JDK 1.2 For Solaris Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 805-5725-06
July 1998, Revision A

Contents

CHAPTER 1

Introduction

The Java™ Development Kit (JDK™) 1.2 for Solaris™ software provides a high-performance Java Virtual Machine (JVM) from Sun Microsystems™. This JDK is optimized for Solaris software server-side applications. Server-side applications are often characterized as:

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

The faster JVM also provides improved performance for client-side applications.

New Features and Enhancements

The JDK 1.2 has been optimized for the Solaris 2.6 and 2.7 system software. It provides these new enhancements and features:

- Substantially increased performance for large applications running large numbers of Java threads on multiprocessor (MP) systems, due to
  - Enhanced garbage collection
  - Fast Java thread synchronization
  - High-performance memory system
  - Fine-grained locking capability
  - Excellent scaling capability on Solaris software
  - Improved optimizations in Just-In-Time (JIT) compiler
  - Improved core class library technology

Contents of this Document

This document discusses the following areas of interest with regard to JDK 1.2:

- Performance Enhancements
- JDK 1.2 Backwards Compatibility
- NMI to JNI Transition Issues

CHAPTER **2**

Performance Enhancements

Garbage Collection and Performance

Garbage collection affects Java program performance. Before JDK 1.2, the garbage collection method was *monolithic*, *conservative*, and handle-based:

- Monolithic, in that the garbage collector examined all allocated objects during each garbage collection cycle
- Conservative, in that some unused objects might not have been recognized as garbage and reclaimed.

This resulted in some garbage not being collected, and slower execution when many objects were created by an application. Compaction in a monolithic, conservative garbage collector also requires objects to have handles, which slows performance.

The JDK 1.2 for Solaris software features enhanced garbage collection, which vastly improves Java application performance.

Enhanced Garbage Collection

The performance of the JDK for Solaris is considerably improved as a result of the following enhancements:

- *Exact* garbage collection
- High-performance memory system

What Is Exact Garbage Collection?

Exact garbage collection increases performance with these characteristics:

- Using direct pointers rather than handles for objects
- Requires the use of the Java Native Interface (JNI) for external code
- Enables *efficient incremental* garbage collection

  Efficient incremental garbage collection produces significantly better performance than monolithic conservative garbage collection.

  Efficient garbage collection uses *generational* techniques. Generational garbage collection examines a subset of all allocated objects for reclamation. This subset is selected based on the age of the objects; younger objects are examined more often than older ones, hence the term “generational”. This approach is used because more recently created objects are more likely to be discarded in Java programs. In addition, pauses for garbage collection are brief.

Reduced Synchronization Overhead

Fast-Sync Monitor Locking/Fast Thread Synchronization

JVM uses a new, patented internal fast-locking algorithm for more efficient method synchronization, which enhances performance.

Lock Contention Minimization

The Solaris JDK 1.2 minimizes lock contention in the JVM by using thread-local data structures and finer-grained locking.

Fine-Grain Locking

The JVM core locking architecture has been significantly reworked to enhance the granularity of the JVM locks. *Fine-grain* locking minimizes the number of contended paths of thread execution. Therefore, it is less likely that one thread’s execution will impede other threads. The more fine-grained the locking, the better the MP scaling capability. Fine-grain locking enables the JVM to conduct a larger number of concurrent operations, boosting performance.

CHAPTER **3**

JDK 1.2 Backwards Compatibility

This chapter is intended to be a complete list of all cases in which a program that works under JDK 1.0 or 1.1 would not compile and/or run under JDK 1.2. There is also a list of JDK tools changes. *JLS* refers to "The Java Language Specification".

Binary Compatibility

JDK 1.2 is upwards binary-compatible with JDK 1.0 and 1.1 except for the incompatibilities listed below.

This means that, except for the noted incompatibilities, class files built with JDK 1.0 or 1.1 compilers will run correctly in 1.2.

