Java 2 SDK for Solaris Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 806–7930–10
May 2002

020115@3062

Contents

Preface

This manual is an introduction to and overview of the new features and enhancements in version 1.4 of the Java™ 2 SDK, Standard Edition, for the Solaris™ Operating Environment.

Who Should Use This Book

This document is intended for application developers who use the Java 2 SDK, Standard Edition, in the Solaris™ Operating Environment. The Java 2 SDK, Standard Edition, software is optimized to deliver superior performance to server- and client-side Java technology applications in an enterprise environment.

This document is a subset of the Java 2 SDK documentation available at http://java.sun.com/j2se/1.4/docs/index.html. Upon final release of this product, consider that online documentation to be the definitive description of version 1.4 of the Java 2 SDK, Standard Edition product.

How This Book Is Organized

Chapter 1 lists the features and enhancements of the product.

Chapter 2 discusses compatibility issues.

Chapter 3 describes command-line options available for maximizing virtual-machine performance.

Chapter 4 describes the new assertions facility in the Java™ 2 Platform.

Related Documentation

These documents also have information related to this release:

- *Solaris Java Plug-in User's Guide*
- *Java 2 SDK, Standard Edition v. 1.4 Release Notes* located online at http://java.sun.com/j2se/1.4/relnotes.html.
- *Java 2 SDK, Standard Edition, v. 1.4 Documentation* located online at http://java.sun.com/j2se/1.4/docs/index.html.
- *Java 2 Platform, Standard Edition, v 1.4 API Specification* located online at http://java.sun.com/j2se/1.4/docs/api/index.html.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`

TABLE P–1 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

New Features and Enhancements

The new features of version 1.4 of the Java™ 2 SDK, Standard Edition, (J2SE™) are listed below. Documentation for the full list of features, including features brought forward from previous versions of the Java 2 SDK, Standard Edition,, is available at http://java.sun.com/j2se/1.4/docs/index.html.

Java 2 Platform

XML Processing

The Java API for XML processing has been added to the Java 2 Platform. J2SE 1.4.0 provides basic support for processing XML documents through a standardized set of APIs. For more information, see http://java.sun.com/j2se/1.4/docs/guide/xml/index.html.

New I/O APIs

The new I/O (NIO) API provides new features and improved performance in the areas of buffer management, character-set support, regular-expression matching, file I/O, and scalable network I/O. For more information, see http://java.sun.com/j2se/1.4/docs/guide/nio/index.html.

Security

- The Java™ Cryptography Extension (JCE), Java™ Secure Socket Extension (JSSE), and Java™ Authentication and Authorization Service (JAAS) security features have now been integrated into J2SE v 1.4 rather than being optional packages.
- There are two new security features:
  - The Java™ GSS-API can be used for securely exchanging messages between communicating applications using the Kerberos V5 mechanism. For more information, see http://java.sun.com/j2se/1.4/docs/guide/security/jgss/tutorials/index.html.
  - The Java™ Certification Path API includes new classes and methods in the `java.security.cert` package that allow you to build and validate certification paths (also known as "certificate chains"). For more information, see http://java.sun.com/j2se/1.4/docs/guide/security/certpath/CertPathProgGuide.html.

- Due to import control restrictions, the JCE jurisdiction policy files shipped with the J2SE, v 1.4 allow "strong" but limited cryptography to be used. An "unlimited" version of these files indicating no restrictions on cryptographic strengths is available.
- The JSSE implementation provided in this release includes the strong cipher suites. However, due to U.S. export control restrictions, it does not allow the default `SSLSocketFactory` and `SSLServerSocketFactory` to be replaced. For more information, please see the *JSSE Reference Guide* at http://java.sun.com/j2se/1.4/docs/guide/security/jsse/JSSERefGuide.html.
- With the integration of JAAS into the J2SE, the `java.security.Policy` API handles Principal-based queries, and the default policy implementation supports Principal-based grant entries. Thus, access control can now be based not just on what code is running, but also on *who* is running it.
- Support for dynamic policies has been added. In J2SE releases prior to version 1.4, classes were statically bound with permissions by querying security policy during class loading. The lifetime of this binding was scoped by the lifetime of the class loader. In version 1.4 this binding is now deferred until needed by a security check. The lifetime of the binding is now scoped by the lifetime of the security policy.

For more information on security in J2SE 1.4, see http://java.sun.com/j2se/1.4/docs/guide/security/index.html.

Java 2D™ Technology

Java 2D™ technology includes many new features including performance improvements, support for hardware acceleration for offscreen images, a pluggable image I/O framework, a new print service API, and several new font features. For more information, see http://java.sun.com/j2se/1.4/docs/guide/2d/new_features.html.

Image I/O Framework

The Java Image I/O Framework provides a pluggable architecture for working with images that are stored in files and accessed across the network. This framework offers substantially more flexibility and power than the pre-J2SE 1.4.0 APIs for loading and saving images. For more information, see http://java.sun.com/j2se/1.4/docs/guide/imageio/index.html.

Java Print Service API

The Java Print Service is a new Java Print API that enables client and server applications to:

- Discover and select print services based on their capabilities

- Specify the format of print data
- Submit print jobs to services that support the document type to be printed.

For more information, see http://java.sun.com/j2se/1.4/docs/guide/jps/index.html.

AWT

Changes to the Abstract Window Toolkit (AWT) package center on improving the robustness, behavior, and performance of programs that present a graphical user interface. Improvements include the following:

- A new focus architecture replaces the previous implementation and addresses many focus-related bugs that were caused by platform inconsistencies, and incompatibilities between AWT and Swing components.
- The new full-screen exclusive mode API supports high-performance graphics by suspending the windowing system so that drawing can be rendered directly to the screen. This capacity benefits applications such as games, or other rendering-intensive applications.
- Headless support is now enabled by new graphics environment methods that indicate whether a display, keyboard, and mouse can be supported in a graphics environment.
- The ability to disable native frame decorations is now available for applications that need complete control of the specification of a frame's appearance. When enabled, it prevents the rendering of a native title bar, system menu, border, or other native screen components.
- The mouse wheel, with a scroll wheel in place of the middle mouse button, is enabled with new built-in Java support for scrolling with the mouse wheel. Also, a new mouse wheel listener class allows customization of mouse wheel behavior.
- The AWT package has been modified to be fully 64-bit compliant and now runs on Solaris machines with 64-bit and 32-bit addresses.

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/awt/AWTChanges.html.

Swing

Many new features have been added to Swing:

- The spinner component is a single line input field that enables the user to select a number or a value by cycling through a sequence of values with a tiny pair of up/down arrow buttons.
- The new formatted text field component allows formatting of dates, numbers, and strings, such as a text field that accepts only decimal currency values.

- A new drag-and-drop architecture provides seamless drag-and-drop support between components as well as an easy way to implement drag-and-drop in your customized Swing components. You need only to write a couple of methods that describe the particulars of your data model.

Several features have been enhanced in Swing:

- The progress bar component has been enhanced to support an indeterminate state. Rather than showing the degree of completeness, the indeterminate progress bar uses constant animation to show that a time-consuming operation is occurring.
- The tabbed pane component has been enhanced to support scrollable tabs. With this feature enabled, if all the tabs cannot fit within a single tab run, the tabbed pane component displays a single, scrollable run of tabs instead of wrapping the tabs onto multiple runs.
- The popup and popup factory classes, which were previously package private, have been exposed and made public so that developers can customize or create their own popups.
- The new focus architecture is fully integrated into Swing.

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/swing/SwingChanges.html.

Drag and Drop

Swing has added support for data transfer between applications. A drag and drop operation is a data transfer request that has been specified by a gesture with a graphical pointing device. In the case of copy/paste, data transfer is often initiated with the keyboard. The ability to transfer data takes two forms: Drag and drop support and clipboard transfer via cut/copy/paste. See
http://java.sun.com/j2se/1.4/docs/guide/swing/1.4/dnd.html.

Logging API

The Java Logging APIs facilitate software servicing and maintenance at customer sites by producing log reports suitable for analysis by end users, system administrators, field service engineers, and software development teams. The Logging APIs capture information such as security failures, configuration errors, performance bottlenecs, and bugs in the application or platform. For more information, see
http://java.sun.com/j2se/1.4/docs/guide/util/logging/index.html.

