Contents

Preface

1. New Features and Enhancements 11
   High-Performance Memory System 12
      Exact Garbage Collection 12
   Multithreading 13
      Reduced Synchronization Overhead 13
      Thread-Specific Data Access 14
   Tools 14
      Heap Inspection Tool 14
      Debugging Utility 14
   Improved JIT Compiler Optimizations 14
      Inlining 14
      Mixed-Mode Execution 15
   Scalability Improvements 15
   Text-Rendering Performance Improvements 15
   Poller Class Demonstration 16

2. Java 2 SDK for Solaris Backward Compatibility 17
   Binary Compatibility 17
   Source Compatibility 18
3. Java Native Interface (JNI)  39
   Making the Transition From NMI to JNI  39
      Porting  39
      javah  39
   General JNI Issues  40
      Compiler Restrictions  40
      Linking Native Solaris Applications  40
      Fast JNI Array Access  41
      Locating Shared Libraries  41
      Signal Processing State  41
4. Command–Line Differences Between the Java 2 SDK and JDK 1.1  43
   VM-Specific (Non-Standard) Options  43
      Option Compatibility  44
      oldjava Utility  45
5. Using SIGQUIT for Debugging  47
   A. Memory Allocation and Constraints  51
      VM Size  51
   B. Interpreting −verbosegc Output  53
      Troubleshooting Garbage Collection  53
      Generational Heap Sizes  53
   C. Poller Class Usage  55
      Poller Class  55

Java 2 SDK for Solaris Developer’s Guide ♦ February 2000
Basics of Poller Class Usage  55

D. Running with Both Java 2 SDK and JDK 1.1  57

Index  59
Preface

Java 2 SDK for Solaris Developer’s Guide is an introduction to and overview of the new features and enhancements in the production version of the Java™ 2 SDK on Solaris™ 8 operating environment.

Who Should Use This Book

This document is intended for application developers who use the Java Development Kit (JDK™). Java 2 Software Development Kit (SDK) for Solaris is optimized to deliver superior performance and scalability to server-side Java technology applications in the enterprise environment. It includes an enhanced Java Virtual Machine (VM) with improved Just-In-Time (JIT) compiler optimizations. The Java 2 SDK for Solaris has been developed to provide substantially increased performance for large server-side applications running large numbers of Java threads on multiprocessor systems. Server-side applications are often characterized as:

- Long lived
- Highly threaded (multiprocessor-capable)
- Network intensive
- Memory intensive

The faster Java VM also provides improved performance for client-side applications.
How This Book Is Organized

Chapter 1 describes the new features and enhancements.

Chapter 2 lists and discusses cases of incompatibilities between the Java 2 SDK for Solaris and JDK 1.1 programs.

Chapter 3 discusses issues of interest to developers making the transition from using the Native Method Interface (NMI) to the Java Native Interface (JNI).

Chapter 4 details the options supported on the Java 2 SDK for Solaris reference and production platforms.

Chapter 5 describes how to use a new debugging process.

Appendix A discusses the way in which the Java 2 SDK for Solaris allocates memory.

Appendix B describes the use of \texttt{−verbosegc} in troubleshooting garbage collection.

Appendix C is a discussion of use of the new \texttt{Poller} class.

Appendix D shows you how to change the default JDK on your system from JDK 1.1 to Java 2 SDK for Solaris.

Related Books

These documents also have information about this release:

- \textit{Java 2 SDK v.1.2.1_04 for Solaris Release Notes}
- \textit{Java 2 SDK v.1.2.1_04 Troubleshooting}

You can download these documents at:

http://www.sun.com/solaris/java/

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at http://www1.fatbrain.com/documentation/sun.
Accessing Sun Documentation Online

The docs.sun.com™ 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.

What Typographic Conventions Mean

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

<table>
<thead>
<tr>
<th>TABLE P–1</th>
<th>Typographic Conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Typeface or Symbol</strong></td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with on-screen computer output</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Command-line placeholder: replace with a real name or value</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words, or terms, or words to be emphasized.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell prompt</td>
<td>machine_name$</td>
</tr>
<tr>
<td>C shell superuser prompt</td>
<td>machine_name#</td>
</tr>
<tr>
<td>Bourne shell and Korn shell prompt</td>
<td>$</td>
</tr>
<tr>
<td>Bourne shell and Korn shell superuser prompt</td>
<td>#</td>
</tr>
</tbody>
</table>
New Features and Enhancements

This chapter describes the new Java 2 SDK for Solaris features and enhancements. If you are developing your application with this release, the advanced feature set of the Java 2 platform provides:

- An improved security model
- Greater interoperability (for database and CORBA-based applications)
- A complete development platform that now includes APIs for collections, internationalization, and GUI development

Additionally, Java 2 SDK for Solaris has significantly improved scalability and performance as the result of an enhanced Java Virtual Machine (VM) with:

- Improved Just-In-Time (JIT) compile optimizations
- Fast-thread synchronization
- State-of-the-art memory system

To help deploy your application with the Java 2 SDK for Solaris production release for the Solaris 8 operating environment, this release provides:

- A secure environment for running your Java application
- Industry leading scalability and performance for large-scale, server-side applications
- Mission critical support as part of your existing Solaris operating environment support contract
High-Performance Memory System

Java 2 SDK for Solaris includes a highly optimized memory system, which makes memory allocation and garbage collection more efficient. This memory system uses direct pointers and is:

- Non-conservative
- Fully compacting
- Generational

These features increase program performance and reduce disruptive garbage collection pauses in interactive programs.

The following enhancements, part of the high-performance memory system, improve system performance.

Exact Garbage Collection

The Java 2 SDK for Solaris features exact garbage collection. Exact garbage collection increases performance by enabling a wider range of garbage collection techniques than conservative systems allow.

The Java 2 SDK for Solaris uses an efficient generational garbage collector. This efficiency produces significantly better performance for many applications. Generational garbage collection examines a subset of objects allocated for reclamation. The garbage collector selects a subset based on the age of the objects where younger generations of objects are examined more often than older ones. This subsetting works because in Java programs, recently created objects are more likely candidates for collection than older objects. This technique provides the added benefit of shorter pauses for garbage collection, because young-generation collections are usually much shorter than full collections.

To achieve this added performance, the entire system must support exact collection, including user-written native code. Fortunately, the Java Native Interface (JNI) allows an implementation that is compatible with exact collection. Because Java 2 SDK for Solaris provides such an implementation, native code written using the JNI works seamlessly with the Java 2 SDK for Solaris VM.

Direct Pointers

Java 2 SDK for Solaris exact garbage collection uses direct pointers for objects, rather than handles. Using direct pointers decrease memory consumption, speed allocation, and increase system performance by eliminating one level of indirection in accessing objects.
Double-Word Alignment (longs and doubles) to 8-Byte Boundaries in the Heap

Double-word values are now aligned to 8-byte boundaries in the heap. This alignment improves the performance of both native code and JIT-compiled Java code while ensuring correctness of volatile double-word values on SPARC systems. However, if your application allocates and retains many small objects, you might need to increase your heap size(s) slightly, as these objects are allocated in multiples of 8 bytes, increasing memory usage.

Multithreading

Reduced Synchronization Overhead

The following features result in enhanced performance as a result of reduced synchronization overhead for multithreaded applications.

Fast-Sync Monitor Locking/Fast Thread Synchronization

The Java 2 SDK for Solaris uses a new, internal, fast-locking algorithm for more efficient method synchronization. The VM has significantly improved implementations of the Java platform’s synchronization primitives. These implementations make concurrent programs more efficient and decrease the impact of the synchronization primitives on single-threaded application performance.

Lock Contention Minimization

The Java 2 SDK for Solaris minimizes lock contention in the VM by using thread-local data structures and finer-grained locking.

The VM core-locking architecture supports better granularity of VM locks. This fine-grained locking minimizes the number of contended paths of thread execution. Fewer contended paths means that one thread’s execution is less likely to impede other threads. This results in better multiprocessor (MP) scaling capability. Fine-grained locking also enables the VM to conduct a larger number of concurrent operations.
Thread-Specific Data Access

The Java 2 SDK for Solaris uses a new internal cache mechanism in order to provide faster access to thread-specific data. This is a general performance enhancement used by the VM and each platform-specific JIT to access execution environment data.

Tools

Heap Inspection Tool

You can access this diagnostic tool for interactively killed programs from the SIGQUIT handler menu. It will find memory leaks in your programs. A memory leak occurs when a program inadvertently retains objects, preventing the garbage collector from reclaiming the memory. Heap inspection presents a per-class breakdown of the objects in the heap, sorted by total amount of memory consumed. You can then examine reference chains to selected objects to see what is keeping them alive.

Debugging Utility

Java 2 SDK for Solaris features a new debugging process. It involves sending a SIGQUIT signal to a Java process running in the foreground. This signal allows you to perform debugging tasks, such as thread and monitor state dumps and deadlock detection, on that process. See Chapter 5 for more information.

Improved JIT Compiler Optimizations

The improved JIT compiler optimizations instructs the Java Virtual Machine to favor ultimate execution speed over the startup time of Java applications. The JIT compiler performs new optimizations for both SPARC and Intel (IA) platforms, including inlining of virtual and non-virtual methods, CSE within extended basic blocks, loop analysis to eliminate array bounds checking, and fast type checks.