Downward binary compatibility is not guaranteed. That is, class files built with a 1.2 compiler relying only on APIs defined in JDK 1.0 or 1.1 will generally run on 1.0 and 1.1 virtual machines. Of course, if the class files depend on any 1.2 APIs, those files will not work on earlier platforms.

Source Compatibility

JDK 1.2 is upwards source-compatible with JDK 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 in JDK 1.2.

Downward source compatibility is not supported. If source files use new language features or 1.2 APIs, they will not be usable with an earlier JDK.

Deprecated APIs are methods and classes supported only for backwards compatibility. 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. It has not been determined when or if the deprecated elements will be removed from the APIs.

Language Incompatibilities

Compilers in JDK 1.0 and 1.1 compile some types of illegal code without producing warnings or error messages. Improvements in JDK 1.2 have made the compiler more stringent in ensuring that code conforms to the JLS. The following list summarizes the types of illegal code that can be compiled with JDK 1.0 or 1.1 compilers but will not compile in 1.2.

1. Some initializations of `int` types to `long` types that were passed by previous compilers will be flagged as errors in JDK 1.2. In the following program,

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

both the initialization to `i` and `j` are in error. Prior to JDK 1.2, only the incorrect initialization of `i` would have been reported. The initialization of `j` would have overflowed silently. (Bug 4035346)

2. Prior to JDK 1.2, implicit assignment conversion from char to byte and short would be allowed in the case of character literals that fit into 8 bits. Such implicit assignment conversion is not allowed by the compiler in JDK 1.2. For example,

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

   will not pass the compiler. An explicit cast is required for such conversions. (Bug 4030496)

3. `0xL` is not a legal hex literal and will not be passed by the 1.2 compiler. It was parsed as zero in previous versions of the JDK software. (Bug 4049982)

4. `'''` (and therefore `'\u0027'`) is not a legal char literal and will not be passed by the 1.2 compiler. Use `'\''` instead. (Bug 1265387)

5. "`\u000D`" is not legal in string and char literals and will not be passed by the 1.2 compiler. Use '`\r`' instead. (Bug 4063147)

   `\u000A` and `\u000D` are the CR and LF characters, which terminate lines, even in comments (bug 4086919). Code such as:

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

6. The type `void[]` is not legal and will not be passed by the 1.2 compiler. (Bug 4034979)

7. It is not legal to combine the `abstract` method modifier with `private`, `final`, `native`, or `synchronized` modifiers. (Bug 1266571)

8. Double-assignment of `final` variables, though illegal, would pass previous compilers in some circumstances. For example, the following two samples would pass 1.1.x compilers, even though both involve double-assignment of a `final` variable. Such assignments will not pass the 1.2 compiler. (Bugs 4066275 and 4056774):

```
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;
     }
    }
}
```

```
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. It is not legal to reference a non-static field with an apparently static field expression of the form 'Classname.fieldname'. Prior to 1.2, 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 JDK 1.2. (Bug 4028359) For example, the following program now generates a compile-time error:

```
interface Noisy {
    int method();
}

interface Quiet {
    void method();
}

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

In JDK 1.2, an assignment expression is not accepted as the third subexpression of a conditional statement. For example, the statement

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

will cause a compiler error in 1.2. If this problem occurs in existing code, placing the offending assignment expression in parentheses will cause the code to compile as it did in previous versions of the JDK:

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

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

```
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) {
    InitTest test = new InitTest();
    System.out.println(test.i);
    }
}
```

Running this program when compiled with earlier versions of `javac` produces the incorrect output:

```
100
```

Running this program when compiled with the 1.2 `javac` produces the correct output:

```
0
```

Single class examples can be formulated as well:

```
public class InitTest2 {
    public int j = method();
    public int i = 0;
    public int method() {
    i = 100;
        return 200;
    }
    public static void main(String[] args) {
        InitTest test = new InitTest();
        System.out.println(test.i);
    }
}
```

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

12. 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 JDK 1.2, the compiler did not enforce this rule correctly, granting access if the member or constructor was declared to permit access without regard to the accessibility of the type to which it belongs. Thus the following illegal program would be accepted:

TABLE 3-1 File `CMain.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);
    }
}
```

TABLE 3-2 File `pack1/P1.java`

```
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 JDK 1.1, a member of a class or interface may be another class or interface, as well as a field or method. The change to the enforcement of the JLS 6.6.1 accessibility rules described here has been extended to inner classes as well.

