The Platform Environment

An application runs in a platform environment, defined by the underlying operating system, the Java virtual machine, the class libraries, and various configuration data supplied when the application is launched. This lesson describes some of the APIs an application uses to examine and configure its platform environment. The lesson consists of three sections:

- Configuration Utilities describes APIs used to access configuration data supplied when the application is deployed, or by the application's user.
- System Utilities describes miscellaneous APIs defined in the System and Runtime classes.
- PATH and CLASSPATH describes environment variables used to configure JDK development tools and other applications.

Configuration Utilities

Properties

Properties are configuration values managed as key/value pairs. In each pair, the key and value are both String values. The key identifies, and is used to retrieve, the value, much as a variable name is used to retrieve the variable's value.

Example: an application capable of downloading files might use a property named "download.lastDirectory" to keep track of the directory used for the last download.

To manage properties, create instances of java.util.Properties. This class provides methods for the following:

- loading key/value pairs into a Properties object from a stream,
- retrieving a value from its key,
- listing the keys and their values,
- enumerating over the keys, and
- saving the properties to a stream

- testing to see if a particular key or value is in the Properties object,
- getting the current number of key/value pairs,
- removing a key and its value,
- adding a key/value pair to the Properties list,
- enumerating over the values or the keys,
- retrieving a value by its key, and
- finding out if the Properties object is empty.

Properties extends java.util.Hashtable. Some of the methods inherited from Hashtable support the following actions:

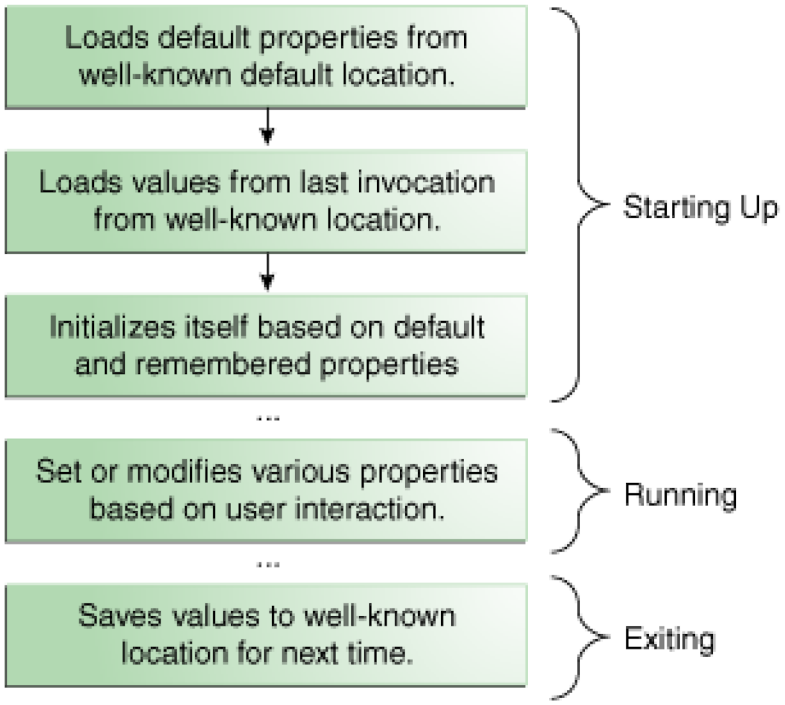
Security Considerations: Access to properties is subject to approval by the current security manager. The example code segments in this section are assumed to be in standalone applications, which, by default, have no security manager. The same code in an applet may not work depending on the browser in which it is running. See What Applets Can and Cannot Do in the Java Applets lesson for information about security restrictions on applets.

The System class maintains a Properties object that defines the configuration of the current working environment. More about these properties at System Properties.
The remainder of this section explains how to use properties to manage application configuration.

The Platform Environment

Properties in the Application Life Cycle

The following figure illustrates how a typical application might manage its configuration data with a Properties object over the course of its execution.



`Starting Up`

- The actions given in the first three boxes occur when the application is starting up. First, the application loads the default properties from a well-known location into a Properties object. Normally, the default properties are stored in a file on disk along with the .class and other resource files for the application.

Next, the application creates another Properties object and loads the properties that were saved from the last time the application was run. Many applications store properties on a per-user basis, so the properties loaded in this step are usually in a specific file in a particular directory maintained by this application in the user's home directory. Finally, the application uses the default and remembered properties to initialize itself.