Inlining

Java 2 SDK for Solaris provides inlining, which eliminates the need to inline manually (and risk destroying program modularity). Automatic inlining is restricted to relatively small, non-synchronous methods without flow control.
Mixed-Mode Execution

Java 2 SDK for Solaris includes a new mixed-mode execution feature that can significantly improve performance. Mixed-mode execution occurs when the VM compiles only the performance-critical methods and interprets the rest. This feature reduces compilation overhead, enables programs to start more quickly, and lets the VM spend more time compiling and optimizing where it matters most, enabling better performance.

Mixed-mode execution is the default mode. In mixed mode, the VM divides the methods into two types:

- Methods that might contain loops
- Methods that do not contain loops

For each type, the JIT compiler determines when compilation occurs. Methods that potentially contain loops are compiled on their first execution. Methods that do not contain loops are compiled on the 15th invocation.

Scalability Improvements

The Java 2 SDK for Solaris improves handling of concurrency primitives and threads, increases the performance of multithreaded (MT) programs, and significantly reduces garbage-collection pause times for programs that use large numbers of threads.

Text–Rendering Performance Improvements

Several graphics optimizations have significantly improved text rendering performance for Java 2 SDK on Solaris platforms without direct graphics Access (DGA) support. These platforms include Ultra™ 5, Ultra 10, Solaris running on Intel (IA), and all remote display systems.
Poller Class Demonstration

The new Poller class provides Java applications with the ability to efficiently access the functionality of the C `poll(2)` routine. The demonstration code with the Java 2 SDK for Solaris is provided with a sample usage server in

```
$(JAVA_HOME)/demo/jni/Poller
```

The demonstration code shows a means of determining the I/O status of multiple I/O objects with one call.

The JNI C code supporting the Poller class is optimized to take full advantage of Solaris 7 operating environment `poll(2)` kernel caching. It can be compiled to take advantage of `/dev/poll`, a faster kernel–polling mechanism available with Solaris 7 system software update 2 and subsequent versions.

See Appendix C for more detailed information about the Poller class.
Java 2 SDK for Solaris Backward Compatibility

This document contains information on the following topics:

- “Binary Compatibility” on page 17
- “Source Compatibility” on page 18
- “Incompatibilities in Java 2 SDK for Solaris” on page 19
  - “Language Incompatibilities” on page 19
  - “Runtime Incompatibilities” on page 27
  - “API Incompatibilities” on page 31
  - “Tool Incompatibilities” on page 37
  - “Serialization Incompatibilities” on page 38

References to the Java Language Specification are denoted by “JLS”.

Binary Compatibility

Java 2 SDK for Solaris is upwards binary-compatible with JDK versions 1.0 and 1.1 except for the incompatibilities listed below. This means that, except for the noted incompatibilities, class files built with JDK version 1.0 or 1.1 compilers run correctly with Java 2 SDK for Solaris.

As long as the javac compiler’s –target 1.2 command-line option is not used, downward binary compatibility is generally supported, though not guaranteed. That is, class files built with a Java 2 SDK for Solaris compiler, but relying only on APIs defined in JDK versions 1.0 or 1.1 of the Java platform, generally run on JDK version
1.0 and JDK version 1.1 of the Java Virtual Machine, but this downwards compatibility has not been extensively tested and cannot be guaranteed. Of course, if the class files depend on any new Java 2 SDK for Solaris APIs, those files do not work on earlier platforms.

In general, the policy is that:

- Maintenance releases (for example JDK versions 1.1.1 and 1.1.2) within a family (JDK 1.1-based releases) maintain both upward and downward binary-compatibility with each other.

- Functionality releases (for example JDK versions 1.1 and 1.2) within a family (JDK 1.1-based releases) maintain upward but not necessarily downward binary-compatibility with each other. Some early Java bytecode obfuscators produced class files that violated the class file format as given in the *Java Virtual Machine Specification, Second Edition*. Such improperly formatted class files do not run on Java 2 SDK for Solaris VM, though some of them might have run on earlier versions of the VM. To remedy this problem, regenerate the class files with a newer obfuscator that produces properly formatted class files.

### Source Compatibility

Java 2 SDK for Solaris is upwards source-compatible with JDK versions 1.0 and 1.1, 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 JDK 1.0 and 1.1 can be compiled and run on Java 2 SDK for Solaris.

Downward source compatibility is not supported. If source files use new language features or Java 2 SDK for Solaris APIs, they are not usable with an earlier version of the Java platform.

In general, the policy is that:

- 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 supported only for backward compatibility. The compiler generates a warning message whenever it encounters deprecated APIs (unless you use the `−nowarn` command–line option). You should eliminate the use of deprecated methods and classes because it has not been determined when or if the deprecated elements will be removed from the APIs.
Incompatibilities in Java 2 SDK for Solaris

The following is intended to be a complete list of all cases in which a program that works under JDK 1.0 or 1.1 of the Java platform would fail to work under Java 2 SDK for Solaris. Most of these incompatibilities involve unusual circumstances and are not expected to affect most existing programs.

Language Incompatibilities

Compilers in JDK 1.0 and 1.1 compiled some types of illegal code without producing warnings or error messages. The Java 2 SDK for Solaris compiler is more stringent in ensuring that code conforms to the JLS. The following list summarizes the types of illegal code that compiled with JDK 1.0 or 1.1 compilers but do not compile with Java 2 SDK for Solaris.

1. Previous compilers passed some initializations of int types to long types. These are flagged as errors in Java 2 SDK for Solaris. In the following program,

   ```java
   public class foo {
   int i = 3000000000;
   int j = 6000000000;
   }
   ```

   both the initialization to i and j are in error. Previous compilers would only report the incorrect initialization of i. The initialization of j would have overflowed silently [bug 4035346].

2. Previous compilers allowed the implicit assignment conversion from char to byte and short for character literals that fit into 8 bits. The compiler does not allow such implicit assignment conversion in Java 2 SDK for Solaris. For example,

   ```java
   byte b = 'b';
   ```

   will not pass the compiler. Use an explicit cast for such conversions [bug 4030496].

   ```java
   byte b = (byte)'b';
   ```

3. The Java 2 SDK for Solaris compiler will not pass 0xL (not a legal hex literal). Previous compilers parsed it as zero [bug 4049982].

4. The Java 2 SDK for Solaris compiler will not pass "" (and therefore ‘\u0027’) (not a legal char literal). Use ‘\’ instead [bug 1265387].
5. The Java 2 SDK for Solaris compiler will not pass "\u000D" (not legal in string and char literals). The CR and LF characters (\u000A and \u000D) terminate lines, even in comments [bug 4086919]. The following code is not:

```java
//This comment about \u000D is not legal; it is really two lines.
```

Use \r instead [bug 4063147].

6. The Java 2 SDK for Solaris compiler will not pass the type `void[]` (not legal) [bug 4034979].

7. Do not combine the abstract method modifier with private, final, native, or synchronized modifiers; it is not legal [bug 1266571].

8. Previous compilers would pass double-assignment of final variables in some circumstances. For example, the following two samples would pass JDK 1.1-based compilers (even though both involve double-assignment of a final variable). The Java 2 SDK for Solaris compiler will not pass such assignments [bugs 4066275 and 4056774].

```java
public class Example1 {
    public static void main(String[] argv) {
        int k=0;
        for (final int a;;) {
            k++;
            a=k;
            System.out.println("a="+a);
            if (k>3)
                return;
        }
    }
}
```

```java
public class Example2 {
    final int k;
    Example2() {
        k = 1;
    }
    Example2(Object whatever) {
        this();
        k = 2;
    }
    static public void main(String[] args) {
        Example2 t = new Example2(null);
        System.out.println("k is ++ t.k");
    }
}
```

9. You cannot reference a non-static field with an apparently static field expression of the form Classname.fieldname. Prior to Java 2 SDK for Solaris, javac silently tolerated such expressions as if they had been written this.fieldname [bug 4087127].
10. Section 5.5 of the JLS specifies that a cast between two interface types is a compile-time error if the interfaces contain methods with the same signature but different return types. The compiler did not generate this compile-time error prior to Java 2 SDK for Solaris [bug 4028359]. For example, the following program now generates a compile-time error:

```java
interface Noisy {
  int method();
}
interface Quiet {
  void method();
}
public class InterfaceCast {
  public static void main(String[] args) {
    Noisy one = null;
    Quiet two = (Quiet) one;
  }
}
```

11. Java 2 SDK for Solaris does not accept an assignment expression as the third subexpression of a conditional statement. For example, the Java 2 SDK for Solaris compiler throws an error in the following statement.

```java
myVal = condition ? x = 7 : x = 3;
```

If this problem occurs in existing code, place the offending assignment expression in parentheses to compile it as in previous versions of the JDK:

```java
myVal = condition ? x = 7 : (x = 3);
```

12. In previous releases of javac, the compiler incorrectly omitted the initialization if a field was initialized to its default value [bug 1227855]. This behavior can lead to different semantics for programs like the following:

```java
abstract class Parent {
  Parent() {
    setI(100);
  }
  abstract public void setI(int value);
}
public class InitTest extends Parent {
  public int i = 0;
  public void setI(int value) {
    i = value;
  }
}
public static void main(String[] args) {
(continued)
```
This program produces incorrect output (100) when compiled with earlier versions of javac. The Java 2 SDK for Solaris javac produces the correct output (0).

