Lesson: 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.

Runtime System (Environment)

A run-time system (also called runtime system or just runtime) is software designed to support the execution of computer programs written in some computer language. The run-time system contains implementations of basic low-level commands and may also implement higher-level commands and may support type checking, debugging, and even code generation and optimization. Some services of the run-time system are accessible to the programmer through an application programming interface, but other services (such as task scheduling and resource management) may be inaccessible.

Distinguish this from Development Environments and Build Environments. You will tend to find a hierarchy here. Run time environment - Everything you need to execute a program, but no tools to change it.

Build environment- Given some code written by someone, everything you need to compile it or otherwise prepare an executable that you put into a Run time environment. Build environments are pretty useless unless you can see tests what you have built, so they often include Run too. In Build you can't actually modify the code. Development environment - Everything you need to write code, build it and test it. Code Editors and other such tools. Typically also includes Build and Run.

I am working on compilers and the Runtime environment means ,it is the structure of the target computers registers and memory that serves to manage memory and maintain information needed to guide the execution process. infact, almost all programming languages use one of three kinds of runtime environment,whose essential structure does not depend on the specific details of the target machine.

These three kind of runtime environmenta are

1. fully static environment(example **FORTRAN77**)
2. stack-based environment(**C,C++**)
3. Fully dynamic environment(**LISP**)

In **computers**, a **utility** is a small program that provides an addition to the capabilities provided by the operating system. In some usages, a **utility** is a special and nonessential part of the operating system. The print "**utility**" that comes with the operating system is an example.

1) Configuration Utilities

This section describes some of the configuration utilities that help an application access its startup context.

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. For 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.

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

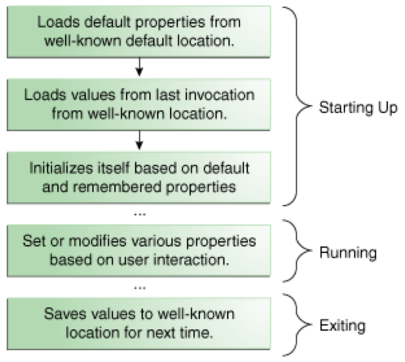
- ➢ testing to see if a particular key or value is in the Properties object, getting the current number of key/value pairs, removing a key and its value, adding a key/value pair to the Properties list, enumerating over the values or the keys, retrieving a value by its key, and finding out if the Properties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

- **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. For more about these properties, see System Properties. **The remainder of this section explains how to use properties to manage application configuration**.

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.



Loads default properties from well-known default location.
Loads values from last invocation from well-known location.
Initializes itself based on default and remembered properties
...
Set or modifies various properties based on user interaction.
...
Saves values to well-known location for next time.
Starting Up
Running
Exiting

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.

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.

Running: 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.

Setting Up the Properties Object

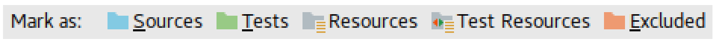
The following Java code performs the first two steps described in the previous section: loading the default properties and loading the remembered properties:

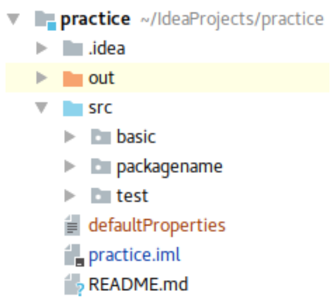
```
. . .
// create and load default properties
Properties defaultProps = new Properties();
FileInputStream in = new FileInputStream("defaultProperties");
defaultProps.load(in);
in.close();

// 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.





✔ In Intellij, the code worked, when I created a "defaultProperties" file in the <u>project root folder</u>. **<u>It does NOT work when the text files will be for example in the src folder</u>**. File → Proj Structure → Modules

```
Properties defaultProps = new Properties();
FileInputStream in = new FileInputStream( s: "defaultProperties");
defaultProps.load(in);
in.close();
```

```
Properties properties = new Properties();
try {
  properties.load(new FileInputStream("path/filename"));
} catch (IOException e) {
  ...
}
```

Iterate as:

```
for(String key : properties.stringPropertyNames()) {
  String value = properties.getProperty(key);
  System.out.println(key + " => " + value);
}
```

Saving Properties

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

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

The **`store`** <u>method needs a stream to write to, as well as a string that it uses as a comment at the top of the output</u>.

Getting Property Information

Once the application has set up its Properties object, the application can query the object for information about various keys and values that it contains. An application gets information from a Properties object after start up so that it can initialize itself based on choices made by the user. The Properties class has several methods for getting property information:

- `contains(Object value)` and `containsKey(Object key)` Returns true if the value or the key is in the Properties object. Properties inherits these methods from Hashtable. <u>Thus they accept Object arguments, but only String values should be used</u>.
- `getProperty(String key)` and `getProperty(String key, String default)` Returns the value for the specified property. The second version provides for a default value. If the key is not found, the default is returned.
- `list(PrintStream s)` and `list(PrintWriter w)` Writes all of the properties to the specified stream or writer. This is useful for debugging.
- `elements()`, `keys()`, and `propertyNames()` Returns an Enumeration containing the keys or values (as indicated by the method name) contained in the Properties object. The keys method only returns the keys for the object itself; the propertyNames method returns the keys for default properties as well.
- `stringPropertyNames()` Like propertyNames, but returns a Set<String>, and only returns names of properties where both key and value are strings. Note that the Set object is not backed by the Properties object, so changes in one do not affect the other.
- `size()` Returns the current number of key/value pairs.

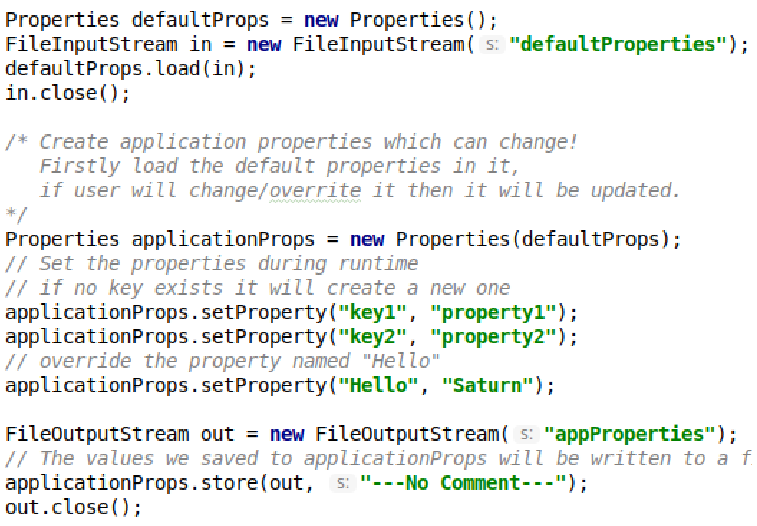
Setting Properties

A user's interaction with an application during its execution may impact property settings. These changes should be reflected in the Properties object so that they are saved when the application exits (and calls the store method). The following methods change the properties in a Properties object:

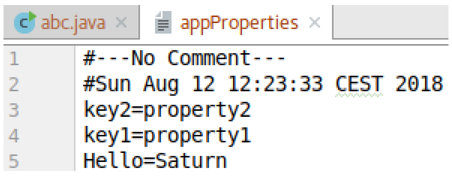
- `setProperty(String key, String value)`
  Puts the key/value pair in the `Properties` object.
- `remove(Object key)`
  Removes the key/value pair associated with key.

Note: Some of the methods described above are defined in `Hashtable`, and thus accept key and value argument types other than `String`. Always use `String`s for keys and values, even if the method allows other types. Also do not invoke `Hashtable.set` or `Hastable.setAll` on `Properties` objects; always use `Properties.setProperty`.

- Because Properties inherits from **`Hashtable`**, the **`put`** and **`putAll`** methods can be applied to a Properties object. **Their use is strongly discouraged** as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail. Similarly, the call to the propertyNames or list method will fail if it is called on a "compromised" Properties object that contains a non-String key.



```
Properties defaultProps = new Properties();
FileInputStream in = new FileInputStream( s: "defaultProperties");
defaultProps.load(in);
in.close();

/* Create application properties which can change!
   Firstly load the default properties in it,
   if user will change/overrite it then it will be updated.
*/
Properties applicationProps = new Properties(defaultProps);
// Set the properties during runtime
// if no key exists it will create a new one
applicationProps.setProperty("key1", "property1");
applicationProps.setProperty("key2", "property2");
// override the property named "Hello"
applicationProps.setProperty("Hello", "Saturn");

