

# *Java™ Product Versioning Specification*

Author: Roger.Riggs@East.Sun.COM

Date: 7/16/03



beta draft

Evolution in open distributed systems must be managed carefully because correct operation depends on a great number of dependencies between packages. The impact of changes within a distributed system has a significant impact on users, support organizations, web administrators, and developers. Packages within a distributed system must operate correctly given only partial knowledge about the state of the whole system. The difficulty is increased because the packages of the system must be able to evolve at different rates. Evolution in such a system is made possible by explicitly managing the dependencies between the packages using techniques of object oriented design to govern how individual packages evolve. The Java language defines packages that are a natural for the consistent unit of update, they expose only public interfaces and consume only public interfaces of other classes.

Copyright 1996, 1997 Sun Microsystems, Inc.  
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.  
All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. (SUN) hereby grants to you a fully-paid, nonexclusive, nontransferable, perpetual, worldwide limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice this specification. This license allows and is limited to the creation and distribution of clean room implementations of this specification that (i) include a complete implementation of the current version of this specification without subsetting or supersetting, (ii) implement all the interfaces and functionality of the standard java.\* packages as defined by SUN, without subsetting or supersetting, (iii) do not add any additional packages, classes or methods to the java.\* packages (iv) pass all test suites relating to the most recent published version of this specification that are available from SUN six (6) months prior to any beta release of the clean room implementation or upgrade thereto, (v) do not derive from SUN source code or binary materials, and (vi) do not include any SUN binary materials without an appropriate and separate license from SUN.

#### RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

#### TRADEMARKS

Sun, the Sun logo, Sun Microsystems, JavaBeans, JDK, Java, HotJava, the Java Coffee Cup logo, Java WorkShop, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK® is a registered trademark of Novell, Inc.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

For further information on Intellectual Property matters contact Sun Legal Department:  
Trademarks, Jan O'Dell at 415-786-8191

# *Java™ Product Versioning*

*1*

## *1.1 Introduction*

In any system support must be provided for the system to evolve over time. Most existing systems have conventions and mechanisms that specify how change is accommodated. These systems have been based on the model where software programs are installed on a computer. Typically, developers have been able to specify what versions of other packages are required and the installation process has helped verify and configure the system.

In open distributed systems the static assumptions of existing systems does not work and evolution is more difficult because it is not possible to control how or when the packages change and correct operation depends on a greater number of dependencies between packages. What is needed is an updated set of conventions and mechanisms that specify how the packages of a system should evolve so that the goal of an open reliable scalable distributed system can be achieved.

This document specifies:

- how classes, resources, and files that make up a Java package are versioned. Packages define the consistent unit that can be developed, packaged, verified, updated, and distributed. Per package manifest information identifies the contents of the package;
- products are distributed by putting packages into archive files. Archives include a manifest to identify the product version and packages it contains;
- the standards and conventions used by developers and administrators to build and deploy products that operate reliably as their packages and the packages they depend on are upgraded.

## 1.2 Requirements

The impact of changes within a distributed system has a significant impact on end users, support organizations, web administrators, and developers.

- End Users
- Product Support Organizations
- Webmasters and Administrators
- Product Developers

Each of these groups has different requirements on products deployed on the net that evolve over time.

### *Users*

For users there is a need to build confidence that Java based products will be reliable and compatible over time. The reluctance to upgrade needs be addressed by building confidence in the Write Once Run Anywhere philosophy. With Java it should no longer be the norm that users will complain “if I upgrade it will break something” or “I won't be able to read/write data others can use”.

- They need to know that upgrading will not break either other programs or will obsolete existing data or produce data unusable by others.
- At the simplest level users want to know if the features they need are in the product version they have and what version to ask for to get particular features.
- More knowledgeable users keep track of what bugs are in what product version so they can work around them or avoid them.

### *Product support*

Product organizations rely on being able to easily and correctly identify the product that is being used, the environment in which it is being used, and the integrity of the product packaging.

- Databases of known problems and solutions are indexed by product identification information.

- The interoperation of products and packages can introduce new kinds of problems and require all of the packages in a system to be identified. Problems can originate from public interfaces that are under-specified, implementations that do not conform to the specification, or clients that use implementation specific details that are not part of the specification.