13. Prior to JDK 1.2, 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. Even though the methods have the same name, no overriding occurs. As an example, compiling these files:

TABLE 3-3 `one/Parent.java`

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

TABLE 3-4 `two/Child.java`

```
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 {
                 ^
```

14. Beginning with JDK 1.2, the compiler properly detects duplicate, nested labels. These are disallowed by the JLS. Therefore, the following example statement is illegal:

```
sameName:
while (condition) {
    sameName:
    while (condition2) {
    break sameName;
    }
}
```

15. Beginning with JDK 1.2, the compiler properly detects a labeled declaration. These are disallowed by the JLS. This is a fix for bug 4039843.

16. Beginning with JDK 1.2, the compiler more strongly enforces the restriction that a package cannot contain a type and a subpackage of the same name. This 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 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. Some examples of classes which will now fail to compile:

```
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, that is, invoke their `finalize` methods, somewhat aggressively. In particular, there are some situations in which all `finalize`-able but `unfinalize`d objects would be `finalize`d at the end of nearly every garbage-collection cycle. Thus code written for these systems may unintentionally depend on prompt finalization for correct operation.

   In JDK 1.2, finalization is not performed directly by the garbage collector; instead, objects are finalized by a low-priority thread. This means that, in a busy program, the time between the moment when an object becomes finalizable and the moment when it is actually `finalize`d may be longer than with previous versions of the runtime system.

   In JDK 1.2, the runtime system properly implements the definition of finalization in the JLS, so this difference is not, strictly speaking, an incompatibility. This change may, however, cause programs to malfunction if they rely on 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 invoke the `Runtime.runFinalization` method at regular intervals.

2. The `String` hash function implemented in 1.1 releases did not match the function specified in the first edition of the JLS. In fact, the specified function is not implementable, 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. (This is due to the "sampling" nature of the function for strings over 15 characters.)

To bring the implementation into accord with the specification, and to fix the performance problems, the specification and implementation have been modified. The new `String` hash function as of JDK 1.2 is specified as:

```
s[0] * 31^(n-1) + s[1] * 31^(n-2) + ... + s[n-1]
```

where `s[i]` is the *i*th character of string `s`.

This change should have no effect on the overwhelming majority of applications. If an application has persistent data that depends on actual `String` hash values, it could theoretically be affected, but 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 will be unaffected. (Bug 4045622)

3. In JDK 1.2, an unimplemented abstract method or interface method will cause an `AbstractMethodError` to be raised at run time when the method is invoked. In previous versions, the error occurred during link time.

4. Prior to 1.2, putting code on `CLASSPATH` could make it more privileged. For example, pre-1.2 VMs would sometimes not require verification of trusted class files. Verification depended on the installed security manager. Changes in 1.2 may cause the following related compatibility problems:

   - Some pre-1.2 compilers had bugs that caused unverifiable class files to be generated, particularly with inner classes. In 1.2, all class files are verified. Such classes would always have failed to run as untrusted applets, but this was not noticed by users running only Java. These classes now fail on all 1.2 VMs.

     Such a problem can often be worked around by either recompiling with the 1.2 `javac`. - invoking java with the `-Xverify:none` option, or redeclaring affected `private` methods and fields to have default (package-private) access.

     For example, if, prior to 1.2, the following method was in a class on `CLASSPATH`, and was called by an applet, then it could read the `user.name property`:

```
String getUser() {
    return System.getProperty("user.name");
}
```

In JDK 1.2, such code must use a `begin/endPrivileged block`:

```
String getUser(){
    try {
    java.security.AccessController.beginPrivileged();
    return System.getProperty("user.name");
    }
    finally {
        java.security.AccessController.endPrivileged();
        }
    }
}
```

API Incompatibilities

1. In JDK 1.2, the class `ActiveEvent` is in the package `java.awt`. In JDK 1.1, `ActiveEvent` was in the package `java.awt.peer`.

2. The `stringFlavor` and `plainTextFlavor` fields of class `java.awt.datatransfer.DataFlavor` have been made `final` in JDK 1.2:

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

   In JDK 1.1, these fields were not `final`.

3. Interface `java.util.List` has been added to JDK 1.2. Existing source code may now produce a namespace conflict between `java.awt.List` and `java.util.List`.

   In JDK 1.2, using the 'wildcard' import statements

   ```
   import java.awt.*;
   import java.util.*;
   ```

   together will cause a compiler error for code that contains the unqualified name `List`. To work around this problem, either add an `import` statement such as

   ```
   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.

4. In JDK 1.2, the field `CHAR_UNDEFINED` in class `java.awt.event.KeyEvent` has the value 0x0ffff. In versions 1.1.x, 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`.

5. In JDK 1.2, the signature of the `java.io.StringReader.ready` method has been changed to allow an `IOException` to be thrown if the `StringReader` has been closed. This change was required in order to make the `StringReader` class properly implement the general contract spelled out in the abstract class `java.io.Reader`, which it extends.

Packaging Incompatibilities

1. In Solaris JDK 1.2, the Java virtual machine implementation and runtime support (*runtime*) are installed in `libjvm.so`. In JDK 1.1, the runtime for Solaris resided in `javai.dll` and `libjava.so`, respectively. The JDK 1.1 shared libraries contained both the runtime and the native methods for some of the class libraries. In JDK 1.2, we have isolated the runtime into a separate binary product; the native methods of the class libraries have been moved to `java.dll` and `libjava.so`. This change is important to users who have either

   - Written non-JNI native methods with the old Native Method Interface (NMI) that was supported by JDK 1.0, or
   - Embedded the runtime via the JNI Invocation Interface.

   If you fall into either of these categories, you must relink your native libraries before they can be used with JDK 1.2. On Solaris, you will have to replace `-ljava` in your link command line with `-ljvm`. Note that this change does not affect JNI programmers implementing native methods.

   If you are an NMI user trying to recompile or relink your libraries in JDK 1.2, please refer to bug 4100539. This bug is fixed in JDK 1.2.

   As of JDK 1.2, Sun supports JNI only. JNI is the standard way to make your native libraries interoperate 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 "NMI to JNI Transition Issues" on page 19 for information about migrating your native methods to JNI.

Tools Incompatibilities

1. In JDK 1.2, the `javac -O` option has a different meaning, and may have different performance effects on generated code, than the previous version. `javac -O` in 1.2 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 will 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 1.2 class files. 1.1 versions of `javac` sometimes generate incorrect inner class attributes. In JDK 1.2, `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. Class files generated by the 1.2 compiler will work on older VMs.

3. In JDK 1.2, the `javakey` tool has been replaced with the `keytool`, `PolicyTool`, and `jarsigner` tools. See the JDK 1.2 security documentation for descriptions of the new tools.

4. In 1.1, `javap -verify` performed a partial verification of the class file. This option was misleading because it did not perform many portions of a full verification. The option has been removed in 1.2.

5. The Java interpreter now has a set of standard command-line options and a set of non-standard options. The standard set is supported in JDK 1.2. The non-standard options may not be supported in future Solaris Software JVMs. See descriptions of the Solaris options.

6. This is an incompatibility between JDK 1.2 and versions prior to 1.1.4. Prior to 1.1.4, the Java interpreter would allow 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 JDK 1.2, the fully qualified class name must be specified. For example, to invoke the class `a.b.c` at `a/b/c.class`, the command `java a.b.c` could be issued from the parent directory of directory `/a`.

