

1

Packaging, Compiling, and Interpreting Java Code

CERTIFICATION OBJECTIVES

- The Java Platform
- Understand Packages
- Understand Package-Derived Classes
- Understand Class Structure
- Compile and Interpret Java Code
- ✓ Two-Minute Drill
- Q&A** Self Test

Since 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 Oracle Certified Associate, Java SE 8 Programmer (OCA), 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 (SE) platform with regard 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, structuring, 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

The Java Platform

Exam Objective Compare and contrast the features and components of Java, such as platform independence, object orientation, encapsulation, and so on

The Java language was first released in 1995 as a beta. At that time the Java team had a radical vision. They envisioned a language that was independent of the platform it was running on. They also wanted to create a language that was object oriented at its core and that used all the principles that this implied. Encapsulation, polymorphism, inheritance, and abstraction are all basic concepts upon which Java is built. This section will review the core philosophy that makes up the Java language.

Platform Independence

When the Java language is compiled, it is targeted for execution on the Java virtual machine, or JVM, instead of a specific hardware architecture. The compiled Java code is called *bytecode*. This is why it is possible to compile the Java language on a

Windows PC and execute the output on a Linux server. The only requirement for the code to work on any computer is the presence of a compatible JVM.

The Java language extends past the PC and server, however. Many mobile phones have embraced the power of Java as their recommended language for apps. This allows the hardware manufacturer to change the hardware between models without breaking the compatibility of the software. Java is even present in embedded systems and appliances. On devices such as Blu-ray players and car infotainment systems, Java software is often present.

It is important that you understand that platform independence does not mean your server code will run on your Blu-ray player. Java has a few different JVM specs for devices with different capabilities. For example, embedded systems use a JVM with only a subset of features, and mobile phones typically use a JVM with mobile optimized user interface libraries. All of these JVMs share a common Java core, but platform independence is limited to compatible versions.

Java's Object-Oriented Philosophy

Java was conceived as an object-oriented language, in contrast to the C language, which is procedural. An object-oriented language organizes related data and code together—a process called *encapsulation*. A properly encapsulated object uses data protection and exposes only some of its data and methods. The data and methods that are designed for internal use in the object are not exposed to other objects.

Object-oriented design also encourages *abstraction*, the ability to generalize algorithms. Abstraction facilitates code reuse and flexibility. These concepts are at the heart of the Java language. Inheritance and polymorphism are key concepts in creating reusable code. Both are covered in much more depth in Chapters 7 and 8 of this book.

Robust and Secure

Security and robustness were major design goals when Java was created. C and C++ suffered from the misuse of pointers, memory management, and buffer overruns. Java was architected to overcome these issues and many more.

Java was designed not to have explicit pointers. In the C language family, pointers store a memory address to an object. This memory address can be directly altered. Java variables store references to objects but do not allow access to, or modification of, the memory address stored in the reference. This simplified development and removed a level of complexity that was often the source of application instability.

Memory management was addressed in Java with the JVM's built-in garbage collector. When Java was introduced, many languages relied on explicit memory

management. This meant that the developer was responsible for both allocating and deallocating the memory that was used for objects. This process could become tedious. If it was done incorrectly, the application could leak memory and/or crash. With Java, the JVM periodically runs the garbage collector, which looks for any objects that have gone out of scope or that are no longer referenced, and it automatically deallocates their memory. This frees the developer from this manual, error-prone task and increases robustness by ensuring that memory is properly managed.

Buffer overruns are a common exploit vector found in software that does not check for them. In a C program, when an array is created, the index used is never automatically checked to ensure it is in bounds. In fact, an out-of-bounds index may not even crash the program. The software will read or write to the memory address whether it is in bounds or out, and this can create unpredictable behavior. This can be used maliciously to alter the program in ways the developer never intended. Java automatically checks the bounds of arrays. If an index is out-of-bounds, an exception is thrown. This level of checking helps create both more robust and secure software.

CERTIFICATION OBJECTIVE

Understand Packages

Exam Objective Import other Java packages to make them accessible in your code

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 in 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 considered containers for classes, but they actually 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 in the same

TABLE 1-1

Package
Attribute
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 a reverse domain name for the package structure. Use lowercase characters delimited with underscores to separate words in package names. |

namespace. The package name plus the class names creates the *fully qualified class name*. 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, consider the key areas shown in Table 1-1.

Let's take a look at a real-world example. As program manager, suppose 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).

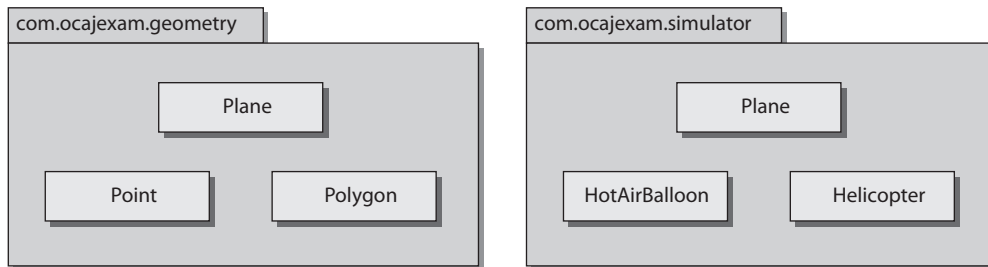
Come delivery time, they both give you a class named `Plane.java`—that is, one for the geometry plane class and one for the airplane class. Now you have a problem, because both of these source files (class files, too) cannot coexist in the same directory because they have the same name. The solution 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 this: Always package your code, unless your coding project is trivial in nature.