Java™ Web Start Product

The Java Web Start product is a new application-deployment technology that is bundled with J2SE 1.4.0. With Java Web Start, you launch applications simply by clicking on a Web page link. If the application is not present on your computer, Java

Web Start automatically downloads all necessary files. It then caches the files on your computer so the application is always ready to be relaunched anytime you want -- either from an icon on your desktop or from the web-page link. For more information, see http://java.sun.com/j2se/1.4/docs/guide/jws/index.html.

Long-term Persistence of JavaBeans™ Components

The new persistence model is designed to handle the process of converting a graph of JavaBeans eans to and from a persistent form. The new API is suitable for creating archives of graphs of JavaBeans components as textual representations of their properties. For more information, see http://java.sun.com/j2se/1.4/docs/guide/beans/changes14.html.

JDBC™ 3.0 API

The JDBC™ 3.0 API, which is composed of packages `java.sql` and `javax.sql`, provides universal data access from the Java programming language. By using the JDBC 3.0 API, you can access virtually any data source, from relational databases to spreadsheets and flat files. JDBC technology also provides a common base on which tools and alternative interfaces can be built.

This API enables you to do the following:

- Set savepoints in a transaction
- Keep result sets open after a transaction is committed
- Reuse prepared statements
- Get metadata about the parameters to a prepared statement
- Retrieve keys that are automatically generated
- Have multiple result sets open at one time

There are two new JDBC data types, `BOOLEAN` and `DATALINK`. The `DATALINK` type enables management of data outside of a data source. This release also establishes the relationship between the JDBC Service Provider Interface and the Connector architecture.

For more information, see http://java.sun.com/j2se/1.4/docs/guide/jdbc/index.html.

Assertion Facility

An assertion facility has been added to the Java 2 Platform. Assertions are boolean expressions that the programmer believes to be true concerning the state of a computer program. For example, after sorting a list, the programmer might assert that

the list is in ascending order. Evaluating assertions at runtime to confirm their validity is one of the most powerful tools for improving code quality, as it quickly uncovers the programmer's misconceptions concerning a program's behavior. For more information, see "Assertion Facility" in *Java 2 SDK for Solaris Developer's Guide*.

Preferences API

This new feature a simple API for managing user preference and configuration data. Applications require preference and configuration data to adapt to different users, environments and needs. Applications need a way to store, retrieve, and modify this data. This need is met by the Preferences API. The Preferences API is intended to replace most common uses of class `java.util.Properties`, rectifying many of its deficiencies, while retaining its light weight. For more information, see http://java.sun.com/j2se/1.4/docs/guide/lang/preferences.html.

Endorsed Standards Override Mechanism

An *endorsed standard* is a Java API defined through a standards process other than the Java Community Process℠ (JCP℠). Because endorsed standards are defined outside the JCP, it is anticipated that such standards may be revised between releases of the Java 2 Platform. In order to take advantage of new revisions to endorsed standards, developers and software vendors may use the Endorsed Standards Override Mechanism to provide newer versions of an endorsed standard than those included in the Java 2 Platform as released by Sun Microsystems. For full information on the Endorsed Standards Override Mechanism, see the documentation on the web at http://java.sun.com/j2se/1.4/docs/guide/standards/.

64–bit Support

J2SE 1.4.0 for the Solaris Operating Environment, SPARC Platform Edition, supports 64-bit operation on 64-bit Sparc-v9 platforms when using the Java HotSpot™ Server VM. This allows support for heaps greater than 4 Gbytes, which is the absolute maximum that can be supported by a 32-bit VM. The Java HotSpot Server VM includes support for either 32-bit or 64-bit operation by using an appropriate command-line flag. When using the 64-bit VM a performance penalty of approximately 15% to 25% may be observed, depending on the amount of time your program spends accessing reference variables. J2SE 1.4.0 does not support 32-bit shared libraries, when running the 64-bit VM. Native (Java Native Interface) code must be recompiled in 64-bit mode.

Java™ HotSpot Virtual Machines

The Java virtual machines in this release include several enhancements.

- Signal-chaining facility.
- 64-bit support for 64-bit Sparc-v9 platforms.
- Error-reporting mechanism.
- New command-line option for performaing additional checks for Java Native Interface (JNI) functions.
- New facility for logging garbage-collection events.

For more information, see http://java.sun.com/j2se/1.4/docs/guide/vm/index.html.

Performance

This release includes performance enhancements. For more information, see http://java.sun.com/j2se/1.4/docs/guide/performance/index.html.

Networking Support, Including IPv6

New features include support for IPv6 in TCP- and UDP-based applications, and support for unconnected/unbound sockets, allowing more flexible socket creation, binding, and connection. A mechanism called Java Secure Socket Extension provides encryption for data sent via sockets, and a new class, `URI`, allows URI construction and parsing without the presence of a protocol handler. The FTP Protocol Handler has been overhauled for conformity to current standards. The default character set is now UTF8, and APIs have been added to enable other character schemes.

A new class, `NetworkInterface`, allows enumeration of interfaces and addresses, and JNDI DNS SP Support in `InetAddress` enables applications to configure a pure Java name service provider. TCP out-of-band data provides support for legacy applications; a UDP Connection function registers destination address with the OS, enabling asynchronous errors to be returned on the UDP socket; and full SOCkS V5 and V4 TCP support includes auto-negotiation with the proxy for which version to use. In addition, there are improvements to streaming, request and response headers processing, and error handling.

For more information, see http://java.sun.com/j2se/1.4/docs/guide/net/enhancements14.html.

RMI

The RMI runtime implementation will now preserve the server-side stack trace information of an exception that is thrown from a remote call, in addition to filling in the client-side stack trace as it did previous releases. Therefore, when such an exception becomes accessible to client code, its stack trace will now contain all of its original server-side trace data followed by the client-side trace.

In J2SE 1.4.0, certain static methods of `java.rmi.server.RMIClassLoader` delegate their behavior to an instance of a new service provider interface, `java.rmi.server.RMIClassLoaderSpi`. This service provider object can be configured to augment RMI's dynamic class-loading behavior for a given application. By default, the service provider implements the standard behavior of all of the static methods in `RMIClassLoader`.

The `java.rmi.server.hostname` property can now be dynamically updated to indicate that future exports should use a new host name. Therefore, the new host name value will be contained in the stub for an object that is exported after the property is updated.

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/rmi/relnotes.html.

Serialization

This release has several changes and enhancements to the serialization API, including

- Support for deserialization of objects that are known to be unshared in the data-serialization stream.
- Support for a class-defined readObjectNoData method.
- Important bug fixes.

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/serialization/relnotes14.html.

Java Naming and Directory Interface™ (JNDI)

Java Naming and Directory Interface™ (JNDI) has the following enhancements in J2SE 1.4.0:

- An Internet Domain Naming System (DNS) service provider is part of J2SE 1.4.0. This component enables applications to read data that is stored in the DNS.
- The JNDI Lightweight Directory Access Protocol (LDAP) service provider has security enhancements that enable applications to establish secure sessions over existing LDAP connections and to use different authentication protocols.

- The JNDI CORBA Object Services (COS) naming service provider supports the Interoperable Naming Service (INS) specification (99-12-03).

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/jndi/index.html#14changes.

CORBA, Java™ IDL, and RMI-IIOP

The ORB shipping as part of Java 2 Platform now includes a Portable Object Adapter (POA) functionality. An ORB makes it possible for client(s) to make method invocations on the objects being supported by server(s) executing on the same or different machine(s). The POA functionality allows programmers to construct object implementations that are portable between different ORB products, provide support for objects with persistent identities, and much more. Other new features include Portable Interceptors, Interoperable Naming Service, GIOP 1.2 support, Dynamic Management of Any values, and new tools that support a persistent naming service and other features.

To learn more about the changes in Java IDL between J2SE v.1.3 and J2SE v.1.4, see http://java.sun.com/j2se/1.4/docs/guide/idl/jidlChanges.html. For general information, see http://java.sun.com/j2se/1.4/docs/guide/idl/index.html.

Java™ Platform Debugger Architecture Product

J2SE 1.4.0 includes an enhanced Java Platform Debugger Architecture that has the following new features.