`Running`

- The key here is consistency. The application must always load and save properties to the same location so that it can find them the next time it's executed.

During the execution of the application, the user may change some settings, perhaps in a Preferences window, and the Properties object is updated to reflect these changes. If the users changes are to be remembered in future sessions, they must be saved.

`Exiting`

- Upon exiting, the application saves the properties to its well-known location, to be loaded again when the application is next started up.

The Platform Environment

Setting Up the Properties Object

The following Java code performs the first two steps described in the previous section: loading the default properties and loading the remembered properties:

```
. . .
// create and load default properties
Properties defaultProps = new Properties();
FileInputStream in = new FileInputStream("defaultProperties");
defaultProps.load(in);
in.close();

// create application properties with default
Properties applicationProps = new Properties(defaultProps);

// now load properties
// from last invocation
in = new FileInputStream("appProperties");
applicationProps.load(in);
in.close();
. . .
```

First, the application sets up a default Properties object. This object contains the set of properties to use if values are not explicitly set elsewhere. Then the load method reads the default values from a file on disk named defaultProperties.

Next, the application uses a different constructor to create a second Properties object, applicationProps, whose default values are contained in defaultProps. The defaults come into play when a property is being retrieved. If the property can't be found in applicationProps, then its default list is searched.

Finally, the code loads a set of properties into applicationProps from a file named appProperties. The properties in this file are those that were saved from the application the last time it was invoked, as explained in the next section.

Saving Properties

The following example writes out the application properties from the previous example using Properties.store. The default properties don't need to be saved each time because they never change.

```
FileOutputStream out = new FileOutputStream("appProperties");
applicationProps.store(out, "---No Comment---");
out.close();
```

The store method needs a stream to write to, as well as a string that it uses as a comment at the top of the output.

The Platform Environment

Configuration Utilities

A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.

The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run. For example, suppose a Java application called Sort sorts lines in a file. To sort the data in a file named friends.txt, a user would enter:

```
java Sort friends.txt
```

When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings. In the previous example, the command-line arguments passed to the Sort application in an array that contains a single String: "friends.txt".

Echoing Command-Line Arguments

The Echo example displays each of its command-line arguments on a line by itself:

```
public class Echo {
    public static void main (String[] args) {
        for (String s: args) {
            System.out.println(s);
        }
    }
}
```

The following example shows how a user might run Echo. User input is in italics.

```
java Echo Drink Hot Java
Drink
Hot
Java
```

Note that the application displays each word — Drink, Hot, and Java — on a line by itself. This is because the space character separates command-line arguments. To have Drink, Hot, and Java interpreted as a single argument, the user would join them by enclosing them within quotation marks.

```
jjava Echo "Drink Hot Java"
Drink Hot Java
```

Parsing Numeric Command-Line Arguments

If an application needs to support a numeric command-line argument, it must convert a String argument that represents a number, such as "34", to a numeric value. Here is a code snippet that converts a command-line argument to an int:

```
int firstArg;
if (args.length > 0) {
    try {
        firstArg = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Argument" + args[0] + " must be an integer.");
        System.exit(1);
    }
}
```

parseInt throws a NumberFormatException if the format of args[0] isn't valid. All of the Number classes — Integer, Float, Double, and so on — have parseXXX methods that convert a String representing a number to an object of their type.

The Platform Environment

Configuration Utilities

Many operating systems use environment variables to pass configuration information to applications. Like properties in the Java platform, environment variables are key/value pairs, where both the key and the value are strings. The conventions for setting and using environment variables vary between operating systems, and also between command line interpreters. To learn how to pass environment variables to applications on your system, refer to your system documentation.

Querying Environment Variables

On the Java platform, an application uses System.getenv to retrieve environment variable values. Without an argument, getenv returns a read-only instance of java.util.Map, where the map keys are the environment variable names, and the map values are the environment variable values. This is demonstrated in the EnvMap example:

```
import java.util.Map;

public class EnvMap {
    public static void main (String[] args) {
        Map<String, String> env = System.getenv();
        for (String envName : env.keySet()) {
            System.out.format("%s=%s%n",
                              envName,
                              env.get(envName));
        }
    }
}
```

With a String argument, getenv returns the value of the specified variable. If the variable is not defined, getenv returns null.