package and import Statements

You should now have a general idea of when and why to package your source files. Next, you need to know exactly how to do this. To place a source file into a package, you use the package statement at the beginning of that file. You may use zero or

6 Chapter 1: Packaging, Compiling, and Interpreting Java Code

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



one package statements per source file. To import classes from other packages into your source file, you may use the `import` statement or you may precede each class name with its package name. 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 return to this listing as we discuss the `package` and `import` statements in detail throughout the chapter.

```
package com.ocaj.exam.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 - run outside your
IDE */
        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:

- They are optional.
- They 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 ocajexam.com may use the following package name for a utilities package: com.ocajexam.utilities.
- Package names equate to directory structures. The package name com.ocajexam.utils would equate to the directory com/ocajexam/utils. If a class includes a package statement that does not map to the relative directory structure, the class will not be usable.
- The package names beginning with java.* and javax.* are reserved.
- 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 Oracle’s Online Javadoc documentation at <http://docs.oracle.com/javase/8/docs/api/>.

On the exam, you will see 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 enables you to include source code from other classes into a source file at compile time. The import statement includes the import keyword followed by the package path delimited with periods and ending with a class

TABLE 1-2

Valid package Statements

| Package Statement | Related Directory Structure |
|---------------------------------|------------------------------------------|
| package java.net; | [directory_path]\java\net\ |
| package com.ocajexam.utilities; | [directory_path]\com\ocajexam\utilities\ |
| package package_name; | [directory_path]\package_name\ |

TABLE 1-3 Valid import Statements

| Import Statement | Definition |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>import java.net.*;</code> | Imports all 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>ColorSpace</code> class of the package <code>java.awt</code> (J2SE 5.0 onward only) |

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 relate to one package only.



For maintenance purposes, it is better that you import your classes explicitly. This will allow the programmer to determine quickly which external classes are used throughout the class. For 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 imports only 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.

| SCENARIO & SOLUTION | |
|--------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| To paint basic graphics and images, which package should you use? | Use the Java AWT API package. <code>import java.awt.*;</code> |
| To use data streams, which package should you use? | Use the Java Basic I/O package. <code>import java.io.*;</code> |
| To develop a networking application, which package should you use? | Use the Java Networking API package. <code>import java.net.*;</code> |
| To work with the collections framework, event model, and date/time facilities, which package should you use? | Use the Java Utilities API package. <code>import java.util.*;</code> |
| To utilize the core Java classes and interfaces, which package should you use? | Use the core Java Language package, which is imported by default. <code>import java.lang.*;</code> |

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 is no direct mapping in functionality.

The static import Statement

Static import statements were introduced in Java SE 5.0. Simply put, static import statements allow you to import static members. The following example statements demonstrate this:

```
/* Import static member ITALY */
import static java.util.Locale.ITALY;
...
System.out.println("Locale: " + ITALY); // Prints "Local: it_IT"
...

/* Imports all static members in class Locale */
import static java.util.Locale.*;
...
System.out.println("Locale: " + ITALY); // Prints "Local: it_IT"
System.out.println("Locale: " + GERMANY); // Prints "Local: de_DE"
System.out.println("Locale: " + JAPANESE); // Print "Local: ja"
...
```

Without the static import statements shown in the example, the direct references to `ITALY`, `GERMANY`, and `JAPANESE` would be invalid and would cause compilation issues.

```
// import static java.util.Locale.ITALY;
...
System.out.println("Locale: " + ITALY); // Won't compile
```

EXERCISE 1-1

Replacing Implicit import Statements with Explicit import Statements

Consider the following sample application:

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

10 Chapter 1: Packaging, Compiling, and Interpreting Java Code

```
public class TestClass {
    public static void main(String[] args) throws IOException {
        /* Ensure directory has been created */
        Files.createDirectories(Paths.get("logs"));
        /* Get the date to be used in the filename */
        DateTimeFormatter df
            = DateTimeFormatter.ofPattern("yyyyMMdd_hhmm");
        LocalDateTime now = LocalDateTime.now();
        String date = now.format(df);
        /* 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 ocajLogger = Logger.getLogger("OCAJ Logger");
        ocajLogger.setLevel(Level.ALL);
        ocajLogger.addHandler(myFileHandler);
        /* Log Message */
        ocajLogger.info("\nThis is a logged information message. ");
        /* Close the file */
        myFileHandler.close();
    }
}
```

There can be implicit `import` statements that allow all necessary classes of a package to be imported:

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

There can be explicit `import` statements that allow only the designated class or interface of a package to be imported:

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

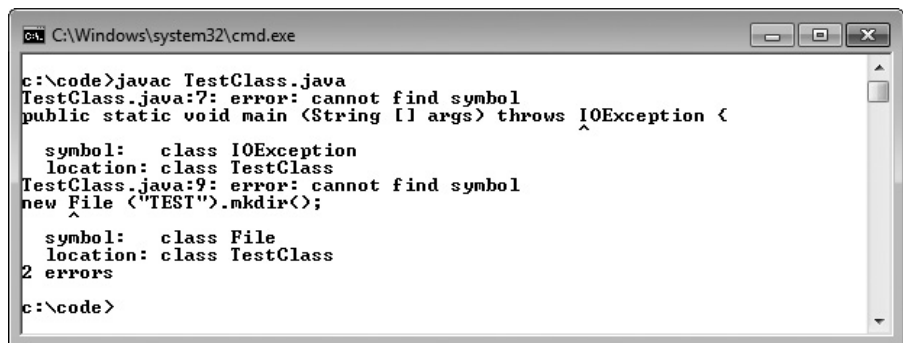
This exercise will have you using explicit `import` statements in lieu of the implicit `import` statements for all of the necessary classes of the sample application. If you are unfamiliar with compiling and interpreting Java programs, finish reading this chapter and then come back to this exercise. 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.time.*;
// import java.time.format.*;
//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 that are displayed when only the `java.io` package has been commented out:



```
C:\Windows\system32\cmd.exe

c:\code>javac TestClass.java
TestClass.java:7: error: cannot find symbol
public static void main (String [] args) throws IOException {
               ^
  symbol:   class IOException
  location: class TestClass
TestClass.java:9: error: cannot find symbol
new File ("TEST").mkdir();
   ^
  symbol:   class File
  location: class TestClass
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 that the application works with the explicit import statements the same way it did with the implicit import statements.

Understand Package-Derived Classes

Oracle includes more than 200 packages in the Java SE 8 API. Each package has a specific focus. Fortunately, you need to be familiar with only a few of them for the OCA exam. These may include packages for Java utilities, basic input/output, networking, Abstract Window Toolkit (AWT), Swing, and data/time. The java data/time classes will be covered in more detail in Chapter 10.

The following sections address these APIs:

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

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 because 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

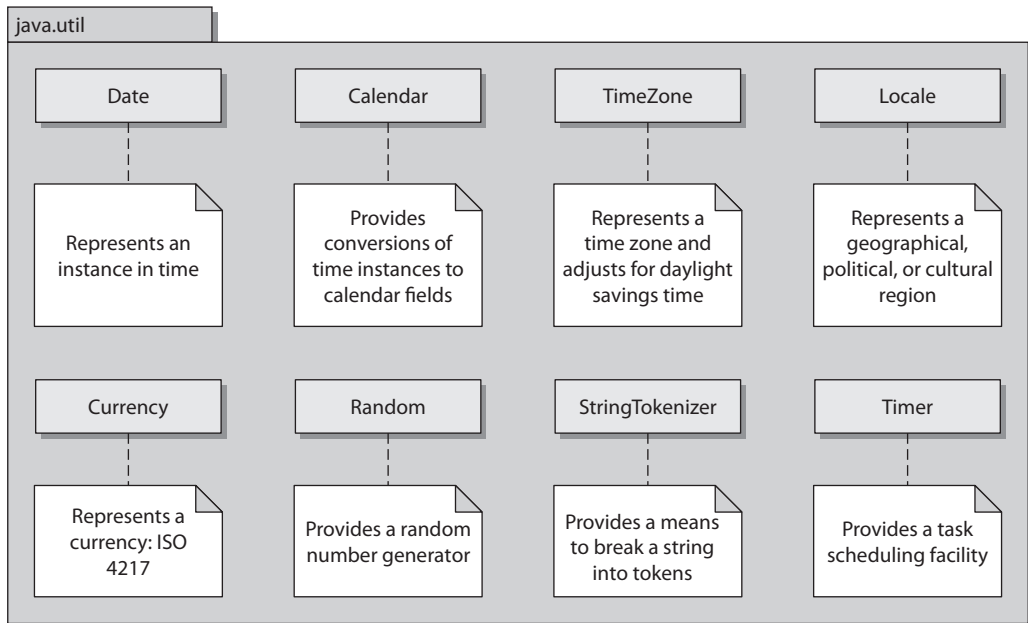
| Interface | Implementations | Description |
|-----------|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| List | ArrayList, LinkedList, Vector | Data structures 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 first in, first out (FIFO) manner. Priority queues order elements according to a supplied comparator. |

Various other classes and interfaces reside in the `java.util` package. Legacy 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`, and these (and the collection interfaces and classes) are classes that you may find yourself commonly using on the job. These utilities classes are represented in Figure 1-2.



Many packages have related classes and interfaces with 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 relation 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.

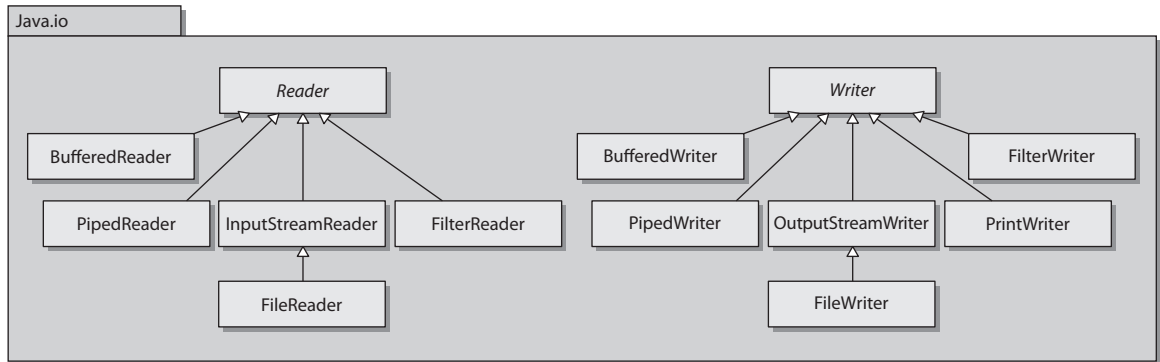
Other important `java.io` classes and interfaces include `File`, `FileDescriptor`, `FilenameFilter`, and `RandomAccessFile`. The `File` class provides a representation of file and directory pathnames. The `FileDescriptor` class provides a means to function as a handle for opening files and sockets. The `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.

