10

Development

CERTIFICATION OBJECTIVES

- Use Packages and Imports
- Determine Runtime Behavior for Classes and Command-Lines
- Use Classes in JAR Files
- Use Classpaths to Compile Code

✓ Two-Minute Drill

Q&A Self Test

You want to keep your classes organized. You need to have powerful ways for your classes to find each other. You want to make sure that when you're looking for a particular class you get the one you want, and not another class that happens to have the same name. In this chapter we'll explore some of the advanced capabilities of the java and javac commands. We'll revisit the use of packages in Java, and how to search for classes that live in packages.

CERTIFICATION OBJECTIVES

Using the javac and java Commands (Objective 7.2)

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

7.2 Given an example of a class and a command-line, determine the expected runtime behavior.

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.

So far in this book, we've probably talked about invoking the `javac` and `java` commands about 1000 times; now we're going to take a closer look.

Compiling with javac

The `javac` command is used to invoke Java's compiler. In Chapter 5 we talked about the assertion mechanism and when you might use the `-source` option when compiling a file. There are many other options you can specify when running `javac`, options to generate debugging information or compiler warnings for example. For the exam, you'll need to understand the `-classpath` and `-d` options, which we'll cover in the next few pages. In addition, it's important to understand the structure of this command. Here's the overview:

```
javac [options] [source files]
```

There are additional command-line options called `@argfiles`, but you won't need to study them for the exam. Both the `[options]` and the `[source files]` are optional parts of the command, and both allow multiple entries. The following are both legal `javac` commands:

```
javac -help
javac -classpath com:. -g Foo.java Bar.java
```

The first invocation doesn't compile any files, but prints a summary of valid options. The second invocation passes the compiler two options (`-classpath`, which itself has an argument of `com:.` and `-g`), and passes the compiler two `.java` files to compile (`Foo.java` and `Bar.java`). Whenever you specify multiple options and/or files they should be separated by spaces.

Compiling with -d

By default, the compiler puts a `.class` file in the same directory as the `.java` source file. This is fine for very small projects, but once you're working on a project of any size at all, you'll want to keep your `.java` files separated from your `.class` files. (This helps with version control, testing, deployment...) The `-d` option lets you tell the compiler in which directory to put the `.class` file(s) it generates (`d` is for destination). Let's say you have the following directory structure:

```
myProject
       |
       |--source
       |       |
       |       |-- MyClass.java
       |
       |-- classes
               |
               |--
```

The following command, issued from the `myProject` directory, will compile `MyClass.java` and put the resulting `MyClass.class` file into the `classes` directory. (Note: This assumes that MyClass does not have a package statement; we'll talk about packages in a minute.)

```
cd myProject
javac -d classes source/MyClass.java
```

This command also demonstrates selecting a `.java` file from a subdirectory of the directory from which the command was invoked. Now let's take a quick look at how packages work in relationship to the `-d` option.

Suppose we have the following `.java` file in the following directory structure:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
       |
       |--source
       |      |
       |      |--com
       |           |
       |           |--wickedlysmart
       |                       |
       |                       |--MyClass.java
       |
       |--classes
       |      |
       |      |--com
       |           |
       |           |--wickedlysmart
       |                       |
       |                       |-- (MyClass.class goes here)
```

If you were in the `source` directory, you would compile `MyClass.java` and put the resulting `MyClass.class` file into the `classes/com/wickedlysmart` directory by invoking the following command:

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

This command could be read: "To set the destination directory, `cd` back to the `myProject` directory then `cd` into the `classes` directory, which will be your destination. Then compile the file named `MyClass.java`. Finally, put the resulting `MyClass.class` file into the directory structure that matches its package, in this case, `classes/com/wickedlysmart`." Because `MyClass.java` is in a package, the compiler knew to put the resulting `.class` file into the `classes/com/wickedlysmart` directory.

Somewhat amazingly, the `javac` command can sometimes help you out by building directories it needs! Suppose we have the following:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
       |
       |--source
       |      |
       |      |--com
       |           |
       |           |--wickedlysmart
       |                       |
       |                       |--MyClass.java
       |
       |--classes
       |      |
```

And the following command (the same as last time):

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

In this case, the compiler will build two directories called `com` and `com/wickedlysmart` in order to put the resulting `MyClass.class` file into the correct package directory (`com/wickedlysmart/`) which it builds within the existing `.../classes` directory.

The last thing about `-d` that you'll need to know for the exam is that if the destination directory you specify doesn't exist, you'll get a compiler error. If, in the previous example, the `classes` directory did NOT exist, the compiler would say something like:

```
java:5: error while writing MyClass: classes/MyClass.class (No
such file or directory)
```

Launching Applications with java

The `java` command is used to invoke the Java virtual machine. In Chapter 5 we talked about the assertion mechanism and when you might use flags such as `-ea` or `-da` when launching an application. There are many other options you can specify

when running the `java` command, but for the exam, you'll need to understand the `-classpath` (and its twin `-cp`) and `-D` options, which we'll cover in the next few pages. In addition, it's important to understand the structure of this command. Here's the overview:

```
java [options] class [args]
```

The `[options]` and `[args]` parts of the `java` command are optional, and they can both have multiple values. You must specify exactly one class file to execute, and the `java` command assumes you're talking about a `.class` file, so you don't specify the `.class` extension on the command line. Here's an example:

```
java -DmyProp=myValue MyClass x 1
```

Sparing the details for later, this command can be read as "Create a *system property* called `myProp` and set its value to `myValue`. Then launch the file named `MyClass.class` and send it two String *arguments* whose values are `x` and `1`."