FileOutputStream out = new FileOutputStream( s: "appProperties");
// The values we saved to applicationProps will be written to a f
applicationProps.store(out,  s: "---No Comment---");
out.close();
```



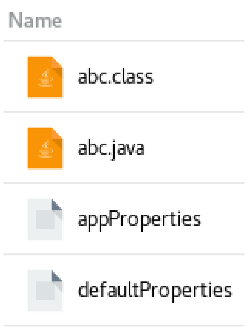
```
#---No Comment---
#Sun Aug 12 12:23:33 CEST 2018
key2=property2
key1=property1
Hello=Saturn
```

FileOutputStream: If there are no folders named "appProperties" it will create a new one.

◆ Don't use save! Use `store()`

`save(OutputStream out, String comments)`
Deprecated.

✔ **If compiled and run manually**, properties files **should be** (and will be created at) **in the same folder** where the .class files are.



```
$ javac abc.java
$ javac abc.java
$ java abc
```

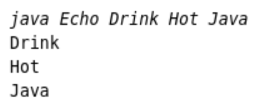
Command-Line Arguments

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.

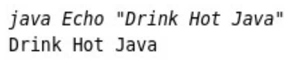
The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run. For example, suppose a Java application called Sort sorts lines in a file. To sort the data in a file named friends.txt, a user would enter: **`java Sort friends.txt`**

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".

Printing them with sout: 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.



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



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

Environment Variables

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:

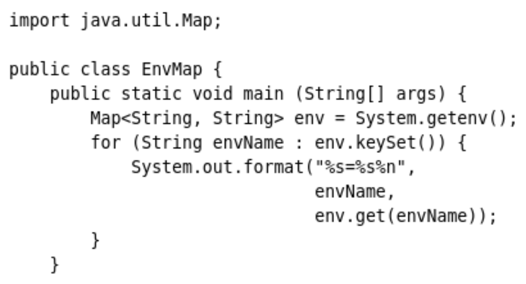
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin

SSH_AUTH_SOCK=/run/user/1000/keyring/ssh

HOME=/home/can

SHELL=/bin/bash...

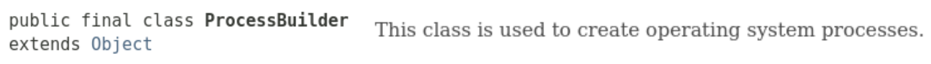
With a **String** argument, **getenv** returns the value of the specified variable. If the variable is not defined, **getenv** returns **null**. The Env example uses getenv this way to query specific environment variables, specified on the command line



```
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));
        }
    }
```

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.



```
public final class ProcessBuilder
extends Object
```

This class is used to create operating system processes.

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. Eğer system property'de varsa, env variable'ı ismi ile çağırma. Sys prop kullan.

Other Configuration Utilities

Here is a summary of some other configuration utilities.

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.

2) System Utilities

The **`System`** class implements a number of system utilities. Some of these have already been covered in the previous section on Configuration Utilities. This section covers some of the other 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. For more information, refer to I/O from the Command Line in the Basic I/O lesson.

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

- Because AMD originally created the 64-bit extensions to the x86 architecture, it is named also “**amd64**”. The most used name is “**x86_64**”.

```
public static void main (String[] args) {
  System.out.println(System.getProperty("os.arch"));
  // prints "amd64"
}
```

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. For more on accessing system properties in applets, refer to System Properties in the Doing More With Java Rich Internet Applications lesson.

Reading System Properties

Reading System Properties The System class has two 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. For example, 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 (all) system property definitions.

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.

Note that the value of system properties can be overwritten! In general, 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.

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.

This section explains how an application interacts with an existing security manager. For more detailed information, including information on how to design a security manager, refer to the Security Guide.

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 read a file:

```
reader = new FileReader("xanadu.txt");
```

In the absence of a security manager, this statement executes without error, provided xanadu.txt exists and is readable. But suppose this statement is inserted in a web applet, which typically runs under a security manager that does not allow file input. The following error messages might result:

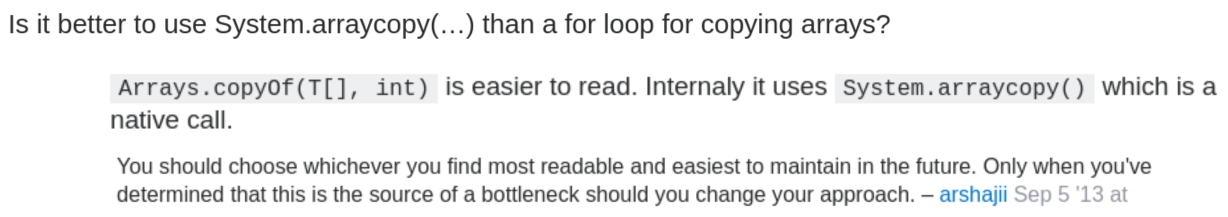
```
appletviewer fileApplet.html
Exception in thread "AWT-EventQueue-1" java.security.AccessControlException: access denied (
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
        at java.security.AccessController.checkPermission(AccessController.java:546)
        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.