### *Webmasters and Administrators*

Webmasters, administrators, and service providers need to deploy applications for their clients via the web or network filesystems in a way that is reliable and supportable.

- The staff of these organizations must be able to support their sites, identifying problems with individual packages and interactions between packages.
- Site configuration must be able to support the scaling up of sites with automated site management tools.
- Installing updating packages must not present a risk to the correction operation of existing packages or active users.

### *Product Developers*

Product developers need to know how to write and deploy applications and libraries that satisfy the requirements of the users, administrators and support personnel. They must be able to make products and packages that:

- can operate correctly in the open dynamic environment of the web,
- can be upgraded without breaking compatibility with clients,
- can take advantage of upgrades in the packages they rely upon,
- can take advantage of dynamic extensions of their packages,
- can identify the packages they rely on for reporting of problems,
- are packaged to support the needs of users, webmasters and support organizations,
- can have known packages and combinations that satisfy the auditing and security requirements appropriate for the application and organization.

## 1.3 *Problems of Evolution in Distributed Systems*

In open distributed systems many problems can occur when the packages evolve and are updated independently. If the specified behavior inherent in the use of public interfaces are not maintained the system may fail in unexpected ways. Open systems are made up of many packages from different companies and organizations. These organizations operate asynchronously, introducing and upgrading their products on their own schedules. The distribution of those upgraded products takes time and adoption is not universal.

In Java, the components of local and distributed systems rely on the public interfaces and contracts for the behavior of other packages. Those packages will themselves evolve over time. In order for a package to operate correctly the packages it depends on must continue to provide the expected behavior even though they have been updated.

In distributed systems only partial consistency is possible, since it is impossible to have knowledge of the entire state of the system. Each process and package of the system has its own partial view of the current state of the system, accumulated incrementally by requesting information from other parts of the distributed system. Each piece of information whether from an applet that was started, a class that was loaded, a remote method invoked, or a web page retrieved must be treated carefully so that it can be used consistently with the rest of that partial view.

Many kinds of errors could result from inconsistencies in the classes that are loaded, including class verification errors, classes could compute incorrectly but without recognizable errors, or user requested functions could exhibit arbitrary failures.

Typical problems include:

- If an applet is running and has loaded only some of its classes when the web server is updated with a new version. When the applet incrementally loads more classes they may be inconsistent with the classes already loaded.
- If an application is using libraries from multiple websites and has loaded only some of the classes it needs, the libraries might be updated raising the potential of incompatibilities that the applet or user is left to detect.
- If an application or applet is running and makes a RMI call that returns an object for which the class needs to be loaded. The class that is loaded could be inconsistent with respect to the other classes already loaded.

- If an application or applet is running and makes a RMI call that returns an object that is for a newer or older version of the class.
- Bugs may exist in a library, if the clients have worked around the bug a cascade of problems may be introduced when the bug is fixed.

These problems cannot be prevented or solved directly because they arise out of inconsistencies between dynamically loaded packages that are not under the control of a single system administrator and so cannot be addressed by current configuration management techniques.

## *1.4 Design for Evolution*

The key to dealing with these problems and meeting the requirements stated above is the careful design of the packages and packaging of the system so that they may be updated, distributed, and loaded in consistent units. Typical to mass produced products is the notion of the field replaceable unit. It is the smallest unit of a product that can be identified with a specification, a supplier, can be distributed and redistributed, and can be replaced if faulty. This same model is used for software distribution, products have a name, a version number, adhere to one or more specifications, are distributed on the network or cdrom and its problems can be reported to support organizations. These packages are the smallest unit that can be distributed, used, validated and replaced or upgraded when necessary. Packages can be assembled with other packages and each package can still be identified, verified, and distributed.

The Java language based package mechanism fits well with the idea of a replaceable unit. Java packages expose only public interfaces and use only the public interfaces of other packages. The Java Language Specification define the approaches for compatible evolution of packages.

### *1.4.1 Java Language Specification on Backwards Compatibility*

The Java Language Specification lays the groundwork for developing packages that can be expected to evolve gracefully over time. It defines how classes can change and still be backward compatible with other classes previously compiled and distributed. Essential to robust evolution is the stability of the public, protected, and package interfaces and behavior as the implementations evolve. It defines “compatible” changes as those changes that do not change existing interfaces or behavior. Thus, if a class defines a method, and the method had a particular behavior, that same contract must be supported by the

all later evolutions of the class. Detailed rules are given in Chapter 13 of the Java Language Specification. One additional incompatible change has been added; it is incompatible to add methods to a public interface.