7. Due to bugs in versions 1.1.x, code that is signed using the 1.1.x `javakey` tool (now obsolete) will be recognized as unsigned in JDK 1.2. Code signed using JDK 1.2 will be recognized as unsigned on 1.1.x versions.

8. Prior to 1.2, `javac` permitted some inconsistent or redundant combinations of command line options. For example, `-classpath` could be specified multiple times with only the last usage taking effect. In 1.2 such usage is no longer allowed.

9. Prior to 1.2, the `-classpath` option of the Java interpreter had been used to set the search path used by the VM when loading system classes. The `java.class.path` property was then automatically set by the VM to reflect this path. Application classes were typically invoked directly from the system class path without an associated class loader.

   In the new model, applications are now started in a special "application" class loader in order to take advantage of both installed extensions and the new security model. The `-classpath` option has been changed to set the classpath that is used by the application class loader for loading classes and resources. Similarly, the `java.class.path` property has been changed to reflect this path. The system classpath used internally by the VM can still be overridden with a new `-Xsysclasspath` option if necessary. In most cases, though, the system classpath should never have to be changed.

   Most applications should not be affected by this change, except that the `java.class.path` property will no longer include those directories and `JAR` files used for loading system classes. Read the new `java.sys.class.path` property for that information. The only applications that might be adversely affected by this change are those that install their own security managers which make assumptions that may no longer be valid in the new model. These applications will have to be rewritten for 1.2, but in the meantime, the `-Xsysclasspath` switch for is provided for backward compatibility.

10. Beginning with JDK 1.2, the `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 JDK 1.2, the filename is

    `package-java.io.html`.

Serialization Incompatibilities

1. Because of problems with the format for `Externalizable` objects, it was necessary to make an incompatible change in JDK 1.2. A 1.1.5 (or earlier) program throws a `StreamCorruptedException` when it tries to read an `Externalizable` object in the new format. Programs based on JDK 1.1.6 (or later) do not have this problem.

   A new JDK 1.2 API supports backward compatibility. To write streams in the old format, call

   ```
   ObjectOutputStream.useProtocolVersion(PROTOCOL_VERSION_1)
   ```

   before doing any writes.

CHAPTER 4

NMI to JNI Transition Issues

NMI is not supported in JDK 1.2 for Solaris. The JNI interfaces are verified by the Java Compatibility Kit™ (JCK).

Note – NMI is known as 'JNI 1.0' , and JNI as 'JNI 1.1' in some installations.

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 using NMI to using JNI interfaces appears to be relatively straightforward. To illustrate, 6000 lines of C code were ported in approximately two days by an engineer who was unfamiliar with the application. The NMI was only lightly used, but the uses were widely distributed.

Running With Two JVMs

Because there are some incompatibilities between the JNI and applications written using the NMI, refer to Chapter 3 "JDK 1.2 Backwards Compatibility" on page 5. Recognizing that the removal of NMI support is an incompatible change, provisions have been made to allow both JDK 1.1 and 1.2 to reside on Solaris systems.

The `/usr/java` symbolic link points to the JDK that is the default on that system. Solaris Software's JDK 1.1 currently installs in /usr/java1.1. Solaris Software's JDK 1.2 will install in /usr/java1.2. The `/usr/java` symbolic link will continue to point to 1.1 after installation of the JDK 1.2 unless manually changed.

To make the JDK 1.2 the default, do the following:

As root,

```
# /usr/bin/mv /usr/java /usr/java.oldlink
# /usr/bin/ln -s /usr/java1.2 /usr/java
```

When you do this, applications using NMI will fail. However, keeping both JDKs on the system makes the workaround supportable and relatively easy to deal with.

To restore JDK 1.1 as the default JDK , do the following:

As root,

```
# /usr/bin/mv /usr/java.oldlink /usr/java
```

javah

The `javah` command generates C header and source files that are needed to implement native methods. The generated header and source files are used by C programs to reference instance variables of an object from native source code. JNI does not require header information or stub files. The `javah` command can still be used 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.

