

Java System Properties

Java programs do not read environment variables the way a C/C++ program would, using `getenv()`. However, an application can read System Properties, which provide information about the local system and configuration. When the java VM starts, it inserts local system properties into a system properties list. You can then use methods defined as part of the System class to access the value of these properties.

Note that for security reasons, applets are not allowed to access all system properties, as Java prevents a rogue applet from trying to learn too much about a local system. The method `System.getProperties` returns an object of type `Properties`, which is a class defined in the `java.util` package as extending a `Hashtable` type. So, the properties are kept in a `Hashtable`, which is accessed using a string key.

```
java.version  
java.vendor  
java.vendor.url  
java.home  
java.class.version  
java.class.path  
os.name  
os.arch  
os.version  
file.separator  
path.separator  
line.separator  
user.name  
user.home  
user.dir
```

Using System Properties

It is commonly the case that you need to determine the user name of the local user in an application, so you can easily access this information using the static `System.getProperty` method.

```
String user = System.getProperty("user.name");
```

Likewise, you can query the local system to find out what type of operating system is being used:

```
String platform = System.getProperty("os.name"); // might return "Solaris" or "Windows NT"
```

To get all the properties, you first get a `Properties` object, and then you can list them all by calling the `Properties.list` method giving it a `PrintStream` as an argument. For example:

```
import java.util.*;

public class Prop {

    public static void main(String[] args)
    {
        Properties p = System.getProperties();
        p.list(System.out);
    }
}
```

The result when this code is executed on Solaris vs Windows NT is shown on the next two slides.

Solaris System Properties for JDK 1.1.5

```
% java Prop
-- listing properties --
user.language=en
java.home=/lusr/java/bin/..
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
file.encoding.pkg=sun.io
java.version=1.1.5
file.separator=/
line.separator=

file.encoding=8859_1
java.vendor=Sun Microsystems Inc.
user.timezone=CST
user.name=lavender
os.arch=sparc
os.name=Solaris
java.vendor.url=http://www.sun.com/
user.dir=/v/hank/v43/lavender/courses/cs371/sp...
java.class.path=/u:/u/lavender:/u/lavender/cs371:/u/l...
java.class.version=45.3
os.version=2.x
path.separator=:
user.home=/u/lavender
```

Windows NT System Properties for JDK 1.2

```
-- listing properties --
java.specification.name=Java Platform API Specification
java.version=1.2
user.timezone=America/Chicago
java.specification.version=1.2
java.vm.vendor=Sun Microsystems Inc.
user.home=C:\WINNT\Profiles\Administrator
java.vm.specification.version=1.0
os.arch=x86
java.vendor.url=http://java.sun.com/
user.region=US
java.home=C:\jdk1.2\jre
java.class.path=C:\
line.separator=

java.io.tmpdir=C:\TEMP\
os.name=Windows NT
java.vendor=Sun Microsystems Inc.
java.library.path=C:\jdk1.2\bin;.;C:\WINNT\System32;C:\...
java.vm.specification.vendor=Sun Microsystems Inc.
user.language=en
user.name=Administrator
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
java.vm.name=Classic VM
java.vm.specification.name=Java Virtual Machine Specification
os.version=4.0
java.vm.version=1.2
java.vm.info=build JDK-1.2-V, native threads, syncjit
path.separator=;
file.separator=\
user.dir=C:\projects\cs371\jmail
```

Enumerating through System Properties

The `java.util` package has a type called an Enumeration type that can be used to enumerate over a Vector or a Hashtable. This is useful when processing the list of system properties.

```
public interface Enumeration {  
  
    // Tests if this enumeration contains more elements.  
    boolean hasMoreElements();  
  
    // Returns the next element of this enumeration.  
    Object nextElement();  
}
```