Single class examples can be formulated as well:

```java
public class InitTest2 {
    public int j = method();
    public int i = 0;
    public int method() {
        i = 100;
        return 200;
    }
}
```

```java
public static void main(String[] args) {
    InitTest2 test = new InitTest2();
    System.out.println(test.i);
}
```

Before, the output was 100. Now it is 0. The same phenomenon can occur in programs using reference types and null.

13. Concerning accessibility in qualified names, JLS 6.6.1 states the following:

“A member (field or method) of a reference (class, interface, or array) type or a constructor of a class type is accessible only if the type is accessible and the member or constructor is declared to permit access.”

Prior to Java 2 SDK for Solaris, the compiler did not enforce this rule correctly. It granted access if the member or constructor was declared to permit access without regard to the accessibility of the type to which it belonged as in the following illegal program.

```java
import pack1.P1;
public class CMain {
    public static void main(String[] args) {
        P1 p1 = new P1();
        // The accesses to field ‘i’ below are illegal, as the type of ‘p2’, the class type ‘pack1.P2’, is not accessible.
        p1.p2.i = 3;
        System.out.println(p1.p2.i);
    }
}
```
package pack1;
public class P1 {
    public P2 p2 = new P2();
} // Note the absence of an access modifier, making
// 'P2' accessible only from within package 'pack1'.
public class P2 {
    public int i = 0;
}

With the introduction of inner classes in Java 2 SDK for Solaris, a member of a
class or interface might be another class or interface, as well as a field or method.
Java 2 SDK for Solaris enforces the accessibility rules described to inner classes as
well.

14. Prior to Java 2 SDK for Solaris, the compiler failed to detect that certain classes
were abstract. This happened when a subclass declared a method with the
same name as an abstract, package-private method defined in a superclass
from a different package. No overriding occurs even though the methods have the
same name. As an example, compiling these files:

package one;
public abstract class Parent {
    abstract void method();
}

package two;
public class Child extends one.Parent {
    void method() {}
}

yields the error message:

two/Child.java:3: class two.Child is not able to provide an
implementation for the method void method() declared in class
one.Parent because it is private to another package. class
two.Child must be declared abstract.
   public class Child extends one.Parent {
^
16. The Java 2 SDK for Solaris compiler properly detects a labeled declaration. These
declarations are disallowed by the JLS. This is a fix for bug 4039843.

17. The Java 2 SDK for Solaris compiler recognizes a new keyword, strictfp, so
programs can no longer use strictfp as an identifier. The Java 2 SDK for Solaris
compiler uses the new keyword to set a modifier bit in the method data
structures. The platform specification previous to Java 2 SDK for Solaris required
that this bit be zeroed. Code written using the new keyword executes in strict
floating-point mode (the default defined for the Java platform). Code that does
not use the new keyword executes in default floating-point mode. This enables
implementations to make better use of some processors to deliver higher
performance.

Some numeric code not marked with the strictfp keyword might behave
differently in Java 2 SDK for Solaris than in previous versions. Such code might
also behave differently depending on the implementation of the Java platform.
Overflow or underflow can occur in different circumstances and create slightly
different results. These differences are not expected to have an impact on most
numeric code. However, code that is vulnerable to floating-point behavior might
be affected.

18. JDK 1.1 extended the syntax of expressions to allow a class name to qualify a
reference to a current instance using the keyword this (as in the following
example).

PrimaryNoNewArray: ...
ClassName . this

The value of such an expression is a reference to the current instance of the
enclosing class (ClassName), which must exist.

Prior to Java 2 SDK for Solaris, the javac compiler treated such expressions
incorrectly. It produced a reference to the current instance of the innermost
enclosing class that was a subtype of the type named by ClassName. The
compiler now implements such expressions properly.

19. JDK 1.1, extended the syntax of expressions allowing a class name to qualify a
reference to a current instance using the keyword this (as in the following
example).
This syntax allows access to members using constructions such as:

```java
ClassName.this.fieldname
ClassName.this.methodname(...)
```

The presence of inner classes made this extension necessary because more than one current instance might occur at a given point in the program. The inner classes specification neglected to include a similar extension for the `super` keyword (as in the following example).

```java
FieldAccess: ...
  ClassName.super.Identifier
MethodInvocation:...
  ClassName.super.Identifier(ArgumentList_opt)
```

Java 2 SDK for Solaris implements these constructs in anticipation of their inclusion in the forthcoming second edition of the JLS.

In each case, the current instance is the current instance of the enclosing class (`ClassName`), which must exist.

The following example shows how the need for the qualified super notation might arise in practice.

```java
class C {
  void f() { ... }
}
class D extends C {
  void f() {
    // overrides f() to run it in a new thread
    new Thread(new Runnable() {
      public void run() {
        D.super.f();
      }
    }).start();
  }
}
```

*Implementation Note:* If you need to use an access method, it must reside in the class named by `ClassName`, not in its superclass. The superclass does not have to be defined in the same compilation unit, and can have been compiled previously.

20. JDK 1.1 extended the syntax of expressions allowing a constructor invocation using the keyword `super` to be qualified with a reference to an outer instance (as in the following example).
The corresponding case for constructor invocations using `this` was inadvertently omitted:

```java
ExplicitConstructorInvocation: ...
Primary.this(ArgumentList_opt);
```

Java 2 SDK for Solaris implements the construct in anticipation of its inclusion in the forthcoming second edition of the JLS.

21. The Inner Classes specification does not permit an inner class to declare a member interface. This rule is enforced in Java 2 SDK for Solaris, but such declarations are silently tolerated in JDK 1.1-based releases. For example, the following program is incorrectly accepted by `javac` in JDK 1.1-based releases, but is rejected in Java 2 SDK for Solaris.

```java
class InnerInterface {
    class Inner {
        interface A {
        }
    }
}
```

A static member class is not an inner class; it is a top-level class (as that term is defined by the Inner Classes specification). The `javac` in both JDK 1.1-based releases and Java 2 SDK for Solaris accepts the following program as correct.

```java
class NestedInterface {
    static class Inner {
        interface A {
        }
    }
}
```

A local class is never a top-level class, so the following example is also illegal; both `javac` JDK 1.1-based releases and Java 2 SDK for Solaris report a syntax error.
22. The Java 2 SDK for Solaris compiler strongly enforces the restriction that a package cannot contain a type and a subpackage of the same name. This change makes it illegal to compile a class whose fully qualified name was the same as the fully qualified name of a package on the classpath. This change also makes it illegal to compile a class whose package (or some proper prefix) would have the same name as an existing class. This is a fix for bug 4101529. The following are some examples of classes that now fail to compile:

```java
\Example 1
package java.lang.String;
class Illegal {
}
\Example 2
package java;
class util {
}
```

Runtime Incompatibilities

1. In JDK 1.0 and 1.1, the runtime systems finalize objects (invoke their finalize methods) somewhat aggressively. Sometimes all eligible but unfinalized objects would be finalized at the end of nearly every garbage-collection cycle. Code written for such systems can unintentionally depend upon this prompt garbage collection-invoked finalization for correct operation, which can lead to complicated bugs and deadlocks.

In Java 2 SDK for Solaris, finalization is not performed directly by the garbage collector. Instead, objects are finalized only by a high-priority thread. So, in a busy program, the time between the moment an object becomes eligible and the moment when it is finalized might be longer than in previous versions of the runtime system.

In Java 2 SDK for Solaris, the runtime system properly implements the definition of finalization in the JLS, so this difference is not, strictly speaking, an incompatibility. This change might, however, cause programs to malfunction if they rely upon prompt finalization. Many such programs can be repaired by using reference objects instead of finalize methods. (Reference objects are implemented by the `java.lang.ref.Reference` class and its subclasses.) A less preferable
workaround is to periodically invoke the `System.runFinalization` method at regular intervals.

2. Virtual machines in releases prior to Java 2 SDK for Solaris accept some class files that should be rejected (according to the JLS). Typically, these class files have one or more of the following problems:

   a. Extra bytes are at the end of the class file.
   b. The class file contains method or field names that do not begin with a letter.
   c. The class attempts to access private members of another class.
   d. The class file has other format errors, including illegal constant pool indexes and illegal UTF-8 strings.

   Java 2 SDK for Solaris VM more closely implements the specification, so it can be stricter on all of these points.

   Until the RC2 release of Java 2 SDK for Solaris, this strict checking was the default behavior. However, final product testing revealed that many existing Java applications failed to run because of some of these checks. In particular, obfuscated code frequently suffers from problems a and b, while some inner-class code generated by previous compilers suffers from problem c. Java 2 SDK for Solaris relaxed some of these checks to make it as easy as possible for developers and end users to upgrade to Java 2 SDK for Solaris.

   The Java 2 SDK for Solaris `-Xfuture` option enables the strictest possible class file format checks, access checks, and verification policies. Developers should start using this option as soon as possible for all new development work. This ensures that new Java applets and applications are prepared for migration to strict behavior when it again becomes the default.

   The Java Plug-in always uses the strict class file checks (as if the `-Xfuture` flag is set). The appletviewer ignores the `-Xfuture` flag and uses the more relaxed set of default checks.

3. In Java 2 SDK for Solaris, an unimplemented `abstract` method or `interface` method causes an `AbstractMethodError` to be raised at runtime when the method is invoked. In previous versions, the error occurred during link time.

