Access Control 6

Chapter Topics

- Understanding the features in Java that aid encapsulation
- Knowing the structure of a Java compilation unit
- Creating and using packages
- Using class-path to search for types during program compilation and execution
- Compiling and running code from packages
- Applying accessibility rules for top-level reference types and their members
- Applying scope rules for class members and local variables
- Implementing immutability

Java SE 17 Developer Exam Objectives	
[3.4] Understand variable scopes, use local variable type inference, apply encapsulation, and make objects immutable ❍ *Variable scope, encapsulation, and immutability are covered in this chapter.* ❍ *For local variable type inference, see §3.13, p. 142.*	§6.1, p. 324 §6.6, p. 352 §6.7, p. 356

Java SE 11 Developer Exam Objectives	
[3.4] Understand variable scopes, applying encapsulation and make objects immutable	§6.1, p. 324 §6.6, p. 352 §6.7, p. 356

Java provides language features (such as classes, packages, and modules) that allow code to be encapsulated. This chapter covers creating and using packages, and compiling and running code from packages. Accessibility rules define how entities (e.g., classes and methods) are accessed in a Java program. Class scope rules define how class members are accessed in a class and block scope rules define how local variables are accessed in a block. Immutability of objects is a common class design strategy. The discussion in this chapter is based on *non-modular* code. Modules are discussed in **Chapter 19**, **p. 1161**.

6.1 Design Principle: Encapsulation

An object has properties and behaviors that are *encapsulated* inside the object. The services that the object offers to its clients make up its *contract*, or public interface. Only the contract defined by the object is available to the clients. The *implementation* of the object's properties and behavior is not a concern of the clients. Encapsulation helps to make clear the distinction between an object's contract and its implementation. This demarcation has major consequences for program development, as the implementation of an object can change without affecting the clients. Encapsulation also reduces complexity, as the internals of an object are hidden from the clients, which cannot alter its implementation.

Encapsulation is achieved through *information hiding*, by making judicious use of language features provided for this purpose. Information hiding in Java can be achieved at the following levels of granularity:

- *Method or block declaration*
  Localizing information in a method or a block hides it from the outside. Local variables can only be accessed in the method or the block according to their scope.
- *Reference type declaration*
  The accessibility of members declared in a reference type declaration (class, enum type, interface) can be controlled through access modifiers (**p. 347**). One much-advocated information-hiding practice is to prevent clients from having direct access to data maintained by an object. The fields of the object are `private`, and the object's contract defines `public` methods for the services provided by the object. Such tight encapsulation helps to separate the use from the implementation of a reference type.
- *Package declaration*
  Top-level reference types that belong together can be grouped into relevant packages by using the `package` statement. An `import` statement can be used to access types in other packages by their simple name. Inter-package accessibility of types can be controlled through `public` or package accessibility (**p. 345**).
- *Module declaration*
  Packages that are related can be grouped into a module by using a `module` declaration. How modules are created, accessed, compiled, and deployed is discussed in detail in **Chapter 19**, **p. 1161**.

Ample examples throughout the book illustrate how encapsulation can be achieved at different levels by using features provided by the Java programming language.

6.2 Java Source File Structure

The structure of a skeletal Java source file is depicted in **Figure 6.1**. A Java source file can have the following elements that, if present, must be specified in the following order:

1. An optional `package` declaration to specify a package name. Packages are discussed in §6.3, p. 326.
2. Zero or more `import` declarations. Since `import` declarations introduce type or static member names in the source code, they must be placed before any type declarations. Both type and static `import` declarations are discussed in §6.3, p. 329.
3. Any number of *top-level* type declarations. Class, enum, and interface declarations are collectively known as *type declarations*. Since these declarations belong to the same package, they are said to be defined at the *top level*, which is the package level.
   The type declarations can be defined in any order. Technically, a source file need not have any such declarations, but that is hardly useful.
   The JDK imposes the restriction that at most one `public` class declaration per source file can be defined. If a `public` class is defined, the file name must match this `public` class. For example, if the `public` class name is `NewApp`, the file name must be `NewApp.java`.
   Classes are discussed in §3.1, p. 99; interfaces are discussed in §5.6, p. 237; and enums are discussed in §5.13, p. 287.

