load impure class
   Explanation :This warning occurs in Sun's Java WorkShop
   (Version 2.0) libraries. According to information on
   http://java.sun.com/100percent, this library is
   certified as 100% Pure Java.
```

**Question 5: How do I mark warnings and errors that exist in non-certified 3rd party products I use in my code?**

If the product has not been certified, you might have to contact the vendor for explanations. Please encourage these vendors to make their products compliant with Sun's purity standard. Remember that any and all classes that you use, rely on, or call, must pass the JavaPureCheck tool testing and have appropriate purity explanations in order to be considered compliant with Sun's purity standard.

**Question 6: I can't get the JavaPureCheck tool to work. Help!**

The JavaPureCheck tool will run on any 1.1 or later version Java Compatible runtime.

Prior to running the JavaPureCheck tool, you can add the `javapurecheck.jar` file to your `CLASSPATH` statement. The syntax for executing the JavaPureCheck tool once your `CLASSPATH` is set is:

```
java JavaPureCheck
```

Alternatively, you can also set the classpath on your java commandline. Consult your Java runtime documentation for details..

**Question 7: Why does the JavaPureCheck tool take so long to process my files, no matter how much memory I give it? What can I do?**

The JavaPureCheck tool does code checking that requires method simulation. This can be particularly memory intensive, and can also take time. Generally, the speed of the JavaPureCheck tool depends entirely on the number of class files that you give it, the class sizes(physical file size), and the number of static strings defined in each class file.

The GUI interface of the JavaPureCheck tool can be rather memory intensive as well. This is especially true when clicking on the "details" button to view purity warnings or errors because of the additional information that is being displayed.

If you find that the JavaPureCheck tool seems to "hang" at this point, run its command line version instead. The documentation that comes with the JavaPureCheck tool explains how to do this. You will still need to use the `-ms` and `-mx` Java virtual machine switches, but this should enable you to check your product and any third party classes that it uses more quickly.

When the JavaPureCheck tool has finished running, it will output a file called `RESULTS.JPC`. This file can be viewed and edited by any standard text editor. Optionally, you can still run the GUI, but click on "exit" rather than viewing the "details" from within the GUI. The `RESULTS.JPC` file will be available on disk and you can view or edit it in a text editor.

**Question 8: When running the JavaPureCheck tool, I see a number of "undefined reference" errors that point to classes that are not mine.**

Example:

```
Class :com.myclasses.classXYZ
   Error: undefined reference:
   com.thirdparty.class123.Init()
   Note :Undefined reference.
   Explanation : <Explanation required>
```

```
Status :ERROR
```

Think of the “undefined reference” error as a message stating: “incomplete product” or “classes missing”. It is important to remember when running the JavaPureCheck tool that you must include all of the classes that your product calls, uses, or relies upon. An “undefined reference” indicates that the JavaPureCheck tool cannot find one or more of these classes. Remember that the CLASSPATH will not be used to resolve these undefined references, and that they must be included in your JavaPureCheck test. Just add the missing class(es) to the JavaPureCheck list of classes to check and the error should go away.

**Question 9: When I run the JavaPureCheck tool on my classes, I get an “undefined reference” error pointing to the same class.**

**Example:**