In JDK 7, the NIO.2 API was introduced in the package `java.nio`. This included the useful `Paths` interface, the `Path` class, and the `Files` class. The `Files` class has `lines`, `list`, `walk`, and `find` methods that work hand-in-hand with the Stream API. All of this is beyond the scope of the exam but is useful to know. The following code snippet provides a quick look at what you can do with the API and other new features of Java. You'll see the lambda expression-related code in Chapter 11 (for example, `p -> { statements; }`).

```
// Print out .txt file names in a given folder
try {
    Files.walk(Paths.get("C:\\opt\\dnaProg\\users\\docs")).forEach(p -> {
        if (p.getFileName().toString().endsWith(".txt")) {
            System.out.println("Text doc:" + p.getFileName());
        }
    });
} catch (IOException e) {
    e.printStackTrace();
}
```

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 will probably see few, if any, of these classes on the exam, but the figure will help you conceptualize

FIGURE 1-3 Reader and Writer class hierarchy

what's in the `java.net` package. The improved performance I/O API (`java.nio`) package, which provides for nonblocking networking and the socket factory support package (`javax.net`), is not included on the exam.

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 with regard 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 has been recommended over AWT, certain pieces of the AWT API still

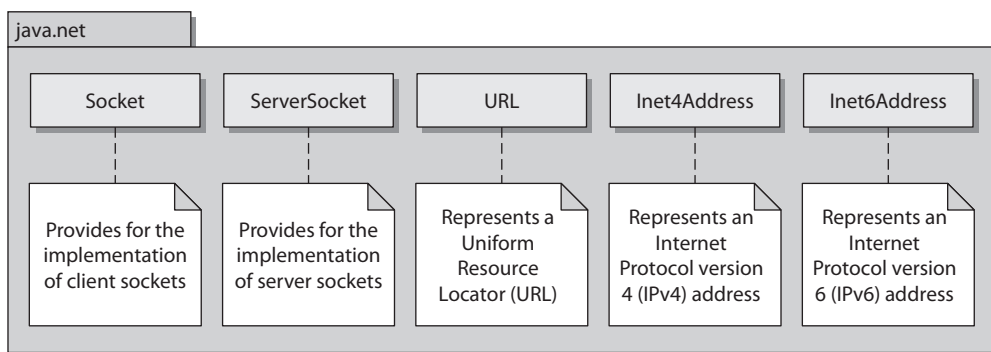
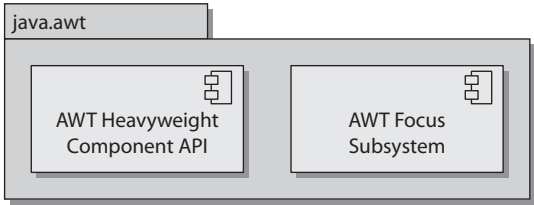
FIGURE 1-4 Various classes of the Networking API

FIGURE 1-5

AWT major elements



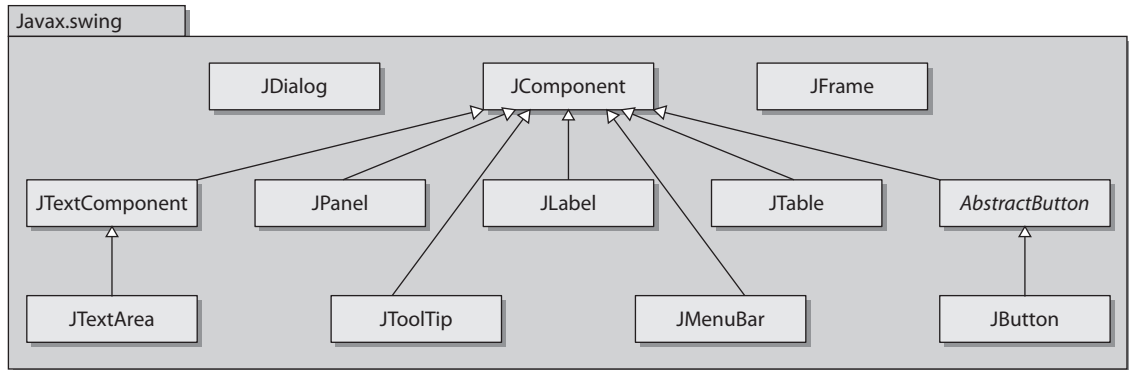
remain commonly used, such as the AWT Focus subsystem that was reworked in J2SE 1.4. The AWT Focus subsystem provides for navigation control between components. Figure 1-5 depicts these major AWT elements.

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, providing a more sophisticated set of GUI components, supersedes the AWT API. Many of the Swing classes are simply prefaced with the addition of “J” in contrast to the legacy AWT component equivalent. 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,

| SCENARIO & 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.*; |

FIGURE 1-6 Various classes of the Swing API

sliders, and progress bars. Some of the Swing API's key classes are represented in Figure 1-6.

The Swing API makes excellent use of subpackages, with 18 of them in Java SE 8. As mentioned earlier, when common classes are separated into their own packages, code usability and maintainability are enhanced.