The Properties class defines a `propertyNames` method that you can use to retrieve an Enumeration of all the key name for the system properties.

```
public Enumeration propertyNames() {  
    Hashtable h = new Hashtable();  
    enumerate(h);  
    return h.keys();  
}  
  
Properties p = System.getProperties();  
Enumeration e = p.propertyNames();  
while (e.hasMoreElements()) {  
    String key = (String)e.nextElement();  
    System.out.println(key + "=" + p.getProperty(key));  
}
```

Finding the Local Internet Address and Hostname

There is no system property for the local host name, although there probably should be! Instead, you have to first obtain the Internet address for the local machine, and then obtain the string hostname using the Internet address object. The `InetAddress` class is found in the `java.net` package. The reason why the hostname is associated with the `InetAddress` object is because the system has to use DNS (Domain Name System) to dynamically look up host information. So, the java developers decided to put the methods to do this in the `InetAddress` class.

The steps are to first get the IP address of the local machine, and then use the address to get the hostname

```
InetAddress addr = InetAddress.getLocalHost();
try {
    String host = addr.getHostName();
}
catch (UnknownHostException e) { System.err.println(e); }
```

If the hostname is not found, then an “`UnknownHostException`” is thrown. You can ask any `InetAddress` object to convert itself to a `String` object:

```
String a = addr.toString();
```

Which is a simple formatted string method that results in a string formatted as “`hostname/%d.%d.%d.%d`”

```
public String toString() {
    return getHostName() + "/" + getHostAddress();
}
```

Other methods are useful to get the `InetAddress` of a remote host using its name.

```
try {
    InetAddress server = InetAddress.getByName("cs.utexas.edu");
}
catch (UnknownHostException e) { ... }
```

A Collector Class

It is often useful to have a class within a package that contains static variables and static methods that can be used by other classes in the same package. I call this a “collector class” since its job is to collect together information that is static and global to other objects in the same package.

```
package projects.cs371.jmail;

import java.net.*; // InetAddress class is defined in this package
import java.util.*; // Date class is defined in this package
import java.text.*; // DateFormat class is defined in this package

public class JMail {
    // static variables needed by other classes in this package
    static String userName;
    static String userHome;
    static String hostName;
    static String ipAddr;
    static String defaultEditor;
    static String platform;

    // get the current date/time using the FULL date/time format, including timezone
    // like: Wednesday, February 10, 1999 12:42:36 AM CST

    static public String getDateTimeString()
    {
        Date d = new Date();
        DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
        return df.format(d);
    }
}
```

Collector Class Continued

A static initializer is a convenient way to initialize all of the static information defined in the collector class

```
static
{
    // find out what platform we are running on and set the
    // default editor for that platform
    platform = System.getProperty("os.name");
    if (platform.startsWith("Windows"))
        defaultEditor = "notepad.exe";
    else // assume some flavor of Unix with vi editor
        defaultEditor = "/bin/vi";

    // login name of the current user by asking for the user.name property
    userName = System.getProperty("user.name");
    userHome = System.getProperty("user.home");

    try { // determine the local host's Internet DNS name
        InetAddress addr = InetAddress.getLocalHost();
        ipAddr = addr.toString();
        hostName = addr.getHostName();
        System.out.println("[JMail 1.0 running on " + ipAddr + "]);
        System.out.println();
    }

    catch (UnknownHostException h) {
        System.err.println("JMail: Unable to determine local host name");
        System.err.println(h);
    }
}
```

Creating a File for Reading/Writing

To work with files in Java, you need to use some of the classes in the `java.io` package. However, it can be a bit confusing. You first create a file object, using the `File` class. For example, to create a file called “msg.txt” in a user’s home directory, you might have something like:

```
fileobj = new File(JMail.userHome, "msg.txt");
```

This `File` class constructor takes a pathname and a filename, and concatenates them together using the value of the `file.separator` system property, which is `/` on Unix and `\` in Windows.