- The Java HotSpot™ virtual machine now uses "full-speed debugging". In the previous version of Java HotSpot, when debugging was enabled, the program executed using only the interpreter. Now, the full performance advantage of Java HotSpot technology is available to programs running with debugging enabled. The improved performance allows long running programs to be more easily debugged. It also allows testing to proceed at full speed and the launch of a debugger to occur on an exception.
- A HotSwap feature has been added to allow a class to be updated while under the control of a debugger.
- `EventRequests` now have the capability of specifying an instance filter, which restricts the events generated by the request to those in which the currently executing instance is the object specified.
- The Java Platform Debugger Architecture has been extended so that non-Java programming language source, which is translated to Java programming language source, can be debugged in the future.
- A request can now be made to control target VM termination notification, allowing clean shutdown synchronization.

Internationalization

Character handling in J2SE 1.4.0 is based on version 3.0 of the Unicode standard. This character handling affects the `Character` and `String` classes in the `java.lang` package as well as the collation and bidirectional text analysis functionality in the `java.text` package.

J2SE 1.4.0 supports Thai and Hindi in all areas of functionality. See the Supported Locales document online for complete information on supported locales and writing systems.

Class `java.util.Currency` was introduced so that currencies can be referenced independent of locales. There are new methods on `java.text.NumberFormat` and related classes to specify the currency for formatting monetary values.

Java™ Plug-in Product

Java Plug-in 1.4 offers the following new features:

- Support of HTTPS through Java Secure Socket Extension (JSSE) rather than the browser
- Enhanced applet caching so that other files, such as GIF, JPEG, and AU, can be cached in addition to JAR and class files
- Applet persistence across browser page changes
- Various applet-compatibility enhancements to allow most JDK 1.1 applets to run seemlessly in Java 2.

Version 1.4 also provides the following:

- Access to the DOM through standard, w3c-defined interfaces
- Assertion and logging support
- Applet usability enhancements that include a customizable progress bar for applet loading
- Enhanced options in the Java Console to allow dynamic reconfiguration of proxy settings, the policy file, etc., while the applet is running.

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/plugin/index.html.

Collections Framework

The collections framework has several enhancements in J2SE 1.4, including a marker interface to advertise random access, an identity-based (rather than equality-based) Map, insertion-order-preserving Map and Set implementations, and several new algorithms for manipulating and returning values from lists. See http://java.sun.com/j2se/1.4/docs/guide/collections/changes4.html for details.

Accessibility

J2SE 1.4.0 offers new support for accessibility in the following areas:

- Mnemonic tab navigation on a JTabbedPane
- Text editing by assistive technologies
- Accessibility of HTML components
- Accessibility of Swing Actions
- List navigation by using the first letter of list items
- Component roles DATE_EDITOR, FONT_CHOOSER, GROUP_BOX, SPIN_BOX, STATUS_BAR
- Properties to indicate the presence of a screen magnifier or screen reader, or to specify assistive technologies to load into the Java virtual machine

For more information, see
http://java.sun.com/j2se/1.4/docs/guide/access/index.html.

Regular Expressions

The new package `java.util.regex` contains classes for matching character sequences against patterns that are specified by regular expressions. For details, see the API specification for `java.util.regex` at
http://java.sun.com/j2se/1.4/docs/api/java/util/regex/package-summary.html.

Math

A new, efficient method for generating prime numbers with no need for the caller to specify a certainty has been added to class `java.math.BigInteger`. For more information, see
http://java.sun.com/j2se/1.4/docs/guide/math/enhancements14.html.

Reflection

Certain reflective operations, specifically `java.lang.reflect.Field`, `java.lang.reflect.Method.invoke()`, `java.lang.reflect.Constructor.newInstance()`, and `Class.newInstance()`, have been rewritten for higher performance. Reflective invocations and instantiations are several times faster than in previous releases. For more information, see http://java.sun.com/j2se/1.4/docs/guide/reflection/index.html.

Java™ Native Interface

The Java Native Interface (JNI) has been enhanced in J2SE 1.4 to reflect a new feature of the `java.nio` package: direct buffers. The contents of a direct buffer can, potentially, reside in native memory outside of the ordinary garbage-collected heap. Also, new Invocation Interface routine `AttachCurrentTreadAsDaemon` allows native code to attach a daemon thread to the virtual machine; this is useful when the VM should not wait for this thread to exit upon shutdown. See *JNI Enhancements* online at http://java.sun.com/j2se/1.4/docs/guide/jni/jni-14.html.

Tools and Utilities

See also the *Tools Changes* online documentation at http://java.sun.com/j2se/1.4/docs/tooldocs/tools-changes.html.

- The *java* application launcher supports command-line options for support of the new Assertions Facility. On the Solaris™ Operating Environment (SPARC™ Platform Edition) new command-line options are available for specifying 64-bit or 32-bit operation. New option `-Xloggc:`*file* logs each garbage-collection event in the specified file. New option `Xcheck:jni()` performs additional checks for Java Native Interface (JNI) functions at the cost of some performance degradation.
- The *javadoc* tool has several new tags, a dozen new options, smarter inheriting of doc comments, more control over HTML output, improvements to the doclet API, better error messages, dozens of bug fixes, and is easier to run. For more details, see *What's New in Javadoc 1.4* online at http://java.sun.com/j2se/1.4/docs/tooldocs/javadoc/whatsnew-1.4.html. Downloadable as a separate product, the MIF doclet http://java.sun.com/j2se/javadoc/mifdoclet) has had a major upgrade -- it generates API documentation in FrameMaker and PDF format.
- The *native2ascii* tool contains bug fixes to process source files encoded with Unicode encoding correctly.
- The *idlj* tool now generates server-side bindings following the Portable Servant Inheritance Model. This change involves producing new POA bindings by default. A new command-line option is added to enable you to continue to generate backwards-compatible server-side bindings. To learn more about the Portable

Servant Inheritance Model, see http://java.sun.com/j2se/1.4/docs/guide/idl/POA.html. For more information on the *idlj* tool, see http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/toJavaPortableUG.html.

- The *orbd* tool, or *Object Request Broker Daemon*, is a new alternative tool for the Transient Naming Service, `tnameserv`. ORBD includes both a Transient Naming Service and a Persistent Naming Service. The orbd tool is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment. The orbd tool incorporates the functionality of a Server Manager, an Interoperable Naming Service, and a Bootstrap Name Server. When used in conjunction with the `servertool`, the Server Manager locates, registers, and activates a server when a client wants to access the server. For more information, see http://java.sun.com/j2se/1.4/docs/guide/idl/orbd.html.
- The servertool is new to J2SE 1.4. The servertool provides an ease-of-use interface for application programmers to register, unregister, startup, and shutdown a server. For more information, see http://java.sun.com/j2se/1.4/docs/guide/idl/servertool.html.
- The rmic compiler has a new option to enable Portable Object Adapter, or POA, support for Remote Method Invocation. The POA enables portability among vendor ORBs, among other uses. To learn more about the POA, see http://java.sun.com/j2se/1.4/docs/guide/idl/POA.html. To enable POA support with the `rmic` compiler, use the arguments `rmic -iiop -poa`. For more information on the rmic compiler, see http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/rmic.html.
- The graphical*Policy Tool* utility has been enhanced to enable specifying a Principal field indicating what user is to be granted specified access control permissions.

Compatibility with Previous Releases

This document contains information on the following topics:

For information about incompatibilities between versions 1.3 and 1.2 of the Java platform, see the compatibility documentation for the Java 2 Platform, v1.3 at http://java.sun.com/j2se/1.3/compatibility.html.

For information about incompatibilities between versions 1.2 and 1.1 of the Java platform, see the compatibility documentation for the Java 2 Platform, v1.2 at http://java.sun.com/j2se/products/jdk/1.2/compatibility.html.

For information about incompatibilities between versions 1.0 and 1.1 of the Java platform, see the compatibility documentation at http://java.sun.com/products/jdk/1.1/compatibility.html for the JDK™ 1.1 software.

Binary Compatibility

The Java™ 2 SDK, Standard Edition, (J2SE™), v1.4 is upwards binary-compatible with J2SE 1.3 except for the incompatibilities listed below. This means that, except for the noted incompatibilities, class files built with version 1.3 compilers will run correctly in the J2SE 1.4.

In general, the policy is that

- Maintenance releases (for example 1.2.1, 1.2.2) within a family (1.2.x) will maintain both upward and downward binary-compatibility with each other.
- Functionality releases (for example 1.3, 1.4) within a family (1.x) will maintain upward but not necessarily downward binary compatibility with each other.

Some early bytecode obfuscators produced class files that violated the class file format as given in the virtual machine specification. Such improperly formatted class files will not run on the J2SE's virtual machine, though some of them may have run on earlier versions of the virtual machine. To remedy this problem, regenerate the class files with a newer obfuscator that produces properly formatted class files.

