



# I

## Packaging, Compiling, and Interpreting Java Code

### CERTIFICATION OBJECTIVES

- Understanding Packages
- Understanding Package-Derived Classes
- Compiling and Interpreting Java Code



Two-Minute Drill

Q&A Self Test

**S**ince you are holding this book or reading an electronic version of it, you must have an affinity for Java. You must also have the desire to let everyone know through the Sun Certified Java Associate certification process that you are truly Java savvy. As such, you should either be—or have the desire to be—a Java programmer, and in the long term, a true Java developer. You may be or plan to be a project manager heading up a team of Java programmers and/or developers. In this case, you will need to acquire a basic understanding of the Java language and its technologies. In either case, this book is for you.

To start, you may be wondering about the core functional elements provided by the basic Java Standard Edition platform in regards to libraries and utilities, and how these elements are organized. This chapter answers these questions by discussing Java packages and classes, along with their packaging, compilation, and interpretation processes.

When you have finished this chapter, you will have a firm understanding of packaging Java classes, high-level details of common Java SE packages, and the fundamentals of Java's compilation and interpretation tools.

### CERTIFICATION OBJECTIVE

## Understanding Packages

*Exam Objective 5.1 Describe the purpose of packages in the Java language, and recognize the proper use of import and package statements.*

Packaging is a common approach used to organize related classes and interfaces. Most reusable code is packaged. Unpackaged classes are commonly found in books and online tutorials, as well as software applications with a narrow focus. This section will show you how and when to package your Java classes and how to import external classes from your Java packages. The following topics will be covered:

- Package design
- Package and import statements

## Package Design

Packages are thought of as containers for classes, but actually they define where classes will be located in the hierarchical directory structure. Packaging is encouraged by Java coding standards to decrease the likelihood of classes colliding. Packaging your classes also promotes code reuse, maintainability, and the object-oriented principle of encapsulation and modularity.

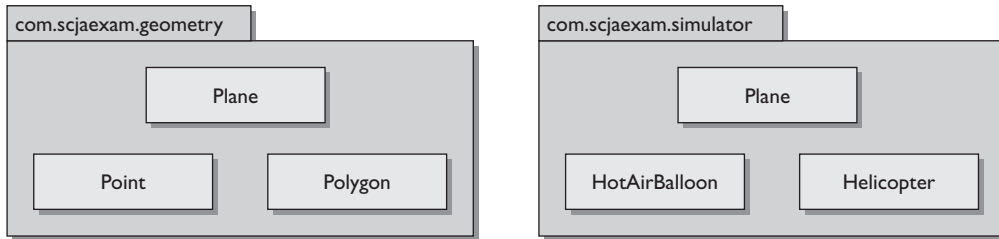
When you design Java packages, such as the grouping of classes, the following key areas (shown in Table 1-1) should be considered.

Let's take a look at a real-world example. As program manager you need two sets of classes with unique functionality that will be used by the same end product. You task Developer A to build the first set and Developer B to build the second. You do not define the names of the classes, but you do define the purpose of the package and what it must contain. Developer A is to create several geometry-based classes including a point class, a polygon class, and a plane class. Developer B is to build classes that will be included for simulation purposes, including objects such as hot air balloons, helicopters, and airplanes. You send them off to build their classes (without having them package their classes). Come delivery time, they both give you a class named `Plane.java`—that is, one for the geometry plane class and one for the airplane class. Now you have a problem because both of these source files (class files, too) cannot coexist in the same directory since they have the same name. The solution is packaging. If you had designated package names to the developers, this conflict never would have happened (as shown in Figure 1-1). The lesson learned is: Always package your code, unless your coding project is trivial in nature.

**TABLE 1-1**

Package  
Attributes  
Considerations

Package Attribute	Benefits of Applying the Package Attribute
Class Coupling	Package dependencies are reduced with class coupling.
System Coupling	Package dependencies are reduced with system coupling.
Package Size	Typically, larger packages support reusability, whereas smaller packages support maintainability.
Maintainability	Often, software changes can be limited to a single package when the package houses focused functionality.
Naming	Consider conventions when naming your packages. Use reverse domain name for the package structure. Use lowercase characters delimited with underscores to separate words in package names.

**FIGURE 1-1** Separate packaging of classes with the same names

## package and import Statements

You should now have a general idea of when and why to package your source files. Now you need to know exactly how. To place a source file into a package, use the package statement at the beginning of that file. You may use zero or one package statements per source file. To import classes from other packages into your source file, use the import statement. The `java.lang` package that houses the core language classes is imported by default.

The following code listing shows usage of the package and import statements. You can continue to come back to this listing as we discuss the package and import statements in further detail throughout the chapter.

```
package com.scjaexam.tutorial; // Package statement
/* Imports class ArrayList from the java.util package */
import java.util.ArrayList;
/* Imports all classes from the java.io package */
import java.io.*;
public class MainClass {
    public static void main(String[] args) {
        /* Creates console from java.io package */
        Console console = System.console();
        String planet = console.readLine("\nEnter your favorite
planet: ");
        /* Creates list for planets */
        ArrayList planetList = new ArrayList();
        planetList.add(planet); // Adds users input to the list
        planetList.add("Gliese 581 c"); // Adds a string to the list
        System.out.println("\nTwo cool planets: " + planetList);
    }
}
$ Enter your favorite planet: Jupiter
$ Two cool planets: [Jupiter, Gliese 581 c]
```

## The package Statement

The package statement includes the package keyword, followed by the package path delimited with periods. Table 1-2 shows valid examples of package statements. Package statements have the following attributes:

- Package statements are optional.
- Package statements are limited to one per source file.
- Standard coding convention for package statements reverses the domain name of the organization or group creating the package. For example, the owners of the domain name `scjaexam.com` may use the following package name for a utilities package: `com.scjaexam.utilities`.
- Package names equate to directory structures. The package name `com.scjaexam.utils` would equate to the directory `com/scjaexam/utils`.
- The package names beginning with `java.*` and `javax.*` are reserved for use by JavaSoft, the business unit of Sun Microsystems that is responsible for Java technologies.
- Package names should be lowercase. Individual words within the package name should be separated by underscores.

The Java SE API contains several packages. These packages are detailed in Sun's Online JavaDoc documentation at <http://java.sun.com/javase/6/docs/api>.

Common packages you will see on the exam are packages for the Java Abstract Window Toolkit API, the Java Swing API, the Java Basic Input/Output API, the Java Networking API, the Java Utilities API, and the core Java Language API. You will need to know the basic functionality that each package/API contains.

## The import Statement

An import statement allows you to include source code from other classes into a source file at compile time. In J2SE 1.4, the import statement includes the

**TABLE 1-2**

Valid package  
Statements

package Statement	Related Directory Structure
<code>package java.net;</code>	<code>[directory_path]\java\net\</code>
<code>package com.scjaexam.utilities;</code>	<code>[directory_path]\com\scjaexam\utilities\</code>
<code>package package_name;</code>	<code>[directory_path]\package_name\</code>

SCENARIO & SOLUTION	
To paint basic graphics and images, which package should you use?	You will need to use the Java AWT API package.
To create lightweight components for GUI, which package should you use?	You will need to use the Java Swing API package.
To utilize data streams, which package should you use?	You will need to use the Java Basic I/O package.
To develop a networking application, which package should you use?	You will need to use the Java Networking API package.
To work with the collections framework, event model, and date/time facilities, which package should you use?	You will need to use the Java Utilities API package.
To utilize the core Java classes and interfaces, which package should you use?	You will need to use the core Java Language package.

