Java Programming
Miscellaneous How-To

TABLE OF CONTENTS (HIDE)

1. How to format a float/double into a String

The method `printf()` can be used to format a `float/double` to a output stream. However, it does not return a `String`. In JDK 1.5, a new `static` method `format()` is added to the `String` class, which is similar to `printf()`, but returns a `String`. For example,

```
String strDouble = String.format("%8.2f", 1.23456);
```

Alternatively, we could use the `java.util.Formatter` class.

```
// Allocate a Formatter on the StringBuilder
StringBuilder sb = new StringBuilder();
Formatter formatter = new Formatter(sb);  // Send all outputs to StringBuilder
// format() has the same syntax as printf()
formatter.format("%.2f", totalPrice);     // 2 decimal places
```

2. How to Clone an Object (Copy by Value)

The easiest way is to write a *copy constructor*, which takes the object to be cloned, construct a new object by copying all the variables (states). For example,

```
1  public class Circle {
2     public double radius;
3
4     public Circle(double radius) {
5        this.radius = radius;
6     }
7     public Circle() {
8        this.radius = 0.0;
9     }
10    // Copy Constructor
11    public Circle(Circle another) {
12       this.radius = another.radius;
13    }
14
15    public void setRadius(double radius) {
16       this.radius = radius;
17    }
18
19    public double getRadius() {
20       return radius;
21    }
22
23    public static void main(String[] args) {
24       Circle c1 = new Circle(1.1);
25       System.out.println(c1.getRadius());  // 1.1
26       Circle c2 = new Circle(c1); // Construct c2 by cloning c1
27       System.out.println(c2.getRadius());  // 1.1
28       c2.setRadius(2.2);
29       System.out.println(c1.getRadius());  // 1.1
30       System.out.println(c2.getRadius());  // 2.2
31
32       Circle c3 = c1; // Assign the reference of c1 to c3
33                       // c1 and c3 points to the same object
34       System.out.println(c3.getRadius());  // 1.1
35       c1.setRadius(3.3);
36       System.out.println(c3.getRadius());  // 3.3
37    }
38 }
```

If the object contains object members, you need to do a *deep copy*, i.e., clone the object members as well.

`Cloneable` Interface and `Object`'s `clone()` method

The `java.lang.Cloneable` interface defines a method called `clone()`. The `java.lang.Object` provides a method called `clone()` to create a clone copy. To use the `Object`'s `clone()` method, the class shall implement the `cloneable` interface and invoke the inherited `clone()` method from `Object`. the For example,

```
public class Circle implements Cloneable { .... }
```

```
Circle c1 = new Circle(1.1);
Circle c2 = (Circle)c1.clone();
   // Invoke the inherited Object's clone() method.
   // clone() returns an Object, need to downcast to Circle
```

However, the `Object`'s `clone()` method makes a *shallow copy*, i.e., it copies only the first level of variables and does not do a *deep copy* for object members. Classes implementating `Cloneable` interface is recommended to override the inherited `Object`'s `clone()` method to perform deep copying. Avoid using `Object`'s `clone()`.

3. How to initialize a `static` array of objects

Suppose that we have a class called `Book`, as follows:

```
public class Book {
   private String title;
   private float price;

   public Book(String title, float price) {  // Constructor
      this.title = title;
      this.price = price;
   }

   public String getTitle() { return title; }
   public float  getPrice() { return price; }
}
```

Suppose that we want to create a `static` array of `Book`s for global access. We need to use the *static initializer* as follows:

```
/**
 * Book Database
 * Contain a static array of Book, and static methods for operation.
 * The array index is used as the bookID.
 */
public class BookDB {
   private static Book[] books;

   static {      // static initializer block
      books = new Book[2];
      books[0] = new Book("Java for Dummies", 19.99f);
      books[1] = new Book("More Java for Dummies", 29.99f);
   }

   public static int size() {
      return books.length;
   }

   public static String getTitle(int bookID) {
      return books[bookID].getTitle();
   }

   public static float getPrice(int bookID) {
      return books[bookID].getPrice();
   }

   // Testing
   public static void main(String[] args) {
      System.out.println(BookDB.size());
      System.out.println(BookDB.getTitle(0));
      System.out.println(BookDB.getPrice(0));
   }
}
```

4. Why Java `Vector` and `Hashtable` are considered obsolete or deprecated?

`Vector` and `Hastable` were introduced in ...

