Part 1

Java Language Tools

This beginning, ground-level part presents reference information for setting up the Java development environment and for compiling and running Java programs. This includes downloading and installing the Java Development Kit (JDK), understanding the Java folder structure, setting path variables for Java, and using Java commands.

In this part . . .

- **Downloading and installing the JDK**
- **The JDK folder structure**
- **Setting path variables**
- **Using Java command line tools**

Downloading and Installing the Java Development Kit

The Java Development Kit (JDK) can be downloaded from the following web address:

```
http://www.oracle.com/technetwork/java/javase/
  downloads/index.html
```

The Java download tab includes links to download the JDK or the Java Runtime Environment (JRE). Follow the JDK link because clicking the JRE link gets you only the JRE, not the complete JDK.

The JDK download comes in two versions: an online version that requires an active Internet connection to install the JDK, and an offline version that lets you download the JDK installation file to your computer and install it later.

I recommend using the offline version. That way, you can reinstall the JDK if you need to without having to download it again.

The exact size of the offline version depends on the platform, but most versions are between 50MB and 60MB, so that means that your download will take a few hours if you don't have a high-speed Internet connection. With a broadband cable, DSL, or T1 connection, though, the download takes less than five minutes.

After you download the JDK file, install it by running its executable file. The procedure varies slightly depending on your operating system, but basically, you just run the JDK installation program file after you download it, as follows:

- On a Windows system, open the folder in which you saved the installation program and double-click the installation program's icon.
- On a Linux or Solaris system, use console commands to change to the directory to which you downloaded the file and then run the program.

After you start the installation program, it prompts you for any information that it needs to install the JDK properly, such as which features you want to install and what folder you want to install the JDK in. You can safely choose the default answer for each option.

Note: If you're an Apple Mac user, you don't have to download and install the Java JDK. Apple pre-installs the JDK on Mac computers.

JDK Folder Structure

The JDK setup program creates several folders on your hard drive. The locations of these folders vary depending on your system, but in all versions of Windows, the JDK root folder is in the path `Program Files\Java` on your boot drive. The name of the JDK root folder also varies, depending on the Java version you've installed. For version 1.7, the root folder is `jdk1.7.0`.

Here are the subfolders created in the JDK root folder. As you work with Java, you'll refer to these folders frequently.

Folder	*Description*
bin	The compiler and other Java development tools
demo	Demo programs you can study to see how to use various Java features
docs	The Java application programming interface (API) documentation
include	The library containing files needed to integrate Java with programs written in other languages
jre	JRE files
lib	Library files, including the Java API class library
Sample	Sample code

In addition to these folders, the JDK installs several files in the JDK root folder.

File	*Description*
`README.html`	The Java README file in HTML format.
`README.txt`	The README file, this time in text format.
`LICENSE`	The Java license you agree to when you download the JDK.
`LICENSE.rtf`	The license file, this time in RTF format.

cont.

File	Description
COPYRIGHT	The copyright notice.
src.zip	The source code for the Java API classes. This folder is created only if you unpack the src.zip file (which may be named src.jar). After you get your feet wet with Java, looking at these source files can be a great way to find out more about how the API classes work.

RTF (rich text format) is a document format that most word-processing programs can understand.

Setting the Path for Java

After you install the JDK, you need to configure your OS so that it can find the JDK command line tools. Start by setting the Path environment variable — a list of folders that the OS uses to locate executable programs. To do this on a Windows XP, Windows Vista, or Windows 7 system, follow these steps:

1. **Click the Windows (Start) button and then right-click Computer (Windows 7 or Vista) or My Computer (XP).**

   This brings up the System Properties page.

2. **Click the Advanced tab (XP) or the Advanced System Settings link (Vista and 7); then click the Environment Variables button.**

   The Environment Variables dialog box appears.

3. **In the System Variables list, scroll to the Path variable, select it, and then click the Edit button.**

   A dialog box pops up to let you edit the value of the Path variable.

4. **Add the JDK bin folder to the beginning of the Path value.**

   Use a semicolon to separate the bin folder from the rest of the information that may already be in the path.

   Note: The name of the bin folder may vary on your system, as in this example:

```
c:\Program Files\Java\jdk1.7.0\bin;other directories...
```

5. **Click OK three times to exit.**

   The first OK gets you back to the Environment Variables dialog box; the second OK gets you back to the System Properties dialog box; and the third OK closes the System Properties dialog box.

java Command

The `java` command runs a Java program from a command prompt. The basic syntax is

```
java filename [options]
```

When you run the `java` command, the JRE is loaded along with the class you specify. Then, the main method of that class is executed.

Here's an example that runs a program named `HelloApp`:

```
C:\java\samples>java HelloApp
```

The bold indicates the part you type.