Incompatible changes are not permitted, but new or similar functionality can always be added in new or existing interfaces or classes.

By choosing the Java package as the unit of update the package and private methods of the classes may change allowing flexibility in the implementation of the package while the public and protected classes and methods maintain the external interfaces and semantics.

### *1.4.2 Object Serialization Specification on Backwards Compatibility*

Robust persistent storage and robust communication between the components is important to distributed systems. Components must be able to maintain persistent storage as they evolve, being able to evolve classes and yet have them read data previously written to storage. Components in a distributed system evolve at different rates and must still be able to reliably communicate.

Adhering to the compatibility requirements of object serialization allows newer and older versions to communicate in a predictable and consistent way. The details are in Chapter 5 of the Java™ Object Serialization Specification.

## *1.5 Package Version Specification*

There are several categories of artifacts that need to be identified including specifications, implementation, the Java Virtual Machine and Java Runtime Environment.

### *1.5.1 Specification Versioning*

Open systems are based on the idea that a specification may have multiple implementations. Specifications evolve under the auspices of an organization or company. It is highly undesirable if a specification has multiple incompatible versions. Each version of a specification or implementation must evolve only into a single subsequent version. The philosophy of requiring specifications to be backward compatible allows specifications to be identified as supersets of the previous specification. Since there is a single sequence of

version specifications they can meaningfully be identified by version numbers with specific semantics that imply the ordering. Specification version numbers use a Dewey decimal notation consisting of numbers separated by periods.

A specification is identified by the:

- Owner of the specification
- Name of the Specification
- Version number - major.minor.micro; major version numbers identify significant functional changes, minor version numbers identify smaller extensions to the functionality, micro versions are even finer grained versions. These version numbers are ordered with larger numbers specifying additions to the specification.

### *1.5.2 Virtual Machine Versioning*

An implementation of the Java Virtual Machine should be identify both the specification and the implementation. These properties should be added to those already available using `java.lang.System.getProperties`.

- `java-vm.specification.version` i.e. 1.2
- `java-vm.specification.vendor` i.e. Sun Microsystems Inc.
- `java-vm.specification.name` i.e. Java™ Virtual Machine Specification
- `java-vm.version` i.e. Solaris 5.5 Native 1.0 build32
- `java-vm.vendor` i.e. Sun Microsystems Inc.
- `java-vm.name` i.e. Solaris 5.x JVM

These properties are accessed using the method `java.lang.System.getProperty` and each returns a string.

### *1.5.3 Version Identification of the Java Runtime*

The requirement to identify the Java Runtime is already partially met via the properties specified by the Java Language Specification, §20.18.7 using `java.lang.System.getProperties`.

- `java.version` i.e. Solaris 1.2
- `java.vendor` i.e. Sun Microsystems Inc.

Currently these identify the implementation of the Java runtime and the core classes that are available. These properties do *not* identify the Java Language Specification version that this JDK implements.

Additional properties are needed to identify the version of the Java Runtime Environment specification that this implementation adhere's to. The properties are:

- `java.specification.version` i.e. 1.1
- `java.specification.name` i.e. Java™ Language Specification
- `java.specification.vendor` i.e. Sun Microsystems Inc.

These properties are accessed using the method `java.lang.System.getProperty` and return their values as strings.

### 1.5.4 Package Versioning

Each Java package is made up of class files plus optional resource files. The information needed to identify the contents of the package is stored with the package contents.

This specification applies to all packages regardless of whether they are developed as a core package distributed with a Java Runtime, a standard extension, an applet or application package.

Unlike version numbers for specifications version information for implementations *cannot* be used to identify the package as being backward compatible with earlier versions. Package version numbers are present to identify differences between the specification and the implementation, i.e. bugs. New versions of implementations are specifically produced to remove (bad or incorrect) behavior and thus are intended *not* to be backward compatible. Thus package version strings can have any unique value and can only be compared for equality. For a complete explanation of the rationale, see 1.5.10, “Rationale for limiting Implementation version numbers to identity”.