Read http://stackoverflow.com/questions/1386275/why-java-vector-class-is-considered-obsolete-or-deprecated.

Use `ArrayList` to replace `Vector`, and `HashMap` for `Hashtable`.

5. Printing Newline with `'\n'`

Line delimiter (or new line) is platform dependent. Windows uses `"\r\n"` (`\r` for *carriage return* with ASCII code `0DH` or decimal `13`; `\n` for line feed with ASCII code `0AH` or decimal `10`); Unix and Mac OS X uses `"\n"` alone; Mac OS up to version 9 uses `"\r"`.

The default line separator is kept in system property `line.separator`. You could print it as follows:

```
String str = System.getProperty("line.separator");
for (int i = 0; i < str.length(); ++i) {
   System.out.printf("%02X(hex) ", (int)str.charAt(i));  // %02X: Pad with 0, 2 spaces, in hex
}
```

```
0D(hex) 0A(hex)
```

Using '\n' in print(), println() and printf() methods to print a line feed (0AH) may result in non-portable codes.

It is recommended to use System.out.println() to print a system-specific new line, or printf() with format specifier "%n", instead of "\n" for system-specific new line.

6. System Properties

Java maintains a set of system properties for its operations. Each system property is a key-value (String-String) pair such as "java.version"="1.7.0_09". You can retrieve all the system properties via System.getProperties(). You can also retrieve individual property via System.getProperty(*key*). For example,

```
import java.util.Properties;
public class PrintSystemProperties {
   public static void main(String[] a) {
      // List all System properties
      Properties pros = System.getProperties();
      pros.list(System.out);

      // Get a particular System property given its key
      // Return the property value or null
      System.out.println(System.getProperty("java.home"));
      System.out.println(System.getProperty("java.library.path"));
      System.out.println(System.getProperty("java.ext.dirs"));
      System.out.println(System.getProperty("java.class.path"));
   }
}
```

The important system properties are:

1. JRE related:
   - **java.home**: JRE home directory, e.g., "C:\Program Files\Java\jdk1.7.0_09\jre".
   - **java.library.path**: JRE library search path for search native libraries. It is usually but not necessarily taken from the environment variable PATH.
   - **java.class.path**: JRE CLASSPATH, e.g., . (for current working directory).
   - **java.ext.dirs**: JRE extension library path(s), e.g, "C:\Program Files\Java\jdk1.7.0_09\jre\lib\ext;C:\Windows\Sun\Java\lib\ext".
   - **java.version**: JRE version, e.g., 1.7.0_09.
   - **java.runtime.version**: JRE version, e.g. 1.7.0_09-b05.
2. File related:
   - **file.separator**: symbol for file directory separator such as d:\test\test.java. The default is \ for windows or / for Unix/Mac.
   - **path.separator**: symbol for separating path entries, e.g., in PATH or CLASSPATH. The default is ; for windows or : for Unix/Mac.
   - **line.separator**: symbol for end-of-line (or new line). The default is "\r\n" for windows or "\n" for Unix/Mac OS X.
3. User related:
   - **user.name**: the user's name.
   - **user.home**: the user's home directory.
   - **user.dir**: the user's current working directory.
4. OS related:
   - **os.name**: the OS's name, e.g., "Windows 7".
   - **os.version**: the OS's version, e.g., "6.1".
   - **os.arch**: the OS's architecture, e.g., "x86".

Access to system properties can be restricted by the Java security manager and policy file. By default, Java programs have unrestricted access to all the system properties.

6.1 Which JRE?

In system property java.home.

[TODO]

6.2 Where are the Java Extension Library Paths?

In system property java.ext.dirs. There could be more than one extension library paths.