Let's look at system properties and command-line arguments more closely.

Using System Properties

Java 5 has a class called `java.util.Properties` that can be used to access a system's persistent information such as the current versions of the operating system, the Java compiler, and the Java virtual machine. In addition to providing such default information, you can also add and retrieve your own properties. Take a look at the following:

```
import java.util.*;
public class TestProps {
  public static void main(String[] args) {
    Properties p = System.getProperties();
    p.setProperty("myProp", "myValue");
    p.list(System.out);
  }
}
```

If this file is compiled and invoked as follows:

```
java -DcmdProp=cmdVal TestProps
```

You'll get something like this:

```
...
os.name=Mac OS X
myProp=myValue
...
java.specification.vendor=Sun Microsystems Inc.
user.language=en
java.version=1.5.0_02
...
cmdProp=cmdVal
...
```

where the . . . represent lots of other name=value pairs. (The *name* and *value* are sometimes called the *key* and the *property*.) Two name=value properties were added to the system's properties: `myProp=myValue` was added via the `setProperty` method, and `cmdProp=cmdVal` was added via the `-D` option at the command line. When using the `-D` option, if your value contains white space the entire value should be placed in quotes like this:

```
java -DcmdProp="cmdVal take 2" TestProps
```

Just in case you missed it, when you use `-D`, the name=value pair must follow *immediately*, no spaces allowed.

The `getProperty()` method is used to retrieve a single property. It can be invoked with a single argument (a String that represents the name (or key)), or it can be invoked with two arguments, (a String that represents the name (or key), and a default String value to be used as the property if the property does not already exist). In both cases, `getProperty()` returns the property as a String.

Handling Command-Line Arguments

Let's return to an example of launching an application and passing in arguments from the command line. If we have the following code:

```
public class CmdArgs {
  public static void main(String[] args) {
    int x = 0;
    for(String s : args)
      System.out.println(x++ + " element = " + s);
  }
}
```

compiled and then invoked as follows

```
java CmdArgs x 1
```

the output will be

```
0 element = x
1 element = 1
```

Like all arrays, `args` index is zero based. Arguments on the command line directly follow the class name. The first argument is assigned to `args[0]`, the second argument is assigned to `args[1]`, and so on.

Finally, there is some flexibility in the declaration of the `main()` method that is used to start a Java application. The order of `main()`'s modifiers can be altered a little, the String array doesn't have to be named `args`, and as of Java 5 it can be declared using var-args syntax. The following are all legal declarations for `main()`:

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[])
```

Searching for Other Classes

In most cases, when we use the `java` and `javac` commands, we want these commands to search for other classes that will be necessary to complete the operation. The most obvious case is when classes we create use classes that Sun provides with J2SE (now sometimes called Java SE), for instance when we use classes in `java.lang` or `java.util`. The next common case is when we want to compile a file or run a class that uses other classes that have been created outside of what Sun provides, for instance our own previously created classes. Remember that for any given class, the `java` virtual machine will need to find exactly the same supporting classes that the `javac` compiler needed to find at compilation time. In other words, if `javac` needed access to `java.util.HashMap` then the `java` command will need to find `java.util.HashMap` as well.

Both `java` and `javac` use the same basic search algorithm:

1. They both have the same list of places (directories) they search, to look for classes.

2. They both search through this list of directories in the same order.
3. As soon as they find the class they're looking for, they stop searching for that class. In the case that their search lists contain two or more files with the same name, the first file found will be the file that is used.
4. The first place they look is in the directories that contain the classes that come standard with J2SE.
5. The second place they look is in the directories defined by classpaths.
6. Classpaths should be thought of as "class search paths." They are lists of directories in which classes might be found.
7. There are two places where classpaths can be declared:

   A classpath can be declared as an operating system environment variable. The classpath declared here is used by default, whenever `java` or `javac` are invoked.

   A classpath can be declared as a command-line option for either `java` or `javac`. *Classpaths declared as command-line options override the classpath declared as an environment variable, but they persist only for the length of the invocation.*

Declaring and Using Classpaths

Classpaths consist of a variable number of directory locations, separated by delimiters. For Unix-based operating systems, forward slashes are used to construct directory locations, and the separator is the colon (:). For example:

```
-classpath /com/foo/acct:/com/foo
```

specifies two directories in which classes can be found: `/com/foo/acct` and `/com/foo`. In both cases, these directories are absolutely tied to the root of the file system, which is specified by the leading forward slash. It's important to remember that when you specify a subdirectory, you're NOT specifying the directories above it. For instance, in the preceding example the directory `/com` will NOT be searched.

exam Watch

Most of the path-related questions on the exam will use Unix conventions. If you are a Windows user, your directories will be declared using backslashes (\) and the separator character you use will be a semicolon (;). But again, you will NOT need any shell-specific knowledge for the exam.

A very common situation occurs in which `java` or `javac` complains that it can't find a class file, and yet you can see that the file is IN the current directory! When searching for class files, the `java` and `javac` commands don't search the current directory by default. You must *tell* them to search there. The way to tell `java` or `javac` to search in the current directory is to add a dot (.) to the classpath:

```
-classpath /com/foo/acct:/com/foo:.
```

This classpath is identical to the previous one EXCEPT that the dot (.) at the end of the declaration instructs `java` or `javac` to *also* search for class files in the current directory. (Remember, we're talking about class files—when you're telling `javac` which `.java` file to compile, `javac` looks in the current directory by default.)

It's also important to remember that classpaths are searched from left to right. Therefore in a situation where classes with duplicate names are located in several different directories in the following classpaths, different results will occur:

```
-classpath /com:/foo:.
```

is not the same as

```
-classpath .:/foo:/com
```

Finally, the `java` command allows you to abbreviate `-classpath` with `-cp`. The Java documentation is inconsistent about whether the `javac` command allows the `-cp` abbreviation. On most machines it does, but there are no guarantees.

Packages and Searching

When you start to put classes into packages, and then start to use classpaths to find these classes, things can get tricky. The exam creators knew this, and they tried to create an especially devilish set of package/classpath questions with which to confound you. Let's start off by reviewing packages. In the following code:

```
package com.foo;
public class MyClass { public void hi() { } }
```

we're saying that `MyClass` is a member of the `com.foo` package. This means that the fully qualified name of the class is now `com.foo.MyClass`. Once a class is in a package, the package part of its fully qualified name is *atomic*—it can never be divided. You can't split it up on the command-line, and you can't split it up in an `import` statement.

Now let's see how we can use `com.foo.MyClass` in another class:

```
package com.foo;
public class MyClass { public void hi() { } }
```

And in another file:

```
import com.foo.MyClass;   // either import will work
import com.foo.*;

public class Another {
  void go() {
    MyClass m1 = new MyClass();                    // alias name
    com.foo.MyClass m2 = new com.foo.MyClass();  // full name
    m1.hi();
    m2.hi();
  }
}
```

It's easy to get confused when you use `import` statements. The preceding code is perfectly legal. The `import` statement is like an alias for the class's fully qualified name. You define the fully qualified name for the class with an `import` statement (or with a wildcard in an `import` statement of the package). Once you've defined the fully qualified name, you can use the "alias" in your code—but the alias is referring back to the fully qualified name.

Now that we've reviewed packages, let's take a look at how they work in conjunction with classpaths and command lines. First we'll start off with the idea that when you're searching for a class using its fully qualified name, that fully qualified name relates closely to a specific directory structure. For instance, relative to your current directory, the class whose source code is

```
package com.foo;
public class MyClass { public void hi() { } }
```

would *have* to be located here:

```
com/foo/MyClass.class
```

In order to find a class in a package, you have to have a directory in your classpath that has the package's leftmost entry (the package's "root") as a subdirectory.

This is an important concept, so let's look at another example:

```
import com.wickedlysmart.Utils;
class TestClass {
  void doStuff() {
    Utils u = new Utils();                        // simple name
    u.doX("arg1", "arg2");
    com.wickedlysmart.Date d =
                new com.wickedlysmart.Date();     // full name
    d.getMonth("Oct");
  }
}
```

In this case we're using two classes from the package `com.wickedlysmart`. For the sake of discussion we imported the fully qualified name for the `Utils` class, and we didn't for the `Date` class. The *only* difference is that because we listed `Utils` in an `import` statement, we didn't have to type its fully qualified name inside the class. In both cases the package is `com.wickedlysmart`. When it's time to compile or run `TestClass`, the classpath will have to include a directory with the following attributes:

- A subdirectory named `com` (we'll call this the "package root" directory)
- A subdirectory in `com` named `wickedlysmart`
- Two files in `wickedlysmart` named `Utils.class` and `Date.class`

Finally, the directory that has all of these attributes has to be accessible (via a classpath) in one of two ways:

1. The path to the directory must be absolute, in other words, from the root (the file system root, not the package root).

   or

2. The path to the directory has to be correct relative to the current directory.

Relative and Absolute Paths

A classpath is a collection of one or more paths. Each path in a classpath is either an absolute path or a relative path. An absolute path in Unix begins with a forward slash (/) (on Windows it would be something like c:\). The leading slash indicates that this path is starting from the root directory of the system. Because it's starting from the root, it doesn't *matter* what the current directory is—*a directory's absolute path is always the same*. A *relative* path is one that does NOT start with a slash. Here's an example of a full directory structure, and a classpath:

```
/ (root)
     |
     |--dirA
          |
          |-- dirB
                |
                |--dirC

-cp dirB:dirB/dirC
```

In this example, `dirB` and `dirB/dirC` are relative paths (they don't start with a slash /). Both of these relative paths are meaningful *only* when the current directory is `dirA`. Pop Quiz! If the current directory is `dirA`, and you're searching for class files, and you use the classpath described above, which directories will be searched?

`dirA? dirB? dirC?`

Too easy? How about the same question if the current directory is the root (/)? When the current directory is `dirA`, then `dirB` and `dirC` will be searched, but not

dirA (remember, we didn't specify the current directory by adding a dot (.) to the classpath). When the current directory is root, since dirB is not a direct subdirectory of root, no directories will be searched. Okay, how about if the current directory is dirB? Again, no directories will be searched! This is because dirB doesn't have a subdirectory named dirB. In other words, Java will look in dirB for a directory named dirB (which it won't find), without realizing that it's already in dirB.

Let's use the same directory structure and a different classpath:

```
/ (root)
     |
     |--dirA
          |
          |-- dirB
                |
                |--dirC