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. For more information, refer to Arrays in the Language Basics lesson.


Is it better to use System.arraycopy(...) than a for loop for copying arrays?

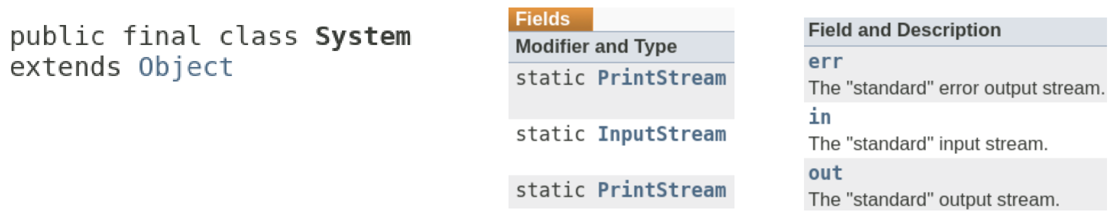
`Arrays.copyOf(T[], int)` is easier to read. Internaly it uses `System.arraycopy()` which is a native call.

You should choose whichever you find most readable and easiest to maintain in the future. Only when you've determined that this is the source of a bottleneck should you change your approach. – arshajii Sep 5 '13 at

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.

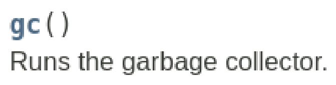
- Note: The accuracy of both **currentTimeMillis** and **nanoTime** is limited by the time services provided by the operating system. Do not assume that currentTimeMillis is accurate to the nearest millisecond or that nanoTime is accurate to the nearest nanosecond. Also, neither currentTimeMillis nor 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.

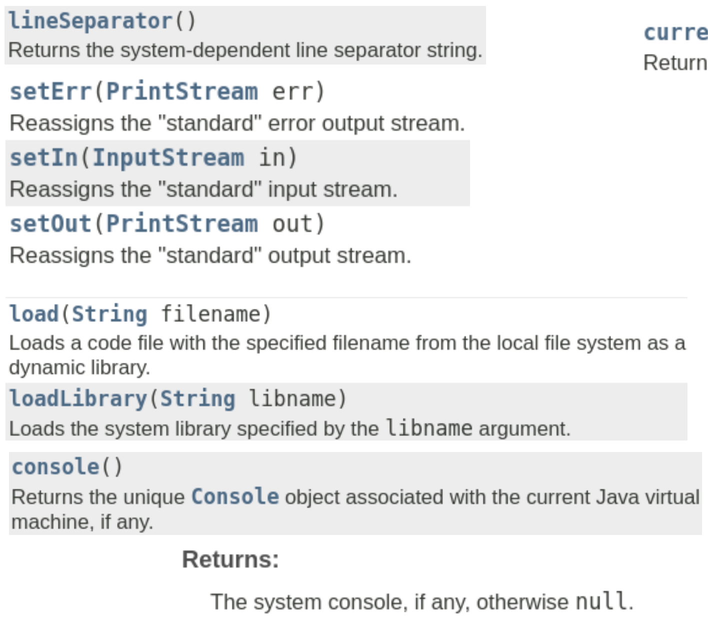

```
public final class System
extends Object
```

Fields

Modifier and Type	Field and Description
static PrintStream	**err** The "standard" error output stream.
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.


`gc()`
Runs the garbage collector.

Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.


`lineSeparator()`
Returns the system-dependent line separator string.

`setErr(PrintStream err)`
Reassigns the "standard" error output stream.

`setIn(InputStream in)`
Reassigns the "standard" input stream.

`setOut(PrintStream out)`
Reassigns the "standard" output stream.

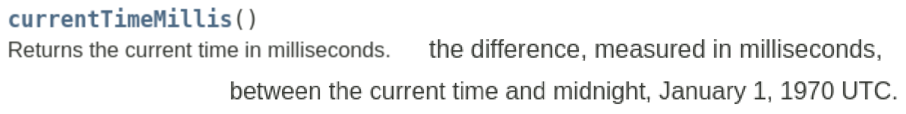
`load(String filename)`
Loads a code file with the specified filename from the local file system as a dynamic library.