[TODO]

7. Java Security Manager and Policy File

Reference:

1. Java Tutorial's "Quick Tour of Controlling Applications".
2. Java Tutorial's Trail "Security Features in Java SE".

3. Permissions in JDK 7 @ http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html.

7.1 Security Manager

Java runtime does NOT automatically install a security manager when it runs Java application. As the result, the Java applications have unrestricted access to all the system. For example, the following Java program (a) read system property "`user.home`", (b) read from file "`in.txt`", (c) write to file "`out.txt`".

```
import java.util.*;
import java.io.*;
public class TestPermissions {
   public static void main(String[] args) throws Exception {
      // Read System Property
      System.out.println(System.getProperty("user.home"));

      // Read File
      Scanner in = new Scanner (new File("in.txt"));
      int num1 = in.nextInt();
      int num2 = in.nextInt();
      System.out.println("The 2 numbers are: " + num1 + ", " + num2);

      // Write File
      Formatter out = new Formatter(new File("out.txt"));
      int sum = num1 + num2;
      System.out.println("The sum is " + sum);
      out.format("%d", sum);
      out.close();
   }
}
```

Create the input file "`in.txt`" with 2 number "`1 2`". Compile and run the program. Everything shall be fine as Java programs have unrestricted permissions, if security manager is not installed.

To explicitly install a Java security manager, run JRE with option `-Djava.security.manager`.

```
java -Djava.security.manager TestPermissions
```

You shall receive a security exception:

```
Exception in thread "main" java.security.AccessControlException: access denied ("java.util.PropertyPermission" "user.home" "read")
```

The security manager, by default, fetches the permissions from `<JRE_HOME>\lib\security\java.policy`. Take a look at this policy file, it grants read permission to many system properties, but NOT including "`user.home`".

If you commneted out the `System.getProperty()` line and run the program with security manager, you will receive this error:

```
Exception in thread "main" java.security.AccessControlException: access denied ("java.io.FilePermission" "out.txt" "write")
```

You have not write access to "`out.txt`". However, you do have read acces to "`in.txt`".

7.2 Granting Permissions

The security manager, by default, fetches the system policy file from `<java.home>\lib\security\java.policy`, where `<java.home>` is the Java system property "`java.home`", which defaults to JRE home directory.

Let's create a user policy file called "`myjava.policy`" to grant the permissions needed for the above program. Suppose that the program and the files are location at "`d:\myproject`".

```
grant codeBase "file:/D:/myproject/" {
   permission java.util.PropertyPermission "user.home", "read";
   permission java.io.FilePermission "D:/myproject/out.txt", "write";
};
```

The `codeBase` indicates the base directory of the executing program, which shall be a URL beginning with a protocol such as `file:` for local programs (or `http:` for applets). A `codeBase` ends with "`/`" matches all class files (not JAR files) in the specified directory. A `codeBase` ends with "`/*`" matches all class files and JAR files. A `codeBase` ends with "`/-`" matches all class files and JAR files in the directory and its subdirectories.

In the above policy file, we grant read permission to system property "`user.home`", and write file permission to "`out.txt`".

To include the user policy file (in addition to the system policy files), use command-line option `-Djava.security.policy`:

```
D:\myproject> java -Djava.security.manager -Djava.security.policy=myjava.policy TestPermissions
```

The program shall now complete its execution.

If you use double equal `==` in assigning policy file, the system policy files will be ignore. That is, only the specified policy file will be used.

```
D:\myproject> java -Djava.security.manager -Djava.security.policy==myjava.policy TestPermissions
```

7.3 Including User Policy Files

By default, the security manager fetches these policy files, as specified in its property file located at `<JRE_HOME>\lib\security\java.security`:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

where "`java.home`" and "`user.home`" are Java system properties.

To include user policy file(s), you could either:

1. Include it in JRE's command-line option `-Djava.security.policy` as in the above example.
2. Include it in the security properties file `<JRE_HOME>\lib\security\java.security`, as follows:

```
policy.url.3=file:/D:/myproejct/myjava.policy
```

7.4 Types of Permission

Each permission has a type (e.g., `java.io.FilePermission`, `java.util.PropertyPermission`), a target name (e.g., property name or filename) and a comma-separated list of one or more actions (e.g., read, write).

Permissions can be granted to programs under a `codeBase`.