`import` keyword followed by the package path delimited with periods and ending with a class name or an asterisk, as shown in Table 1-3. These `import` statements occur after the optional `package` statement and before the class definition. Each `import` statement can only relate to one package.

TABLE 1-3

Valid import Statements

import Statement	Definition
<code>import java.net.*;</code>	Imports all of the classes from the package <code>java.net</code> .
<code>import java.net.URL;</code>	Imports only the <code>URL</code> class from the package <code>java.net</code> .
<code>import static java.awt.Color.*;</code>	Imports all static members of the <code>Color</code> class of the package <code>java.awt</code> (J2SE 5.0 onward only).
<code>import static java.awt.color.ColorSpace .CS_GRAY;</code>	Imports the static member <code>CS_GRAY</code> of the <code>Color</code> class of the package <code>java.awt</code> (J2SE 5.0 onward only).



**For maintenance purposes, it is better to explicitly import your classes. This will allow the programmer to quickly determine which external classes are used throughout the class. As an example, rather than using `import java.util.*`, use `import java.util.Vector`. In this real-world example, the coder would quickly see (with the latter approach) that the class only imports one class and it is a collection type. In this case, it is a legacy type and the determination to update the class with a newer collection type could be done quickly.**

C and C++ programmers will see some look-and-feel similarities between Java's import statement and C/C++'s `#include` statement, even though there isn't a direct mapping in functionality.

## exam

### Watch

**Static imports are a new feature to Java SE 5.0. Static imports allow you to import static members. The following example statements would be valid in Java SE 5.0, but would be invalid for J2SE 1.4.**

```
/* Import static member ITALY */
import static java.util.Locale.ITALY;
/* Imports all static members in class Locale */
import static java.util.Locale.*;
```

## EXERCISE 1-1

### Replacing Implicit `import` Statements with Explicit `import` Statements

Consider the following sample application:

```
import java.io.*;
import java.text.*;
import java.util.*;
import java.util.logging.*;

public class TestClass {
    public static void main(String[] args) throws IOException {
```

```

        /* Ensure directory has been created */
        new File("logs").mkdir();
        /* Get the date to be used in the filename */
        DateFormat df = new SimpleDateFormat("yyyyMMddhhmmss");
        Date now = new Date();
        String date = df.format(now);
        /* Set up the filename in the logs directory */
        String logFileName = "logs\\testlog-" + date + ".txt";
        /* Set up Logger */
        FileHandler myFileHandler = new FileHandler(logFileName);
        myFileHandler.setFormatter(new SimpleFormatter());
        Logger scjaLogger = Logger.getLogger("SCJA Logger");
        scjaLogger.setLevel(Level.ALL);
        scjaLogger.addHandler(myFileHandler);
        /* Log Message */
        scjaLogger.info("\nThis is a logged information message.");
        /* Close the file */
        myFileHandler.close();
    }
}

```

There can be implicit imports that allow all necessary classes of a package to be imported.

```
import java.io.*; // Implicit import example
```

There can be explicit imports that only allow the designated class or interface of a package to be imported.

```
import java.io.File; // Explicit import example
```

This exercise will have you using explicit import statements in lieu of the implicit import statements for all of the necessary classes of the sample application. If you are unfamiliar with compiling and interpreting Java programs, complete this chapter and then come back to this exercise. Otherwise, let's begin.

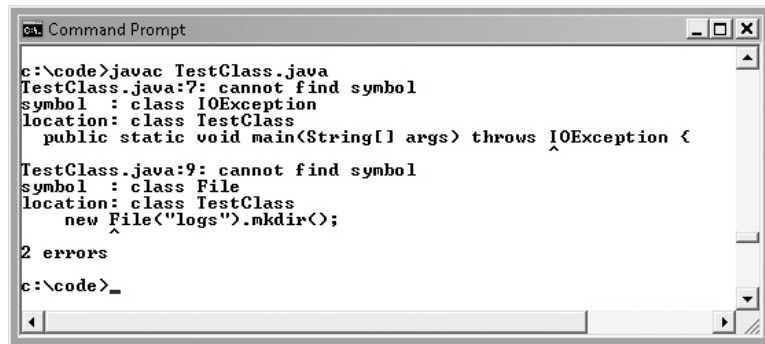
1. Type the sample application into a new file and name it *TestClass.java*. Save the file.
2. Compile and run the application to ensure that you have created the file contents without error; `javac TestClass.java` to compile, `java TestClass` to run. Verify that the log message prints to the screen. Also verify that a file has been created in the *logs* subdirectory with the same message in it.



3. Comment out all of the `import` statements.

```
//import java.io.*;  
//import java.text.*;  
//import java.util.*;  
//import java.util.logging.*;
```

4. Compile the application; `javac TestClass.java`. You will be presented with several compiler errors related to the missing class imports. As an example, the following illustration demonstrates the errors seen when only the `java.io` package has been commented out.



```
cmd: Command Prompt  
c:\code>javac TestClass.java  
TestClass.java:7: cannot find symbol  
symbol : class IOException  
location: class TestClass  
    public static void main(String[] args) throws IOException {  
TestClass.java:9: cannot find symbol  
symbol : class File  
location: class TestClass  
    new File("logs").mkdir();  
2 errors  
c:\code>_
```

5. For each class that cannot be found, use the online Java Specification API to determine which package it belongs to and then update the source file with the necessary explicit `import` statement. Once completed, you will have replaced the four *implicit* `import` statements with nine *explicit* `import` statements.
6. Run the application again to ensure the application works the same with the explicit imports as it did with the implicit import.

## CERTIFICATION OBJECTIVE

# Understanding Package-Derived Classes

*Exam Objective 5.3 Describe the purpose and types of classes for the following Java packages: `java.awt`, `javax.swing`, `java.io`, `java.net`, `java.util`.*

Sun includes over 100 packages in the core Java SE API. Each package has a specific focus. Fortunately, you only need to be familiar with a few of them for the SCJA exam. These include packages for Java utilities, basic input/output, networking, AWT and Swing.

- Java Utilities API
- Java Basic Input/Output API
- Java Networking API
- Java Abstract Window Toolkit API
- Java Swing API

### Java Utilities API

The Java Utilities API is contained in the package `java.util`. This API provides functionality for a variety of utility classes. The API's key classes and interfaces can be divided into several categories. Categories of classes that may be seen on the exam include the Java Collections Framework, date and time facilities, internationalization, and some miscellaneous utility classes.

Of these categories, the Java Collections Framework pulls the most weight since it is frequently used and provides the fundamental data structures necessary to build valuable Java applications. Table 1-4 details the classes and interfaces of the Collections API that you may see referenced on the exam.

To assist collections in sorting where the ordering is not natural, the Collections API provides the `Comparator` interface. Similarly, the `Comparable` interface that resides in the `java.lang` package is used to sort objects by their natural ordering.