The following attribute names are defined for a package. The value of each attribute is a string:

- `Package-Title` Title of the package
- `Package-Version` Version number
- `Package-Vendor` Vendors company or organization
- `Specification-Title` Title of the specification
- `Specification-Version` Version number
- `Specification-Vendor` Vendors company or organization

These attributes are stored in the manifest and retrieved by programs using the `java.lang.Package` API described below.

## 1.5.5 API to Package Version Information

The `java.lang.Package` class provides a object to locate and access information about the package.

Package objects are created explicitly by class loaders and should be created before the first class in the package is defined. The attributes of each package are stored in the manifest and are retrieved by the classloader.

```
package java.lang;
public class Package {

    // Return the name of this package.
    public String getName();

    // Return the title of the specification of this package.
    public String getSpecificationTitle();

    // Return the version of the specification of this package.
    public String getSpecificationVersion();

    // Return the vendor of the specification of this package.
    public String getSpecificationVendor();

    // Return the title of the implementation of this package.
    public String getImplementationTitle();

    // Return the version of the implementation of this package.
    public String getImplementationVersion();

    // Return the vendor of the implementation of this package.
    public String getImplementationVendor();

    // Is this package is compatible with the requested version
    public boolean isCompatibleWith(String desired);

    // Get the Package for the named class
    public static Package getPackage(String classname);

    // Return the packages for currently loaded classes.
    public static Package[] getAllPackages();

    // Return true if this package is equal to another object.
    public boolean equals(Object obj);
```

```
    // Return the hashCode for this object
    public int hashCode();

    // Return the string describing this package.
    public String toString();
}
```

The `getName` method returns this package's name, for example, `java.lang`.

The `getSpecificationTitle` method return this packages specification title if it is known and null otherwise.

The `getSpecificationVersion` method return this the version number of the specification this package implements. Null is returned if the version is not known.

The `getSpecificationVendor` returns the organization, group or vendor that owns the specification.

The `getImplementationTitle` method return this packages implementation title if it is known and null otherwise.

The `getImplementationVersion` method return this the version number of the implementation this package implements. Null is returned if the version is not known.

The `getImplementationVendor` returns the organization, group or vendor that owns this implementation.

The `isCompatibleWith` method returns true if this package's specification version number is compatible with the desired version number. True is returned if this packages specification version number is greater than the supplied version string. A version string is a series of positive numbers separated by periods. The numbers are compared component by component from left to right. If any number is greater than the corresponding number of the supplied string the method returns true. If the number is less than it returns false. If the corresponding numbers are equal the next number is examined.

The `getPackage` method locates the package for the class by name. The current class loader is consulted to map the package name to the package object in that class loader. It returns the package object containing the attributes for the package. Null is returned if the package information has not yet been loaded or if no package information was defined by the classloader.

The `getAllPackages` method will return an array of the packages known to the current classloader. It includes the packages of both the system and classloaded classes. It does not identify all packages available to be loaded by the classloader. It only identifies those packages for which the classloader has provided information.

The `equals` method returns true if this package has the same name and classloader as the object passed in.

The `hashCode` method returns a hashcode consistent with definition of `equals` as required by the Java Language Specification.

The `toString` method returns a string consisting of “package” and the package name. If available the specification title and specification version number are appended to the string.

### 1.5.6 *java.lang.Class Additions*

A method has been added to `java.lang.Class` to get the package for this class.

```
package java.lang;
public class Class {
    ...
    public Package getPackage();
    ...
}
```

### 1.5.7 *java.lang.ClassLoader Additions*

In order to support Packages the classloader is extended to keep track of the mapping from classes to packages and to allow classloaders to define the Package instances for the classes they load. The additional methods are defined to allow subclasses to define packages in this classloader to allow the Package implementation to get information about packages defined by this classloader.

The `java.lang.Package` implementation needs to identify the current classloader in order to call it from system code.

```
package java.lang;
public class ClassLoader {
    ...
    // Return the non-null classloader of callers
    public static ClassLoader currentClassLoader();
}
```

```

// Define a Package
protected Package(String pkgname,
                  String spectitle, String specversion,
                  String specvendor, String impltitle,
                  String implversion, String implvendor);

...
}

```

The `currentClassLoader` method is used to find the current `ClassLoader` even if called from a system class. When called from a classloader loaded class it will return the equivalent of `this.getClass().getClassLoader()`. It's behavior is identical to the current `SecurityManager.currentClassLoader` method but is public.