-cp /dirB:/dirA/dirB/dirC
```

In this case, what directories will be searched if the current directory is dirA? How about if the current directory is root? How about if the current directory is dirB? In this case, both paths in the classpath are absolute. It doesn't matter what the current directory is; since absolute paths are specified the search results will always be the same. Specifically, only dirC will be searched, regardless of the current directory. The first path (/dirB) is invalid since dirB is not a direct subdirectory of root, so dirB will never be searched. And, one more time, for emphasis, since dot (.) is not in the classpath, the current directory will only be searched if it happens to be described elsewhere in the classpath (in this case, dirC).

CERTIFICATION OBJECTIVE

JAR Files (Objective 7.5)

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.

JAR Files and Searching

Once you've built and tested your application, you might want to bundle it up so that it's easy to distribute and easy for other people to install. One mechanism that Java provides for these purposes is a JAR file. JAR stands for Java Archive. JAR files are used to compress data (similar to ZIP files) and to archive data.

Let's say you've got an application that uses many different classes that are located in several different packages. Here's a partial directory tree:

```
test
   |
   |--UseStuff.java
   |--ws
       |
       |--(create MyJar.jar here)
       |--myApp
             |
             |--utils
             |      |
             |      |--Dates.class       (package myApp.utils;)
             |      |--Conversions.class    "          "
             |
             |--engine
                    |
                    |--rete.class        (package myApp.engine;)
                    |--minmax.class         "          "
```

You can create a single JAR file that contains all of the class files in `myApp`, and also maintains `myApp`'s directory structure. Once this JAR file is created, it can be moved from place to place, and from machine to machine, and all of the classes in the JAR file can be accessed via classpaths and used by `java` and `javac`. All of this can happen without ever unJARing the JAR file. Although you won't need to know how to make JAR files for the exam, let's make the current directory `ws`, and then make a JAR file called `MyJar.jar`:

```
cd ws
jar -cf MyJar.jar myApp
```

The `jar` command will create a JAR file called `MyJar.jar` and it will contain the `myApp` directory and `myApp`'s entire subdirectory tree and files. You can look at the contents of the JAR file with the next command (this isn't on the exam either):

```
jar -tf MyJar.jar
```

which will list the JAR's contents something like this:

```
META-INF/
META-INF/MANIFEST.MF
myApp/
myApp/.DS_Store
myApp/utils/
myApp/utils/Dates.class
myApp/utils/Conversions.class
myApp/engine/
myApp/engine/rete.class
myApp/engine/minmax.class
```

Okay, now back to exam stuff. Finding a JAR file using a classpath is similar to finding a package file in a classpath. The difference is that when you specify a path for a JAR file, *you must include the name of the JAR file at the end of the path.* Let's say you want to compile `UseStuff.java` in the `test` directory, and `UseStuff.java` needs access to a class contained in `myApp.jar`. To compile `UseStuff.java` you would say

```
cd test
javac -classpath ws/myApp.jar UseStuff.java
```

Compare the use of the JAR file to using a class in a package. If `UseStuff.java` needed to use classes in the `myApp.utils` package, and the class was not in a JAR, you would say

```
cd test
javac -classpath ws UseStuff.java
```

Remember when using a classpath, the last directory in the path must be the super-directory of the *root* directory for the package. (In the preceding example, `myApp` is the root directory of the package `myApp.utils`.) Notice that `myApp` can be the root directory for more than one package (`myApp.utils` and `myApp.engine`), and the `java` and `javac` commands can find what they need across multiple *peer* packages like this. In other words, if `ws` is on the classpath and `ws` is the super-directory of `myApp`, then classes in both the `myApp.utils` and `myApp.engine` packages will be found.

exam Watch

When you use an import statement you are declaring only one package. When you say `import java.util.*;` ***you are saying "Use the short name for all of the classes in the*** `java.util` ***package." You're NOT getting the*** `java.util.jar` ***classes or*** `java.util.regex` ***packages! Those packages are totally independent of each other; the only thing they share is the same "root" directory, but they are not the same packages. As a corollary, you can't say*** `import java.*;` ***in the hopes of importing multiple packages—just remember, an*** `import` ***statement can import only a single package.***

Using .../jre/lib/ext with JAR files

When you install Java, you end up with a huge directory tree of Java-related stuff, including the JAR files that contain the classes that come standard with J2SE. As we discussed earlier, `java` and `javac` have a list of places that they access when searching for class files. Buried deep inside of your Java directory tree is a subdirectory tree named `jre/lib/ext`. If you put JAR files into the `ext` subdirectory, `java` and `javac` can find them, and use the class files they contain. You don't have to mention these subdirectories in a classpath statement—searching this directory is a function that's built right into Java. Sun recommends, however, that you use this feature only for your own internal testing and development, and not for software that you intend to distribute.

exam Watch

It's possible to create environment variables that provide an alias for long classpaths. The classpath for some of the JAR files in J2SE can be quite long, and so it's common for such an alias to be used when defining a classpath. If you see something like `JAVA_HOME` ***or*** `$JAVA_HOME` ***in an exam question it just means "That part of the absolute classpath up to the directories we're specifying explicitly." You can assume that the*** `JAVA_HOME` ***literal means this, and is pre-pended to the partial classpath you see.***

CERTIFICATION OBJECTIVE

Using Static Imports (Exam Objective 7.1)

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

Note: In Chapter 1 we covered most of what's defined in this objective, but we saved static imports for this chapter.

Static Imports

We've been using `import` statements throughout the book. Ultimately, the only value `import` statements have is that they save typing and they can make your code easier to read. In Java 5, the `import` statement was enhanced to provide even greater keystroke-reduction capabilities…although some would argue that this comes at the expense of readability. This new feature is known as *static imports*. Static imports can be used when you want to use a class's `static` members. (You can use this feature on classes in the API and on your own classes.) Here's a "before and after" example:

Before static imports:

```
public class TestStatic {
  public static void main(String[] args) {
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Integer.toHexString(42));
  }
}
```

After static imports:

```
import static java.lang.System.out;             // 1
import static java.lang.Integer.*;              // 2
public class TestStaticImport {
  public static void main(String[] args)  {
    out.println(MAX_VALUE);                     // 3
    out.println(toHexString(42));               // 4
  }
}
```

Both classes produce the same output:

```
2147483647
2a
```

Let's look at what's happening in the code that's using the static import feature:

1. Even though the feature is commonly called "static import" the syntax MUST be `import static` followed by the fully qualified name of the `static` member you want to import, or a wildcard. In this case we're doing a static import on the `System` class's `out` object.
2. In this case we might want to use several of the `static` members of the `java.lang.Integer` class. This static import statement uses the wildcard to say "I want to do static imports of ALL the `static` members in this class."
3. Now we're finally seeing the *benefit* of the static import feature! We didn't have to type the `System` in `System.out.println`! Wow! Second, we didn't have to type the `Integer` in `Integer.MAX_VALUE`. So in this line of code we were able to use a shortcut for a `static` method AND a constant.
4. Finally, we do one more shortcut, this time for a method in the `Integer` class.

We've been a little sarcastic about this feature, but we're not the only ones. We're not convinced that saving a few keystrokes is worth possibly making the code a little harder to read, but enough developers requested it that it was added to the language.

Here are a couple of rules for using static imports:

- You must say `import static`; you can't say `static import`.
- Watch out for ambiguously named `static` members. For instance, if you do a static import for both the `Integer` class and the `Long` class, referring to `MAX_VALUE` will cause a compiler error, since both `Integer` and `Long` have a `MAX_VALUE` constant, and Java won't know which `MAX_VALUE` you're referring to.
- You can do a static import on `static` object references, constants (remember they're `static` and `final`), and `static` methods.

CERTIFICATION SUMMARY

We started by exploring the `javac` command more deeply. The `-d` option allows you to put class files generated by compilation into whatever directory you want to. The `-d` option lets you specify the destination of newly created class files.

Next we talked about some of the options available through the `java` application launcher. We discussed the ordering of the arguments `java` can take, including `[options] class [args]`. We learned how to query and update system properties in code and at the command line using the `-D` option.

The next topic was handling command-line arguments. The key concepts are that these arguments are put into a String array, and that the first argument goes into array element 0, the second argument into array element 1, and so on.

We turned to the important topic of how `java` and `javac` search for other class files when they need them, and how they use the same algorithm to find these classes. There are search locations predefined by Sun, and additional search locations, called *classpaths* that are user defined. The syntax for Unix classpaths is different than the syntax for Windows classpaths, and the exam will tend to use Unix syntax.

The topic of packages came next. Remember that once you put a class into a package, its name is atomic—in other words, it can't be split up. There is a tight relationship between a class's fully qualified package name and the directory structure in which the class resides.

JAR files were discussed next. JAR files are used to compress and archive data. They can be used to archive entire directory tree structures into a single JAR file. JAR files can be searched by `java` and `javac`.

We finished the chapter by discussing a new Java 5 feature, static imports. This is a convenience-only feature that reduces keying long names for `static` members in the classes you use in your programs.

TWO-MINUTE DRILL

Using javac and java (Objective 7.2)

- ❑ Use `-d` to change the destination of a class file when it's first generated by the `javac` command.
- ❑ The `-d` option can build package-dependent destination classes on-the-fly if the *root* package directory already exists.
- ❑ Use the `-D` option in conjunction with the `java` command when you want to set a system property.
- ❑ System properties consist of name=value pairs that must be appended directly behind the `-D`, for example, `java -Dmyproperty=myvalue`.
- ❑ Command-line arguments are always treated as Strings.
- ❑ The `java` command-line argument 1 is put into array element 0, argument 2 is put into element 1, and so on.

Searching with java and javac (Objective 7.5)

- ❑ Both `java` and `javac` use the same algorithms to search for classes.
- ❑ Searching begins in the locations that contain the classes that come standard with J2SE.
- ❑ Users can define secondary search locations using classpaths.
- ❑ Default classpaths can be defined by using OS environment variables.
- ❑ A classpath can be declared at the command line, and it overrides the default classpath.
- ❑ A single classpath can define many different search locations.
- ❑ In Unix classpaths, forward slashes (`/`) are used to separate the directories that make up a path. In Windows, backslashes (`\`) are used.

- In Unix, colons (:) are used to separate the paths within a classpath. In Windows, semicolons (;) are used.
- In a classpath, to specify the current directory as a search location, use a dot (.)
- In a classpath, once a class is found, searching stops, so the order of locations to search is important.

Packages and Searching (Objective 7.5)

- When a class is put into a package, its fully qualified name must be used.
- An `import` statement provides an alias to a class's fully qualified name.
- In order for a class to be located, its fully qualified name must have a tight relationship with the directory structure in which it resides.
- A classpath can contain both relative and absolute paths.
- An absolute path starts with a `/` or a `\`.
- Only the final directory in a given path will be searched.

JAR files (Objective 7.5)

- An entire directory tree structure can be archived in a single JAR file.
- JAR files can be searched by `java` and `javac`.
- When you include a JAR file in a classpath, you must include not only the directory in which the JAR file is located, but the name of the JAR file too.
- For testing purposes, you can put JAR files into `.../jre/lib/ext`, which is somewhere inside the Java directory tree on your machine.

Static Imports (Objective 7.1)

- You must start a static import statement like this: `import static`
- You can use static imports to create shortcuts for `static` members (static variables, constants, and methods) of any class.

SELF TEST

1. Given these classes in different files:

```
package xcom;
public class Useful {
  int increment(int x) { return ++x; }
}

import xcom.*;                                    // line 1
class Needy3 {
  public static void main(String[] args) {
    xcom.Useful u = new xcom.Useful();            // line 2
    System.out.println(u.increment(5));
  }
}
```

Which statements are true? (Choose all that apply.)

A. The output is 0.

B. The output is 5.

C. The output is 6.

D. Compilation fails.

E. The code compiles if line 1 is removed.

F. The code compiles if line 2 is changed to read

```
Useful u = new Useful();
```

2. Given the following directory structure:

```
org
  | -- Robot.class
  |
  | -- ex
       |-- Pet.class
       |
       |-- why
             |-- Dog.class
```

And the following source file:

```
class MyClass {
  Robot r;
  Pet p;
  Dog d;
}
```

Which statement(s) *must* be added for the source file to compile? (Choose all that apply.)

A. `package org;`

B. `import org.*;`

C. `package org.*;`

D. `package org.ex;`

E. `import org.ex.*;`

F. `package org.ex.why;`

G. `package org.ex.why.Dog;`

3. Given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax.

4. Given:

```
import static java.lang.System.*;
class _ {
  static public void main(String... __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `-A.`

E. `_-A.`

F. Compilation fails.

G. An exception is thrown at runtime.

5. Given the default classpath:

```
/foo
```

And this directory structure:

```
foo
  |
  test
```

```
|
xcom
   |--A.class
   |--B.java
```

And these two files:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Which allows `B.java` to compile? (Choose all that apply.)

A. Set the current directory to `xcom` then invoke
```
javac B.java
```

B. Set the current directory to `xcom` then invoke
```
javac -classpath . B.java
```

C. Set the current directory to `test` then invoke
```
javac -classpath . xcom/B.java
```

D. Set the current directory to `test` then invoke
```
javac -classpath xcom B.java
```

E. Set the current directory to `test` then invoke
```
javac -classpath xcom:. B.java
```

6. Given two files:

```
package xcom;
public class Stuff {
  public static final int MY_CONSTANT = 5;
  public static int doStuff(int x) { return (x++)*x; }
}

import xcom.Stuff.*;
import java.lang.System.out;
class User {
  public static void main(String[] args) {
    new User().go();
```

```
  }
  void go() { out.println(doStuff(MY_CONSTANT)); }
}
```

What is the result?

A. 25

B. 30

C. 36

D. Compilation fails.

E. An exception is thrown at runtime.

7. Given two files:

```
a=b.java
c_d.class
```

Are in the current directory, which command-line invocation(s) could complete without error? (Choose all that apply.)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

8. Given three files:

```
package xcom;
public class A {
  // insert code here
}

package xcom;
public class B extends A {public void doB() { System.out.println("B.doB"); } }
```

```
import xcom.B;
class TestXcom {
  public static void main(String[] args) {
    B b = new B();  b.doB();  b.go();
  }
}
```

Which, inserted at `// insert code here` will allow all three files to compile? (Choose all that apply.)

A. `void go() { System.out.println("a.go"); }`

B. `public void go() { System.out.println("a.go"); }`

C. `private void go() { System.out.println("a.go"); }`

D. `protected void go() { System.out.println("a.go"); }`

E. None of these options will allow the code to compile.

9. Given:

```
class TestProps {
  public static void main(String[] args) {
    String s = System.getProperty("aaa","bbb");
  }
}
```

And the command-line invocation:

```
java -Daaa=ccc TestProps
```

What is always true? (Choose all that apply.)

A. The value of property `aaa` is `aaa`.

B. The value of property `aaa` is `bbb`.

C. The value of property `aaa` is `ccc`.

D. The value of property `bbb` is `aaa`.

E. The value of property `bbb` is **ccc**.

F. The invocation will not complete without error.

10. If three versions of `MyClass.java` exist on a file system:

Version 1 is in `/foo/bar`
Version 2 is in `/foo/bar/baz`
Version 3 is in `/foo/bar/baz/bing`

And the system's classpath includes:

```
/foo/bar/baz
```

And this command line is invoked from `/foo`

```
javac -classpath /foo/bar/baz/bing:/foo/bar MyClass.java
```

Which version will be used by `javac`?

A. `/foo/MyClass.java`
B. `/foo/bar/MyClass.java`
C. `/foo/bar/baz/MyClass.java`
D. `/foo/bar/baz/bing/MyClass.java`
E. The result is not predictable.