**TABLE 1-4**

Various Classes  
of the Java  
Collections  
Framework

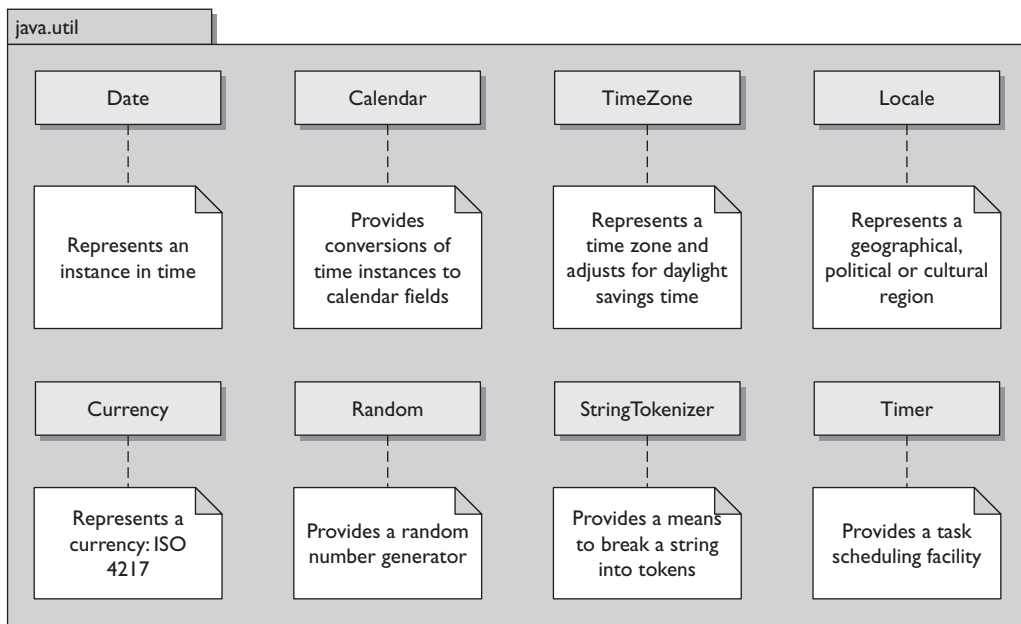
Interface	Implementations	Description
List	ArrayList, LinkedList, Vector	Data structures based on positional access.
Map	HashMap, Hashtable, LinkedHashMap, TreeMap	Data structures that map keys to values.
Set	HashSet, LinkedHashSet, TreeSet	Data structures based on element uniqueness.
Queue	PriorityQueue	Queues typically order elements in a FIFO manner. Priority queues order elements according to a supplied comparator.

Various other classes and interfaces reside in the `java.util` package. Date and time facilities are represented by the `Date`, `Calendar`, and `TimeZone` classes. Geographical regions are represented by the `Locale` class. The `Currency` class represents currencies per the ISO 4217 standard. A random number generator is provided by the `Random` class. And `StringTokenizer` breaks strings into tokens. Several other classes exist within `java.util`, but these (and the collection interfaces and classes) are the ones most likely to be seen on the exam. The initially discussed classes are represented in Figure 1-2.



**Many packages have related classes and interfaces that have unique functionality, so they are included in their own subpackages. For example, regular expressions are stored in a subpackage of the Java utilities (`java.util`) package. The subpackage is named `java.util.regex` and houses the `Matcher` and `Pattern` classes. Where needed, consider creating subpackages for your own projects.**

**FIGURE 1-2** Various utility classes



## Java Basic Input/Output API

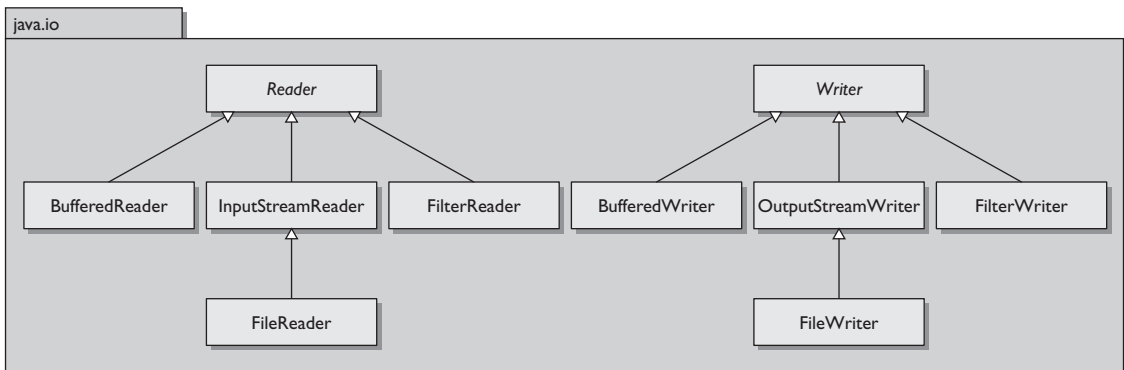
The Java Basic Input/Output API is contained in the package `java.io`. This API provides functionality for general system input and output in relationships to data streams, serialization, and the file system. Data stream classes include byte-stream subclasses of the `InputStream` and `OutputStream` classes. Data stream classes also include character-stream subclasses of the `Reader` and `Writer` classes. Figure 1-3 depicts part of the class hierarchy for the `Reader` and `Writer` abstract classes.

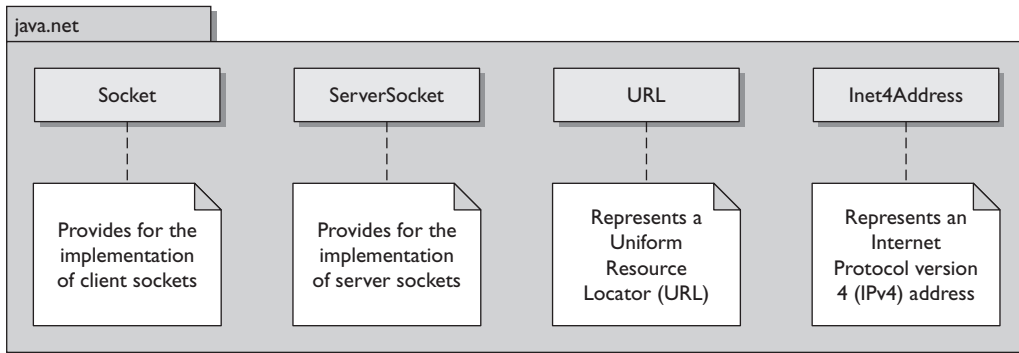
Other important `java.io` classes and interfaces include `File`, `FileDescriptor`, `FilenameFilter`, and `RandomAccessFile`. The `File` class provides a representation of file and directory pathnames. The `FileDescriptor` class provides a means to function as a handle for opening files and sockets. The `FilenameFilter` interface, as its name implies, defines the functionality to filter filenames. The `RandomAccessFile` class allows for the reading and writing of files to specified locations.

## The Java Networking API

The Java Networking API is contained in the package `java.net`. This API provides functionality in support of creating network applications. The API's key classes and interfaces are represented in Figure 1-4. You probably will not see few, if any, of these classes on the exam but the figure will help you conceptualize what's in the `java.net` package. The improved performance New I/O API (`java.nio`) package, which provides for nonblocking networking and the socket factory support package (`javax.net`), is not on the exam.