Source Compatibility

The J2SE 1.4 is upwards source-compatible with earlier versions, except for the incompatibilities listed below. This means that, except for the noted incompatibilities, source files written to use the language features and APIs defined for earlier releases can be compiled and run in the J2SE 1.4.

Downward source compatibility is not supported. If source files use new language features or Java 2 Platform APIs, they will not be usable with an earlier version of the Java platform.

In general, the policy is:

- Maintenance releases do not introduce any new language features or APIs, so they maintain source-compatibility in both directions.
- Functionality releases and major releases maintain upwards but not downwards source-compatibility.

Deprecated APIs are methods and classes that are supported *only* for backwards compatibility, and the compiler will generate a warning message whenever one of these is used, unless the `-nowarn` command-line option is used. It is recommended that programs be modified to eliminate the use of deprecated methods and classes, though there are no current plans to remove such methods and classes entirely from the system.

Some APIs in the `sun.*` packages have changed. These APIs are not intended for use by developers. Developers importing from `sun.*` packages do so at their own risk. For more details, see *Why Developers Should Not Write Programs That Call* `sun.*` *Packages* at http://java.sun.com/products/jdk/faq/faq-sun-packages.html.

Incompatibilities in the Java 2 Platform, Standard Edition, v1.4

J2SE 1.4 is strongly compatible with previous versions of the Java 2 Platform. Almost all existing programs should run on J2SE 1.4 without modification. However, there are some minor potential incompatibilities that involve rare circumstances and "corner cases" that we are documenting here for completeness.

1. Beginning with version 1.4 of the Java 2 Platform, class `javax.swing.tree.DefaultTreeModel` allows a null root node. In previous versions, `DefaultTreeModel` would not allow a null root, even though the specification for `TreeModel` indicated a null root was valid. `DefaultTreeModel` now allows setting a null root, as well as a null root in the constructor. As part of this change, the specification for `DefaultTreeModel.setRoot()` has been revised. The old specification for `DefaultTreeModel.setRoot()` was:

   Sets the root to root. This will throw an IllegalArgumentException if root is null.

   The new `DefaultTreeModel.setRoot()` specification says:

   Sets the root to root. A null root implies the tree is to display nothing, and is legal.

2. If a serializable inner class contains explicit references to its class object, then the computed value of the serial version UID for the class will be different in J2SE 1.3 and J2SE 1.4. The difference is due to the fact that the computation of the serial version UID is sensitive to modifications made in the javac compiler between J2SE 1.3 and J2SE 1.4.

   To avoid this problem affecting your applications, we recommend that you add an explicit serial version UID to your serializable classes. You can use the serialver tool to obtain the serial version UID of classes compiled with the J2SE 1.3 javac compiler.

3. Beginning with J2SE 1.4.0, public static field `DefaultPainter` in class `javax.swing.text.DefaultHighlighter` is final. In previous versions of the Java 2 Platform, Standard Edition, this field was non-final.

4. The way in which HTML forms are modeled internally in the implementation of the Java 2 Platform has changed in J2SE 1.4.0. Previously, any attributes of a form would be stored in the attributeset of all of the children character elements. In J2SE 1.4.0, an element is created to represent the form, better matching that of the html file itself. This allows for better modeling of the form, as well as consistent writing of the form. This change was made to address bug 4200439.

   This change will effect developers who relied on forms being handled loosely. As an example, pre-1.4.0 implementations would previously treat the following invalid html

   ```
   <table>
   <form>
   </table>
   ```

```
</form>
```

as

```
<form>
<table>
</table>
</form>
```

J2SE 1.4.0 instead treats it as:

```
<table>
<form>
</form>
</table>
```

While this change likely will not impact many developers, there is the possibility you will need to update your code. If you had previously been expecting the attributes of the leaf Elements to contain the Form's attributes, you will now have to get the attributes from the Form Element's AttributeSet.

5. The value of static final field `MOUSE_LAST` in class `java.awt.event.MouseEvent` has changed to `507` beginning in J2SE 1.4.0. In previous versions of the Java 2 Platform, the value of `MOUSE_LAST` was `506`.

   Because compilers hard-code static final values at compile-time, code that refers to `MOUSE_LAST` and that was compiled against a pre-1.4.0 version of `java.awt.event.MouseEvent` will retain the old value. Such code should be recompiled with the version 1.4.0 compiler in order to work with J2SE 1.4.0.

6. Changes have been made to make the APIs for the CORBA technology shipped in J2SE 1.4.0 compliant to the CORBA 2.3 mapping as specified by the OMG documents referenced in *CORBA Compatibility Information*, available online at http://java.sun.com/j2se/1.4/compatibility-CORBA.html. Consult that document for information regarding all of the API changes related to CORBA functionality between J2SE v1.3 and v1.4.0, as well as a listing of all OMG specifications with which J2SE 1.4.0 complies.

7. Beginning with J2SE 1.4.0, ObjectOutputStream's public one-argument constructor requires the "enableSubclassImplementation" SerializablePermission when invoked (either directly or indirectly) by a subclass which overrides `ObjectOutputStream.putFields` or `ObjectOutputStream.writeUnshared`.

   Also beginning with J2SE 1.4.0, ObjectInputStream's public one-argument constructor requires the "enableSubclassImplementation" SerializablePermission when invoked (either directly or indirectly) by a subclass which overrides `ObjectInputStream.readFields`or `ObjectInputStream.readUnshared`.

   This change will not affect the great majority of applications. However, it will affect any ObjectInputStream/ObjectOutputStream subclasses which override the `putFields` or `readFields` methods without also overriding the rest of the serialization infrastructure.

8. The Javac bytecode compiler in J2SE 1.4.0 is more strict than in previous versions in enforcing compliance with the Java Language Specification. The new compiler may not compile existing code that does not strictly conform to the Java Language Specification, even though that code may have compiled with earlier versions of the compiler.

   The following are some examples of situations in which the J2SE 1.4.0 compiler is stricter.

   - The compiler now detects unreachable empty statements as required by the language specification. Here are two examples of fairly common cases that the compiler now detects and rejects.

     ```
     return 0;/* exit success */;
     ```
     and
     ```
     {
         return f();
     } catch (Whatever e) {
         throw new Whatever2();
     };
     ```

     Note the extra semicolon in both cases, which the compiler now correctly regards as unreachable empty statements. In addition, some automatically generated source code may generate unreachable empty statements.

   - The compiler now rejects import statements that import a type from the unnamed namespace. Previous versions of the compiler would accept such import declarations, even though they were arguably not allowed by the language (because the type name appearing in the import clause is not in scope). The specification is being clarified to state clearly that you cannot have a simple name in an import statement, nor can you import from the unnamed namespace.

     To summarize, the syntax

     ```
     import SimpleName;
     ```

     is no longer legal. Nor is the syntax

     ```
     import ClassInUnnamedNamespace.Nested;
     ```

     which would import a nested class from the unnamed namespace. To fix such problems in your code, move all of the classes from the unnamed namespace into named namespace.

9. As of J2SE 1.4.0, the javac bytecode compiler uses "`-target 1.2`" by default as opposed to the previous "`-target 1.1`" behavior. See the reference page for the javac compiler at http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javac.html for descriptions of these behaviors. One of the changes involved in targeting 1.2 is that the compiler no longer generates and inserts method declarations into class files when the class inherits unimplemented methods from interfaces. These inserted methods, like all other non-private methods, are included in the default serialVersionUID computation. As a result, if you define an abstract serializable

class which directly implements an interface but does not implement one or more of its methods, then its default serialVersionUID value will vary depending on whether it is compiled with the J2SE 1.4 version of javac or a previous javac.

For background information on these methods inserted by earlier versions of javac, see the report for bug 4043008, available online at http://java.sun.com/jdc/bugParade/bugs/4043008.html.