4. Prior to Java 2 SDK for Solaris, putting code on `CLASSPATH` could make it more privileged. For example, prior to Java 2 SDK for Solaris, VMs would sometimes not require that some trusted class files pass verification. Verification depended on the installed security manager. For example, if the following method was in a class on `CLASSPATH`, and was called by an applet, then it could read the `user.name` property:

   ```java
   String getUser() {
       return System.getProperty("user.name");
   }
   ```
In Java 2 SDK for Solaris, this kind of code requires a call to do Privileged (as in the following example).

```java
String getUser(){
    return (String) java.security.AccessController.doPrivileged(new PrivilegedAction(){
        public Object run() {
            return System.getProperty("username");
        }
    });
}
```

The security model in Java 2 SDK for Solaris changes the way in which resources are accessed. When a security manager is in force, resources must reside at a URL that has been granted appropriate security permissions by a policy file. The default policy file, java.policy, grants all security permissions to resources located in the lib/ext directory, where extensions are stored. The default policy file is located at /lib/security/java.policy.

In addition, an invocation to access system resources, as for example by ClassLoader.getResource, must be enclosed within an AccessController.doPrivileged call. Attempts to access system resources without use of a doPrivileged statement will fail. This is true even if the resources that are granted security permissions by the policy file.

5. Prior to JDK 1.1.6, the default ISO 8859-1 character encoding had the name “8859_1”. With JDK 1.1.6, this name changed to “ISO8859_1”. The old name works when passed as an argument to methods in the API, but does not work when used in font.properties files.

6. Each of the following classes in package java.util.zip has a constructor that accepts an int parameter to specify the buffer size:
   - DeflaterOutputStream
   - InflaterInputStream
   - GZIPInputStream
   - GZIPOutputStream

   In Java 2 SDK for Solaris, an IllegalArgumentException is thrown if the input parameter for buffer size is less than or equal to 0. In the JDK 1.1 platform, these constructors do not throw IllegalArgumentException if the size parameter is less than or equal to 0.

7. When the Java 2 SDK for Solaris VM attempts to load a class file that is not of the proper class file format, a java.lang.ClassFormatError is thrown. A java.lang.UnsupportedClassVersionError is thrown when the virtual machine attempts to load a class file which is not of a supported major or minor version. Earlier versions of the virtual machine threw
java.lang.NoClassDefFoundError when either of the above class file problems was encountered.

8. In Java 2 SDK for Solaris, application classes are loaded by an actual ClassLoader instance. This makes it possible for application classes to use installed extensions and also separates the application class path, which is specified by the user, from the bootstrap class path, which is fixed and normally should not be modified by the user. The \-Xbootclasspath option can be used to override the bootstrap class path if necessary.

However, this means that in Java 2 SDK for Solaris, application classes no longer have all permissions by default. Instead, they are granted permissions based on the system’s configured security policy. This might cause some applications that write their own security code based on the original security model in JDK 1.0 and 1.1 to throw an exception and not start in Java 2 SDK for Solaris. To work around this problem, run these applications with the oldjava application launcher, which is documented on the reference page for the Java application launcher. See Chapter 4 for more information on using the oldjava utility.

See the Extension Mechanism Specification at http://java.sun.com/products/jdk/1.2/docs/guide/extensions/spec.html for more information regarding the new extension mechanism and its effect on class loading. The document provides relevant information regarding the new class loader delegation model and class loader API changes.

See the JDK security documentation at http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html for information regarding the security model of Java 2 SDK for Solaris.

9. In Java 2 SDK for Solaris, the constructors of some classes in package java.io check for null input parameters. Such checks were not performed in earlier versions of the platform.

- The constructors PrintStream(OutputStream out) and PrintStream(OutputStream out, boolean autoFlush) throws a NullPointerException if parameter out is null.
- Similar checks have also been added in Java 2 SDK for Solaris to constructors InputStreamReader(InputStream in) and InputStreamReader(InputStream in, String enc). These constructors throw NullPointerException if parameter in is null.
- In Java 2 SDK for Solaris, the constructors Reader(Object lock) and Writer(Object lock) now throw NullPointerException if parameter lock is null.

10. In JDK 1.1 software, Thread.stop is able to interrupt a thread blocked in Object.wait or Thread.sleep. However, in Java 2 SDK for Solaris software running on the Solaris 8 operating environment with native threads, Thread.stop cannot interrupt a blocked thread. Java 2 SDK for Solaris behaves the same as the JDK 1.1 release with respect to Thread.stop when running on other operating systems or on a Solaris 2.6 operating platform.
11. Java 2 SDK for Solaris contains a revised class-loading mechanism. Under the new class loader, if any class file belonging to a package in a jar file is signed, all class files belonging to the same package must have been signed by the same signers. It is no longer possible to use a jar file in which some classes of a package are signed and others are unsigned or signed by a different signer. jar files can still contain packages that are unsigned. However, if any packages contain signed classes, all class files of that package must be signed by the same signer. Existing jar files that don’t meet this criterion are not usable with Java 2 SDK for Solaris or Runtime Environment.

12. Foreground and background colors of native components can be set explicitly using the setForeground() and setBackground() methods. If the colors are not set explicitly, default colors are used as follows:

- In the Java 2 platform, default colors of native components are taken to be the colors defined by the underlying operating system.
- Prior to Java 2 SDK for Solaris, default colors were pre-defined by the Java platform itself.

Code compiled with JDK 1.1 that relies on the default colors can exhibit different and sometimes undesirable component appearance when run on Java 2 SDK for Solaris. For example, if the JDK 1.1 code explicitly sets the foreground color to white for a component label but uses the default background color, the label is not visible when run on Java 2 SDK for Solaris if the underlying operating system’s default background color is white.

API Incompatibilities

1. The ActiveEvent class now resides in the package java.awt. It used to be in java.awt.peer.

2. The Swing and Accessibility packages (formerly in the com.sun.java.* namespace) have moved to the javax.* namespace. These packages now have the following names:

- javax.swing
- javax.swing.border
- javax.swing.colorchooser
- javax.swing.event
- javax.swing.filechooser
- javax.swing.plaf
- javax.swing.plaf.basic
- javax.swing.plaf.metal
- javax.swing.plaf.multi
- javax.swing.table
Applications that use the old com.sun.java.swing* names for these packages from Swing 1.0 do not work in the Java 2 SDK for Solaris platform. Update these applications to use the new java.swing package names. A PackageRenamer tool is available at http://java.sun.com/products/jfc/PackageRenamer for making this conversion.

Note - The packages com.sun.java.swing.plaf.windows and com.sun.java.plaf.motif have not changed names.

If necessary, you can force applications using the old package names to work on the Java 2 SDK for Solaris platform by placing the Swing 1.0 jar file first on the boot class path:

```java
java -Xbootclasspath:<path to 1.0 swingall.jar>:<path to Java 2 SDK rt.jar> ...
```

Note - The Java 2 SDK for Solaris Swing package has been modified to deal with Java 2 SDK for Solaris security. Therefore if you use this technique to run the Swing 1.0 classes with Java 2 SDK for Solaris in an environment where a security manager is present (that is, applets in a browser), the program might not run correctly.

3. The following fields of class java.awt.datatransfer have been made final in Java 2 SDK for Solaris: stringFlavor and plainTextFlavor:

```java
public static final DataFlavor stringFlavor
public static final DataFlavor plainTextFlavor
```

4. Java 2 SDK for Solaris includes the java.util.List interface. As a result, existing source code might possibly produce a namespace conflict between java.awt.List and java.util.List.

In Java 2 SDK for Solaris, using the wildcard import statements together causes a compiler error for code that contains the unqualified name List, as in the following example:
To work around this problem, either add an import statement such as

```java
import java.awt.List;
```

to resolve the conflict throughout the file, or fully qualify the class name by the desired package at each point of use.

5. In Java 2 SDK for Solaris, the field CHAR_UNDEFINED in class java.awt.event.KeyEvent has the value 0xffff. In JDK 1.1-based releases, the field had a value of 0x0. This change was made because 0 is a valid Unicode character and therefore cannot be used to define CHAR_UNDEFINED.

6. In Java 2 SDK for Solaris, the signature of the java.io.StringReader.ready method changed so an IOException can be thrown if the StringReader is closed. Now the StringReader class properly implements the general contract spelled out in the abstract class java.io.Reader, which it extends.

7. In Java 2 SDK for Solaris, the specifications for Integer.decode() and Short.decode() are explicit about the correct representation of negative numbers. Negative numbers always begin with a minus sign (-). If the number is hexadecimal or octal, the base specifier (0x, #, or 0), must come after the minus sign.

In Java 2 SDK for Solaris, the specifications did not clearly state whether a minus sign should come before or after a base specifier. The actual implementations expected the base specifier (if any) to come before the minus sign (if any).

For example, in Java 2 SDK for Solaris, decoding -0x5 returns -5 and decoding 0x-5 throws a NumberFormatException. In JDK 1.1, the results were opposite: -0x5 threw a NumberFormatException and 0x-5 returned -5.

The decoding rules used in JDK 1.1 were unconventional and undocumented. However, some programs might rely on this behavior.

8. In Java 2 SDK for Solaris, classes java.util.Vector and java.util.Hashtable have been retrofitted to implement the relevant interfaces in the new Collections Framework (java.util.Map, respectively). As a consequence, the semantics of the equals and hashCode methods have changed to provide value equality, rather than reference equality, as per the general contracts set forth in List.equals and Map.equals.