**FIGURE 1-3** `Reader` and `Writer` class hierarchy



**FIGURE 1-4** Various classes of the Networking API

## Java Abstract Window Toolkit API

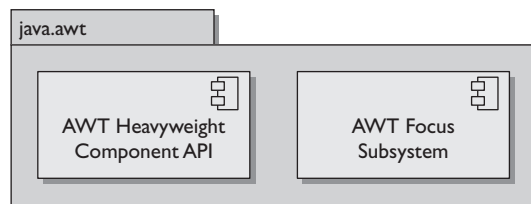
The Java Abstract Window Toolkit API is contained in the package `java.awt`. This API provides functionality for creating heavyweight components in regards to creating user interfaces and painting associated graphics and images. The AWT API was Java's original GUI API and has been superseded by the Swing API. Where Swing is now recommended, certain pieces of the AWT API still remain commonly used, such as the AWT Focus subsystem that was reworked in J2SE 1.4. The AWT Focus subsystem provides for navigation control between components. Figure 1-5 depicts these major AWT elements.

## Java Swing API

The Java Swing API is contained in the package `javax.swing`. This API provides functionality for creating lightweight (pure-Java) containers and components. The Swing API superseded the AWT API. Many of the new classes were simply prefaced with the addition of "J" in contrast to the legacy AWT component equivalent.

**FIGURE 1-5**

AWT major  
elements



## SCENARIO &amp; SOLUTION

You need to create basic Java Swing components such as buttons, panes, and dialog boxes. Provide the code to import the necessary classes of a package.

```
// Java Swing API package
import javax.swing.*;
```

You need to support text-related aspects of your Swing components. Provide the code to import the necessary classes of a package.

```
// Java Swing API text subpackage
import javax.swing.text.*;
```

You need to implement and configure basic pluggable look-and-feel support. Provide the code to import the necessary classes of a package.

```
// Java Swing API plaf subpackage
import javax.swing.plaf.*;
```

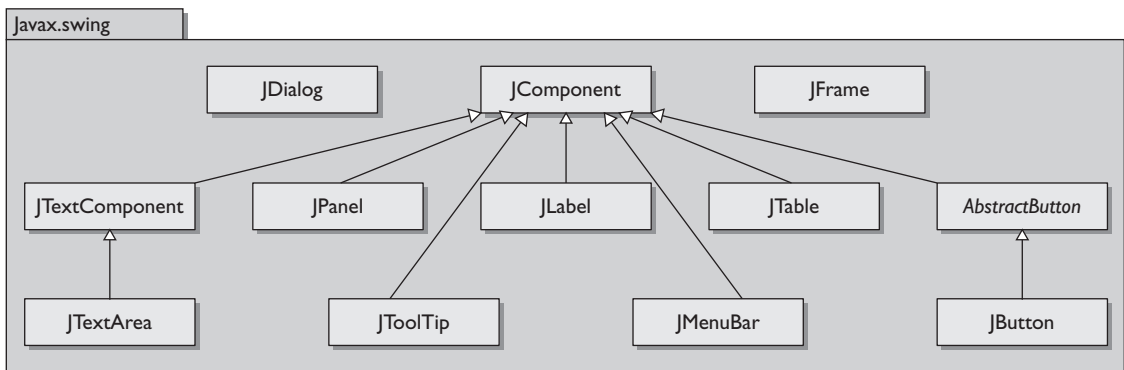
You need to use Swing event listeners and adapters. Provide the code to import the necessary classes of a package.

```
// Java Swing API event subpackage
import javax.swing.event.*;
```

For example, Swing uses the class `JButton` to represent a button container, whereas AWT uses the class `Button`.

Swing also provides look-and-feel support, allowing for universal style changes of the GUI's components. Other features include tooltips, accessibility functionality, an event model, and enhanced components such as tables, trees, text components, sliders, and progress bars. Some of the Swing API's key classes are represented in Figure 1-6. See Chapter 11 for more information on the Swing API as a client-side user-interface solution.

**FIGURE 1-6** Various classes of the Swing API



The Swing API makes excellent use of subpackages, with 16 of them total in Java SE 6. As mentioned earlier, when common classes are separated into their own packages, code usability and maintainability is enhanced.

Swing takes advantage of the model-view-controller architecture (MVC). The model represents the current state of each component. The view is the representation of the components on the screen. The controller is the functionality that ties the UI components to events. While understanding the underlying architecture of Swing is important, it's not necessary for the exam. For comprehensive information on the Swing API, look to the book *Swing: A Beginner's Guide*, by Herbert Schildt (McGraw-Hill Professional, 2007).

## exam

### Watch

***Be familiar with the package prefixes `java` and `javax`. The prefix `java` is commonly used for the core packages. The prefix `javax` is commonly used for packages comprised of Java standard extensions. Take special notice of the prefix usage in the AWT and Swing APIs: `java.awt` and `javax.swing`.***

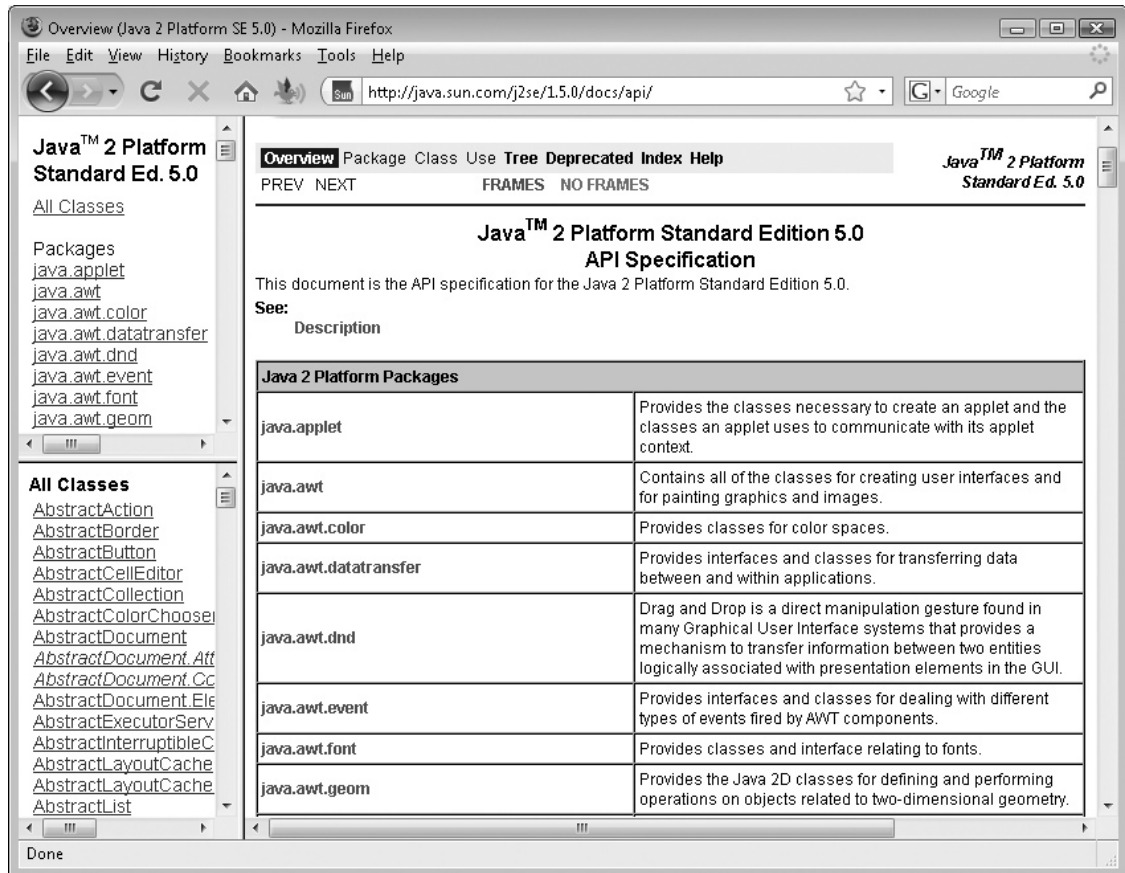
## EXERCISE 1-2

### Understanding Extended Functionality of the Java Utilities API

In Java SE 6, a total of ten packages are in direct relationship to the Java Utilities API, with the base package being named `java.util`. J2SE 5.0 has only nine packages; J2SE 1.4, just six. This exercise will have you exploring the details of the Java Utilities API subpackages that were added in subsequent releases of the Java SE 1.4 platform.