The Env example uses getenv this way to query specific environment variables, specified on the command line:

```
public class Env {
    public static void main (String[] args) {
        for (String env: args) {
            String value = System.getenv(env);
            if (value != null) {
                System.out.format("%s=%s%n",
                                  env, value);
            } else {
                System.out.format("%s is"
                    + " not assigned.%n", env);
            }
        }
    }
}
```

Passing Environment Variables to New Processes

When a Java application uses a ProcessBuilder object to create a new process, the default set of environment variables passed to the new process is the same set provided to the application's virtual machine process. The application can change this set using ProcessBuilder.environment.

Platform Dependency Issues

There are many subtle differences between the way environment variables are implemented on different systems. For example, Windows ignores case in environment variable names, while UNIX does not. The way environment variables are used also varies. For example, Windows provides the user name in an environment variable called USERNAME, while UNIX implementations might provide the user name in USER, LOGNAME, or both.

To maximize portability, never refer to an environment variable when the same value is available in a system property. For example, if the operating system provides a user name, it will always be available in the system property user.name

Configuration Utilities

Summary

The Preferences API allows applications to store and retrieve configuration data in an implementation-dependent backing store. Asynchronous updates are supported, and the same set of preferences can be safely updated by multiple threads and even multiple applications. For more information, refer to the Preferences API Guide.

An application deployed in a JAR archive uses a manifest to describe the contents of the archive. For more information, refer to the Packaging Programs in JAR Files lesson.
The configuration of a Java Web Start application is contained in a JNLP file. For more information, refer to the Java Web Start lesson.

The configuration of a Java Plug-in applet is partially determined by the HTML tags used to embed the applet in the web page. Depending on the applet and the browser, these tags can include <applet>, <object>, <embed>, and <param>. For more information, refer to the Java Applets lesson.

The class java.util.ServiceLoader provides a simple service provider facility. A service provider is an implementation of a service — a well-known set of interfaces and (usually abstract) classes. The classes in a service provider typically implement the interfaces and subclass the classes defined in the service. Service providers can be installed as extensions (see The Extension Mechanism). Providers can also be made available by adding them to the class path or by some other platform-specific means.

The Platform Environment

System Utilities

Command-Line I/O Objects

System provides several predefined I/O objects that are useful in a Java application that is meant to be launched from the command line. These implement the Standard I/O streams provided by most operating systems, and also a console object that is useful for entering passwords.

System Properties

In Properties, we examined the way an application can use Properties objects to maintain its configuration. The Java platform itself uses a Properties object to maintain its own configuration. The System class maintains a Properties object that describes the configuration of the current working environment. System properties include information about the current user, the current version of the Java runtime, and the character used to separate components of a file path name.

The following table describes some of the most important system properties

Key	Meaning
`"file.separator"`	Character that separates components of a file path. This is "/" on UNIX and "\" on Windows.
`"java.class.path"`	Path used to find directories and JAR archives containing class files. Elements of the class path are separated by a platform-specific character specified in the `path.separator` property.
`"java.home"`	Installation directory for Java Runtime Environment (JRE)
`"java.vendor"`	JRE vendor name
`"java.vendor.url"`	JRE vendor URL
`"java.version"`	JRE version number
`"line.separator"`	Sequence used by operating system to separate lines in text files
`"os.arch"`	Operating system architecture
`"os.name"`	Operating system name
`"os.version"`	Operating system version
`"path.separator"`	Path separator character used in `java.class.path`
`"user.dir"`	User working directory
`"user.home"`	User home directory
`"user.name"`	User account name

Security consideration: Access to system properties can be restricted by the Security Manager. This is most often an issue in applets, which are prevented from reading some system properties, and from writing any system properties.

Reading System Properties

The System class has 2 methods used to read system properties:

`getProperty` and `getProperties`

The System class has two different versions of getProperty. Both retrieve the value of the property named in the argument list. The simpler of the two getProperty methods takes a single argument, a property key For example, to get the value of path.separator, use the following statement:

```
System.getProperty("path.separator");
```

The getProperty method returns a string containing the value of the property. If the property does not exist, this version of getProperty returns null.