Swing takes advantage of the model-view-controller (MVC) architecture. 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. Although 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).



It's good to 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 that comprise Java standard extensions. Take special notice of the prefix usage in the AWT and Swing APIs: `java.awt` and `javax.swing`. Also note that JavaFX will be replacing Swing as the GUI toolkit for Java SE. Its prefix is `java.fx`.

JavaFX API

JavaFX is Java's latest technology for creating rich user interfaces. It is designed to provide lightweight, hardware-accelerated interfaces. JavaFX provides a similar set of features to the Swing library. JavaFX is intended to replace Swing in the same manner that Swing replaced AWT. The JavaFX libraries are part of the `java.fx` package.

JavaFX best practices suggest that the MVC architecture be used when designing applications. FXML, an XML-based markup language, has been created for defining user interfaces. Many of the more than 60 UI controls can be styled by using Cascading Style Sheets (CSS). These features together represent a powerful new way to create user interfaces. JavaFX makes going from whiteboard design to implemented software faster than ever before. A great reference for JavaFX is *Introducing JavaFX 8 Programming*, by Herbert Schildt (Oracle Press).



JavaFX is the latest technology for creating user interfaces. Oracle is actively promoting this technology as the go-to tool kit. However, the Swing libraries are not going away anytime soon. In Java 8, both JavaFX and Swing are fully supported and can be used interchangeably. The `SwingNode` class allows Swing elements to be embedded in JavaFX. The `JFXPanel` will allow the reverse so that JavaFX elements can be used in a Swing applications.

CERTIFICATION OBJECTIVE

Understand Class Structure

Exam Objective Define the structure of a Java class

You must understand the structure of a Java class to do well on the exam and to have a promising career with Java. It would help to have a fundamental knowledge of Java naming conventions as well as knowledge of the typical separators that are seen in Java source code (such as comment separators and brackets for enclosing entities). These topics are covered in the following sections:

- Naming Conventions
- Separators and Other Java Source Symbols
- Java Class Structure

Naming Conventions

Naming conventions are rules for the usage and application of characters in creation of identifiers, methods, class names, and so forth, throughout your code base. If some of your team members are not applying naming conventions to their code, you should encourage them to do so, for the good of the effort and for maintainability aspects for after the code is deployed.



The popular article, “How to Write Unmaintainable Code,” by Roedy Green, is worth reading (<http://thc.org/root/phun/unmaintain.html>). It brings to light, in a comical way, the challenges that can occur with maintaining code when there is a blatant or intentional disregard to software development best practices. On the flip-side, *The Passionate Programmer: Creating a Remarkable Career in Software Development*, by Chad Fowler (Pragmatic Bookshelf, 2009), encourages the software developer to be the best that he or she can be.

You may encounter people who will come up with their own naming conventions. Although this is better than not applying any convention, an outsider trying to maintain that person’s code would need to learn the original convention and apply it as well for consistency. Fortunately, the Java community does subscribe to a shared thought on how naming conventions should be applied to the many different elements in Java. Table 1-5 describes these conventions in a simple manner. When applying

TABLE 1-5 Java Naming Conventions

| Element | Lettering | Characteristic | Example |
|-------------------------------------|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|-------------------------------------------------------------|
| Class name | Begins uppercase, continues CamelCase | Noun | SpaceShip |
| Interface name | Begins uppercase, continues CamelCase | Adjective ending with “able” or “ible” when providing a capability; otherwise a noun | Dockable |
| Method name | Begins lowercase, continues CamelCase | Verb, may include adjective or noun | orbit |
| Instance and static variables names | Begins lowercase, continues CamelCase | Noun | moon |
| Parameters and local variables | Begins lowercase, continues CamelCase if multiple words are necessary | Single words, acronyms, or abbreviations | lop (line of position) |
| Generic type parameters | Single uppercase letter | The letter <i>T</i> is recommended | T |
| Constant | All uppercase letters | Multiple words separated by underscores | LEAGUE |
| Enumeration | Begins uppercase, continues CamelCase; the set of objects should be all uppercase | Noun | enum Occupation {MANNED, SEMI_MANNED, UNMANNED} |
| Package | All lowercase letters | Public packages should be the reversed domain name of the org | com.ocajexam.sim |

naming conventions, you should strive to use meaningful and unambiguous names. And remember that naming conventions exist for the primary goal of making Java programs more readable, and therefore maintainable. The practice of using CamelCase—using uppercase letters for the first characters in compound words—is part of the Java naming conventions.

Separators and Other Java Source Symbols

The Java programming language makes use of several separators and symbols to aid in the structuring of the source code in a software program. Table 1-6 details these separators and symbols.