This raises several compatibility issues:

- “Self-reference” (inserting a Vector or Hashtable into itself) can now cause stack overflows under certain circumstances:
  a. If a Hashtable is inserted into itself as a key, the Hashtable becomes corrupt, and subsequent operations can cause stack overflow. (It has never been permissible to mutate an object that is serving as a Hashtable key.
Inserting an object into a Hashtable now qualifies as mutation, since it affects equals comparison.

b. If a Hashtable contains itself as a value (otherwise known as an element), equals and hashCode are undefined by the contract. If the hashCode or equals methods are called on such a “self-referential” Hashtable, a stack overflow can result.

c. If a Vector contains itself as an element, equals and hashCode are undefined by the contract, and can cause a stack overflow.

Because equals and hashCode now depend on the contents of the Vector or Hashtable, they are now synchronized against other operations on the collection. Prior to Java 2 SDK for Solaris, they were unsynchronized. The additional synchronization could cause liveness problems if a client depended on the fact that equals or hashCode were unsynchronized.

Clients with explicit dependence on the “reference equality” behavior of equals for Vector or Hashtable do not operate properly. For example, suppose a program kept a Hashtable whose keys were all of the Vectors (or Hashtables) in some system. It was previously the case that each Vector (or Hashtable) was a distinct key, regardless of its contents. It is now the case that any two Vectors (or Hashtables) with the same contents are considered equal. Further, it is no longer legal to modify a Vector or Hashtable that is currently serving as the key in a Hashtable.

Because equals and hashCode now examine the entire contents of the Vector or Hashtable, they might run more slowly than they used to for large collections.

9. The new version of the File class is compatible with both its originally intended behavior and its current common uses. However, some minor differences in behavior might cause some programs to fail.

The new version of the File class also removes redundant separator characters, including those at the end of a pathname string. So the expression new File("foo//bar/").getPath() evaluates to the string foo/bar on a Unix system.

10. In Java 2 SDK for Solaris, the checkAccess method of java.lang.Thread is final. In the JDK 1.1 platform, checkAccess was not final [bug 4151102].

11. In Java 2 SDK for Solaris, the getInterfaces method of class java.lang.Class returns an array containing Cloneable and Serializable class objects when invoked on a class representing an array. In JDK 1.1, getInterfaces returned an empty array when invoked thus.

12. The Java 2 SDK for Solaris abstract classes java.text.BreakIterator, java.text.SimpleTextBoundary, and java.text.Collator (and Collator’s subclass java.text.RuleBasedCollator) no longer implement the Serializable interface as they did in JDK 1.1 [bug 4152965].

13. The Java 2 SDK for Solaris abstract method drawString(AttributedCharacterIterator, int, int) has been added to
the java.awt.Graphics class as part of the Input Method Framework for use by the PersonalJava™ platform. Subclasses of Graphics in applications based on the JDK 1.1 platform are likely to be rare. However, any such subclasses needs to provide an implementation of this new abstract method for use in conjunction with the Java 2 SDK for Solaris platform [bug 4128205].

14. The String hash function implemented in JDK 1.1 releases did not match the function specified in the first edition of the JLS. In fact, the specified function cannot be implemented, in that it addresses characters outside of the input string. Additionally, the implemented function did not perform well on certain classes of strings, including URLs.

To bring the implementation into accord with the specification, and to fix the performance problems, the specification and implementation have been modified. The Java 2 SDK for Solaris String hash() function is specified as:

\[ s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1] \]

where \( s[i] \) is the \( i \)th character of string \( s \).

This change should have no effect on the majority of applications. If an application has persistent data that depend on actual String hash values, it could theoretically be affected. However, the serialized representation of a Hashtable does not depend on the actual hash values of the keys stored in the Hashtable. Thus, applications relying on serialization for storage of persistent data are unaffected [bug 4045622].

15. Java 2 SDK for Solaris does not support the Native Method Interface (NMI).

If you have either written non-JNI native methods with the NMI supported by JDK 1.0, or embedded the runtime by way of the JNI Invocation Interface, you must re-link your native libraries before they can be used with Java 2 SDK for Solaris. In the Solaris environment, you have to replace -ljava in your link command line with -ljvm. You also have to rewrite to the JNI.

**Note** - This change does not affect JNI programmers implementing native methods.

As of Java 2 SDK for Solaris, Sun supports JNI only. JNI is the standard way to make your native libraries inter-operate with the Java programming language. In addition, JNI offers VM independence for any native code that you write. If your methods use NMI, please refer to Chapter 3 for information about migrating your native methods to JNI.

16. Using Thread.suspend() can lead to a deadlock condition. This method is inherently unsafe and has been deprecated (along with other asynchronous thread methods, such as Thread.stop() and Thread.resume()) [bug Id 4203325].

Thread.suspend() is unsafe since it might cause a thread to be suspended, for instance, while it is in the midst of a synchronized method. This action can potentially lock out other threads, and might cause deadlock (please see http://java.sun.com/products//jdk/1.2/docs/guide/misc/).
17. In JDK 1.1, Thread.stop can interrupt a thread blocked in Object.wait or Thread.sleep. In Java 2 SDK for Solaris running on the Solaris 2.6 environment with native threads, Thread.stop cannot interrupt a blocked thread. Java 2 SDK for Solaris behaves the same as JDK 1.1 with respect to Thread.stop when running on other operating systems or on the Solaris 2.6 environment.

18. This section describes compatibility issues affecting developers who implement the interfaces in the java.sql package (primarily those implementing JDBC drivers). The following interfaces contain new methods in Java 2 SDK for Solaris:
   - java.sql.Connection
   - java.sql.DatabaseMetaData
   - java.sql.ResultSetMetaData
   - java.sql.ResultSet
   - java.sql.CallableStatement
   - java.sql.PreparedStatement
   - java.sql.Statement

   Source code that implements the JDK 1.1 versions of these interfaces do not compile properly under Java 2 SDK for Solaris unless you change it to implement the new methods.

   The following interfaces support new types in Java 2 SDK for Solaris:
   - java.sql.ResultSet
   - java.sql.CallableStatement
   - java.sql.PreparedStatement

   Source code that implements these interfaces and compiles correctly in Java 2 SDK for Solaris do not compile under the JDK 1.1 as the new types added to java.sql are not present.

19. A public field, serialVersionUID, was introduced into the java.io.Serializable interface in Java 2 SDK for Solaris. This field should never have been introduced into the interface. Its meaning is unspecified, and its presence contradicts the java.io.Serializable specification.

   The API specification for java.io.Serializable states
   “The serialization interface has no methods or fields.”

   Furthermore, according to the serialization specification, the only way to get the serial version UID of a class is through the ObjectStreamClass.lookup(className).getSerialVersionUID() method. It does not make sense to try to get the serial version UID through a field that might or might not exist.

   The serial version UID is sometimes computed and sometimes explicitly defined in a class, but it is never inherited from a superclass or interface. Since interfaces
cannot be instantiated directly, a serialVersionUID field in an interface can be of no use to the serialization system. In fact, the serialization system makes no use of this field.

For these reasons, the public serialVersionUID field was removed in Java 2 SDK for Solaris. It is expected that this change will not break any existing applications.

**Tool Incompatibilities**

1. In Java 2 SDK for Solaris, the javac −O option has a different meaning, and might have different performance effects on generated code than it does in JDK 1.1. javac −O directs the compiler to generate faster code. It no longer inlines methods across classes, or implicitly turns −depend on or −g off. Because it no longer implicitly turns on −depend, you need to add −depend to the command line where desired.

2. Use only versions of javac released after JDK 1.1.4 to compile against Java 2 SDK for Solaris class files. JDK 1.1 versions of javac sometimes generate incorrect inner class attributes (the Java 2 SDK for Solaris javac generates correct attributes). The JDK 1.1.4 javac compiler and earlier compilers could crash when encountering the correct form. Beginning with JDK 1.1.5, javac handles both the old and the new corrected attributes. This is only a compile-time problem. Java 2 SDK for Solaris compiler generates class files that work on older VMs.


4. In JDK 1.1, javap −verify performed a partial verification of the class file. Java 2 SDK for Solaris does not include this option. The option was misleading because it did not perform many portions of a full verification.

5. Prior to JDK 1.1.4, the Java interpreter allowed the class file at /a/b/c.class to be invoked from within the /a/b directory by the command java c (even if the class c was in package a.b.*). In JDK 1.1.4 and Java 2 SDK for Solaris, you must specify the fully qualified class name. For example, to invoke the class a.b.c at a/b/c.class, you could issue the command java a.b.c from the parent directory of directory /a.

6. Because of bugs in JDK 1.1-based releases, code signed using the JDK 1.1 javakey tool appears to be unsigned to the Java 2 SDK for Solaris. Code signed using Java 2 SDK for Solaris appears to be unsigned to JDK 1.1-based releases.

7. Prior to Java 2 SDK for Solaris, javac permitted some inconsistent or redundant combinations of command line options. For example, you could specify −classpath multiple times with only the last usage taking effect. You can no longer use this behavior.

8. Prior to Java 2 SDK for Solaris, the −classpath option of the Java interpreter set the search path used by the VM when loading system classes. The VM then set the java.class.path property to reflect this path. Typically, application classes
were invoked directly from the system class path without an associated class loader.