Modules introduce another Java source file that contains a single module declaration (§19.3, p. 1168).

Note that except for the `package` and the `import` statements, all code is encapsulated in classes, interfaces, enums, and records. No such restriction applies to comments and whitespace.

```
// File: NewApp.java
// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilepackage;
// PART 2: (ZERO OR MORE) import declarations
import java.io.*;
import java.util.*;
import static java.lang.Math.*;
// PART 3: (ZERO OR MORE) top-level declarations
public class NewApp { }
class A { }
interface IX { }
record B() { }
enum C { }
// end of file
```

Figure 6.1 *Java Source File Structure*

6.3 Packages

A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces, enums, and records.

Figure 6.2 shows an example of a package hierarchy comprising a package called `wizard` that contains two other packages: `pandorasbox` and `spells`. The package `pandorasbox` has a class called `Clown` that implements an interface called `Magic`, also found in the same package. In addition, the package `pandorasbox` has a class called `LovePotion` and a subpackage called `artifacts` containing a class called `Ailment`. The package `spells` has two classes: `Baldness` and `LovePotion`. The class `Baldness` is a subclass of class `Ailment` found in the subpackage `artifacts` in the package `pandorasbox`.

The dot (`.`) notation is used to uniquely identify package members in the package hierarchy. The class `wizard.pandorasbox.LovePotion`, for example, is different from the class `wizard.spells.LovePotion`. The `Ailment` class can be easily identified by the name `wizard.pandorasbox.artifacts.Ailment`, which is known as the *fully qualified name* of the type. Note that the fully qualified name of the type in a named package comprises the fully qualified name of the package and the simple name of the type. The *simple type name* `Ailment` and the *fully qualified package name* `wizard.pandorasbox.artifacts` together define the *fully qualified type name* `wizard.pandorasbox.artifacts.Ailment`.

Java programming environments usually map the fully qualified name of packages to the underlying (hierarchical) file system. For example, on a Unix system, the class file `LovePotion.class` corresponding to the fully qualified name `wizard.pandorasbox.LovePotion` would be found under the directory `wizard/pandorasbox`.