The class must be contained in a file with the same name as the class, and its filename must have the extension `.class`. You typically don't have to worry about the name of the class file because it's created automatically when you compile the program with the `javac` command.

The Java runtime command lets you specify options that can influence its behavior.

Option	*Description*
`-?` or `-help`	Lists standard options
`-classpath` *directories and archives*	Lists the directories or JAR or Zip archive files used to search for class files
`-client`	Runs the client virtual machine
`-cp <search path>`	Does the same thing as `-classpath`
`-D` *name=value*	Sets a system property

cont.

Option	Description
`-dsa` or `-disablesystemassertions`	Disables system assertions
`-ea` *`classes or packages`*	Enables assertions for the specified classes or packages
`-ea` or `-enableassertions`	Enables the `assert` command
`-esa` or `-enablesystemassertions`	Enables system assertions
`-server`	Runs the server virtual machine, which is optimized for server systems
`-showversion`	Displays the JRE version number and then continues
`-verbose`	Enables verbose output, which displays more comprehensive messages
`-version`	Displays the JRE version number and then stops
`-X`	Lists nonstandard options

javac Command

The `javac` command compiles a program from a command prompt. It reads a Java source program from a text file and creates a compiled Java class file. The basic form of the `javac` command is

```
javac filename [options]
```

For example, to compile a program named `HelloWorld.java`, use this command:

```
javac HelloWorld.java
```

Normally, the `javac` command compiles only the file that you specify on the command line, but you can coax `javac` into compiling more than one file at a time by using any of the following techniques:

- If the Java file you specify on the command line contains a reference to another Java class that's defined by a `java` file in the same folder, the Java compiler automatically compiles that class, too.
- You can list more than one filename in the `javac` command. The following command compiles three files:

```
javac TestProgram1.java TestProgram2.java
  TestProgram3.java
```

- You can use a wildcard to compile all the files in a folder, like this:

```
javac *.java
```

- If you need to compile a lot of files at the same time but don't want to use a wildcard (perhaps you want to compile a large number of files but not all the files in a folder), you can create an *argument file,* which lists the files to compile. In the argument file, you can type as many filenames as you want, using spaces or line breaks to separate them. Here's an argument file named `TestPrograms` that lists three files to compile:

```
TestProgram1.java
TestProgram2.java
TestProgram3.java
```

  You can compile all the programs in this file by using an `@` character, followed by the name of the argument file on the `javac` command line, like this:

```
javac @TestPrograms
```

The `javac` command has a gaggle of options that you can use to influence how it compiles your programs.

Option	*Description*
`-bootclasspath <path>`	Overrides locations of bootstrap class files. (The bootstrap class files are the classes that implement the Java runtime. You will rarely use this option.)
`-classpath <path>`	Specifies where to find user class files. Use this option if your program makes use of class files that you've stored in a separate folder.

cont.

Option	*Description*
`-cp <path>`	Same as `classpath`.
`-d <directory>`	Specifies where to place generated class files.
`-deprecation`	Outputs source locations where deprecated APIs are used. Use this option if you want the compiler to warn you whenever you use API methods that have been deprecated.
`-encoding <encoding>`	Specifies character encoding used by source files.
`-endorseddirs <dirs>`	Overrides location of endorsed standards path.
`-extdirs <dirs>`	Overrides locations of installed extensions.
`-g`	Generates all debugging info.
`-g:{lines,vars,source}`	Generates only some debugging info.
`-g:none`	Generates no debugging info.
`-help`	Prints a synopsis of standard options.
`-J<flag>`	Passes `<flag>` directly to the runtime system.
`-nowarn`	Generates no warnings.
`-source <release>`	Provides source compatibility with specified release.
`-sourcepath <path>`	Specifies where to find input source files.
`-target <release>`	Generates class files for specific virtual machine version.
`-verbose`	Outputs messages about what the compiler is doing.
`-version`	Provides version information.
`-X`	Prints a synopsis of nonstandard options.

A *class file* is a compiled Java program that can be executed by the `java` command. The Java compiler reads source files and creates class files.

A *deprecated API* is a feature that is considered obsolete.

To use one or more of these options, type the option before or after the source filename. Either of the following commands, for example, compiles the `HelloApp.java` file with the `-verbose` and `-deprecation` options enabled:

```
javac HelloWorld.java –verbose –deprecation
javac –verbose –deprecation HelloWorld.java
```

javap Command

The `javap` command is called the Java "disassembler" because it takes apart class files and tells you what's inside them. You won't use this command often, but using it to find out how a particular Java statement works is fun, sometimes. You can also use it to find out what methods are available for a class if you don't have the source code that was used to create the class.