The commonly-used permission types are:

1. **`java.security.AllPermission`**: All the permissions. For example, the default system policy file grants `AllPermission` to all the classes and JAR files in the Java extension paths:

   ```
   grant codeBase "file:${{java.ext.dirs}}/*" {
           permission java.security.AllPermission;
   };
   ```

   Granting `AllPermission` practically disables security manager and shall be done with great care.

2. **`java.io.FilePermission`**: grant permission to file and directory.
   A pathname ends with "`/*`" indicates all files in that directory; "`/-`" indicates all file in that directory and subdirectories; "*" indicates all file in the current working directory; "`-`" indicates all files in the current working directory and its subdirectories; `<<ALL FILES>>` indicates any file.
   The actions (comma-separated) include: read, write, delete, and execute (allow `Runtime.exec()`).

3. **`java.net.SocketPermission`**: permit access to a network via socket.
   The target name consists of `hostname:port`. A port specification of "`n-`" indicate port number `n` and above; "`-n`" all port number `n` and below.
   The actions include: `accept`, `connect`, `listen` and `resolve` (DNS lookup). When use with localhost, `listen` is the only meaningful action.
   For example, the default system policy file grant listen permission to localhost of port 1024 and above (un-privilege port).

   ```
   permission java.net.SocketPermission "localhost:1024-", "listen";
   ```

4. **`java.util.PropertyPermission`**: The target name is a property name. The actions include: `read` and `write`.
5. **`java.lang.RuntimePermission`**: Only target name without action, e.g., the target "`stopThread`" permits stopping of threads via calls to the `Thread`'s `stop()` method.
6. **`java.sql.SQLPermission`**: [TODO]

7.5 Signing Code and Granting Permission

[TODO]

8. Run External Process from Java

You could use either `Runtime.getRuntime.exec()` or a `ProcessBuilder`.

8.1 `exec()` method

The API doc for `exec()` of `java.lang.Runtime` is:

```
public Process exec(String command) throws IOException
// Executes the specified string command in a separate process.
// It returns a java.lang.Process object.
```

The `Process` object provides a `waitFor()` method, which "causes the current thread to wait, if necessary, until the process represented by this `Process` object has terminated." `waitFor()` returns the exit value of the subprocess (with 0 for normal termination).

Example

```
public class TestExec {
   public static void main(String[] a) {
      // Invoke external command via exec(), which returns a Process
      Process p = null;
      try {
         p = Runtime.getRuntime().exec("javac Add2Numbers.java");  // Compile
      } catch (java.io.IOException ex) {}

      // Wait for the process to complete
      try {
         int exitValue = p.waitFor();
         System.out.println("Process Completed with exit value of " + exitValue);
      } catch (InterruptedException ex) {}
   }
}
```

You can optionally specifies an environment (a set of name=value pairs) and an initial working directory.

The `Add2Numbers` reads 2 integer from `System.in` and prints their sum to `System.out`, as follows:

```
import java.util.Scanner;
public class Add2Numbers {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int num1 = in.nextInt();
      int num2 = in.nextInt();
      System.out.println(num1 + num2);
   }
}
```

With Input/Output for the Subprocess

By default, the subprocess' standard input, output and error will be piped from/to the parent process. You can access the standard input via stream obtained from subprocess' `getOutputStream()` (ouput stream of the parent process is piped into the subprocess' standard input). Similarly, you can obtain the standard output stream of the subprocess via `getInputStream()` (input to parent process); and standard error via `getErrorStream()`. For example,

```
import java.io.*;
public class TestExecRedirect {
   public static void main(String[] a) {
      try {
         Process p = Runtime.getRuntime().exec("java Add2Numbers");  // Execute with input/output

         // Write into the standard input of the subprocess
         PrintStream pin = new PrintStream(new BufferedOutputStream(p.getOutputStream()));
         // Read from the standard output of the subprocess
         BufferedReader pout = new BufferedReader(new InputStreamReader(p.getInputStream()));

         // Pump in input
         pin.print("1 2");
         pin.close();

         // Save the output in a StringBuffer for further processing
         StringBuffer sb = new StringBuffer();
         int ch;
         while ((ch = pout.read()) != -1) {
            sb.append((char)ch);
         }
         System.out.println(sb);

         int exitValue = p.waitFor();
         System.out.println("Process Completed with exit value of " + exitValue);
      } catch (IOException ex) {
      } catch (InterruptedException ex) {}
   }
}
```

8.2 ProcessBuilder

JDK 1.5 introduces a new `ProcessBuilder` class in `java.lang`, which manages command, environment, initial working directory, as well as standard input, output and error of a process. To use the `ProcessBuilder`, construct an instance and invoke its `start()` method. `start()` returns a `Process` object.