Java 2 SDK for Solaris now starts applications in an application class loader to take advantage of both installed extensions and the new security model. The −classpath option now sets the classpath used by the application class loader (for loading classes and resources). Similarly, the java.class.path property now reflects this path. You can still override the system classpath used internally by the VM with a new −Xbootclasspath option. In most cases, you should not have to change the system classpath.

Most applications should not be affected by this change. However, the java.class.path property no longer include those directories and JAR files used for loading system classes. Read the new sun.boot.class.path property for that information. Applications that install their own security managers might be adversely affected. You have to rewrite these applications for Java 2 SDK for Solaris, but in the meantime, the −Xbootclasspath switch is provided for backward compatibility.

9. The Java 2 SDK for Solaris javadoc tool produces filenames of the form

   package−<package name>.html

   instead of the previous

   Package−<package name>.html

   for package-level API output. For example, previously the default package-level output for package java.io was in file

   Package-java.io.html

   beginning with Java 2 SDK for Solaris, the filename is

   package-java.io.html.

10. The Java 2 SDK for Solaris javah is different as NMI is not supported, thus the −nmi flag is no longer provided. For more information, see “javah” on page 39.

### Serialization Incompatibilities

Because of rare and infrequent problems with the format for Externalizable objects, it was necessary to make an incompatible change in Java 2 SDK for Solaris. A JDK 1.1.5 (or earlier) program throws a StreamCorruptedException when it tries to read an Externalizable object generated in Java 2 SDK for Solaris format. Programs based on JDK software version JDK 1.1.6 or later do not have this problem.

However, a new Java 2 SDK for Solaris API supports backward compatibility. To write streams in the old format, before doing any writes, call

   ObjectOutputSteam.useProtocolVersion(PROTOCOL_VERSION_1)
Java Native Interface (JNI)

Making the Transition From NMI to JNI

NMI is not supported in Java 2 SDK for Solaris software. In some installations, NMI is known as JNI 1.0, and JNI as JNI 1.1.

Note - Any application containing native methods is platform dependent and not Write Once Run Anywhere™. This is true regardless of whether NMI or JNI is used. Applets cannot be affected by a change in native method support.

Porting

Porting applications that use NMI to ones that use JNI interfaces is straightforward. For example, an engineer who was unfamiliar with the application ported 6000 lines of C code in approximately two days. In this instance, NMI was only lightly used, but the uses were widely distributed.

javah

The javah command generates C header and source files that are needed to implement native methods. C programs use the generated header and source files to reference instance variables of an object from native source code. JNI does not require header information or stub files. You can still use the javah command with the jni option to generate native method function prototypes needed for JNI-style native methods. The result is placed in the .h file.
In JDK 1.1 default mode, the javah command generates NMI output. Because the NMI interfaces are completely incompatible with the Java 2 SDK for Solaris implementation, Java 2 SDK for Solaris does not support the production of NMI-style output by way of the −old flag. Rather the −old flag is parsed and, if it is found, an −old not supported message occurs and javah exits.

General JNI Issues

Compiler Restrictions

Certain optimizations that might be performed by compilers compiling native methods can cause the VM to fail. The VM relies on the ability to examine the stack frames of functions on a thread’s stack. Therefore, the code for native functions must always create stack frames as specified in the system calling conventions for non-leaf functions. Additionally, the frame pointer register must always point to a valid stack frame.

- On the Intel (IA) platform, the Sun Workshop™ C compiler can generate code that violates this requirement at optimization levels -xO4 and -xO5. You should not use an optimization level above -xO3.
- On the SPARC platform, all optimization levels are safe. If you are using another compiler, check its documentation to ensure that it produces code that meets this requirement.

Note - A native method that violates this restriction can cause the VM to abort randomly.

Linking Native Solaris Applications

You should link native Solaris applications with −lthread. Not doing so can cause incorrect behavior.

You must specify the libthread.so library (−lthread option) before the libc.so library (−lc option) when linking up native applications that use JNI. The Sun C compiler option −mt automatically adds the −lthread option, and the −lc option is typically not specified (it defaults to the end of the list). So when using Sun compilers or the Sun linker, you must supply either the −mt or the −lthread options.
Fast JNI Array Access

You can get faster JNI array access with this JDK by using
jni_GetPrimitiveArrayCritical() and
jni_ReleasePrimitiveArrayCritical() instead of Get*ArrayElements() calls.

- Code that uses these *Critical() operations must comply with certain restrictions:
  - You must release the array elements quickly after you get them.
  - You cannot call back into Java while holding onto the array elements.
  - You cannot call other JNI operations.

Because of these restrictions, the array elements can be accessed without pinning and without copying. For more information, see:


Locating Shared Libraries

If you are using Solaris 2.5.1 or 2.6 operating environment and you have installed the JDK in any location other than the default location (/usr/java1.2), JNI native applications might have problems locating the JDK shared libraries. You can work around this problem by setting the environment variable LD_LIBRARY_PATH to include the jre/lib/sparc or jre/lib/i386 directory of the JDK installation. This problem does not exist on either Solaris 7 software or Solaris 8 software.

Signal Processing State

Native code using JNI should not modify the signal-processing state. The VM uses signals and any change to the signal handling can result in VM failures.
CHAPTER 4

Command–Line Differences Between the Java 2 SDK and JDK 1.1

VM-Specific (Non-Standard) Options

The reference implementation of Java 2 SDK for Solaris divides options into two groups. One group includes options specific to a particular VM; the other group applies to all VMs. Each group has its own option syntax. VM-specific options all start with \(-X\); for example, \(-Xdebug\) to enable debugging. Using the \(-X\) by itself produces a help message listing all the VM-specific options that this implementation accepts.

Java 2 SDK for Solaris supports these \(-X\) options shown in the table below:

**TABLE 4–1** Currently Accepted \(-X\) Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-X)</td>
<td>Lists available options</td>
</tr>
<tr>
<td>(-Xbootclasspath[/a</td>
<td>/p] : &lt;path&gt;)</td>
</tr>
<tr>
<td>(-Xdebug)</td>
<td>Enables remote debugging</td>
</tr>
<tr>
<td>(-Xmaxjitcodesize)</td>
<td>Sets the maximum size in bytes for the JIT compiler code area</td>
</tr>
<tr>
<td>(-Xms)</td>
<td>Sets initial Java heap size</td>
</tr>
<tr>
<td>(-Xmx &lt;size&gt;)</td>
<td>Sets maximum Java heap size</td>
</tr>
</tbody>
</table>
### TABLE 4–1  Currently Accepted −X Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-XnoFieldAccessGC</td>
<td>Disables field access garbage collection</td>
</tr>
<tr>
<td>-Xoptimize</td>
<td>(SPARC only) Experimental only. Spend more time optimizing methods in the JIT. This option will most likely benefit long-running CPU-bound applications and might result in increased performance of your application.</td>
</tr>
<tr>
<td>-Xoss&lt;size&gt;</td>
<td>Sets maximum Java stack size for any thread</td>
</tr>
<tr>
<td>-Xrs</td>
<td>Reduces the use of OS signals</td>
</tr>
<tr>
<td>-Xrunhprof[:file=&lt;file&gt;,depth=&lt;n&gt;]</td>
<td>Outputs heap profile to java.hprof.txt or &lt;file&gt;</td>
</tr>
<tr>
<td>-Xsqnopause</td>
<td>Does not pause for user interaction on SIGQUIT.</td>
</tr>
<tr>
<td>-Xss &lt;size&gt;</td>
<td>Sets the maximum native stack size for any thread</td>
</tr>
<tr>
<td>-Xt</td>
<td>Turns on instruction tracing</td>
</tr>
<tr>
<td>-Xtm</td>
<td>Turns on method tracing</td>
</tr>
</tbody>
</table>

**Note**: The −X options are subject to change without notice.

Many of these options correspond to JDK 1.1.1–1.1.6 options. Refer to the next section for more information.

Also, although these −X options are VM specific, many of them are sufficiently generic that any reasonably useful VM would provide them. Also, Java 2 SDK for Solaris does not support the following reference platform −X option whereas it is supported in the reference release:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Xnoasyncgc</td>
<td>Disables asynchronous garbage collection</td>
</tr>
</tbody>
</table>

### Option Compatibility

The following JDK 1.1 compatibility options are currently allowed:

- −verbosegc
- −t
- −tm
- −debug

---

Java 2 SDK for Solaris Developer’s Guide ♦ February 2000
-noasyncgc
-noclasspath
-verify
-verifyremote
-noverify
-prof[:<file->]
-cs
-checksource
-ss
-oss
-ms
-mx
-l

oldjava Utility

The oldjava utility provides greater compatibility with the JDK 1.1–based java utility. When you invoke oldjava, the -classpath command–line option and CLASSPATH environmental variable are treated as in JDK 1.1 release. Certain new Java 2 SDK for Solaris options are disabled (in particular the -jar option).
Using SIGQUIT for Debugging

Java 2 SDK for Solaris features a new debugging process. It sends a SIGQUIT signal to a Java process running in the foreground. This signal causes the process to pause for user input after displaying the following menu:

```
1: Terminate program
2: Find & print one deadlock
3: Find & print all deadlocks
4: Print thread stacks
5: Print lock registry
6: Continue program
```

- Use the second and third options to search for Java level deadlocks (where two or more threads are involved in a cyclic wait for monitors). Careless use of synchronized methods in a multithreaded Java program can cause this kind of deadlock.
- The fourth option provides a list of active threads in the system and lets you selectively obtain Java stack traces of these threads.
- The fifth option produces a dump of the VM’s internal locks. You might find this information useful when reporting VM-related problems or bugs.