Here is the general format:

```
javap filename [options]
```

The following is typical of the information you get when you run the `javap` command:

```
C:\java\samples>javap HelloApp
Compiled from "HelloApp.java"
public class HelloApp extends java.lang.
  Object{
    public HelloApp();
    public static void main(java.lang.
  String[]);
}
```

As you can see, the `javap` command indicates that the `HelloApp` class was compiled from the `HelloApp.java` file and that it consists of a `HelloApp public` class and a `main public` method.

You may want to use two options with the `javap` command. If you use the `-c` option, the `javap` command displays the actual Java bytecodes created by the compiler for the class. (*Java bytecode* is the executable program compiled from your Java source file.) And if you use the `-verbose` option, the bytecodes — plus a ton of other fascinating information about the innards of the class — are displayed. Here's the `-c` output for a class named `HelloApp`:

```
C:\java\samples>javap HelloApp -c
Compiled from «HelloApp.java»
public class HelloApp extends java.lang.Object{
public HelloApp();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method
   java/lang/Object.»<init>»:()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field
   java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc     #3; //String Hello, World!
   5:   invokevirtual   #4; //Method
   java/io/PrintStream.println:(Ljava/lang/
  String;)V
   8:   return

}
```

jar Command

You use the `jar` command to create a *JAR file,* which is a single file that can contain more than one class in a compressed format that the Java Runtime Environment can access quickly. (*JAR* stands for *Java archive.*) A JAR file can have a few or thousands of classes in it. In fact, the entire Java API is stored in a single JAR file named `rt.java`. (The `rt` stands for *runtime.*) It's a big file — over 35MB — but that's not bad considering that it contains more than 12,000 classes.

JAR files are similar in format to *Zip files,* a compressed format made popular by the PKZIP program. The main difference is that JAR files contain a special file, called the *manifest file,* which contains information about the files in the archive. This manifest is automatically created by the `jar` utility, but you can supply a manifest of your own to provide additional information about the archived files.

JAR files are the normal way to distribute finished Java applications. After finishing your application, you run the `jar`

command from a command prompt to prepare the JAR file. Then, another user can copy the JAR file to his or her computer. The user can then run the application directly from the JAR file.

JAR files are also used to distribute class libraries. You can add a JAR file to the ClassPath environment variable. Then, the classes in the JAR file are automatically available to any Java program that imports the package that contains the classes.

The basic format of the `jar` command is

```
jar options jar-file [manifest-file]
  class-files...
```

The options specify the basic action you want `jar` to perform and provide additional information about how you want the command to work. Here are the options:

Option	*Description*
`c`	Creates a new `jar` file.
`u`	Updates an existing `jar` file.
`x`	Extracts files from an existing `jar` file.
`t`	Lists the contents of a `jar` file.
`f`	Indicates that the `jar` file is specified as an argument. You almost always want to use this option.
`v`	Verbose output. This option tells the `jar` command to display extra information while it works.
`0`	Doesn't compress files when it adds them to the archive. This option isn't used much.
`m`	Specifies that a manifest file is provided. It's listed as the next argument following the `jar` file.
`M`	Specifies that a manifest file should not be added to the archive. This option is rarely used.

Note that you must specify at least the `c`, `u`, `x`, or `t` option to tell `jar` what action you want to perform.

To create an archive, follow these steps:

1. **Open a command window.**

   The easiest way is to choose Start⇨Run, type **`cmd`** in the Open text box, and then click OK.

2. **Use a `cd` command to navigate to your package root.**

   For example, if your packages are stored in `c:\java classes`, use this command:

   ```
   cd \javaclasses
   ```

3. **Use a `jar` command that specifies the options `cf`, the name of the `jar` file, and the path to the class files you want to archive.**

   For example, to create an archive named `utils.jar` that contains all the class files in the `com.lowewriter.util` package, use this command:

   ```
   jar cf utils.jar com\lowewriter\util\*.
     class
   ```

4. **To verify that the `jar` file was created correctly, use the `jar` command that specifies the options `tf` and the name of the `jar` file.**

   For example, if the `jar` file is named `utils.jar`, use this command:

   ```
   jar tf utils.jar
   ```

   This lists the contents of the jar file so you can see what classes were added. Here's some typical output from this command:

   ```
   META-INF/
   META-INF/MANIFEST.MF
   com/lowewriter/util/Console.class
   com/lowewriter/util/Random.class
   ```

   As you can see, the `utils.jar` file contains the two classes in my `com.lowewriter.util` package, `Console` and `Random`.

5. **You're done!**

   You can leave the `jar` file where it is, or you can give it to your friends so they can use the classes it contains.