Example

```
public class TestProcessBuilder {
   public static void main(String[] a) {
      try {
         // Allocate a ProcessBuilder for the command
         ProcessBuilder pb = new ProcessBuilder("javac", "Add2Numbers.java");  // Compile (no standard input)
         // Start the process
         Process p = pb.start();
         // Wait for the process to complete
         int exitValue = p.waitFor();
         System.out.println("Process Completed with exit value of " + exitValue);
      } catch (java.io.IOException ex) {
      } catch (InterruptedException ex) {}
   }
}
```

Redirecting Input/Ouput of the Subprocess to Files

Same as `exec()`, by default, subprocess reads input from a pipe and write output and error to a pipe of the parent process. In `ProcessBuilder`, you can conveniently redirect the subprocess' input, output and error to a file (as of JDK 1.7), as follows:

```
import java.io.File;
public class TestProcessBuilderRedirect {
   public static void main(String[] a) {
      try {
         ProcessBuilder pb = new ProcessBuilder("java", "Add2Numbers");  // Execute (with standard input and output)

         File log = new File("error.log");
//         pb.redirectErrorStream(true);  // merge output and error streams
         pb.redirectInput(ProcessBuilder.Redirect.from(new File("in.txt")));
         pb.redirectOutput(ProcessBuilder.Redirect.to(new File("out.txt")));
         pb.redirectError(ProcessBuilder.Redirect.appendTo(new File("error.log")));
```

```
         Process p = pb.start();
         int exitValue = p.waitFor();
         System.out.println("Process Completed with exit value of " + exitValue);
      } catch (java.io.IOException ex) {
      } catch (InterruptedException ex) {}
   }
}
```

However, the redirect methods work only for files. For other IO streams, you need to access via `Process`' `getOutputStream()` and `getInputStream()` (as in the `exec()` section's example).

9. How to Redirect Standard Input, Output and Error Streams

You can re-direct the standard input (`System.in`), standard output (`System.out`) and standard error (`System.err`) to another IO stream (such as file or network socket) via `static` methods `System.setIn()`, `System.setOut()` and `System.setErr()`. The signture of the methods are:

```
public static void setIn(InputStream in)
public static void setOut(PrintStream out)
public static void setErr(PrintStream err)
```

For example,

```
1  import java.io.*;
2  import java.util.*;
3  public class TestRedirect {
4     public static void main(String[] args) throws IOException {
5        PrintStream sysout =
6           new PrintStream(
7              new BufferedOutputStream(
8                 new FileOutputStream("out.txt", true))); // append outputs to file
9
10
11       InputStream sysin =
12          new BufferedInputStream(
13             new FileInputStream("in.txt"));
14
15       // Redirect to file
16       System.setIn(sysin);
17       System.setOut(sysout);
18       System.setErr(sysout);  // merge error and output streams
19
20
21       // Test inputting/outputting
22       Scanner in = new Scanner(System.in);
23       System.err.println("Let's begin...");
24       System.out.print("Enter two integers: ");
25       int num1 = in.nextInt();
26       int num2 = in.nextInt();
27       int sum = num1 + num2;
28       System.out.println("The sum is " + sum);
29
30       sysout.flush();
31       sysout.close();
32       sysin.close();
33    }
34 }
```

[TODO] Check on `FileWriter` and `FileReader`, instead of `FileInputStream` and `FileOutputStream`.

10. How to Append to a File

Use these constructors of the `FileOutputStream` and `FileWriter`, which take a second `boolean` argument to indicate append:

```
FileWriter(File file, boolean append)
FileWriter(String fileName, boolean append)
FileOutputStream(File file, boolean append)
FileOutputStream(String name, boolean append)
```

See "How to Redirect Standard Input, Output and Error Streams" for example.

REFERENCES & RESOURCES

1. [TODO]