10. *Source incompatibility* - The JDBC 3.0 API, included as part of the J2SE 1.4, introduces two new interfaces and adds several new methods to existing interfaces. Drivers and applications that use earlier versions of the JDBC API are binary compatible with the J2SE 1.4 and will run with no problem. However, the changes made in the JDBC 3.0 API are not source compatible. Drivers and applications that implement the JDBC interfaces must be updated to reflect the changes in order to build successfully. Chapter 6 of The JDBC 3.0 Specification gives a complete list of what must be done to be compliant with the JDBC 3.0 API, and thereby be source compatible with J2SE 1.4.
11. Prior to J2SE 1.4, a `FileNotFoundException` would be thrown if the file type was known and the response code was greater than or equal to 400. Otherwise no exception would be thrown. The correct behavior as implemented in J2SE 1.4 is for `URLConnection.getInputStream` to throw an `IOException` for all http errors regardless of the file type and throw `FileNotFoundException`, which is a subclass of `IOException`, only if the http response indicates that the resource was not found. In other words, the `FileNotFoundException` is thrown only if the response code is 404 or 410. As part of this change, `HttpURLConnection.getErrorStream` now can be used to read the error page returned from the server. Prior to J2SE 1.4, `getErrorStream` always returned `null`. Also, method `HttpURLConnection.getResponseCode` works correctly in J2SE 1.4.
12. The `assert` keyword has been added to the Java Programming Language in J2SE 1.4. Because of the new keyword, existing programs that use "assert" as an identifier will not be in conformance with J2SE 1.4. The addition of this keyword does not, however, cause any problems with the use of preexisting binaries (.class files). In order to ease the transition from pre-J2SE 1.4 releases in which “assert” is a legal identifier to J2SE 1.4 where it isn’t, the Javac bytecode compiler supports two modes of operation in J2SE 1.4.

    In the normal mode of operation, the compiler accepts programs conforming to the specification for the previous release (J2SE 1.3). Assertions are not permitted, and the compiler generates a warning if the assert keyword is used as an identifier.

    In an alternate mode of operation, the compiler accepts programs conforming to the specification for J2SE 1.4. Assertions are permitted, and the compiler generates an error message if the assert keyword is used as an identifier.

    To enable assertions, use the `-source 1.4` command line switch. In the absence of this flag, the behavior defaults to "1.3" for maximal source compatibility. Support for 1.3 source compatibility is likely to be phased out over time.

13. The API specification for interface `java.applet.AppletContext` has been modified to enable applet developers to stream data and objects for persistent use during a browser session. This eliminates the need for developers to use static classes to cache data and objects, but it introduces a potential binary incompatibility. Any existing applications containing classes that implement the `AppletContext` interface will not be compatible with the new `AppletContext` specification. Such classes will have to be modified to implement the revised `AppletContext` API. In practice, the only applications that commonly implement `AppletContext` are those that act as applet containers, such as Java™ Plug-in and appletviewer. The impact of this potential incompatibility is therefore expected to be minimal.
14. J2SE 1.4 introduces significant changes to the Socket API, one of which is the addition of a new abstract method to the `SocketImpl` abstract class. Because of the new method, if a subclass of `SocketImpl` was created prior to J2SE 1.4, then compilation against the 1.4 javac will fail because there is no implementation provided for the new method. Note that *binary compatibility* is enforced, and existing class files for such a subclass will still work as expected.

    Few applications subclass `SocketImpl`, and the impact of this potential incompatibility is expected to be minimal. Developers who do subclass `SocketImpl` have two ways to deal with this change:

    - Use a class file compiled on J2SE 1.3.x (or earlier).
    - Provide an implementation for the new method.

Java HotSpot VM Options

This chapter provides information on the command-line options and environment variables that can affect the performance characteristics of the Java HotSpot™ Virtual Machine. Unless otherwise noted, all information in this document pertains to both the Java HotSpot Client VM and the Java HotSpot Server VM. Refer to http://java.sun.com/docs/hotspot for more information on garbage collection (GC), threading, and performance FAQs for the Java platform.

The chapter contains the following sections.

Categories of Java HotSpot VM Options

Standard options recognized by the Java HotSpot VM are described on the man page for the Java Application Launcher (the java utility) and in the online documentation at http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/java.html. This chapter describes non-standard options recognized by the Java HotSpot VM:

- Options that begin with `-X` are non-standard (not guaranteed to be supported on all VM implementations), and are subject to change without notice in subsequent releases of the Java 2 SDK.
- Because the `-XX` options have specific system requirements for correct operation and may require privileged access to system configuration parameters, they are not recommended for casual use. These options are also subject to change without notice.

Java HotSpot VM Equivalents of Exact VM Options

For additional information on issues related to the Java HotSpot VM, see the online documentation at http://java.sun.com/j2se/1.4/docs/guide/vm/index.html.

Prior to version 1.3.0, the production releases of the Java 2 SDK for the Solaris Operating Environment shipped with a virtual-machine implementation known as the Exact VM (EVM). Beginning with version 1.3.0, the Exact VM is replaced by the Java HotSpot VM.

Some options supported by the Exact VM have changed names or become obsolete in the Java HotSpot VM. These EVM options and their Java HotSpot VM equivalents in Java 2 SDK v 1.4.0 are given in the following table.

EVM Option	Description	Java HotSpot VM equivalent
`-Xt`	Instruction tracing	None (obsolete option)
`-Xtm`	Method tracing	None (obsolete option)
`-Xoss`	Maximum java stack size	None (Java HotSpot VM doesn't have separate native and Java programming language stacks)

`-Xverifyheap`	Verify heap integrity	`-XX:+VerifyBeforeGC`
		`-XX:+VerifyAfterGC`
		`-XX:+VerifyTLAB`
		`-XX:+VerifyBeforeScavenge`
		`-XX:+VerifyAfterScavenge` (all debug only)

`-Xmaxjitcodesize`	Maximum compiled code size	`-Xmaxjitcodesize` (used to be `-Xmaxjitcodesize=32m`, now `-Xmaxjitcodesize32m`)
`-Xgenconfig`	Configure the heap	(See "The `-Xgenconfig` Option" on page 35 below.)
`-Xoptimize`	Use optimizing JIT Compiler	`-server`
`-Xconcgc`	Use concurrent garbage collector (1.2.2_07+)	None (not yet in 1.3, 1.4)

The Java HotSpot VM currently recognizes the following `-X` options that were not supported by the Exact VM.

Option	Description
`-Xincgc`	Use Train GC
`-Xnoincgc`	Do not use Train Garbage Collection (default)
`-Xmaxf`<*Maximum*>	Max percentage o fhead free after GC to avoid shrinking (default 0.7)
`-Xminf`<*Minimum*>	Min percentage of heap free after GC to avoid expansion (default 0.4)
`-Xint`	Interpreter only
`-Xboundthreads`	Bind user-level threads (default in 1.4, not in 1.3)
`-Xmn`<*Size*>	Set the size of the young generation (1.4 only)

The `-Xgenconfig` Option

The young generation in the Java HotSpot VM consists of an eden and two equally sized semi-spaces, whereas the young generation in EVM consists of two equally size semi-spaces (with no eden). A command such as `-Xgenconfig:32m,64m,semispaces:128m,512m,markcompact` states that there are two 64mb semispaces each starting at 32mb and expandable to 64mb, and an old generation which starts at 128mb and can expand to 512mb. This creates a heap which is 640mb in size (maximum). In the Java HotSpot VM the equivalent command would be: `-Xms256m -Xmx640m -XX:NewSize=32m -XX:MaxNewSize=64m` As you can see, the Java HotSpot VM sets the total size of the heap with `-Xms/-Xmx` and then carves out the young generation from that space, whereas when using `-Xgenconfig` you must specify each generation's size.

Java HotSpot VM Equivalents to `_JIT_ARGS` Environment Variables

Most `_JIT_ARGS` environment variables are internal debugging options only and have no corresponding Java HotSpot VM options. Most simply turn off some form of optimization which may have caused instability when first introduced and could be used by the internal testing group to track down problems.