You can send a SIGQUIT signal using `kill(1)` or by typing Ctrl-Backslash to a foreground process.

If you send the signal to a Java process running in the background, the output of options three, four, and five is dumped to the standard error device but the program continues to execute without pausing for user input.

The following trace illustrates a typical interaction scenario following the receipt of a SIGQUIT signal by the Java process.
A SIGQUIT has been received. Do you want to:
1) terminate program
2) check & print one deadlock
3) check & print all deadlocks
4) dump thread stacks
5) dump lock registry
6) continue program
Select Action: 2
Found 0 deadlock

Do you want to:
1) terminate program
2) check & print one deadlock
3) check & print all deadlocks
4) dump thread stacks
5) dump lock registry
6) continue program
Select Action: 6

List of Java Threads:
[Thread# 1] t@1: (0xef715af4) GC-like thread, or damaged thread
[Thread# 2] t@2: "Thread-4"
[Thread# 3] t@3: "SoftReference sweeper"
[Thread# 4] t@4: "Finalizer"
[Thread# 5] t@5: "Reference handler"
[Thread# 6] t@6: "Signal dispatcher"
[Thread# 7] t@7: "main"

Choose an index (1 to 7) to dump thread stack.
8 or greater returns you to the previous menu: 7

"main" (TID:0x36adc, sys_thread_t:0x36a50, state:R, thread_t: t@1, threadID:0x20f68, stack_bottom:0xf0000000, stack_size:0x20000) prio=5

[1] java.lang.Thread.yield(Thread.java)
[2] PingPoll.run(Compiled Code @ 0x117300)
[3] PingPoll.main(Compiled Code @ 0x1172c8)

Choose an index (1 to 7) to dump thread stack.
8 or greater returns you to the previous menu: 8

Do you want to:
1) terminate program
2) check & print one deadlock
3) check & print all deadlocks
4) dump thread stacks
5) dump lock registry
6) continue program
Select Action: 6
Continuing Program
...

Using SIGQUIT for Debugging  49
Memory Allocation and Constraints

This appendix describes how Java 2 SDK for Solaris allocates memory. The only memory constraints are the Solaris 32-bit addressing limitations minimum and the swap space on the machine.

VM Size

The calculation for the size of a single instance of the VM is as follows:

- long and double fields = 8 bytes.
- All other field types = 4 bytes.
- Instance size = size of all non-static fields (including inherited fields) 8-byte header. No alignment or other implicit costs.
- An array has a 12-byte header plus storage for all its elements (rounded up to a multiple of 4 bytes). The size of a single element is:
  - byte[], boolean[] = 1 byte
  - short[], char[] = 2 bytes
  - long[], double[] = 8 bytes
- All other arrays = 4 bytes

arrays are also not aligned beyond the 4 bytes that the rounding ensures.
Interpreting \texttt{--verbosegc} Output

Troubleshooting Garbage Collection

Use \texttt{--verbosegc} to determine whether your application has garbage-collected only young space or both young and old space (complete). Run the option as follows:

\begin{verbatim}
% java -verbosegc
\end{verbatim}

This action would produce output similar to the following:

\begin{verbatim}
GC[1] in 305 ms: (6144kb, 6% free) \rightarrow (14Mb, 61% free)
\end{verbatim}

In this case \texttt{GC[1]} indicates a complete garbage collection. \texttt{GC[0]} would indicate a young space only garbage collection. The collection took 305\,ms. At the start of the collection there was a 6144KByte heap with 6\% free. The heap expanded during collection and at the end of the collection there was a 14MByte heap with 61\% free.

Generational Heap Sizes

The heap is divided into a young generation and an old generation. For the sizes of each of the generations, use the following command.

\begin{verbatim}
% java -verbosegc -verbosegc
\end{verbatim}
Sample Output

Examine the following sample output of `–verbosegc –verbosegc`:

```
Gen0 (semi-spaces): size=4096kb, free=0kb, maxAlloc=0kb
  From space: size=524288 words, used=524286 words, free=2
  To space: size=524288 words, used=1 words, free=524287
Gen0 (semi-spaces)-GC #4 tenure-thresh=0 61 ms 0%->28% free
  From space: size=524288 words, used=378157 words, free=146131
  To space: size=524288 words, used=1 words, free=524287
Gen1 (mark-compact): size=4096kb, free=0kb, maxAlloc=0kb
Gen1 (mark-compact)-GC #1 Gen1: 850kb dense
  262 ms 0%->2% free
Gen1 (mark-compact): size=4096kb, free=80kb, maxAlloc=80kb
```

From this output, you can derive the following information about each generation:

Gen0 is a semi-space copying collector that started at 4096 KBytes. Before the young space collection there was 0 Kb of free memory; after the young space collection there were 378157 words free (28%). Gen1 is a mark-compact collector, which freed 2% during the collection.

A `–verbosegc –verbosegc –verbosegc` mode gives even more information.
Poller Class Usage

Poller Class

The Poller class demonstration code provides a means of accessing the functionality of the C poll(2) API. It attempts to mirror the C poll(2) API only as much as is possible while allowing for optimal performance. To remove the impact of shuttling large arrays of I/O objects between Java and the kernel, management of the files/sockets to be polled has been moved into the JNI C code (or the /dev/poll device driver itself, if available). The README.txt file included in the Poller class demonstration directory gives more detailed information on using it.

Basics of Poller Class Usage

```java
Poller Mux = new Poller();

int serverFd = Mux.add(serverSocket, Poller.POLLIN);
int fd1 = Mux.add(socket1, Poller.POLLIN);
...
int fdN = Mux.add(socketN, Poller.POLLIN);
long timeout = 1000; // one second

int numEvents = Mux.waitMultiple(100, fds, revents, timeout);

for (int i = 0; i < numEvents; i++) {
    /*
    * Probably need more sophisticated mapping scheme than this!
    */
} (continued)
```
if (fds[i] == serverFd) {
    System.out.println("Got new connection.");
    newSocket = serverSocket.accept();
    newSocketFd = Mux.add(newSocket, Poller.POLLIN);
} else if (fds[i] == fd1) {
    System.out.println("Got data on socket1");
    socket1.getInputStream().read(byteArray);
    // Do something based upon state of fd1 connection
}
...
Running with Both Java 2 SDK and JDK 1.1

The `/usr/java` symbolic link is used to define the default Java environment on a Solaris system when more than one Java environment is installed. Currently, JDK 1.1 is installed in `/usr/java1.1` and the Java 2 SDK for Solaris is installed in `/usr/java1.2`.

Prior to the Solaris 8 release, the `/usr/java` symbolic link pointed to `/usr/java1.1` if both JDK 1.1 and Java 2 SDK for Solaris are installed. Starting with the Solaris 8 release, the `/usr/java` symbolic link points to `/usr/java1.2` by default if both JDK 1.1 and Java 2 SDK for Solaris are installed.

Since there are symbolic links in `/usr/bin` (also known as `/bin`) that use `/usr/java` (for example, `/usr/bin/java` refers to `/usr/java/bin/java`) this `/usr/java` link can change the default “java” seen by most users. Many Java applications run with either Java 2 SDK for Solaris or JDK 1.1, but users and applications might wish to be selective about which “java” they use.

Java users that want to use JDK 1.1 should add `/usr/java1.1/bin` to their PATH settings before `/usr/bin`. Java users that want to use Java 2 SDK for Solaris should add `/usr/java1.2/bin` to their PATH settings before `/usr/bin`. Also, depending on the situation, you might need to make changes to other environment variables such as CLASSPATH, LD_LIBRARY_PATH, or JAVA_HOME, although none of these environment variables are required.

Java applications that require JDK 1.1 should refer to `/usr/java1.1` and those that require Java 2 SDK for Solaris should refer to `/usr/java1.2`.

Changing the symbolic link `/usr/java` is not recommended. Changing the symbolic link can cause problems for Java applications that were bundled with a Solaris release prior to Solaris 8 or that are expecting to use JDK 1.1.
Index

A
abstract 34
AbstractMethodError 28
AccessController.doPrivileged 29
ActiveEvent 31
API incompatibilities 31
  abstract 34
  ActiveEvent class 31
  boot class path 32
  CHAR_UNDEFINED 33
  checkAccess 34
  class java.lang.Class 34
  Cloneable 34
  Collator 34
  com.sun.java.plaf.motif 32
  com.sun.java.swing* 32
  com.sun.java.swing.plaf.windows 32
  File class 34
    redundant separator characters 34
  hashCode 33, 34
  Hashtable 33
    as a key 34
    contains itself 34
    import statement 33
    Integer.decode() 33
    IOException 33
    java.awt 31
    java.awt.datatransfer 32
    java.awt.event.KeyEvent 33
    java.awt.Graphics 35
    java.awt.List 32
    java.io.Serializable 36
    java.io.StringReader.ready 33
    java.lang.Thread 34
    java.sql 36
    java.swing 32
    java.text.BreakIterator 34
    java.text.Collator 34
    java.text.RuleBasedCollator 34
    java.text.SimpleTextBoundary 34
    java.util.Hashtable 33
    java.util.List 32
    java.util.Map 33
    java.util.Vector 33
    JNI 35
    List.equals 33
    Map.equals 33
    minus sign 33
    NMI 35
    NumberFormatException 33
    Object.wait 36
    PackageRenamer 32
    plainTextFlavor 32
    Serializable 34
    serialVersionUID 36, 37
    Short.decode() 33
    String 35
    stringFlavor 32
    Swing and Accessibility packages 31
    Thread.sleep 36
    Thread.stop 35
    Thread.suspend 35
    unqualified name List 33
    Vector 33
      contains itself 34
    wildcard import statements 33
with security manager 32