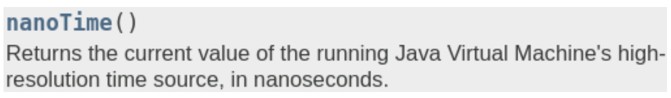
`loadLibrary(String libname)`
Loads the system library specified by the `libname` argument.

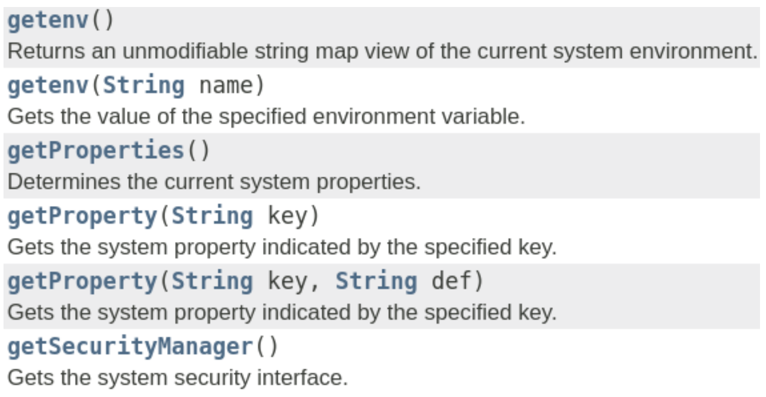
`console()`
Returns the unique `Console` object associated with the current Java virtual machine, if any.

Returns:

The system console, if any, otherwise `null`.


`currentTimeMillis()`
Returns the current time in milliseconds. the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.


`nanoTime()`
Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.


`getenv()`
Returns an unmodifiable string map view of the current system environment.

`getenv(String name)`
Gets the value of the specified environment variable.

`getProperties()`
Determines the current system properties.

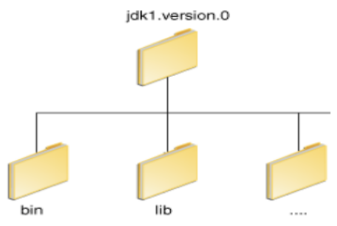
`getProperty(String key)`
Gets the system property indicated by the specified key.

`getProperty(String key, String def)`
Gets the system property indicated by the specified key.

`getSecurityManager()`
Gets the system security interface.

3) 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 below.



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:

Windows Vista:

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**.

Windows 7:

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**.

Note: You may see a PATH environment variable similar to the following when editing it from the Control Panel:

```
%JAVA_HOME%\bin;%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem
```

Variables enclosed in percentage signs (%) are existing environment variables. If one of these variables is listed in the **Environment Variables** window from the Control Panel (such as JAVA_HOME), then you can edit its value. If it does not appear, then it is a special environment variable that the operating system has defined. For example, SystemRoot is the location of the Microsoft Windows system folder. To obtain the value of a environment variable, enter the following at a command prompt. (This example obtains the value of the SystemRoot environment variable):

```
echo %SystemRoot%
```

Update the PATH Variable (Solaris and Linux)

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 C shell (`csh`):

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

For `ksh`, `bash`, or `sh`:

```
    % . /.profile
    % 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. The class path is the path that the Java Runtime Environment (JRE) searches for classes and other resource files. (The class path tells the JDK tools and applications where to find third-party and user-defined classes that are not extensions or part of the Java platform). (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 class path needs to find any classes you have compiled with the javac compiler. The **default** is the **current directory** to conveniently enable those classes to be found. The JDK, the JVM and other JDK tools find classes by searching the Java platform (bootstrap) classes, any extension classes, and the class path, in that order.

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.

Class libraries for most applications use the extensions mechanism. You only need to set the class path when you want to load a class that is (a) not in the current directory or in any of its subdirectories, and (b) not in a location specified by the extensions mechanism.

If you upgrade from an earlier release of the JDK, then your startup settings might include CLASSPATH settings that are no longer needed. You should remove any settings that are not application-specific, such as classes.zip. Some third-party applications that use the Java Virtual Machine (JVM) can modify your CLASSPATH environment variable to include the libraries they use. Such settings can remain.

You can change the class path by using the **-classpath** or **-cp** option of some Java commands when you call the JVM or other JDK tools or by using the CLASSPATH environment variable. See JDK Commands Class Path Options. Using the -classpath option is **preferred** over setting the CLASSPATH environment variable because you can set it individually for each application without affecting other applications and without other applications modifying its value. See CLASSPATH Environment Variable.

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.