_JIT_ARGS Environment	Java HotSpot VM Option	Description
jit/jbe	`-client/-server`	jbe is the same as `-Xoptimize` in 1.2–based systems, jit is the default. Use `-server` to replace `-Xoptimize` (or jbe) in 1.2.
trace	`-XX:+PrintCompilation`	traces methods as they are compiled (debug only in 1.3, available in 1.4)
V8/V9	`-XX:+UseV8InstrsOnly`	Done automatically on both systems, force architecture using these flags (Sparc/debug only)

Java HotSpot VM Equivalents to `_JVM_ARGS` Environment Variables

_JVM_ARGS Environment	Java HotSpot VM Option	Description
bound_threads	`-Xboundthreads`	This option forces all threads to be created as bound threads (default in 1.4, not in 1.3).
fixed_size_young_gen	`-Xmn<size>`	Disable young generation resizing. To do this on the Java HotSpot VM, simply set the size of the young generation to a constant (in 1.4; in 1.3, use `-XX:MaxNewSize=<size>` `-XX:MaxNewSize=<size>`).
gc_stats	`-verbose:gc` and/or `-XX:+PrintGCDetails`	Turns on various forms of gc statistics gathering.
ims_concurrent	none	

_JVM_ARGS Environment	Java HotSpot VM Option	Description
inline_instrs	`-XX:MaxInlineSize=<size>`	Integer specifying maximum number of bytecode instructions in a method which gets inlined.
inline_print	`-XX:+PrintInlining`	Print message about inlined methods (debug only)
no_parallel_gc	none	
sync_final	none	
yield_interval	`-XX:DontYieldALotInterval=<ms>`	(debug only) Interval in milliseconds between yields
monitor_order	none	

Additional Java HotSpot VM arguments

Numbers can include 'k' or 'K' for kilobytes, 'm' or 'M' for megabytes, 'g' or 'G' for gigabytes, and 't' or 'T' for terabytes (for example, 32k is the same as 32768). Turn on a boolean flag with `-XX:+<option>` and off with `-XX:-<option>`.

Flag and Default	Description
`-XX:-AllowUserSignalHandlers`	Do not complain if the application installs signal handlers
`-XX:AltStackSize=16384`	Alternate signal stack size (in Kbytes)
`-XX:+MaxFDLimit`	Bump the number of file descriptors to max
`-XX:MaxHeapFreeRatio=70`	Maximum percentage of heap free after GC to avoid shrinking
`-XX:MinHeapFreeRatio=40`	Minimum percentage of heap free after GC to avoid expansion
`-XX:ReservedCodeCacheSize=32m`	Reserved code cache size (in bytes) — maximum code cache size
`-XX:+UseBoundThreads`	Bind user level threads to LWPs (default on in 1.4)
`-XX:+UseLWPSynchronization`	Use LWP-based instead of thread based synchronization (default on in 1.4)

Flag and Default	Description
`-XX:+UseThreadPriorities`	Use native thread priorities
`-XXMaxPermSize=64m`	Size of the permanent genration.
`-XX:-CITime`	Time spent in JIT Compiler (1.4 only)
`-XX:-PrintTenuringDistribution`	Print tenuring age information
`-XX:TargetSurvivorRatio=50`	Desired percentage of survivor space used after scavenge
`-XX:-DisableExplicitGC`	Disable calls to `System.gc()`. VM still performs garbage collection when necessary.
`-XX:-OverrideDefaultLibthread`	On Solaris 9, this option is not necessary. On Solaris 8, J2SE™ versions 1.3.1_02+ and 1.4+ require this option when using the alternate threads library. This option is not possible on pre-Solaris 8 operating environments.

For more information on threads libraries, see the threads document at http://java.sun.com/docs/hotspot/threads/threads.html.

Those flags differing per architecture/OS. "Flag and Default" has the default of Sparc/`-server`.

Flag and Default	Description
`-XX:CompileThreshold=10000`	Number of method invocations/branches before compiling [10,000 —server, 1,000 — client on Sparc, 1,500 on client x86]
`-XX:MaxNewSize=unlimited`	Maximum size of new generation (in bytes) [32m Sparc, 2.5m x86 for 1.3, no limit for 1.4 as NewRatio is now used to determine MaxNewSize]]
`-XX:NewRatio=2`	Ratio of new/old generation sizes [Sparc -server: 2; Sparc -client: 4 (1.3), 8 (1.3.1+); x86: 12]
`-XX:NewSize=2228224`	Default size of new generation (in bytes) [Sparc 2.125M, x86: 640k]
`-XX:SurvivorRatio=64`	Ratio of eden/survivor space size [Solaris: 64]
`-XX:ThreadStackSize=512`	Thread Stack Size (in Kbytes) (0 means use default stack size [Solaris Sparc 32–bit: 512, Solaris Sparc 64–bit: 1024, Solaris x86: 256])

Flag and Default	Description
`-XX:+UseTLAB (XX:+UseTLE` in J2SE 1.3)	Use thread-local object allocation [Sparc -server: true, all others: false]
`-XX:+UseISM`	See *Intimate Shared Memory*, online at http://java.sun.com/docs/hotspot/ism.html.

Assertion Facility

An *assertion* is understood to be a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed. For example, after unmarshalling all of the arguments from a data buffer, a programmer might assert that the number of bytes of data remaining in the buffer is zero. The system executes the assertion by evaluating the boolean expression and reporting an error if the expression evaluates to false. By verifying that the boolean expression is indeed true, the system corroborates the programmer's knowledge of the program thus increasing the possibility that the program is free of errors.

Assertion checking may be disabled for increased performance. Typically, assertion checking is enabled during program development and testing and disabled for deployment.

Because assertions may be disabled, programs must not assume that the boolean expressions contained in assertions will be evaluated. Thus these expressions should be free of side effects. That is, evaluating such an expression should not affect any state that is visible after the evaluation is complete. Although it is not illegal for a boolean expression contained in an assertion to have a side effect, it is generally inappropriate, as it could cause program behavior to vary depending on whether assertions are enabled or disabled.

Similarly, assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException or NullPointerException). An assertion failure will not throw an appropriate exception.

For more details, choose from the following links:

- "Compiling" on page 42
- "Syntax" on page 42
- "Semantics" on page 42

Compiling

In order for the `javac` bytecode compiler to accept code containing assertions, you must use the `-source 1.4` command-line option as in this example:

```
javac -source 1.4 MyClass.java
```

Syntax

A new keyword is added to the language. Use of the `assert` keyword is governed by one modified production and one new production in the grammar:

```
StatementWithoutTrailingSubstatement:
           <All current possibilities, as per JLS,
              Section 14.4> AssertStatement
AssertStatement:
   assert Expression1;
   assert Expression1 : Expression2;
```

In both forms of the assert statement, $Expression_1$ must be of type boolean or a compile-time error occurs.

Semantics

If assertions are disabled in a class, the assert statements contained in that class have no effect. If assertions are enabled, the first expression is evaluated. If it evaluates to false, an `AssertionError` is thrown. If the assertion contains a second expression (preceded by a colon), this expression is evaluated and passed to the constructor of the AssertionError; otherwise the parameterless constructor is used. (If the first expression evaluates to true, the second expression is not evaluated.)

If an exception is thrown while either expression is being evaluated, the assert statement completes abruptly, throwing this exception.

Enabling and Disabling Assertions

By default, assertions are disabled. Two command-line switches allow you to selectively enable or disable assertions.

The following switch enables assertions at various granularities:

```
java [ -enableassertions | -ea  ] [:<package name>"..." | :<class name> ]
```

With no arguments, the switch enables assertions by default. With one argument ending in "...", assertions are enabled in the specified package and any subpackages by default. If the argument is simply "...", assertions are enabled in the unnamed package in the current working directory. With one argument not ending in "...", assertions are enabled in the specified class.

The following switch disables assertions in similar fashion:

```
java [ -disableassertions | -da ] [:<package name>"..." | :<class name> ]
```

If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. For example, to run a program with assertions enabled only in package `com.wombat.fruitbat` (and any subpackages), the following command could be used:

```
java -ea:com.wombat.fruitbat... java -ea:com.wombat.fruitbat... <Main class>
```

To run a program with assertions enabled in package `com.wombat.fruitbat` but disabled in class `com.wombat.fruitbat.Brickbat`, the following command could be used:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat <class>
```

The above switches apply to all class loaders, and to system classes (which do not have a class loader). There is one exception to this rule: in their no-argument form, the switches do *not* apply to system classes. This makes it easy to turn on asserts in all classes except for system classes. A separate switch is provided to enable asserts in all system classes (i.e., to set the default assertion status for system classes to *true*).

```
 java [ -enablesystemassertions | -esa ]
```

For symmetry, a corresponding switch is provided to enable asserts in all system classes.