The other version of getProperty requires two String arguments: the first argument is the key to look up and the second argument is a default value to return if the key cannot be found or if it has no value. EG: the following invocation of getProperty looks up the System property called subliminal.message. This is not a valid system property, so instead of returning null, this method returns the default value provided as a second argument: "Buy StayPuft Marshmallows!"

```
System.getProperty("subliminal.message", "Buy StayPuft Marshmallows!");
```

The last method provided by the System class to access property values is the getProperties method, which returns a Properties object. This object contains a complete set of system property definitions.

System Utilities

Writing System Properties

To modify the existing set of system properties, use System.setProperties. This method takes a Properties object that has been initialized to contain the properties to be set. This method replaces the entire set of system properties with the new set represented by the Properties object.

Warning: Changing system properties is potentially dangerous and should be done with discretion. Many system properties are not reread after start-up and are there for informational purposes. Changing some properties may have unexpected side-effects.

PropertiesTest, creates a Properties object and initializes it from myProperties.txt .

```
subliminal.message=Buy StayPuft Marshmallows!
```

PropertiesTest then uses System.setProperties to install the new Properties objects as the current set of system properties.

```
import java.io.FileInputStream;
import java.util.Properties;

public class PropertiesTest {
    public static void main(String[] args)
        throws Exception {

        // set up new properties object
        // from file "myProperties.txt"
        FileInputStream propFile =
            new FileInputStream( "myProperties.txt");
        Properties p =
            new Properties(System.getProperties());
        p.load(propFile);

        // set the system properties
        System.setProperties(p);
        // display new properties
        System.getProperties().list(System.out);
    }
}
```

Note how PropertiesTest creates the Properties object, p, which is used as the argument to setProperties:

```
Properties p = new Properties(System.getProperties());
```

This statement initializes the new properties object, p, with the current set of system properties, which in the case of this small application, is the set of properties initialized by the runtime system. Then the application loads additional properties into p from the file myProperties.txt and sets the system properties to p. This has the effect of adding the properties listed in myProperties.txt to the set of properties created by the runtime system at startup. Note that an application can create p without any default Properties object, like this:

```
Properties p = new Properties();
```

Also note that the value of system properties can be overwritten! For example, if myProperties.txt contains the following line, the java.vendor system property will be overwritten: `java.vendor=Acme Software Company`

be careful not to overwrite system properties.

The setProperties method changes the set of system properties for the current running application. These changes are not persistent. That is, changing the system properties within an application will not affect future invocations of the Java interpreter for this or any other application. The runtime system re-initializes the system properties each time its starts up. If changes to system properties are to be persistent, then the application must write the values to some file before exiting and read them in again upon startup.

System Utilities

The Security Manager

A security manager is an object that defines a security policy for an application. This policy specifies actions that are unsafe or sensitive. Any actions not allowed by the security policy cause a SecurityException to be thrown. An application can also query its security manager to discover which actions are allowed.
Typically, a web applet runs with a security manager provided by the browser or Java Web Start plugin. Other kinds of applications normally run without a security manager, unless the application itself defines one. If no security manager is present, the application has no security policy and acts without restrictions.

Interacting with the Security Manager

The security manager is an object of type SecurityManager; to obtain a reference to this object, invoke System.getSecurityManager. `SecurityManager appsm = System.getSecurityManager();`

If there is no security manager, this method returns null.

Once an application has a reference to the security manager object, it can request permission to do specific things. Many classes in the standard libraries do this. For example, System.exit, which terminates the Java virtual machine with an exit status, invokes SecurityManager.checkExit to ensure that the current thread has permission to shut down the application.
The SecurityManager class defines many other methods used to verify other kinds of operations. For example, SecurityManager.checkAccess verifies thread accesses, and SecurityManager.checkPropertyAccess verifies access to the specified property. Each operation or group of operations has its own checkXXX() method.
In addition, the set of checkXXX() methods represents the set of operations that are already subject to the protection of the security manager. Typically, an application does not have to directly invoke any checkXXX() methods.