TABLE 1-6 Symbols and Separators

| Symbol | Description | Purpose |
|--------|---------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| () | Parentheses | Encloses set of method arguments, encloses cast types, adjusts precedence in arithmetic expressions |
| { } | Braces | Encloses blocks of codes, initializes arrays |
| [] | Box brackets | Declares array types, initializes arrays |
| < > | Angle brackets | Encloses generics |
| ; | Semicolon | Terminates statement at the end of a line |
| , | Comma | Separates identifiers in variable declarations, separates values, separates expressions in a for loop |
| . | Period | Delineates package names, selects an object’s field or method, supports method chaining |
| : | Colon | Follows loop labels |
| ` ` | Single quotes | Defines a single character |
| -> | Arrow operator | Separates left-side parameters from the right-side expression |
| " " | Double quotes | Defines a string of characters |
| // | Forward slashes | Indicates a single-line comment |
| /* */ | Forward slashes with asterisks | Indicates a blocked comment for multiple lines |
| /** */ | Forward slashes with a double and single asterisk | Indicates Javadoc comments |

Java Class Structure

Every Java program has at least one class. A Java class has a signature, optional constructors, optional data members (fields), and optional methods, as outlined here:

```
[modifiers] class classIdentifier [extends superClassIdentifier]
[implements interfaceIdentifier1, interfaceIdentifier2, etc.] {
    [data members]
    [constructors]
    [methods]
}
```

Each class may extend one and only one superclass. Each class may implement one or more interfaces. Interfaces are separated by commas.

The following `SpaceShip` class shows typical elements annotated with comments. The file containing this `SpaceShip` class must be called `SpaceShip.java`. Note that the class declaration extends the `Ship` class and implements the `Dockable` interface. The `Dockable` interface includes the `dockShip` method, which is overridden here. `Ship` class methods would be inherited by the `SpaceShip` class. Chapters 4–7 go into more comprehensive details about creating and working with classes.

The following code shows the structure of a typical class:

```
package com.ocajexam.craft_simulator;

public class SpaceShip extends Ship implements Dockable {

    // Data Members
    public enum ShipType {
        FRIGATE, BATTLESHIP, MINELAYER, ESCORT, DEFENSE
    }
    ShipType shipType = ShipType.BATTLESHIP;

    // Constructors
    public SpaceShip() {
        System.out.println("\nSpaceShip created with default ship type.");
    }
    public SpaceShip(ShipType shipType) {
        System.out.println("\nSpaceShip created with specified ship type.");
        this.shipType = shipType;
    }

    // Methods
    @Override
```

```

    public void dockShip () {
        // TODO
    }
    @Override
    public String toString() {
        String shipTypeRefined = this.shipType.name().toLowerCase();
        return "The pirate ship is a " + shipTypeRefined + " ship.";
    }
}

```

This `SpaceShip` class can be instantiated as demonstrated in the following code:

```

package com.ocajexam.craft_simulator;
import com.ocajexam.craft_simulator.PirateShip.ShipType;
public class SpaceShipSimulator {

    public static void main(String[] args) {

        // Create SpaceShip object with default ship type
        SpaceShip ship1 = new SpaceShip ();
        // Prints "The pirate ship is a battleship."
        System.out.println(ship1);

        // Create SpaceShip object with specified ship type
        SpaceShip ship2 = new SpaceShip (ShipType.FRIGATE);
        // Prints "The pirate ship is a frigate ship."
        System.out.println(ship2);
    }
}

```



The override annotation (`@Override`) indicates that a method declaration intends on overriding a method declaration in the class's supertype.

CERTIFICATION OBJECTIVE

Compile and Interpret Java Code

Exam Objective Create executable Java applications with a main method, run a program from the command line, including console output

The Java Development Kit (JDK) 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

Because we'll need a sample application to use for our Java compiler and interpreter exercises, we'll employ the simple `GreetingsUniverse.java` source file, shown in the following listing, throughout the section. This sample includes the main method used as the entry point of the executed code. When the program is started, this is the first method to be called by the JVM. The main method shown here contains one line of code. This line,

```
System.out.println("Greetings, Universe!")
```

will print

```
Greetings, Universe!
```

to standard output. This output would typically be displayed on a Java application that was started from a console.

```
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 OCA exam, you will need to know the details surrounding only 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

FIGURE 1-7 Java Development Kit utilities

```

C:\Windows\system32\cmd.exe

c:\Program Files\Java\jdk1.7.0_04\bin>dir *.exe /w
Volume in drive C is OS
Volume Serial Number is 6059-69EE

Directory of c:\Program Files\Java\jdk1.7.0_04\bin

appletviewer.exe  apt.exe          extcheck.exe      idlj.exe
jar.exe           jarsigner.exe    java-rmi.exe       java.exe
javac.exe         javadoc.exe      javah.exe          javap.exe
javaw.exe         javaws.exe       jcmd.exe           jconsole.exe
jdb.exe          jhat.exe         jinfo.exe          jmap.exe
jps.exe          jrunscript.exe   jsadebugd.exe      jstack.exe
jstat.exe        jstatd.exe       jvisualvm.exe      keytool.exe
kinit.exe        klist.exe        ktab.exe           native2ascii.exe
orbd.exe         pack200.exe      policytool.exe     rmic.exe
rmid.exe        .rmiregistry.exe schemagen.exe       serialver.exe
servertool.exe   tnameserv.exe    unpack200.exe      wsgen.exe
wsimport.exe     xjc.exe

                46 File(s)          1,478,192 bytes
                0 Dir(s)        265,837,461,504 bytes free

c:\Program Files\Java\jdk1.7.0_04\bin>

```

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.



You will find that many projects use Apache Ant and/or Maven build environments. Understanding the fundamentals of the command-line tools is necessary for writing and maintaining the scripts associated with these build products.

Compiling Your Source Code with the -d Option

You may want to specify explicitly where you would like the compiled bytecode class files to go. You can accomplish this by 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\ocajexam\tutorial\
GreetingsUniverse.class
```

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 options or clicking a hot key. The disadvantage is that even though you may establish your settings through a configuration dialog 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 Code with the -classpath Option