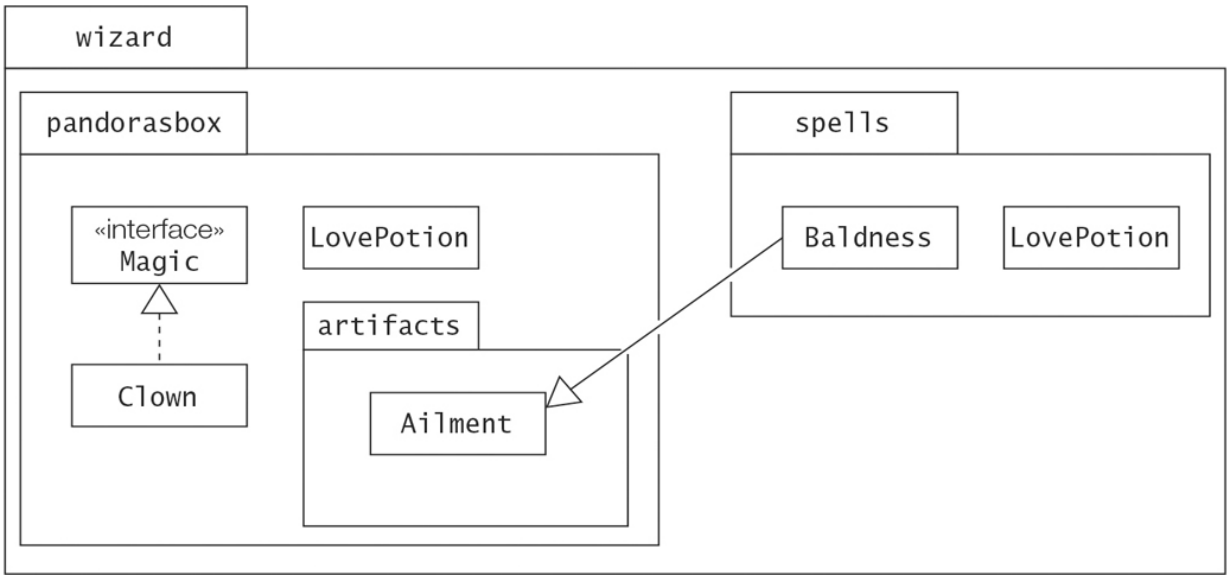


Figure 6.2 *Package Structure*

Conventionally, the *reverse* DNS (*Domain Name System*) notation based on the Internet domain names is used to uniquely identify packages. If the package `wizard` was implemented by a company called Sorcerers Limited that owns the domain `sorcerersltd.com`, its fully qualified name would be

```
com.sorcerersltd.wizard
```

Because domain names are unique, packages with this naming scheme are globally identifiable. It is not advisable to use the top-level package names `java` and `sun`, as these are reserved for the Java designers.

Note that each component of a package name must be a legal Java identifier. The following package would be illegal:

```
org.covid-19.2022.vaccine
```

The package name below is legal:

```
org.covid_19._2022.vaccine
```

A subpackage would be located in a subdirectory of the directory corresponding to its parent package. Apart from this locational relationship, a subpackage is an independent package with no other relation to its parent package. The subpackage `wizard.pandorasbox.artifacts` could easily have been placed elsewhere, as long as it was uniquely identified. Subpackages in a package do not affect the accessibility of the other package members. For all intents and purposes, subpackages are more an *organizational* feature than a language feature. Accessibility of members defined in type declarations is discussed in §6.5, p. 345.

Defining Packages

A package hierarchy represents an organization of the Java classes and interfaces. It does *not* represent the *source code* organization of the classes and interfaces. The source code is of no consequence in this regard. Each Java source file (also called *compilation unit*) can contain zero or more type declarations, but the compiler produces a separate *class* file containing the Java bytecode for each of them. A type declaration can indicate that its Java bytecode should be placed in a particular package, using a `package` declaration.

The `package` statement has the following syntax:

Click here to view code image

```
package fully_qualified_package_name;
```

At most, one `package` declaration can appear in a source file, and it must be the first statement in the source file. The package name is saved in the Java bytecode of the types contained in the package. Java naming conventions recommend writing package names in lowercase letters.

Note that this scheme has two consequences. First, all the classes and interfaces in a source file will be placed in the same package. Second, several source files can be used to specify the contents of a package.

If a `package` declaration is omitted in a compilation unit, the Java bytecode for the declarations in the compilation unit will belong to an *unnamed package* (also called the *default package*), which is typically synonymous with the current working directory on the host system.

Example 6.1 illustrates how the packages in Figure 6.2 can be defined using the `package` declaration. There are four compilation units. Each compilation unit has a `package` declaration, ensuring that the type declarations are compiled into the correct package. The complete code can be found in Example 6.7, p. 345.

Example 6.1 *Defining Packages and Using Type Import*

Click here to view code image

```
// File name: Clown.java                          // This file has 2 type declarations
package wizard.pandorasbox;                       // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing specific class

public class Clown implements Magic { /* ... */ }

interface Magic { /* ... */ }
```

Click here to view code image

```
// File name: LovePotion.java
package wizard.pandorasbox;                       // Package declaration

public class LovePotion { /* ... */ }
```

Click here to view code image

```
// File name: Ailment.java
package wizard.pandorasbox.artifacts;             // Package declaration

public class Ailment { /* ... */ }
```

Click here to view code image

```
// File name: Baldness.java                       // This file has 2 type declarations
package wizard.spells;                            // Package declaration

import wizard.pandorasbox.*;                      // (1) Type-import-on-demand
import wizard.pandorasbox.artifacts.*;            // (2) Import from subpackage

public class Baldness extends Ailment {           // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;           // (3) Fully qualified class name
  LovePotion tlcTwo;                              // Class in same package
  // ...
}
```

```
class LovePotion { /* ... */ }
```

Using Packages

The import facility in Java makes it easier to use the contents of packages. This subsection discusses importing *reference types* and *static members of reference types* from packages.