1. Go to the online J2SE 1.4.2 API specification: <http://java.sun.com/j2se/1.4.2/docs/api/>.
2. Use the web browser's scroll bar to scroll down to the Java Utilities API packages.
3. Click the link for each related package. Explore the details of the classes and interfaces within each package.

4. Go to the online J2SE 5.0 API specification: <http://java.sun.com/j2se/1.5.0/docs/api/>. This is the API specification you should be referencing for the exam. It is shown in the following illustration.



5. Use the web browser's scroll bar to scroll down to the Java Utilities API packages.
6. Determine which three new subpackages were added to the Java Utilities API. Click the link for each of these new packages. Explore the details of the classes and interfaces within each package.
7. Go to the online Java SE 6 API specification: <http://java.sun.com/javase/6/docs/api/>.



8. Use the web browser's scroll bar to scroll down to the Java Utilities API packages.
  9. Determine which new subpackage was added to the Java Utilities API. Click the link for the new package. Explore the details of the classes within the package.
- 

## CERTIFICATION OBJECTIVE

# Compiling and Interpreting Java Code

*Exam Objective 5.2 Demonstrate the proper use of the “javac” command (including the command-line options: -d and -classpath) and demonstrate the proper use of the “java” command (including the command-line options: -classpath, -D and -version).*

The Java Development Kit includes several utilities for compiling, debugging, and running Java applications. This section details two utilities from the kit: the Java compiler and the Java interpreter. For more information on the JDK and its other utilities, see Chapter 10.

## Java Compiler

We will need a sample application to use for our Java compiler and interpreter exercises. We shall employ the simple *GreetingsUniverse.java* source file, shown here in the following listing, throughout the section.

```
public class GreetingsUniverse {  
    public static void main(String[] args) {  
        System.out.println("Greetings, Universe!");  
    }  
}
```

Let's take a look at compiling and interpreting simple Java programs along with their most basic command-line options.

## Compiling Your Source Code

The Java compiler is only one of several tools in the JDK. When you have time, inspect the other tools resident in the JDK's bin folder, as shown in Figure 1-7. For the scope of the SCJA exam, you will only need to know the details surrounding the compiler and interpreter.

The Java compiler simply converts Java source files into bytecode. The Java compiler's usage is as follows:

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

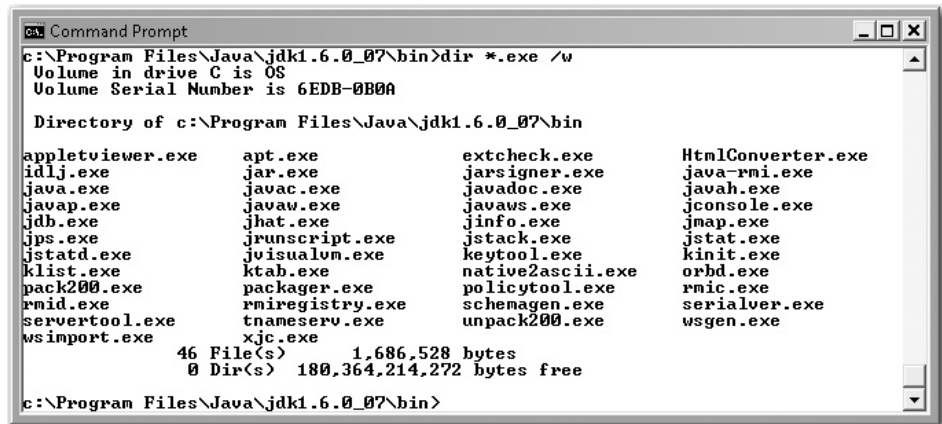
The most straightforward way to compile a Java class is to preface the Java source files with the compiler utility from the command line: `javac.exe FileName.java`. The `.exe` is the standard executable file extension on Windows machines and is optional. The `.exe` extension is not present on executables on Unix-like systems.

```
javac GreetingsUniverse.java
```

This will result in a bytecode file being produced with the same preface, such as `GreetingsUniverse.class`. This bytecode file will be placed into the same folder as the source code, unless the code is packaged and/or it's been told via a command-line option to be placed somewhere else.

**FIGURE 1-7**

Java Development  
Kit utilities



## INSIDE THE EXAM

### Command-Line Tools

Most projects use Integrated Development Environments (IDEs) to compile and execute code. The clear benefit in using IDEs is that building and running code can be as easy as stepping through a couple of menu buttons or even just hitting a hot key. The disadvantage is that even though you may establish your settings through a configuration dialog box and see the commands and subsequent arguments in one of the workspace windows, you are not getting direct experience in repeatedly creating the complete structure of

the commands and associated arguments by hand. The exam is structured to validate that you have experience in scripting compiler and interpreter invocations. Do not take this prerequisite lightly. Take the exam only after you have mastered when and how to use the tools, switches, and associated arguments. At a later time, you can consider taking advantage of the “shortcut” features of popular IDEs such as those provided by NetBeans, Eclipse, IntelliJ IDEA, and JDeveloper.

### Compiling Your Source Code with the `-d` Option

You may wish to explicitly specify where you would like the compiled bytecode class files to go. You can accomplish this using the `-d` option.

```
javac -d classes GreetingsUniverse.java
```

This command-line structure will place the class file into the `classes` directory, and since the source code was packaged (that is, the source file included a package statement), the bytecode will be placed into the relative subdirectories.

```
[present working directory]\classes\com\scjaexam\tutorial\
GreetingsUniverse.class
```

### Compiling Your Code with the `-classpath` Option

If you wish to compile your application with user-defined classes and packages, you may need to tell the JVM where to look by specifying them in the classpath. This classpath inclusion is accomplished by telling the compiler where the desired classes and packages are with the `-cp` or `-classpath` command-line option. In the

following compiler invocation, the compiler includes in its compilation any source files that are located under the `3rdPartyCode\classes` directory, as well as any classes located in the present working directory (the period). The `-d` option (again) will place the compiled bytecode into the `classes` directory.

```
javac -d classes -cp 3rdPartyCode\classes\;. GreetingsUniverse
.java
```

Note that you do not need to include the classpath option if the classpath is defined with the `CLASSPATH` environment variable, or if the desired files are in the present working directory.

On Windows systems, classpath directories are delimited with backward slashes, and paths are delimited with semicolons:

```
-classpath .;\dir_a\classes_a\;\dir_b\classes_b\
```

On POSIX-based systems, classpath directories are delimited with forward slashes and paths are delimited with colons:

```
-classpath .:/dir_a/classes_a/;/dir_b/classes_b/
```

Again, the period represents the present (or current) working directory.