11. Which are true? (Choose all that apply.)

A. The `java` command can access classes from more than one package, from a single JAR file.
B. JAR files can be used with the `java` command but not with the `javac` command.
C. In order for JAR files to be used by `java`, they MUST be placed in the /jre/lib/ext subdirectory within the J2SE directory tree.
D. In order to specify the use of a JAR file on the command line, the JAR file's path and filename MUST be included.
E. When a part of a directory tree that includes subdirectories with files is put into a JAR file, all of the files are saved in the JAR file, but the subdirectory structure is lost.

12. Given two files:

```
package pkg;
public class Kit {
  public String glueIt(String a, String b) { return a+b; }
}

import pkg.*;
class UseKit {
  public static void main(String[] args) {
    String s = new Kit().glueIt(args[1], args[2]);
    System.out.println(s);
  }
}
```

And the following sub-directory structure:

```
test
   |--UseKit.class
   |
   com
     |--KitJar.jar
```

If the current directory is test, and the file `pkg/Kit.class` is in `KitJar.jar`, which command line will produce the output `bc` ? (Choose all that apply.)

A. `java UseKit b c`
B. `java UseKit a b c`
C. `java -classpath com UseKit b c`
D. `java -classpath com:. UseKit b c`
E. `java -classpath com/KitJar.jar UseKit b c`
F. `java -classpath com/KitJar.jar UseKit a b c`
G. `java -classpath com/KitJar.jar:. UseKit b c`
H. `java -classpath com/KitJar.jar:. UseKit a b c`

SELF TEST ANSWERS

1. Given these classes in different files:

```
package xcom;
public class Useful {
  int increment(int x) { return ++x; }
}

import xcom.*;                                  // line 1
class Needy3 {
  public static void main(String[] args) {
    xcom.Useful u = new xcom.Useful();          // line 2
    System.out.println(u.increment(5));
  }
}
```

Which statements are true? (Choose all that apply.)

A. The output is 0.

B. The output is 5.

C. The output is 6.

D. Compilation fails.

E. The code compiles if line 1 is removed.

F. The code compiles if line 2 is changed to read

```
Useful u = new Useful();
```

Answer:

☑ **D** is correct. The `increment()` method must be marked `public` to be accessed outside of the package. If `increment()` was `public`, **C, E,** and **F** would be correct.

☒ **A** and **B** are incorrect output, even if `increment()` is `public`. (Objective 7.1)

2. Given the following directory structure:

```
org
  | -- Robot.class
```

```
|
| -- ex
      |-- Pet.class
      |
      |-- why
            |-- Dog.class
```

And the following source file:

```
class MyClass {
  Robot r;
  Pet p;
  Dog d;
}
```

Which statement(s) *must* be added for the source file to compile? (Choose all that apply.)

A. `package org;`

B. `import org.*;`

C. `package org.*;`

D. `package org.ex;`

E. `import org.ex.*;`

F. `package org.ex.why;`

G. `package org.ex.why.Dog;`

Answer:

☑ **B, E,** and **F** are required. The only way to access class Dog is via **F**, which is a package statement. Since you can have only one package statement in a source file, you have to get access to class Robot and class Pet using `import` statements. Option **B** accesses Robot, and option **E** accesses Pet.

☒ **A, C, D,** and **G** are incorrect based on the above. Also, **C** and **G** are incorrect syntax.

(Objective 7.1)

3. Given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax.

Answer:

☑ **C** and **E** are correct syntax for static imports. Line 4 isn't making use of `static imports`, so the code will also compile with none of the imports.

☒ **A, B, D,** and **F** are incorrect based on the above. (Objective 7.1)

4. Given:

```
import static java.lang.System.*;
class _ {
  static public void main(String... __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `-A.`

E. `_-A.`

F. Compilation fails.

G. An exception is thrown at runtime.

Answer:

☑ **B** is correct. This question is using valid (but inappropriate and weird) identifiers, static imports, var-args in `main()`, and pre-incrementing logic.

☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (Objective 7.2)

5. Given the default classpath:

```
/foo
```

And this directory structure:

```
foo
  |
  test
     |
     xcom
        |--A.class
        |--B.java
```

And these two files:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Which allows `B.java` to compile? (Choose all that apply.)

A. Set the current directory to `xcom` then invoke
```
javac B.java
```

B. Set the current directory to `xcom` then invoke
```
javac -classpath . B.java
```

C. Set the current directory to `test` then invoke
```
javac -classpath . xcom/B.java
```

D. Set the current directory to `test` then invoke
```
javac -classpath xcom B.java
```

E. Set the current directory to `test` then invoke
```
javac -classpath xcom:. B.java
```

Answer:

☑ **C** is correct. In order for `B.java` to compile, the compiler first needs to be able to find `B.java`. Once it's found `B.java` it needs to find `A.class`. Because `A.class` is in the `xcom` package the compiler won't find `A.class` if it's invoked from the `xcom` directory. Remember that the `-classpath` isn't looking for `B.java`, it's looking for whatever classes `B.java` needs (in this case `A.class`).

☒ **A, B,** and **D** are incorrect based on the above. **E** is incorrect because the compiler can't find `B.java`. (Objective 7.2)

6. Given two files:

```
package xcom;
public class Stuff {
  public static final int MY_CONSTANT = 5;
  public static int doStuff(int x) { return (x++)*x; }
}

import xcom.Stuff.*;
import java.lang.System.out;
class User {
  public static void main(String[] args) {
    new User().go();
  }
  void go() { out.println(doStuff(MY_CONSTANT)); }
}
```

What is the result?

A. 25

B. 30

C. 36

D. Compilation fails.

E. An exception is thrown at runtime.

Answer:

☑ **D** is correct. To import static members, an `import` statement must begin: `import static`.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 7.1)

7. Given two files:

```
a=b.java
c_d.class
```

Are in the current directory, which command-line invocation(s) could complete without error? (Choose all that apply.)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

Answer:

☑ **A** is correct. The `-D` flag is NOT a compiler flag, and the name=value pair that is associated with the `-D` must follow the `-D` with no spaces.

☒ **B, C,** and **D** are incorrect based on the above. (Objective 7.2)

8. Given three files:

```
package xcom;
public class A {
  // insert code here
}
```

```
package xcom;
public class B extends A {public void doB() { System.out.println("B.doB"); } }

import xcom.B;
class TestXcom {
  public static void main(String[] args) {
    B b = new B();  b.doB();  b.go();
  }
}
```

Which, inserted at `// insert code here` will allow all three files to compile? (Choose all that apply.)

A. `void go() { System.out.println("a.go"); }`

B. `public void go() { System.out.println("a.go"); }`

C. `private void go() { System.out.println("a.go"); }`

D. `protected void go() { System.out.println("a.go"); }`

E. None of these options will allow the code to compile.

Answer:

☑ **B** is correct. The `public` access modifier is the only one that allows code from outside a package to access methods in a package—regardless of inheritance.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 7.1)

9. Given:

```
class TestProps {
  public static void main(String[] args) {
    String s = System.getProperty("aaa","bbb");
  }
}
```

And the command-line invocation:

```
java -Daaa=ccc TestProps
```

What is always true? (Choose all that apply.)

A. The value of property `aaa` is `aaa`.
B. The value of property `aaa` is `bbb`.
C. The value of property `aaa` is `ccc`.
D. The value of property `bbb` is `aaa`.
E. The value of property `bbb` is `ccc`.
F. The invocation will not complete without error.

Answer:

☑ **C** is correct. The value of aaa is set at the command line. If `aaa` had no value when getProperty was invoked, then `aaa` would have been set to `bbb`.
☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.2)

10. If three versions of `MyClass.java` exist on a file system:

Version 1 is in `/foo/bar`
Version 2 is in `/foo/bar/baz`
Version 3 is in `/foo/bar/baz/bing`

And the system's classpath includes:

```
/foo/bar/baz
```

And this command line is invoked from `/foo`

```
javac -classpath /foo/bar/baz/bing:/foo/bar MyClass.java
```

Which version will be used by `javac`?

A. `/foo/MyClass.java`
B. `/foo/bar/MyClass.java`
C. `/foo/bar/baz/MyClass.java`
D. `/foo/bar/baz/bing/MyClass.java`
E. The result is not predictable.

Answer:

☒ **D** is correct. A -classpath included with a `javac` invocation overrides a system classpath. When `javac` is using any classpath, it reads the classpath from left to right, and uses the first match it finds.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 7.5)

11. Which are true? (Choose all that apply.)

A. The `java` command can access classes from more than one package, from a single JAR file.

B. JAR files can be used with the `java` command but not with the `javac` command.

C. In order for JAR files to be used by `java`, they MUST be placed in the /jre/lib/ext subdirectory within the J2SE directory tree.

D. In order to specify the use of a JAR file on the command line, the JAR file's path and filename MUST be included.

E. When a part of a directory tree that includes subdirectories with files is put into a JAR file, all of the files are saved in the JAR file, but the subdirectory structure is lost.

Answer:

☑ **A** and **D** are correct.

☒ **B** is incorrect because `javac` can also use JAR files. **C** is incorrect because JARs can be located in ../jre/lib/ext, but they can also be accessed if they live in other locations. **E** is incorrect, JAR files maintain directory structures. (Objective 7.5)

12. Given two files:

```
package pkg;
public class Kit {
  public String glueIt(String a, String b) { return a+b; }
}
import pkg.*;
class UseKit {
  public static void main(String[] args) {
    String s = new Kit().glueIt(args[1], args[2]);
    System.out.println(s);
  }
}
```

And the following sub-directory structure:

```
test
   |--UseKit.class
   |
   com
     |--KitJar.jar
```

If the current directory is `test`, and the file `pkg/Kit.class` is in `KitJar.jar`, which command line will produce the output `bc` ? (Choose all that apply.)

A. `java UseKit b c`

B. `java UseKit a b c`

C. `java -classpath com UseKit b c`

D. `java -classpath com:. UseKit b c`

E. `java -classpath com/KitJar.jar UseKit b c`

F. `java -classpath com/KitJar.jar UseKit a b c`

G. `java -classpath com/KitJar.jar:. UseKit b c`

H. `java -classpath com/KitJar.jar:. UseKit a b c`

Answer:

☑ **H** is correct.

☒ **A, C, E,** and **G** are incorrect if for no other reason than `args[]` is 0-based.
B, D, and **F** are incorrect because java needs a classpath that specifies two directories, one for the class file (the . directory), and one for the JAR file (the `com` directory). Remember, to find a JAR file, the classpath must include the name of the JAR file, not just its directory. (Objective 7.5)