If you want 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...]
```

Interpreting Your Bytecode

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

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

```
java MainClass
```

FIGURE 1-8 Bytecode conversion



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, because 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 want 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.ocajexam.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 comprising the `PropertiesManager` class prints out all of the system properties:

```
import java.util.Properties;
public class PropertiesManager {
    public static void main(String[] args) {
        if (args.length == 0) {System.exit(0);}
        Properties props = System.getProperties();
        /* New property example */

        props.setProperty("new_property2", "new_value2");
        switch (args[0]) {
            case "-list_all":
                props.list(System.out); // Lists all properties
```

```

        break;
    case "-list_prop":
        /* Lists value */
        System.out.println(props.getProperty(args[1]));
        break;
    default:
        System.out.println("Usage: java
        PropertiesManager [-list_all]");
        System.out.println("        java
        PropertiesManager [-list_prop [property]]");
        break;
    }
}
}

```

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 1-7 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 putting 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.



Check out the other JDK utilities at your disposal. Java Flight Recorder and Java Mission Control in particular are valuable GUI-based tools that are used to monitor, profile, and collect runtime information.

TABLE 1-7 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:\Windows\system32\cmd.exe

c:\code>java -version
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java -version INVALID_ARGUMENT
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

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

c:\code>_

```


EXERCISE 1-2**Compiling and Interpreting Packaged Software**

When you compile and run packaged software from an IDE, the execution process can be as easy as clicking a run icon, as the IDE will maintain the classpath for you and will also let you know if anything is out of sorts. 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 placed in the `com.ocajexam.tutorial` package (that is, the `com/ocajexam/tutorial` directory).

```
package com.ocajexam.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.ocajexam.tutorial.GreetingsUniverse
```