## exam

### Watch

**Know your switches.** *The designers of the exam will try to throw you by presenting answers with mix-matching compiler and interpreter switches. You may even see some make-believe switches that do not exist anywhere. For additional preparation, query the commands' complete set of switches by typing "java -help" or "javac -help". Switches are also known as command-line parameters, command-line switches, options, and flags.*

## Java Interpreter

Interpreting the Java files is the basis for creating the Java application, as shown in Figure 1-8. Let's examine how to invoke the interpreter and its command-line options.

```
java [-options] class [args...]
```

**FIGURE I-8** Bytecode conversion

## Interpreting Your Bytecode

The Java interpreter is invoked with the `java [ .exe ]` command. It is used to interpret bytecode and execute your program.

You can easily invoke the interpreter on a class that's not packaged as follows:

```
java MainClass
```

You can optionally start the program with the `javaw` command on Microsoft Windows to exclude the command window. This is a nice feature with GUI-based applications since the console window is often not necessary.

```
javaw.exe MainClass
```

Similarly, on POSIX-based systems, you can use the ampersand to run the application as a background process.

```
java MainClass &
```

## Interpreting Your Code with the `-classpath` Option

When interpreting your code, you may need to define where certain classes and packages are located. You can find your classes at runtime when you include the `-cp` or `-classpath` option with the interpreter. If the classes you wish to include are packaged, then you can start your application by pointing the full path of the application to the base directory of classes, as in the following interpreter invocation:

```
java -cp classes com.scjaexam.tutorial.MainClass
```

The delimitation syntax is the same for the `-cp` and `-classpath` options, as defined earlier in the “Compiling Your Code with the `-classpath` Option” section.

## Interpreting Your Bytecode with the -D Option

The `-D` command-line option allows for the setting of new property values. The usage is as follows:

```
java -D<name>=<value> class
```

The following single-file application comprised of the `PropertiesManager` class prints out all of the system properties.

```
import java.util.Properties;
public class PropertiesManager {
    public static void main(String[] args) {
        Properties props = System.getProperties();
        /* New property example */
        props.setProperty("new_property2", "new_value2");
        if (args[0].equals("-list_all")) {
            props.list(System.out); // Lists all properties
        } else if (args[0].equals("-list_prop")) {
            /* Lists value */
            System.out.println(props.getProperty(args[1]));
        } else {
            System.out.println("Usage: java PropertiesManager
[-list_all]");
            System.out.println("          java PropertiesManager
[-list_prop [property]]");
        }
    }
}
```

Let's run this application while setting a new system property called "new\_property1" to the value of "new\_value1".

```
java -Dnew_property1=new_value1 PropertiesManager -list_all
```

You'll see in the standard output that the listing of the system properties includes the new property that we set and its value.

```
...
new_property1=new_value1
java.specification.name=Java Platform API Specification
...
```

Optionally, you can set a value by instantiating the `Properties` class, and then setting a property and its value with the `setProperty` method.

To help you conceptualize system properties a little better, Table 1-5 details a subset of the standard system properties.

Retrieving the Version of the Interpreter with the `-version` Option

The `-version` command-line option is used with the Java interpreter to return the version of the JVM and exit. Don't take the simplicity of the command for granted, as the designers of the exam may try to trick you by including additional arguments after the command. Take the time to toy with the command by adding arguments and placing the `-version` option in various places. Do not make any assumptions about how you think the application will respond. Figure 1-9 demonstrates varying results based on where the `-version` option is used.

TABLE 1-5      Subset of System Properties

System Property	Property Description
<code>file.separator</code>	The platform specific file separator ('/' for POSIX, '\' for Windows)
<code>java.class.path</code>	The classpath as defined for the system's environment variable
<code>java.class.version</code>	The Java class version number
<code>java.home</code>	The directory of the Java installation
<code>java.vendor</code>	The vendor supplying the Java platform
<code>java.vendor.url</code>	The vendor's Uniform Resource Locator
<code>java.version</code>	The version of the Java Interpreter/JVM
<code>line.separator</code>	The platform-specific line separator ("\r" on Mac OS 9, "\n" for POSIX, "\r\n" for Microsoft Windows)
<code>os.arch</code>	The architecture of the operating system
<code>os.name</code>	The name of the operating system
<code>os.version</code>	The version of the operating system
<code>path.separator</code>	The platform-specific path separator (":" for POSIX, ";" for Windows)
<code>user.dir</code>	The current working directory of the user
<code>user.home</code>	The home directory of the user
<code>user.language</code>	The language code of the default locale
<code>user.name</code>	The username for the current user
<code>user.timezone</code>	The system's default time zone

**FIGURE 1-9**

The `-version` command-line option

```

c:\code>java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Client VM (build 10.0-b23, mixed mode, sharing)

c:\code>java -version INVALID_ARGUMENT
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Client VM (build 10.0-b23, mixed mode, sharing)

c:\code>java HelloWorld -version
Hello, World!

```



*Check out the other JDK utilities at your disposal. You can find them in the `bin` directory of your JDK. `JConsole` in particular is a valuable GUI-based tool that is used to monitor and manage Java applications. Among the many features, `JConsole` allows for viewing memory and thread usages. `JConsole` was released with `J2SE 5.0`.*

## EXERCISE 1-3

### Compiling and Interpreting Packaged Software

When you compile and run packaged software from an IDE, it can be as easy as clicking a run icon as IDE's support, setting the necessary paths that will be used by the compiler and interpreters. However, when you try to compile and interpret the code yourself from the command line, you will need to know exactly how to path your files. Consider our sample application that is now placed in the `com.scjaexam.tutorial` package.

```

package com.scjaexam.tutorial;

public class GreetingsUniverse {
    public static void main(String[] args) {
        System.out.println("Greetings, Universe!");
    }
}

```



This exercise will have you compiling and running the application with new classes created in a separate package.

1. Compile the program.

```
javac -d . GreetingsUniverse.java
```

2. Run the program to ensure it is error-free.

```
java -cp . com.scjaexam.tutorial.GreetingsUniverse.
```

3. Create three classes named `Earth`, `Mars`, and `Venus` and place them in the `com.scja.exam.tutorial.planets` package. Create constructors that will print the names of the planets to standard out. Note that the details for the `Earth` class are given here as an example of what you will need to do.

```
package com.scja.exam.tutorial.planets;
public class Earth {
    public Earth {
        System.out.println("Hello from Earth!");
    }
}
```

4. Instantiate each class from the main program, by adding the necessary code to the `GreetingsUniverse` class.

```
Earth e = new Earth();
```

5. Ensure that all of the source code is in the paths `src/com/scjaexam/tutorial/` and `src/com/scjaexam/tutorial/planets/`, respectively.
6. Determine the command-line arguments needed to compile the complete program. Compile the program, and debug where necessary.
7. Determine the command-line arguments needed to interpret the program. Run the program.

The standard output will read:

```
$ Greetings, Universe!
Hello from Earth!
Hello from Mars!
Hello from Venus!
```

---

## **CERTIFICATION SUMMARY**

This chapter discussed packaging, compiling, and interpreting Java code. The chapter started with a discussion on the importance of organizing your classes into packages as well as using the `package` and `import` statements to define and include different pieces of source code. Through the middle of the chapter, we discussed the key features of the most commonly used Java packages: `java`, `java.awt`, `javax.swing`, `java.net`, `java.io`, and `java.util`. We concluded the chapter by providing detailed information on how to compile and interpret Java source and class files and how to work with their command-line options. At this point, you should be able to independently (outside of an IDE) package, build, and run basic Java programs.