```
Class :com.myclasses.classFoobar
   Error: undefined reference: com.myclasses.classFoobar
   Note :Undefined reference.
   Explanation : <Explanation required>
   Status :ERROR
```

This error sometimes occurs with `.class` files that have been “modified” after being compiled. Specifically, the problem might occur after running a “post processor” (e.g.. an obfuscator or optimizer) on your complied files. If you run into this problem, please contact the post-processorvendor for assistance.

**Question 10: I get an error for the use of “Runtime.exec()” in my class files. Is this permitted?**

This issue is completely documented in “Using Runtime.exec” in Chapter 2. To summarize, use of `Runtime.exec()` is permitted only when certain conditions are met. The exact criteria for use of the `Runtime.exec()` method are:

- The invocation must be a direct result of a specific user action; the user must know they're executing a separate application.
- The user must be able to choose, by configuration or as part of the invocation action, which application is executed.
- A failure of the `Runtime.exec()` method, especially one caused by the absence of the requested application, must be handled cleanly.

For each class that uses `Runtime.exec()` you must include information concerning each of these aspects to demonstrate that you meet the defined criteria.

**Question 11: Can a developer call all future releases of a certified program 100% Pure Java without undergoing further tests?**

No. If you release a new version of the product that is for bug fixes only, you should run JavaPureCheck on the updated product. If the new version includes bug fixes and new features, functionality or enhancements, then the product must be verified again to be considered compliant with the purity standard.

**Question 12: What Variances requests are allowble?**

- (#1) Use of `Runtime.exec` or `Method.invoke` to start a compiler.
- (#5) Use of `Runtime.exec` or `Method.invoke` to start a browser.
- (#17) Use of `Runtime.exec` or `Method.invoke` to perform a configurable user action.
- (#44) Use of transitional beans in 1.0.2 applications.
- (#45) Use of native code to enable common functionality (e.g., allowing access to port 80 on UNIX) -- RESTRICTED.
- (#47) Use of class libraries containing unused native code or unused external references --RESTRICTED.
- (#67) Use of Netscape™ and Internet Explorer security managers.
- (#68) Inclusion of Java platform version-specific (hybrid) functionality.

*Note:* **These variances are fully explained in the "Variances" section of this guide.**

**Question 13: What Variance requests are not acceptable?**

- Use of native code with pure workarounds for unsupported platforms.
- Any use of `sun.*` packages.
- Use of optional native libraries as plug-in replacements for pure libraries.
- Use of external class dependencies (e.g., JavaScript support in browser).
- Dependencies on non-portable (i.e., impure) standard extensions.

**Question 14: When running the JavaPureCheck tool, I note that some of my classes are missing from the results but I know that I included them because they are in the same directory as all of the other classes. I also will sometimes see the message "error: bad class file" appear in the console where the JavaPureCheck tool is being run. What's happening?**

This problem generally occurs because the file was compiled incorrectly or has been modified in such a way as to be invalid. We have seen this issue occur when compiling using Microsoft Visual J++ and marking the "debug" option. If you remove this option, then re-compile your classes, the "bad class file" message should go away.

**Question 15: Can I evaluate programs with dependencies on "optional packages"?**

Yes and no. Java optional packages are implementations of Sun authorized APIs that extend the core Java platform. These implementations can, in some cases, be portable and be written completely in the Java application

programming language, or they may be non-portable and make specific platform dependent references to native code or to the underlying operating system. Only programs bundled with portable optional packages can be considered compliant with the purity standard.

Portable optional packages are normally considered compliant with the 100% Pure Java standard. When bundled with a dependent program seeking certification, these portable optional packages are treated in the same manner as pure class libraries or beans.

*Note:* **Sun has defined an exception to this "must bundle" requirement for the Servlet optional package. Because the Servlet API is normally implemented as part of a web server, it is not reasonable to bundle a 100% Pure Java program implementation of the API with every servlet because there is no guaranteed way to invoke it.**

**Question 16: Can I use native JDBC drivers with my product?**

Yes, native JDBC drivers can be included in your program as long as you certify your program using a pure driver (i.e., references to native drivers may not be exclusively hard-coded into your program). See "Pitfall: "JDBC™ Driver Loading" " in Chapter 3 for more information about using JDBC drivers.

You can either include a 100% Pure Java certified driver in your verification package for certification, or use an uncertified (but pure) driver that will be included as part of the static check of your program.

**Question 17: How can my applet work outside the sandbox (i.e., access security-constrained methods) in Netscape Navigator and Internet Explorer?**

Use Variance #67. This variance allows your applet to make direct calls to the native Netscape and Microsoft security manger APIs. See "Variance: #67—Use of Netscape Navigator and Internet Explorer Security Managers." Contact Netscape or Microsoft for more information about developing to these APIs.

*Note:* **Verification of your program to the purity standard should always be done using Java Compatible platforms. Specific code written to enable a browser's ability to access secure methods will only be exercised if that browser uses a Java Compatible runtime environment. See Appendix A, "Available Java Platforms" for information about the platforms used for certification.**