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

```
package com.ocajexam.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/ocajexam/tutorial/` and `src/com/ocajexam/tutorial/planets/`.
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 as follows:

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

CERTIFICATION SUMMARY

This chapter discussed packaging, structuring, 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 commonly used Java packages: `java.awt`, `javax.swing`, `java.net`, `java.io`, and `java.util`. We discussed the basic structure of a Java class. We then 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 (outside of an IDE) package, build, and run basic Java programs independently.



TWO-MINUTE DRILL

Understand 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.
- ☐ 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.

Understand 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 include classes and interfaces that support the Java Collections Framework, legacy collection classes, event model, date and time facilities, and internationalization functionality.

Understand Class Structure

- ☐ Naming conventions are used to make Java programs more readable and maintainable.
- ☐ Naming conventions are applied to several Java elements, including class names, interface names, method names, instance and static variable names, parameter and local variable names, generic type parameter names, constant names, enumeration names, and package names.
- ☐ The preferred order of presenting elements in a class is data members, followed by constructors, followed by methods. Note that the inclusion of each type of element is optional.

Compile and Interpret 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.
- ☐ The `-h` command-line option can be applied either to the compiler or the interpreter to print out the tool's usage information.

SELF TEST

Understanding Packages

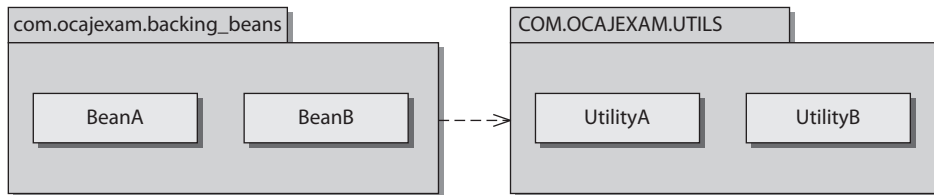
1. Which two import statements will allow for the import of the `HashMap` class?
 - A. `import java.util.HashMap;`
 - B. `import java.util.*;`
 - C. `import java.util.HashMap.*;`
 - D. `import java.util.hashMap;`
2. Which statement would designate that your file belongs in the package `com.ocajexam.utilities`?
 - A. `pack com.ocajexam.utilities;`
 - B. `package com.ocajexam.utilities.*`
 - C. `package com.ocajexam.utilities.*;`
 - D. `package com.ocajexam.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`

Understand Package-Derived Classes

4. The `JCheckBox` and `JComboBox` classes belong to which package?
 - A. `java.awt`
 - B. `javax.awt`
 - C. `java.swing`
 - D. `javax.swing`
5. Which package contains the Java Collections Framework?
 - A. `java.io`
 - B. `java.net`
 - C. `java.util`
 - D. `java.utils`

36 Chapter 1: Packaging, Compiling, and Interpreting Java Code

6. 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
7. Which API provides a lightweight solution for GUI components?
- A. AWT
 - B. Abstract Window Toolkit
 - C. Swing
 - D. AWT and Swing
8. Consider the following illustration. What problem exists with the packaging? You may wish to reference Appendix G of the Unified Modeling Language (UML) for assistance.



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

Understand Class Structure

9. When apply naming conventions, which Java elements should start with an uppercase letter and continue on using the CamelCase convention?
- A. Class names
 - B. Interface names
 - C. Constant names
 - D. Package names
 - E. All of the above

10. When instantiating an object with generics, should you use angle brackets, box brackets, curly brackets, or double-quotation marks to enclose the generic type? Select the appropriate answer.
- A. `List<Integer> a = new ArrayList<Integer>();`
 - B. `List[Integer] a = new ArrayList[Integer]();`
 - C. `List{Integer} a = new ArrayList{Integer}();`
 - D. `List"Integer" a = new ArrayList"Integer"();`
11. When you're organizing the elements in a class, which order is preferred?
- A. Data members, methods, constructors
 - B. Data members, constructors, methods
 - C. Constructors, methods, data members
 - D. Constructors, data members, methods
 - E. Methods, constructors, data members

Compile and Interpret Java Code

12. 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`
13. 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`
14. 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 -classpaths /project/classes/ MainClass.java`

- 15.** Which command-line usages appropriately set a system property value?
- A. `java -Dcom.ocajexam.propertyValue=003 MainClass`
 - B. `java -d com.ocajexam.propertyValue=003 MainClass`
 - C. `java -prop com.ocajexam.propertyValue=003 MainClass`
 - D. `java -D:com.ocajexam.propertyValue=003 MainClass`

SELF TEST ANSWERS

Understand 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 wildcard 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.ocajexam.utilities`?
 - A. `pack com.ocajexam.utilities;`
 - B. `package com.ocajexam.utilities.*`
 - C. `package com.ocajexam.utilities.*;`
 - D. `package com.ocajexam.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. **A** is incorrect because the word `pack` is not a valid keyword. **B** is incorrect because a `package` statement must end with a semicolon, and you cannot use asterisks in `package` statements. **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.

Understand Package-Derived Classes

4. 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 an uppercase *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.

5. 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`.

6. 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. Internationalization (i18n), RMI, JDBC, JNDI, and the Collections framework are not included in the Basic I/O API.

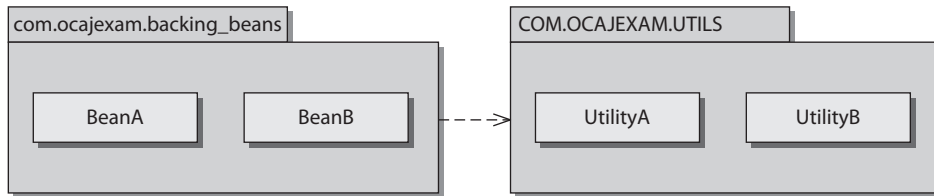
7. Which API provides a lightweight solution for GUI components?
- A. AWT
 - B. Abstract Window Toolkit
 - C. Swing
 - D. AWT and Swing

Answer:

- ☒ **C.** The Swing API provides a lightweight solution for GUI components, meaning that the Swing API's classes render using pure Java code and not native platform widgets.
- ☒ **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.

42 Chapter 1: Packaging, Compiling, and Interpreting Java Code

8. Consider the following illustration. What problem exists with the packaging? You may wish to reference Appendix G of the Unified Modeling Language (UML) for assistance.



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

Answer:

- ☒ **D.** `COM.OCAJEXAM.UTILS` fails to meet the appropriate package naming conventions. Package names should be lowercase and should use an underscore between words. However, the words in `ocajexam` are joined in the URL; therefore, excluding the underscore here is acceptable. The package name should read `com.ocajexam.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.ocajexam.backing_beans` meets appropriate packaging naming conventions.

Understand Class Structure

9. When applying naming conventions, which Java elements should start with an uppercase letter and continue on using the CamelCase convention?
- A. Class names
 - B. Interface names
 - C. Constant names
 - D. Package names
 - E. All of the above

Answer:

- ☒ **A** and **B**. Class names and interface names should start with an uppercase letter and continue on using the CamelCase convention.
- ☒ **C** and **D** are incorrect. **C** is incorrect because constant names should be all uppercase letters separated by underscores. **D** is incorrect because package names do not include uppercase letters, nor do they subscribe to the CamelCase convention.

- 10.** When instantiating an object with generics, should you use angle brackets, box brackets, parentheses, or double-quotation marks to enclose the generic type? Select the appropriate answer.
- A. `List<Integer> a = new ArrayList<Integer>();`
 - B. `List[Integer] a = new ArrayList[Integer]();`
 - C. `List{Integer} a = new ArrayList{Integer}();`
 - D. `List"Integer" a = new ArrayList"Integer"();`

Answer:

- ☒ **A**. Generics use angle brackets.
- ☒ **B**, **C**, and **D** are incorrect. Box brackets (**B**), curly brackets (**C**), and double quotation marks (**D**) are not used to enclose the generic type.

- 11.** When you're organizing the elements in a class, which order is preferred?
- A. Data members, methods, constructors
 - B. Data members, constructors, methods
 - C. Constructors, methods, data members
 - D. Constructors, data members, methods
 - E. Methods, constructors, data members

Answer:

- ☒ **B**. The preferred order in presenting elements in a class is to present the data members first, followed by constructors, followed by methods.
- ☒ **A**, **C**, **D**, and **E** are incorrect. Although ordering the elements in these manners will not cause any functional or compilation errors, none of these is the preferred order.

Compile and Interpret Java Code

12. 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.

13. 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.

- 14.** 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 -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. The option flags `-sp` (**B**) and `-classpaths` (**D**) are invalid.

- 15.** Which command-line usages appropriately set a system property value?
- A. `java -Dcom.ocajexam.propertyValue=003 MainClass`
 - B. `java -d com.ocajexam.propertyValue=003 MainClass`
 - C. `java -prop com.ocajexam.propertyValue=003 MainClass`
 - D. `java -D:com.ocajexam.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. The `-d` (**B**), `-prop` (**C**), and `-D:` (**D**) flags are invalid ways to designate a system property.