The protected access `definePackage` method is used by subclasses to define the packages of the classes it is loading. Packages with a given name may only be defined once and must be defined before the first class of that package is loaded. The classloader should supply the versioning attributes from the manifest if they are available.

### 1.5.8 JAR Manifest Format

The current manifest format is extended to allow the specification of the attributes for package versioning information. A manifest entry should be created for each Java package. The name of the entry will be the directory within the archive that contains the package's class and resource files. For example:

```

Manifest-Version: 1.0

Name: java/util/
Specification-Title: "Java Utility Classes"
Specification-Version: "1.2"
Specification-Vendor: "Sun Microsystems Inc."
Package-Title: "java.util"
Package-Version: "build57"
Package-Vendor: "Sun Microsystems. Inc."

```

These attributes can be inserted in the manifest by creating a prototype manifest file and using the `-m` switch of the `jar` tool to merge them into the manifest when it is built. `JarTool` will be extended to browse and set the versioning attributes in the manifest.

### 1.5.9 How Users Know What is Running

Users need to be able to report the identities of the packages in use when bugs occur. It's up to the application, applet or browser to expose the available information to the user on demand or when an error occurs. API's are available to allow the following to be reported:

- What are the packages loaded?  
The `package.getAllPackages` method will return the active packages.
- What are the package versions?  
The `java.lang.Package` methods allow the attributes for names and versions to be examined as listed in 1.5.4, "Package Versioning".
- What version of the Java Runtime is active?  
The `System.getProperties` method can be used to get the properties of this virtual machine listed in 1.5.3, "Version Identification of the Java Runtime".
- What version of the Java VM is active?  
The `System.getProperties` method can be used to get the properties of this virtual machine listed in 1.5.2, "Virtual Machine Versioning".

### 1.5.10 Rationale for limiting Implementation version numbers to identity

Implementations evolve independently over time to fix bugs, improve performance or add new functions called for by later revisions of the specifications. Packages implement specifications and must identify which version of each specification they implement. Interactions occur between packages only through their public and protected interfaces and classes. It is the public api and behavior that must remain stable over time so that changes can be allowed in the implementation of one package without affecting the behavior of another package.

If the classes of a package *always* faithfully implemented the specification it would be sufficient just to identify the specification. Since in the real world this rarely happens packages need to identify themselves so that bugs can be reported against the packages that may have contributed to the problem.

There is a significant tendency to try to attach some significance to version identifiers of implementations. If the purpose is to allow the tracking of bugs then a unique number is sufficient. It is also sufficient for a client package to workaround a bug in a particular version of a vendors package since that version can be tested for and the bug avoided.

However, many additional problems can occur when one package attempts to work around bugs in other packages. They need to identify behavior that is not part of the specification and may try to use behavior that is only part of one implementation. Such implementation specific behavior cannot be relied upon to be in any particular version other than the one(s) seen and tested by the developer.

A bug first appears in some version of a vendors package and may or may not continue to be a problem in subsequent versions. If the client of the buggy package uses a work around based on version numbers it could correctly work around the bug in the specific version. Now, if the buggy package was fixed, how would the client package know whether the bug was fixed or not. If it assumed that higher versions still contained the bug it would still try to work around the bug. The work around itself might not work correctly with the non-buggy package. This could cause a cascade of bugs caused by fixing a bug. Only the developer, through testing with a new version, can determine whether or not the workaround for a bug is still necessary or whether it will cause problems with the correctly behaving package. The developer only knows that the bug exists in a particular individual versions.

## *1.6 Documenting How to Develop*

This section should discuss each aspect of product development and distributions giving direction on how to achieve a robust evolvable product.

- Read the JLS, be backward compatible
- Read Serialization Specification, be backward compatible
- Develop on a platform that includes all of the functionality required.
- Test for new functionality not included in each previous version of the platform specification and fall back or put up sensible messages
- Author the package and product versioning information and create the necessary files
- Depending on the archive model the entire contents will need to be signed and the manifests be created
- Test on oldest platform
- Distribute only in archives with manifests to insure consistency and integrity
- Update only whole java packages or whole archive files