B
binary compatibility 17
general policy 18
functional releases 18
maintenance releases 18
regenerating class files 18
violating class file formats 18
-targer 1.2 option 17
with Java 2 SDK APIs 18
with javac compiler 17
with JDK 1.0 and 1.1 17
with JDK 1.0 or 1.1 APIs 18

C
CHAR_UNDEFINED 33
ccheckAccess 34
cchecksource option 45
class java.lang.Class 34
ClassLoader 30
CLASSPATH 28
Cloneable 34
Collator 34
com.sun.java.plaf.motif 32
com.sun.java.swing* 32
com.sun.java.swing.plaf.windows 32
tcompiler restrictions 40
tcompilers 19
tconcurrency primitives and threads 15
tcs option 45

deadlock detection 47
ddebug option 45
deflateroutputstream 29

e
Externalizable 38

f
features and enhancements 11
Find & print all deadlocks 47
Find & print one deadlock 47

G
garbage collection
full collections 12
generational 12
old space 53
older generations 12
shorter pauses 12
using verbosegc option 53
young space 53
younger generations 12
garbage collection, exact 12
direct pointers 12
allocation 12
indirection 12
memory consumption 12
handles 12
JNI 12
requirements 12
generational heap sizes 53
old generation 53
young generation 53
getArrayElements 41
GZIPInputStream 29
GZIPOutputStream 29

H
hashCode 33, 34
Hashtable 33
heap alignment 13
8-byte boundaries 13
JIT-compiled code 13
memory usage 13
native code 13
performance 13
heap inspection 14
finding memory leaks 14
killing programs 14
memory consumed 14
objects in heap 14
reference chains 14
SIGQUIT 14
high-performance memory system
direct pointers 12
fully compacting 12
generational 12
non-conservative 12
IllegalArgumentException 29
incompatibilities 19
language incompatibilities 19
and JLS 19
more stringent 19
InflaterInputStream 29
inlining 14
automatic 14
when to use 14
InputStreamReader(InputStream in) 30
InputStreamReader(InputStream in, String enc) 30
Integer.decode() 33
IOException 33

J
Java Language Specification, see JLS
java.awt 31
java.awt.datatransfer 32
java.awt.event.KeyEvent 33
java.awt.Graphics 35
java.awt.List 32
java.class.path 38
java.io 30
java.io.Serializable 36
java.io.StringReader.ready 33
java.lang.ClassFormatError 29
java.lang.NoClassDefFoundError 30
java.lang.Thread 34
java.lang.UnsupportedClassVersionError 30
java.policy 29
java.sql.CallableStatement 36
java.sql.Connection 36
java.sql.DatabaseMetaData 36
java.sql.PreparedStatement 36
java.sql.ResultSet 36
java.sql.ResultSetMetaData 36
java.sql.Statement 36
java.swing 32
java.text.BreakIterator 34
java.text.Collator 34
java.text.RuleBasedCollator 34
java.text.SimpleTextBoundary 34
java.util.Hashtable 33
java.util.List 32
java.util.Map 33
java.util.Vector 33
javah command 39
NMI-style not supported 40
old flag 40
javakey 37
javax.accessibility 32
javax.swing 31
javax.swing.border 31
javax.swing.colorchooser 31
javax.swing.event 31
javax.swing.filechooser 31
javax.swing.plaf 31
javax.swing.plaf.basic 31
javax.swing.plaf.metal 31
javax.swing.plaf.multi 31
javax.swing.table 32
javax.swing.text 32
javax.swing.text.html 32
javax.swing.tree 32
javax.swing.undo 32
JIT compiler 14
fast type checks 14
inlining 14
loop analysis 14
optimizing 14
virtual and non-virtual methods 14
JLS 17
jni_GetPrimitiveArrayCritical() 41
jni_ReleasePrimitiveArrayCritical() 41

K
keytool 37
kill() 47

L
l option 45
language incompatibilities 19
abstract classes 23
abstract method modifier 20
with final 20
with native 20
with private 20
with synchronized 20
cast between two interface types 21
char type to byte type 19
char type to short type 19
ClassName 24
classpath 27
constructor invocation using this 26
double assignment of final variables 20
duplicate nested labels 23
field initialized to default value 21
illegal char literals 20
illegal hex literals 19
illegal string literals 20
illegal void 20
inner classes 23, 26
int type to long type 19
labeled declaration 24
local class 26
null type 22
package-private 23
qualified names 22
static fields expression 21
static member class
top-level class 26
strict floating point 24
strictfp code behavior 24
strictfp identifies 24
strictfp keyword 24
super keyword 26
third subexpression 21
using parentheses to correct 21
this keyword 24
type and subpackage with same name, restriction 27
libthread.so library 40
List, unqualified name 33
List.equals 33

M
Map.equals 33
—lc option 40
—lthread option 40
mixed-mode execution 15
compilation overhead 15
default mode 15
effect on performance 15
methods with loops 15
methods without loops 15
threshold integer 15
with performance-critical methods 15
ms option 45
multithreading
contention paths 13
core-locking 13
fast-locking algorithm 13
fine-grain locking 13
lock contention, minimizing 13
mx option 45

N
NMI 39
noasyncgc option 45
noclassgc option 45
noverify option 45
-nowarn command-line option 18
NullPointerException 30
NumberFormatException 33

O
Object.wait 36
oldjava 30
oldjava utility 45
oss option 45

P
PackageRenamer 32
plainTextFlavor 32
poll function 16
poll() 55
Poller class
demonstration code 16
determining I/O status 16
JNI support 16
kernel–polling mechanism 16
Poller class demonstration code 55
porting to JNI 39
Print lock registry 47
Print thread stacks 47
PrintStream(OutputStream out) 30
PrintStream(OutputStream out, boolean
autoFlush) 30
Privileged 29
prof option 45
Reader(Object lock) 30
runtime incompatibilities 27
abstract method 28
AbstractMethodError 28
AccessController.doPrivileged call 29
buffer size less than or equal to zero 29
class file rejection 28
accessing private members of another class 28
constant pool indexes 28
extra bytes 28
format errors 28
illegal UTF-8 strings 28
method or field name not beginning with a letter 28
ClassLoader 30
ClassLoader.getResource 29
CLASSPATH 28
default behavior 28
default colors 31
default Java 2 SDK background color 31
DeflaterOutputStream 29
doPrivileged statement 29
finalize methods 27
garbage collection 27
high priority threads 27
when finalized 27
GZIPInputStream 29
GZIPOutputStream 29
IllegalArgumentException 29
InflaterInputStream 29
InputStreamReader(InputStream in) 30
InputStreamReader(InputStream in, String enc) 30
interface method 28
ISO 8859–1 29
“ISO8859_1” 29
jar file 31
Java Plug-in 28
java.io 30
java.lang.ClassFormatError 29
java.lang.NoClassDefFoundError 30
java.lang.UnsupportedClassVersionError 30
java.policy 29
NullPointerException 30
oldjava application launcher 30
PrintStream(OutputStream out) 30
PrintStream(OutputStream out, boolean autoFlush) 30
Privileged 29
program malfunctions 27
Reader(Object lock) 30
reference objects 27
setBackground() 31
setForeground() 31
setting foreground and background colors 31
system configuration security policy 30
System.runFinalization method 28
Thread.sleep 31
Thread.stop 31
Writer(Object lock) 30
Xfuture option 28

S
Scalability 15
security model 29
Serializable 34
serialization incompatibilities 38
Externalizable 38
StreamCorruptedException 38
serialVersionUID 36, 37
setBackground() 31
setForeground() 31
Short.decode() 33
signal processing state 41
SIGQUIT 14, 47
deadlock detection 14
for debugging 14
state dumps 14
thread monitoring 14
source compatibility 18
deprecated APIs 18
-nowarn command-line option 18
downward source-compatibility 18
genral policy 18
functional releases 18
maintenance releases 18
upward source-compatibility 18
ss option 45
StreamCorruptedException 38
stringFlavor 32
sun.boot.class.path 38

T

t option 44
Thread.sleep 31, 36
Thread.stop 31, 35
Thread.suspend 35
tm option 44
Tool incompatibilities 37
classpath 38
java.class.path 38
javakey 37
keytool 37
sun.boot.class.path 38
verify 37
Xbootclasspath 38

V

Vector 33
verbosegc option 44
verify 37
verify option 45
verifyremote option 45
VM size calculation 51
VM-specific options 43

W

wildcard import statement 33
Write Once Run Anywhere 39
Writer(Object lock) 30

X

X option 43
Xbootclasspath 38
Xdebug option 43
-Xfuture option 28
Xmaxjitcodesize option 43
Xms option 43
Xmx option 43
Xnoasycgc option 44
Xnoclassgc 44
Xoptomize 44
Xoss option 44
Xrs option 44
Xrunhprof option 44
Xsqnopause option 44
Xss option 44
Xt option 44