Recognizing a Security Violation

Many actions that are routine without a security manager can throw a SecurityException when run with a security manager. This is true even when invoking a method that isn't documented as throwing SecurityException. For example, consider the following code used to write to a file: `reader = new FileWriter("xanadu.txt");`

```
appletviewer fileApplet.html
    Exception in thread "AWT-EventQueue-1"
java.security.AccessControlException: access denied (java.io.FilePermission
xanadu.txt write)
        at
java.security.AccessControlContext.checkPermission(AccessControlContext.java:32
3)
        at
java.security.AccessController.checkPermission(AccessController.java:546)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
        at java.lang.SecurityManager.checkWrite(SecurityManager.java:962)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:169)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:70)
        at java.io.FileWriter.<init>(FileWriter.java:46)
```

Note that the specific exception thrown in this case, java.security.AccessControlException, is a subclass of SecurityException.

System Utilities

Miscellaneous Methods in System

This section describes some of the methods in System that aren't covered in the previous sections.

The arrayCopy method efficiently copies data between arrays.

The currentTimeMillis and nanoTime methods are useful for measuring time intervals during execution of an application. To measure a time interval in milliseconds, invoke currentTimeMillis twice, at the beginning and end of the interval, and subtract the first value returned from the second. Similarly, invoking nanoTime twice measures an interval in nanoseconds.

Method Detail

getInstance

```
public static Calendar getInstance()
```

Gets a calendar using the default time zone and locale. The `Calendar` returned is based on the current time in the default time zone with the default `FORMAT` locale.

Returns:
`a Calendar.`

getInstance

```
public static Calendar getInstance(TimeZone zone)
```

Gets a calendar using the specified time zone and default locale. The `Calendar` returned is based on the current time in the given time zone with the default `FORMAT` locale.

Parameters:
`zone - the time zone to use`

Returns:
`a Calendar.`

Note: The accuracy of both `currentTimeMillis` and `nanoTime` is limited by the time services provided by the operating system. Do not assume that `currentTimeMillis` is accurate to the nearest millisecond or that `nanoTime` is accurate to the nearest nanosecond. Also, neither `currentTimeMillis` nor `nanoTime` should be used to determine the current time. Use a high-level method, such as `java.util.Calendar.getInstance`.

The exit method causes the Java virtual machine to shut down, with an integer exit status specified by the argument. The exit status is available to the process that launched the application. By convention, an exit status of 0 indicates normal termination of the application, while any other value is an error code.

exit

```
public static void exit(int status)
```

Terminates the currently running Java Virtual Machine. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.

This method calls the `exit` method in class `Runtime`. This method never returns normally.

The call `System.exit(n)` is effectively equivalent to the call:

```
Runtime.getRuntime().exit(n)
```

Parameters:
`status - exit status.`

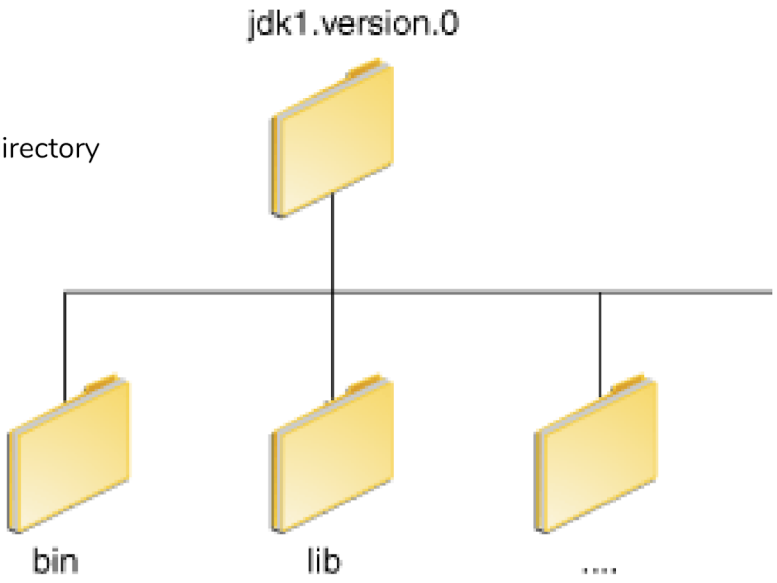
Throws:
`SecurityException - if a security manager exists and its checkExit method doesn't allow exit with the specified status.`

See Also:
`Runtime.exit(int)`

PATH and CLASSPATH

This section explains how to use the PATH and CLASSPATH environment variables on Microsoft Windows, Solaris, and Linux. Consult the installation instructions included with your installation of the Java Development Kit (JDK) software bundle for current information.

After installing the software, the JDK directory will have the structure shown here:



The bin directory contains both the compiler and the launcher.