```
java [ -disablesystemassertions | -dsa ]
```

Enabling and Disabling Assertions Programmatically

Most programmers will not need to use the following methods. They are provided for those writing interpreters or other execution environments.

Setting the Default Assertion Status for a Class Loader

Each class loader maintains a *default assertion status*, a boolean value that determines whether assertions are, by default, enabled or disabled in new classes that are subsequently initialized by the class loader. A newly created class loader's default assertion status is false (disabled). It can be changed at any time by invoking a new method in class ClassLoader:

```
public void setDefaultAssertionStatus(boolean enabled)
```

If, at the time that a class is loaded, its class loader has been given specific instructions regarding the assertion status of the class's package name or its class name (via either of the two new methods in ClassLoader described below), those instructions take precedence over the class loader's default assertion status. Otherwise, the class's assertions are enabled or disabled as specified by its class loader's default assertion status.

Setting the Assertion Status for a Package and its Subpackages

The following method allows the invoker to set a per-package default assertion status. Note that a per-package default actually applies to a package and any subpackages.

```
public void setPackageAssertionStatus(String packageName, boolean
enabled);
```

Setting the Assertion Status for a Class and its Nested Classes

The following method is used to set assertion status on a per-class basis:

```
public void setClassAssertionStatus(string className, boolean
enabled);
```

Resetting to the Class Loader Default Assertion Status

The following method clears any assertion status settings associated with a class loader:

```
public void clearAssertStatus();
```

Usage Notes

The material contained in this section is not part of the assert specification, instead it is intended to provide information about the use of the facility. In the parlance of the standards community, the information in this section is non-normative.

You will find examples of appropriate and inappropriate use of the assert construct. The examples are not exhaustive and are meant to convey the intended usage of the construct.

Internal Invariants

In general, it is appropriate to frequently use short assertions indicating important assumptions concerning a program's behavior.

In the absence of an assertion facility, many programmers use comments in the following way:

```
if (i%3 == 0) {
        ...
    } else if (i%3 == 1) {
```

```
        ...
    } else { // (i%3 == 2)
        ...
    }
```

When your code contains a construct that asserts an invariant, you should change it to an assert. To change the above example (where an assert protects the `else` clause in a multiway if-statement), you might do the following:

```
if (i % 3 == 0) {
        ...
    } else if (i%3 == 1) {
        ...
    } else {
        assert i%3 == 2;
        ...
    }
```

Note, the assertion in the above example may fail if `i` is negative, as the `%` operator is not a true mod operator, but computes the remainder, which may be negative.

Control-Flow Invariants

Another good candidate for an assertion is a switch statement with no default case.

For example:

```
switch(suit) {
      case Suit.CLUBS:
        ...
        break;

      case Suit.DIAMONDS:
        ...
        break;

      case Suit.HEARTS:
        ...
        break;

      case Suit.SPADES:
        ...
    }
```

The programmer probably assumes that one of the four cases in the above switch statement will always be executed. To test this assumption, add the following default case:

```
default:
        assert false;
```

More generally, the following statement should be placed at any location the programmer assumes will not be reached.

```
assert false;
```

For example, suppose you have a method that looks like this:

```
void foo() {
       for (...) {
           if (...)
               return;
        }
        // Execution should never reach this point!!!
   }
```

Replace the final comment with:

```
assert false;
```

Note, use this technique with discretion. If a statement is unreachable as defined in (JLS 14.19), you will see a compile time error if you try to assert that it is unreached.

Preconditions, Postconditions, and Class Invariants

While the assert construct is not a full-blown design-by-contract facility, it can help support an informal design-by-contract style of programming.

Preconditions

By convention, preconditions on public methods are enforced by explicit checks inside methods resulting in particular, specified exceptions. For example:

```
/**
    * Sets the refresh rate.
    *
    * @param  rate refresh rate, in frames per second.
    * @throws IllegalArgumentException if rate <= 0 or
    *          rate > MAX_REFRESH_RATE.
    */
    public void setRefreshRate(int rate) {
        // Enforce specified precondition in public method
        if (rate <= 0 || rate > MAX_REFRESH_RATE)
            throw new IllegalArgumentException("Illegal rate: " + rate);

        setRefreshInterval(1000/rate);
    }
```

This convention is unaffected by the addition of the assert construct. An assert is inappropriate for such preconditions, as the enclosing method guarantees that it will enforce the argument checks, whether or not assertions are enabled. Further, the assert construct does not throw an exception of the specified type.

If, however, there is a precondition on a nonpublic method and the author of a class believes the precondition to hold no matter what a client does with the class, then an assertion is entirely appropriate. For example:

```
/**
  * Sets the refresh interval (must correspond to a legal frame rate).
  *
  * @param  interval refresh interval in milliseconds.
  */
  private void setRefreshInterval(int interval) {
      // Confirm adherence to precondition in nonpublic method
      assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE;

      ... // Set the refresh interval
  }
```

Note, the above assertion will fail if `MAX_REFRESH_RATE` is greater than `1000` and the user selects a refresh rate greater than `1000`. This would, in fact, indicate a bug in the library!

Postconditions

Postcondition checks are best implemented via assertions, whether or not they are specified in public methods. For example:

```
/**
    * Returns a BigInteger whose value is (this-1 mod m).
    *
    * @param  m the modulus.
    * @return this-1 mod m.
    * @throws ArithmeticException  m <= 0, or this BigInteger
    *         has no multiplicative inverse mod m (that is, this BigInteger
    *         is not relatively prime to m).
    */
   public BigInteger modInverse(BigInteger m) {
       if (m.signum <= 0)
         throw new ArithmeticException("Modulus not positive: " + m);
       if (!this.gcd(m).equals(ONE))
         throw new ArithmeticException(this + " not invertible mod " + m);

       ... // Do the computation

       assert this.multiply(result).mod(m).equals(ONE);
       return result;
   }
```

In practice, one would not check the second precondition `(this.gcd(m).equals(ONE))` prior to performing the computation, because it is wasteful. This precondition is checked as a side effect of performing the modular multiplicative inverse computation by standard algorithms.

Occasionally, it is necessary to save some data prior to performing a computation in order to check a postcondition after it is complete. This can be done with two assert statements and the help of a simple inner class designed to save the state of one or more variables so they can be checked (or rechecked) after the computation. For example, suppose you have a piece of code that looks like this:

```
void foo(int[] array) {
        // Manipulate array
        ...

        // At this point, array will contain exactly the ints that it did
        // prior to manipulation, in the same order.
    }
```

Here is how you could modify the above method to turn the textual assertion into a functional one:

```
 void foo(final int[] array) {

        class DataCopy {
          private int[] arrayCopy;

          DataCopy() { arrayCopy = (int[])(array.clone()); }

          boolean isConsistent() { return Arrays.equals(array, arrayCopy); }
        }

        DataCopy copy = null;

        // Always succeeds; has side effect of saving a copy of array
        assert (copy = new DataCopy()) != null;

        ... // Manipulate array

        assert copy.isConsistent();
     }
```

Note that this idiom easily generalizes to save more than one data field, and to test arbitrarily complex assertions concerning pre-computation and post-computation values.

The first assert statement (which is executed solely for its side-effect) could be replaced by the more expressive:

```
  copy = new DataCopy();
```

but this would copy the array whether or not asserts were enabled, violating the dictum that asserts should have no cost when disabled.

Class Invariants

As noted above, assertions are appropriate for checking internal invariants. The assertion mechanism itself does not enforce any particular style for doing so. It is sometimes convenient to combine many expressions that check required constraints into a single internal method that can then be invoked by assertions. For example, suppose one were to implement a balanced tree data structure of some sort. It might be appropriate to implement a private method that checked that the tree was indeed balanced as per the dictates of the data structure:

```
// Returns true if this tree is properly balanced
   private boolean balanced() {
       ...
   }
```

This method is a class invariant. It should always be true before and after any method completes. To check that this is indeed the case, each public method and constructor should contain the line:

```
assert balanced();
```

immediately prior to each return. It is generally overkill to place similar checks at the head of each public method unless the data structure is implemented by native methods. In this case, it is possible that a memory corruption bug could corrupt a "native peer" data structure in between method invocations. A failure of the assertion at the head of such a method would indicate that such memory corruption had occurred. Similarly, it may be advisable to include class invariant checks at the head of methods in classes whose state is modifiable by other classes. (Better yet, design classes so that their state is not directly visible by other classes!)

Removing all Trace of Assertions from Class Files

Programmers developing for resource-constrained devices may wish to strip assertions out of class files entirely. While this makes it impossible to enable assertions in the field, it also reduces class file size, possibly leading to improved class loading performance. In the absence of a high quality JIT, it could lead to decreased footprint and improved runtime performance.

The assertion facility offers no direct support for stripping assertions out of class files. However, the assert statement may be used in conjunction with the "conditional compilation" idiom described in JLS 14.19:

```
static final boolean asserts = ... ; // false to eliminate asserts

    if (asserts)
        assert <expr> ;
```

If asserts are used in this fashion, the compiler is free to eliminate all traces of these asserts from the class files that it generates. It is recommended that this be done where appropriate to support generation of code for resource-constrained devices.

Requiring that Assertions are Enabled

Programmers of certain critical systems might wish to ensure that assertions are not disabled in the field. Here is an idiom that prevents a class from being loaded if assertions have been disabled for that class:

```
static {
      boolean assertsEnabled = false;
      assert assertsEnabled = true; // Intentional side effect!!!
```

```
        if (!assertsEnabled)
            throw new RuntimeException("Asserts must be enabled!!!");
    }
```

Source Compatibility

The addition of the `assert` keyword to the Java Programming Language causes existing programs that use assert as an identifier to become invalid. The addition of this keyword does not, however, cause any problems with the use of preexisting binaries (.class files). In order to ease the transition from a world where assert is a legal identifier to one where it isn't, the compiler supports two modes of operation in this release.

- In the normal mode of operation the compiler accepts programs conforming to the specification for the previous release (1.3). Assertions are not permitted, and the compiler generates a warning if the assert keyword is used as an identifier or label.
- In an alternate mode of operation, the compiler accepts programs conforming to the specification for release 1.4. Assertions are permitted, and the compiler generates an error message if the assert keyword is used as an identifier or label.

To enable assertions, use the following command line switch.

```
-source 1.4
```

In the absence of this flag, the behavior defaults to "1.3" for maximal source compatibility. Support for 1.3 source compatibility is likely to be phased out over time.

Design FAQ

Following is a collection of frequently asked questions concerning the design of the assertion facility.

- General Questions
- Compatibility
- Syntax and Semantics
- The AssertionError Class
- Enabling and Disabling Assertions

Design FAQ - General Questions

1. *Why provide an assertion facility at all, given that one can program assertions atop the Java programming language with no special support?*

   Although ad hoc implementations are possible, they are of necessity either ugly (requiring an if statement for each assertion) or inefficient (evaluating the condition even if assertions are disabled). Further, each ad hoc implementation has its own means of enabling and disabling assertions, which lessens the utility of these implementations, especially for debugging in the field. As a result of these shortcomings, assertions have never become a part of the Java culture. Adding assertion support to the platform stands a good chance of rectifying this situation.

2. *Why does this facility justify a language change, as opposed to a library solution?*

   We recognize that a language change is a serious effort, not to be undertaken lightly. The library approach was considered. It was, however, deemed essential that the runtime cost of assertions be negligible if they are disabled. In order to achieve this with a library, the programmer is forced to hard-code each assertion as an if statement. Many programmers would not do this. Either they would omit the if statement and performance would suffer, or they would ignore the facility entirely. Note also that assertions were contained in James Gosling's original specification for Java. Assertions were removed from the Oak specification because time constraints prevented a satisfactory design and implementation.

3. *Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?*

   We considered providing such a facility, but were unable to convince ourselves that it is possible to graft it onto the Java programming language without massive changes to the Java platform libraries, and massive inconsistencies between old and new libraries. Further, we were not convinced that such a facility would preserve the simplicity that is Java's hallmark. On balance, we came to the conclusion that a simple boolean assertion facility was a fairly straight-forward solution and far less risky. It's worth noting that adding a boolean assertion facility to the language doesn't preclude adding a full-fledged design-by-contract facility at some time in the future.

   The simple assertion facility does enable a limited form of design-by-contract style programming. The assert statement is appropriate for postcondition and class invariant checking. Precondition checking should still be performed by checks inside methods that result in particular, documented exceptions, such as IllegalArgumentException and IllegalStateException.

4. *In addition to boolean assertions, why not provide an assert-like construct to suppress the execution of an entire block of code if assertions are disabled?*

   Providing such a construct would encourage programmers to put complex assertions inline, when they are better relegated to separate methods.

Design FAQ - Compatibility

1. *Won't the new keyword cause compatibility problems with existing programs that use assert as an identifier?*

   Yes, for source files. (Binaries for classes that use assert as an identifier will continue to work fine.) To ease the transition, we describe a strategy whereby developers can continue using assert as an identifier during a transitional period. See "Source Compatibility" on page 51.

Design FAQ - Syntax and Semantics

1. *Why allow primitive types in Expression$_2$?*

   There is no compelling reason to restrict the type of this expression. Allowing arbitrary types provides convenience for developers who for example want to associate a unique integer code with each assertion. Further, it makes this expression seem like `System.out.print(...)`, which is desirable.

Design FAQ - The AssertionError Class

1. *When an AssertionError is generated by an assert statement in which Expression$_2$ is absent, why isn't the program text of the asserted condition used as the detail message (e.g., "height < maxHeight")?*

   While doing so might improve out-of-the-box usefulness of assertions in some cases, the benefit doesn't justify the cost of adding all those string constants to .class files and runtime images.

2. *Why doesn't an AssertionError allow access to the object that generated it? Similarly, why not pass an arbitrary object from the assertion to the AssertionError in place of a detail message?*

   Access to these objects would encourage programmers to attempt to recover from assertion failures, which defeats the purpose of the facility.

3. *Why not provide context accessors (like getFile, getLine, getMethod) on AssertionError?*

   This facility is best provided on Throwable, so it may be used for all throwables, and not just assertion errors. We intend to enhance Throwable to provide this functionality in the same release in which the assertion facility first appears.

4. *Why is AssertionError a subclass of Error rather than RuntimeException?*

   This issue was controversial. The expert group discussed it at length, and came to the conclusion that Error was more appropriate to discourage programmers from attempting to recover from assertion failures. It is, in general, difficult or impossible to localize the source of an assertion failure. Such a failure indicates that the program is operating "outside of known space," and attempts to continue execution are likely to be harmful. Further, convention dictates that methods specify most

runtime exceptions they may throw (via "@throws" doc comments). It makes little sense to include in a method's specification the circumstances under which it may generate an assertion failure. Such information may be regarded as an implementation detail, which can change from implementation to implementation and release to release.

Design FAQ - Enabling and Disabling Assertions

1. *Why not provide a compiler flag to completely eliminate assertions from object files?*

   It is a firm requirement that it be possible to enable assertions in the field, for enhanced serviceability. It would have been possible to also permit developers to eliminate assertions from object files at compile time. However, since assertions can contain side effects (though they should not), such a flag could alter the behavior of a program in significant ways. It is viewed as good thing that there is only one semantics associated with each valid Java program. Also, we want to encourage users to leave asserts in object files so they can be enabled in the field. Finally, the standard Java "conditional compilation idiom" described in JLS 14.19 can be used to achieve this effect for developers who really want it.

2. *Why does setPackageAssertionStatus have package-tree semantics instead of the more obvious package semantics?*

   Hierarchical control is useful, as programmers really do use package hierarchies to organize their code. For example, package-tree semantics allow assertions to be enabled or disabled in all of Swing at one time.

3. *Why does setClassAssertionStatus return a boolean instead of throwing an exception if it is invoked when it's too late to set the assertion status (i.e., the named class has already been loaded)?*

   No action (other than perhaps a warning message) is necessary or desirable if it's too late to set the assertion status. An exception seems unduly heavyweight.

4. *Why not overload a single method to take the place of setDefaultAssertionStatus and setAssertionStatus?*

   Clarity in method naming is for the greater good.

5. *Why is there no RuntimePermission to prevent applets from enabling/disabling assertions?*

   While applets have no reason to call any of the ClassLoader methods for modifying assertion status, allowing them to do so seems harmless. At worst, an applet can mount a weak denial-of-service attack by turning on asserts in classes that have yet to be loaded. Moreover, applets can only affect the assert status of classes that are to be loaded by class loaders that the applets can access. There already exists a RuntimePermission to prevent untrusted code from gaining access to class loaders (getClassLoader).

6. *Why not provide a construct to query the assert status of the containing class?*

   Such a construct would encourage people to inline complex assertion code, which we view as a bad thing:

```
if (assertsEnabled()) {
    ...
}
```

Further, it is straightforward to query the assert status atop the current API, if you feel you must:

```
boolean assertsEnabled = false;
        assert assertsEnabled = true;  // Intentional side-effect!!!
        // Now assertsEnabled is set to the correct value
```