Since the NMI interfaces are completely incompatible with the JDK 1.2 implementation, JDK 1.2 will not support the production of NMI-style output via the `-old` flag. Rather the `-old` flag will be parsed and, if found, an `-old not supported` message will be output and `javah` will exit.

Command Line Differences

JVM-specific Options

Java Software's JDK 1.2 divides options into two groups according to whether a given option is specific to the particular JVM at hand or applicable to all JVMs. This division manifests itself in the option syntax. In the 1.2 scheme, JVM-specific options all start with `-X`, for example, `-Xdebug` to enable debugging. `-X` by itself produces a help message that names all the JVM-specific options that the implementation at hand accepts.

Solaris Software will support this new scheme. `-X` options currently accepted by Solaris Software:

```
% java -X
```

TABLE 4-1 Currently Accepted -X Options

`-Xsysclasspath`	set system class path
`-Xdebug`	enable remote debugging
`-Xnoclassgc`	disable class garbage collection
`-Xss`<*size*>	set maximum native stack size for any thread
`-Xoss`<*size*>	set maximum Java stack size for any thread
`-Xms`<*size*>	set initial Java heap size
`-Xmx`<*size*>	set maximum Java heap size
`Xprof[:`<*file*>|*file*=<*file*>`,type=inst|cpu,format =perthread|old]`	output profiling data to java.prof or <file>
`-Xverifyheap`	verify heap integrity before and after garbage collection
`-Xverify[:all|remote|none]`	verify all remote (default), or no classes
`-Xdescribe`	describe the memory system
`-Xmaxjitcodesize`<*size*>	set the maximum size in bytes for the JIT code area

The `-X` options are non-standard and subject to change without notice.

You'll note that many of these options correspond to 1.1.x options. More on this in the next section.

Also, although these -X options are nominally JVM-specific, many of them are sufficiently generic that any reasonably useful JVM would provide them. Both Java Software's and Solaris Software's JDK 1.2 support everything in the list above except for the last two entries. Solaris Software does not support these two -X options:

-Xhprof[:*file*=<*file*>,depth=<*n*>]	output heap profile to java.hprof.txt or <file>
-Xnoasyncgc	disable asynchronous garbage collection

Option Compatibility

The following JDK 1.1 compatibility options are currently allowed:

TABLE 4-2 JDK 1.1 Compatibility Options

-verbosegc
-verbosejni
-t
-tm
-debug
-noasyncgc
-noclassgc
-verify
-verifyremote
-noverify
-prof[:<*file*>]
-cs
-checksource
-ss
-oss
-ms
-mx
-l

−new vs java _old

The scheme currently implemented in Java Software's JDK 1.2 to accomplish migration from the 1.1.x command syntax to the 1.2 syntax is to add a "-new" flag. When set, other options become available and the compatibility forms listed above are disabled. (In addition to its syntactic effect, -new affects the semantics of the -classpath and -cp options.) Compare:

% java -help

Usage: java [-options] class [args...]

where options include:

`−classpath` *<directories and zip/jar files separated by :>*	set search path for system classes
`−D`*<name>*`=`*<value>*	set a system property
`−v −verbose[:`*class*`|gc|jni]`	enable verbose output
`−version`	print product version
`? −help`	- print this help message
`−X`	print help on non-standard options

to:

`java −new −help`

Usage: `java [-options] class [args...]`	(to execute a class)
or `java −jar [-options] jarfile [args...]`	(to execute a JAR file)

where options include:

`-cp −classpath` *<directories and zip/jar files separated by :>*	set search path for application classes (ignored with -jar option)
`-usepolicy[:`*file*`]`	enable policy checking with optional policy file
`−D`*<name>*`=`*<value>*	set a system property
`−v −verbose[:`*class*`|gc|jni]`	enable verbose output
`−version print product version`	-? -help print this help message
`−X`	print help on non-standard options