## TWO-MINUTE DRILL

### Understanding Packages

- ☐ Packages are containers for classes.
- ☐ A package statement defines the directory path where files are stored.
- ☐ A package statement uses periods for delimitation.
- ☐ Package names should be lowercase and separated with underscores between words.
- ☐ Package names beginning with `java.*` and `javax.*` are reserved for use by JavaSoft.
- ☐ There can be zero or one package statement per source file.
- ☐ An `import` statement is used to include source code from external classes.
- ☐ An `import` statement occurs after the optional package statement and before the class definition.
- ☐ An `import` statement can define a specific class name to import.
- ☐ An `import` statement can use an asterisk to include all classes within a given package.

### Understanding Package-Derived Classes

- ☐ The Java Abstract Window Toolkit API is included in the `java.awt` package and subpackages.
- ☐ The `java.awt` package includes GUI creation and painting graphics and images functionality.
- ☐ The Java Swing API is included in the `javax.swing` package and subpackages.
- ☐ The `javax.swing` package includes classes and interfaces that support lightweight GUI component functionality.
- ☐ The Java Basic Input/Output-related classes are contained in the `java.io` package.
- ☐ The `java.io` package includes classes and interfaces that support input/output functionality of the file system, data streams, and serialization.

- ❑ Java networking classes are included in the `java.net` package.
- ❑ The `java.net` package includes classes and interfaces that support basic networking functionality that is also extended by the `javax.net` package.
- ❑ Fundamental Java utilities are included in the `java.util` package.
- ❑ The `java.util` package and subpackages includes classes and interfaces that support the Java Collections Framework, legacy collection classes, event model, date and time facilities, and internationalization functionality.

### Compiling and Interpreting Java Code

- ❑ The Java compiler is invoked with the `javac[.exe]` command.
- ❑ The `.exe` extension is optional on Microsoft Windows machines and is not present on UNIX-like systems.
- ❑ The compiler's `-d` command-line option defines where compiled class files should be placed.
- ❑ The compiler's `-d` command-line option will include the package location if the class has been declared with a `package` statement.
- ❑ The compiler's `-classpath` command-line option defines directory paths in search of classes.
- ❑ The Java interpreter is invoked with the `java[.exe]` command.
- ❑ The interpreter's `-classpath` switch defines directory paths to use at runtime.
- ❑ The interpreter's `-D` command-line option allows for the setting of system property values.
- ❑ The interpreter's syntax for the `-D` command-line option is `-Dproperty=value`.
- ❑ The interpreter's `-version` command-line option is used to return the version of the JVM and exit.

# SELF TEST

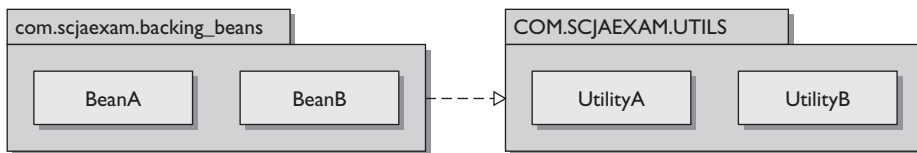
## Understanding Packages

1. Which two `import` statements will allow for the import of the `HashMap` class?
  - A. `import java.util.HashMap;`
  - B. `import java.util.*;`
  - C. `import java.util.HashMap.*;`
  - D. `import java.util.hashMap;`
2. Which statement would designate that your file belongs in the package `com.scjaexam.utilities`?
  - A. `pack com.scjaexam.utilities;`
  - B. `package com.scjaexam.utilities.*`
  - C. `package com.scjaexam.utilities.*;`
  - D. `package com.scjaexam.utilities;`
3. Which of the following is the only Java package that is imported by default?
  - A. `java.awt`
  - B. `java.lang`
  - C. `java.util`
  - D. `java.io`
4. What Java-related features are new to J2SE 5.0?
  - A. Static imports
  - B. `package` and `import` statements
  - C. Autoboxing and unboxing
  - D. The enhanced `for`-loop

## Understanding Package-Derived Classes

5. The `JCheckBox` and `JComboBox` classes belong to which package?
  - A. `java.awt`
  - B. `javax.awt`
  - C. `java.swing`
  - D. `javax.swing`

6. Which package contains the Java Collections Framework?
  - A. `java.io`
  - B. `java.net`
  - C. `java.util`
  - D. `java.utils`
7. The Java Basic I/O API contains what types of classes and interfaces?
  - A. Internationalization
  - B. RMI, JDBC, and JNDI
  - C. Data streams, serialization, and file system
  - D. Collection API and data streams
8. Which API provides a lightweight solution for GUI components?
  - A. AWT
  - B. Abstract Window Toolkit
  - C. Swing
  - D. AWT and Swing
9. Consider the following illustration. What problem exists with the packaging? You may wish to reference Chapter 9 for assistance with UML.



- A. You can only have one class per package.
- B. Packages cannot have associations between them.
- C. Package `com.scjaexam.backing_beans` fails to meet the appropriate packaging naming conventions.
- D. Package `COM.SCJAEXAM.UTILS` fails to meet the appropriate packaging naming conventions.

## Compiling and Interpreting Java Code

- 10.** Which usage represents a valid way of compiling a Java class?
- A. `java MainClass.class`
  - B. `javac MainClass`
  - C. `javac MainClass.source`
  - D. `javac MainClass.java`
- 11.** Which two command-line invocations of the Java interpreter return the version of the interpreter?
- A. `java -version`
  - B. `java --version`
  - C. `java -version ProgramName`
  - D. `java ProgramName -version`
- 12.** Which two command-line usages appropriately identify the classpath?
- A. `javac -cp /project/classes/ MainClass.java`
  - B. `javac -sp /project/classes/ MainClass.java`
  - C. `javac -classpath /project/classes/ MainClass.java`
  - D. `javac -classpath /project/classes/ MainClass.java`
- 13.** Which command-line usages appropriately set a system property value?
- A. `java -Dcom.scjaexam.propertyValue=003 MainClass`
  - B. `java -d com.scjaexam.propertyValue=003 MainClass`
  - C. `java -prop com.scjaexam.propertyValue=003 MainClass`
  - D. `java -D:com.scjaexam.propertyValue=003 MainClass`

## SELF TEST ANSWERS

### Understanding Packages

1. Which two import statements will allow for the import of the `HashMap` class?

- A. `import java.util.HashMap;`
- B. `import java.util.*;`
- C. `import java.util.HashMap.*;`
- D. `import java.util.hashMap;`

Answer:

- ☒ **A** and **B**. The `HashMap` class can be imported directly via `import java.util.HashMap` or with a wild card via `import java.util.*;`
- ☒ **C** and **D** are incorrect. **C** is incorrect because the answer is a static import statement that imports static members of the `HashMap` class, and not the class itself. **D** is incorrect because class names are case-sensitive, so the class name `hashMap` does not equate to `HashMap`.

2. Which statement would designate that your file belongs in the package `com.scjaexam.utilities`?

- A. `pack com.scjaexam.utilities;`
- B. `package com.scjaexam.utilities.*`
- C. `package com.scjaexam.utilities.*;`
- D. `package com.scjaexam.utilities;`

Answer:

- ☒ **D**. The keyword `package` is appropriately used, followed by the package name delimited with periods and followed by a semicolon.
- ☒ **A**, **B**, and **C** are incorrect answers. **A** is incorrect because the word `pack` is not a valid keyword. **B** is incorrect because a package statement must end with a semicolon. **C** is incorrect because you cannot use asterisks in package statements.