```java
public class abc {
  public static void main (String[] args) throws Exception {
    UseMe a = new UseMe();
    System.out.println(a.use());
```

UseMe.java

```java
public class UseMe {
  public String use() {
    return "EVELE HÜVELE";
  }
}
```

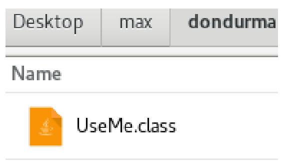
1. Eğer **abc.java** ile **UseMe.class** <u>aynı dosyada</u> ise `javac abc.java` ve sonra `java abc` olur.
   Default classpath **'.'** yani compiler ve jvm gerekli class fileları bulmak için aynı klasörün içini arıyorlar
2. Eğer **UseMe.class** <u>başka dosyada ise</u>: `javac -cp ':/home/can/Desktop/' abc.java` .Burda compilera söylediğimiz abc'de kullanılan class dosyalarını /home/can/Desktop'da bulabilirsin. Orayı ara. Çalıştırırken de class'ın nerde oldugunu söylemek lazım `java -cp ':/home/can/Desktop/' abc`
   Burda da class dosyalarını nerde olduğunu JVM'e söylüyoruz sanırım(?)
3. Eğer **UseMe.class**'ı **JAR** haline sokmak istiyorsak: `jar cf UseMe.jar UseMe.class`
4. JAR içindeki class dosyalarını kullanmak istiyorsak (ve eğer JAR içinde herhangi bir package yok ise ve sadece .class dosyaları varsa):
   - **JAR** ve **abc.java** aynı klasörde ise: `javac -cp ':UseMe.jar' abc.java` `java -cp ':UseMe.jar' abc`
   - **JAR** başka bir dosyada ise:

```
javac -cp ':/home/can/Desktop/Alle Module/UseMe.jar' abc.java
java -cp ':/home/can/Desktop/Alle Module/UseMe.jar' abc
```

Üstteki bütün adımlarda, **abc.java** içinde herhangi bir **`import`** yapmamıza GEREK YOK.

```java
package max.dondurma;

public class UseMe {
  public String use() {
    return "EVELE HÜVELE";
  }
```

```
javac UseMe.java
jar cf max.dondurma.jar max

javac -cp ':/home/can/Desktop/max.dondurma.jar' abc.java
java -cp ':/home/can/Desktop/max.dondurma.jar' abc
```



Burda **import** yazmak gerekli oluyor. `import max.dondurma.UseMe`;

Benim tahminim: Biz IDE içinde hangi jarlar projemize include olsun diyoruz ve IDE ler de dynamic olarak classpathları kendileri tasarlıyorlar. **-cp** vb.

Using bash, how do you make a classpath out of all files in a directory?

Details

Given a directory:

```
LIB=/path/to/project/dir/lib
```

that contains nothing but *.jar files such as:

```
junit-4.8.1.jar
jurt-3.2.1.jar
log4j-1.2.16.jar
mockito-all-1.8.5.jar
```

I need to create a colon-separated classpath variable in the form:

```
CLASSPATH=/path/to/project/dir/lib/junit-4.8.1.jar:/path/to/project/dir/lib/jurt-3.2.
```

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.

New Answer
(October 2012)

There's no need to manually build the classpath list. Java supports a convenient wildcard syntax for directories containing jar files.

```
java -cp "$LIB/*"
```

(Notice that the `*` is *inside* the quotes.)

Explanation from `man java`:

> As a special convenience, a class path element containing a basename of * is considered equivalent to specifying a list of all the files in the directory with the extension `.jar` or `.JAR` (a java program cannot tell the difference between the two invocations).
>
> For example, if directory foo contains `a.jar` and `b.JAR`, then the class path element `foo/*` is expanded to a `A.jar:b.JAR`, except that the order of jar files is unspecified. All jar files in the specified directory, even hidden ones, are included in the list. A classpath entry consisting simply of `*` expands to a list of all the jar files in the current directory. The `CLASSPATH` environment variable, where defined, will be similarly expanded. Any classpath wildcard expansion occurs before the Java virtual machine is started — no Java program will ever see unexpanded wildcards except by querying the environment.