Once you have created a `File` object, which represents a filename, if that file does not exist, then you need to create it by passing the file object into a `RandomAccessFile` constructor, with the mode for the file. By default, it is read, but you can make it read/write by passing a mode string of “rw”. Note that you have to do this in a try block since an `IOException` might be thrown

```
PrintWriter out = null;
try {
    file = new RandomAccessFile(fileobj, "rw");
    out = new PrintWriter(new FileWriter(fileobj), true);
}
catch (IOException e) { System.err.println(e); }
```

Since we are going to want to write the file a line at a time, we would like a line buffered writer object associated with the file. That is, we want to automatically flush output to the file each time we call `out.println(someString)`. So, we need to construct a `FileWriter` object using the `File` object, and then use the `FileWriter` object to construct a `PrintWriter` that is line buffered. We indicate that the `PrintWrite` should be line buffered by passing a boolean argument value of `true` as the second argument. When doing most formatted output, you will want to use a `PrintWriter`, because that class has lots of methods for doing format output, which makes life easy. It’s then easy to write text to the file:

```
out.println("Date: " + JMail.getDateTimeString());
out.println("From: " + JMail.userName + "@" + JMail.hostName);
```

A Simple Editor Class

The following is a very simple Editor class, that either uses a default editor defined in a collector class, or allows the editor to be specified when passed into the constructor. The doEdit method takes a filename of a file to edit, and uses the `java.lang.Runtime` and `java.lang.Process` classes to arrange to “fork” a subprocess to run the editor. NOTE that this simple Editor class is independent of the platform that the code is written for. It works in conjunction with the JMail collector class, which isolates the platform specific aspects of the editor. In object oriented programming, this type of **separation of concerns** is very important for making software portable, maintainable, and safe.

```
package projects.cs371.jmail;

public class Editor {
    private String editor;

    // default constructor use the JMail default editor
    public Editor () { this.editor = JMail.defaultEditor; }

    // user can specify which editor to us
    public Editor (java.lang.String editor) { this.editor = editor; }

    // fork off the editor as a subprocess of the JVM
    public int doEdit(java.lang.String filename) {
        Runtime r = Runtime.getRuntime();
        String[] s = new String[] { editor, filename };
        try {
            Process p = r.exec(s);
            return p.waitFor();
        }
        catch (Exception e) { System.err.println(e); return -1; }
    }
}
```

Using System.in and System.out for I/O

```
package projects.cs371.jmail;
import java.io.*;

public class Main {
    private Message m;

    public static void main (String[] args)
    {
        BufferedReader rdr = new BufferedReader(new InputStreamReader(System.in));

        Message m = new Message(System.in);
        // get the header fields
        ... // m.setTo(), m.setCc, m.setBcc, m.setSubject
        // get the body
        System.out.println("-> Enter the message, followed by a CTRL-Z and RETURN:");
        System.out.println();
        m.setBody();

        // figure out whether or not to invoke the editor, default is N
        System.out.println("-> Do you wish to edit the message? (y/N)");
        try {
            String result = rdr.readLine();
            if (result.startsWith("y") || result.startsWith("Y"))
                m.edit();
            else
                m.save();
        }
        catch (IOException e) { System.err.println(e); System.exit(-1); }
    }
}
```

Using a “Driver” to Execute a Java Program

To execute a Java program, the CLASSPATH environment variable has to be set correctly. It is advisable to implement a driver script that will establish the environment for the program, and run the Java VM.

On Unix, using a shell script, you might have the following ‘jmail’ shell script (with execute permissions):

```
#!/bin/sh
# a Unix shell script called 'jmail'

# define CLASSPATH
CLASSPATH=/u/lavender
export CLASSPATH

# execute jmail main method, which is located in $CLASSPATH/projects/cs371/jmail
exec /usr/java/bin/java projects.cs371.jmail.Main
```

On Windows, you would write a short jmail.bat file:

```
@echo off
REM a BAT file to run the java VM to start the program

set CLASSPATH=C:\
REM execute jmail main method, which is located in C:\projects\cs371\jmail
C:\jdk1.2\bin\java projects.cs371.jmail.Main
```