Importing Reference Types

The accessibility of types (classes, interfaces, and enums) in a package determines their access from other packages. Given a reference type that is accessible from outside a package, the reference type can be accessed in two ways. One way is to use the fully qualified name of the type. However, writing long names can become tedious. The second way is to use the `import` declaration that provides a shorthand notation for specifying the name of the type, often called *type import*.

The `import` declarations must be the first statement after any `package` declaration in a source file. The simple form of the `import` declaration has the following syntax:

Click here to view code image

```
import fully_qualified_type_name;
```

This is called *single-type-import*. As the name implies, such an `import` declaration provides a shorthand notation for a single type. The *simple* name of the type (i.e., its identifier) can now be used to access this particular type. Given the `import` declaration

Click here to view code image

```
import wizard.pandorasbox.Clown;
```

the simple name `Clown` can be used in the source file to refer to this class.

Alternatively, the following form of the `import` declaration can be used:

Click here to view code image

```
import fully_qualified_package_name.*;
```

This is called *type-import-on-demand*. It allows *any* type from the specified package to be accessed by its simple name. Given the `import` declaration

Click here to view code image

```
import wizard.pandorasbox.*;
```

the classes `Clown` and `LovePotion` and the interface `Magic` that are in the package `wizard.pandorasbox` can be accessed by their simple name in the source file.

An `import` declaration does not recursively import subpackages, as such nested packages are autonomous packages. The declaration also does not result in inclusion of the source code of the types; rather, it simply imports type names (i.e., it makes type names available to the code in a compilation unit).

All compilation units implicitly import the `java.lang` package (§8.1, p. 425). This is the reason why we can refer to the class `String` by its simple name, and need not use its fully qualified name `java.lang.String` all the time.

Import statements are *not* present in the compiled code, as all type names in the source code are replaced with their fully qualified names by the compiler.

Example 6.1 shows several usages of the `import` statement. Here we will draw attention to the class `Baldness` in the file `Baldness.java`. This class relies on two classes that have the same simple name `LovePotion` but are in different packages: `wizard.pandorasbox` and `wizard.spells`. To distinguish between the two classes, we can use their fully qualified names. However, since one of them is in the same package as the class `Baldness`, it is enough to fully qualify the class from the other package. This solution is used in Example 6.1 at (3). Note that the import of the `wizard.pandorasbox` package at (1) becomes redundant. Such name conflicts can usually be resolved by using variations of the `import` declaration together with fully qualified names.

The class `Baldness` extends the class `Ailment`, which is in the subpackage `artifacts` of the `wizard.pandorasbox` package. The `import` declaration at (2) is used to import the types from the subpackage `artifacts`.

The following example shows how a single-type-import declaration can be used to disambiguate a type name when access to the type is ambiguous by its simple name. The following `import` statement allows the simple name `List` to be used as shorthand for the `java.awt.List` type as expected:

Click here to view code image

```
import java.awt.*;            // imports all reference types from java.awt
```

Given the two `import` declarations

Click here to view code image

```
import java.awt.*;            // imports all type names from java.awt
import java.util.*;           // imports all type names from java.util
```

the simple name `List` is now ambiguous because both the types `java.util.List` and `java.awt.List` match.

Adding a single-type-import declaration for the `java.awt.List` type allows the simple name `List` to be used as a shorthand notation for this type:

Click here to view code image

```
import java.awt.*;            // imports all type names from java.awt
import java.util.*;           // imports all type names from java.util
import java.awt.List;         // imports the type List from java.awt explicitly
```

Importing Static Members of Reference Types

Analogous to the type import facility, Java also allows import of *static members* of reference types from packages, often called *static import*. Imported static members can be used by their simple names, and therefore need not be qualified. Importing static members of reference types from the unnamed package is not permissible.