Update the PATH Environment Variable (Microsoft Windows)

You can run Java applications just fine without setting the PATH environment variable. Or, you can optionally set it as a convenience.

Set the PATH environment variable if you want to be able to conveniently run the executables (javac.exe, java.exe, javadoc.exe, and so on) from any directory without having to type the full path of the command. If you do not set the PATH variable, you need to specify the full path to the executable every time you run it, such as:

```
C:\Java\jdk1.7.0\bin\javac MyClass.java
```

The PATH environment variable is a series of directories separated by semicolons (;). Microsoft Windows looks for programs in the PATH directories in order, from left to right. You should have only one bin directory for the JDK in the path at a time (those following the first are ignored), so if one is already present, you can update that particular entry.

The following is an example of a PATH environment variable:

```
C:\Java\jdk1.7.0\bin;C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem
```

It is useful to set the PATH environment variable permanently so it will persist after rebooting. To make a permanent change to the PATH variable, use the System icon in the Control Panel. The precise procedure varies depending on the version of Windows...

PATH and CLASSPATH

Update the PATH Environment Variable (Microsoft Windows)

Windows XP

1. Select Start, select Control Panel. double click System, and select the Advanced tab.
2. Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
3. In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable. Click OK. Close all remaining windows by clicking OK.

WindowsVista

1. From the desktop, right click the My Computer icon.
2. Choose Properties from the context menu.
3. Click the Advanced tab (Advanced system settings link in Vista).
4. Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
5. In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable. Click OK. Close all remaining windows by clicking OK.

Windows7

1. From the desktop, right click the Computer icon.
2. Choose Properties from the context menu.
3. Click the Advanced system settings link.
4. Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
5. In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable. Click OK. Close all remaining windows by clicking OK.

PATH and CLASSPATH

Update the PATH Variable (Solaris and Linux)

You can run the JDK just fine without setting the PATH variable, or you can optionally set it as a convenience. However, you should set the path variable if you want to be able to run the executables (javac, java, javadoc, and so on) from any directory without having to type the full path of the command. If you do not set the PATH variable, you need to specify the full path to the executable every time you run it, such as:

```
% /usr/local/jdk1.7.0/bin/javac MyClass.java
```

To find out if the path is properly set, execute:

```
% java -version
```

This will print the version of the java tool, if it can find it. If the version is old or you get the error java: Command not found, then the path is not properly set.

To set the path permanently, set the path in your startup file.

For C shell (csh), edit the startup file (~/.cshrc):

```
set path=(/usr/local/jdk1.7.0/bin $path)
```

For bash, edit the startup file (~/.bashrc):

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
export PATH
```

For ksh, the startup file is named by the environment variable, ENV. To set the path:

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
export PATH
```

For sh, edit the profile file (~/.profile):

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
export PATH
```

Then load the startup file and verify that the path is set by repeating the java command:

For ksh, bash, or sh:

```
% . /.profile
% java -version
```

For C shell (csh):

```
% source ~/.cshrc
% java -version
```

Checking the CLASSPATH Variable (All platforms)

The CLASSPATH variable is one way to tell applications, including the JDK tools, where to look for user classes. (Classes that are part of the JRE, JDK platform, and extensions should be defined through other means, such as the bootstrap class path or the extensions directory.)

The preferred way to specify the class path is by using the -cp command line switch. This allows the CLASSPATH to be set individually for each application without affecting other applications. Setting the CLASSPATH can be tricky and should be performed with care.

The default value of the class path is ".", meaning that only the current directory is searched. Specifying either the CLASSPATH variable or the -cp command line switch overrides this value.

To check whether CLASSPATH is set on Microsoft Windows NT/2000/XP, execute the following:

```
C:> echo %CLASSPATH%
```

On Solaris or Linux, execute the following:

```
% echo $CLASSPATH
```

f CLASSPATH is not set you will get a CLASSPATH: Undefined variable error (Solaris or Linux) or simply %CLASSPATH% (Microsoft Windows NT/2000/XP).

To modify the CLASSPATH, use the same procedure you used for the PATH variable.

Class path wildcards allow you to include an entire directory of .jar files in the class path without explicitly naming them individually. For more information, including an explanation of class path wildcards, and a detailed description on how to clean up the CLASSPATH environment variable, see the Setting the Class Path technical note.