3. Which of the following is the only Java package that is imported by default?

- A. `java.awt`
- B. `java.lang`
- C. `java.util`
- D. `java.io`

Answer:

- ☒ B. The `java.lang` package is the only package that has all of its classes imported by default.
- ☒ A, C, and D are incorrect. The classes of packages `java.awt`, `java.util`, and `java.io` are not imported by default.

4. What Java-related features are new to J2SE 5.0?

- A. Static imports
- B. `Package` and `import` statements
- C. Autoboxing and unboxing
- D. The enhanced for-loop

Answer:

- ☒ A, C, and D. Static imports, autoboxing/unboxing, and the enhanced for-loop are all new features of J2SE 5.0.
- ☒ B is incorrect because basic package and import statements are not new to J2SE 5.0.

## Understanding Package-Derived Classes

5. The `JCheckBox` and `JComboBox` classes belong to which package?

- A. `java.awt`
- B. `javax.awt`
- C. `java.swing`
- D. `javax.swing`

Answer:

- ☒ **D.** Components belonging to the Swing API are generally prefaced with a capital J. Therefore, `JCheckBox` and `JComboBox` would be part of the Java Swing API and not the Java AWT API. The Java Swing API base package is `javax.swing`.
- ☒ **A, B, and C** are incorrect. **A** is incorrect because the package `java.awt` does not include the `JCheckBox` and `JComboBox` classes since they belong to the Java Swing API. Note that the package `java.awt` includes the `CheckBox` class, as opposed to the `JCheckBox` class. **B** and **C** are incorrect because the package names `javax.awt` and `java.swing` do not exist.

**6.** Which package contains the Java Collections Framework?

- A. `java.io`
- B. `java.net`
- C. `java.util`
- D. `java.utils`

Answer:

- ☒ **C.** The Java Collections Framework is part of the Java Utilities API in the `java.util` package.
- ☒ **A, B, and D** are incorrect. **A** is incorrect because the Java Basic I/O API's base package is named `java.io` and does not contain the Java Collections Framework. **B** is incorrect because the Java Networking API's base package is named `java.net` and also does not contain the Collections Framework. **D** is incorrect because there is no package named `java.utils`.

**7.** The Java Basic I/O API contains what types of classes and interfaces?

- A. Internationalization
- B. RMI, JDBC, and JNDI
- C. Data streams, serialization, and file system
- D. Collection API and data streams

Answer:

- ☒ C. The Java Basic I/O API contains classes and interfaces for data streams, serialization, and the file system.
- ☒ A, B, and D are incorrect because internationalization (i18n), RMI, JDBC, JNDI, and the Collections Framework are not included in the Basic I/O API.

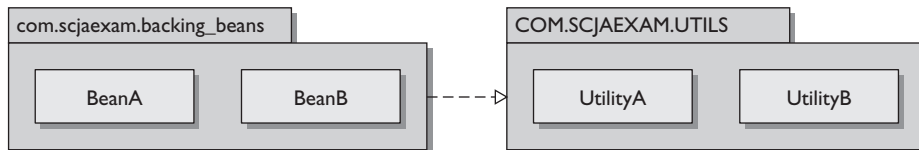
8. Which API provides a lightweight solution for GUI components?

- A. AWT
- B. Abstract Window Toolkit
- C. Swing
- D. AWT and Swing

Answer:

- ☒ C. The Swing API provides a lightweight solution for GUI components, meaning that the Swing API's classes are built from pure Java code.
- ☒ A, B, and D are incorrect. AWT and the Abstract Window Toolkit are one and the same and provide a heavyweight solution for GUI components.

9. Consider the following illustration. What problem exists with the packaging? You may wish to reference Chapter 9 for assistance with UML.



- A. You can only have one class per package.
- B. Packages cannot have associations between them.
- C. Package `com.scjaexam.backing_beans` fails to meet the appropriate packaging naming conventions.
- D. Package `COM.SCJAEXAM.UTILS` fails to meet the appropriate packaging naming conventions.

Answer:

- ☒ **D.** `COM.SCJAEEXAM.UTILS` fails to meet the appropriate packaging naming conventions. Package names should be lowercase. Note that package names should also have an underscore between words; however, the words in “scjaexam” are joined in the URL, therefore excluding the underscore here is acceptable. The package name should read `com.scjaexam.utils`.
- ☒ **A, B, and C** are incorrect. **A** is incorrect because being restricted to having one class in a package is ludicrous. There is no limit. **B** is incorrect because packages can and frequently do have associations with other packages. **C** is incorrect because `com.scjaexam.backing_beans` meets appropriate packaging naming conventions.

## Compiling and Interpreting Java Code

**10.** Which usage represents a valid way of compiling a Java class?

- A. `java MainClass.class`
- B. `javac MainClass`
- C. `javac MainClass.source`
- D. `javac MainClass.java`

Answer:

- ☒ **D.** The compiler is invoked by the `javac` command. When compiling a java class, you must include the filename, which houses the main classes including the `.java` extension.
- ☒ **A, B, and C** are incorrect. **A** is incorrect because `MainClass.class` is bytecode that is already compiled. **B** is incorrect because `MainClass` is missing the `.java` extension. **C** is incorrect because `MainClass.source` is not a valid name for any type of Java file.

**11.** Which two command-line invocations of the Java interpreter return the version of the interpreter?

- A. `java -version`
- B. `java --version`
- C. `java -version ProgramName`
- D. `java ProgramName -version`

Answer:

- ☒ **A** and **C**. The `-version` flag should be used as the first argument. The application will return the appropriate strings to standard output with the version information and then immediately exit. The second argument is ignored.
- ☒ **B** and **D** are incorrect. **B** is incorrect because the version flag does not allow double dashes. You may see double dashes for flags in utilities, especially those following the GNU license. However, the double dashes do not apply to the version flag of the Java interpreter. **D** is incorrect because the version flag must be used as the first argument or its functionality will be ignored.

**12.** Which two command-line usages appropriately identify the class path?

- A.** `javac -cp /project/classes/ MainClass.java`
- B.** `javac -sp /project/classes/ MainClass.java`
- C.** `javac -classpath /project/classes/ MainClass.java`
- D.** `javac -classpaths /project/classes/ MainClass.java`

Answer:

- ☒ **A** and **C**. The option flag that is used to specify the classpath is `-cp` or `-classpath`.
- ☒ **B** and **D** are incorrect because the option flags `-sp` and `-classpaths` are invalid.

**13.** Which command-line usages appropriately set a system property value?

- A.** `java -Dcom.scjaexam.propertyValue=003 MainClass`
- B.** `java -d com.scjaexam.propertyValue=003 MainClass`
- C.** `java -prop com.scjaexam.propertyValue=003 MainClass`
- D.** `java -D:com.scjaexam.propertyValue=003 MainClass`

Answer:

- ☒ **A**. The property setting is used with the interpreter, not the compiler. The property name must be sandwiched between the `-D` flag and the equal sign. The desired value should immediately follow the equal sign.
- ☒ **B**, **C**, and **D** are incorrect because `-d`, `-prop`, and `"-D:"` are invalid ways to designate a system property.