The two forms of static import are shown here:

- *Single static import*: imports a specific static member from the designated type
  `import static` *fully_qualified_type_name* `.` *static_member_name* `;`
- *Static import on demand*: imports all static members in the designated type
  `import static` *fully_qualified_type_name* `.*;`

Both forms require the use of the keyword `import` followed by the keyword `static`, although the feature is called *static import*. In both cases, the *fully qualified name of the reference type* we are importing from is required.

The first form allows *single static import* of individual static members, and is demonstrated in Example 6.2. The constant `PI`, which is a static field in the class `java.lang.Math`, is imported at (1). Note the use of the fully qualified name of the type in the static import statement. The static method named `sqrt` from the class `java.lang.Math` is imported at (2). Only the *name* of the static method is specified in the static import statement; no parameters are listed. Use of any other static member from the `Math` class requires that the fully qualified name of the class be specified. Since types from the `java.lang` package are imported implicitly, the fully qualified name of the `Math` class is not necessary, as shown at (3).

Static import on demand is easily demonstrated by replacing the two `import` statements in Example 6.2 with the following `import` statement:

Click here to view code image

```
import static java.lang.Math.*;
```

We can also dispense with the use of the class name `Math` at (3), as all static members from the `Math` class are now imported:

Click here to view code image

```
double hypotenuse = hypot(x, y);   // (3') Type name can now be omitted.
```

Example 6.2 *Single Static Import*

Click here to view code image

```
import static java.lang.Math.PI;           // (1) Static field
import static java.lang.Math.sqrt;         // (2) Static method
// Only specified static members are imported.

public class Calculate3 {
  public static void main(String[] args) {
    double x = 3.0, y = 4.0;

    double squareroot = sqrt(y);           // Simple name of static method
    double hypotenuse = Math.hypot(x, y);  // (3) Requires type name
    double area = PI * y * y;              // Simple name of static field
    System.out.printf("Square root: %.2f, hypotenuse: %.2f, area: %.2f%n",
                        squareroot, hypotenuse, area);
  }
}
```

Output from the program:

Click here to view code image

```
Square root: 2.00, hypotenuse: 5.00, area: 50.27
```

Example 6.3 illustrates how static import can be used to access interface constants (§5.6, p. 254). The static `import` statement at (1) allows the interface constants in the package `mypkg` to be accessed by their simple names. The static import facility avoids the `MyFactory` class having to *implement* the interface so as to access the constants by their simple name (often referred to as the *interface constant antipattern*):

Click here to view code image

```
public class MyFactory implements mypkg.IMachineState {
 // ...
}
```

Example 6.3 *Avoiding the Interface Constant Antipattern*

Click here to view code image

```
package mypkg;

public interface IMachineState {
  // Fields are public, static, and final.
  int BUSY = 1;
  int IDLE = 0;
  int BLOCKED = -1;
}
```

Click here to view code image

```
import static mypkg.IMachineState.*;    // (1) Static import interface constants

public class MyFactory {
  public static void main(String[] args) {
    int[] states = { IDLE, BUSY, IDLE, BLOCKED }; // (2) Access by simple name
    for (int s : states)
      System.out.print(s + " ");
  }
}
```

Output from the program:

```
0 1 0 -1
```

Static import is ideal for importing enum constants from packages, as such constants are static members of an enum type (§5.13, p. 287). Example 6.4 combines type and static imports. The enum constants can be accessed at (5) using their simple names because of the static import statement at (2). The type import at (1) is required to access the enum type `State` by its simple name at (4) and (6).

Example 6.4 *Importing Enum Constants*

Click here to view code image

```
package mypkg;

public enum State { BUSY, IDLE, BLOCKED }
```

Click here to view code image

```
// File: Factory.java (in unnamed package)
import mypkg.State;                  // (1) Single type import

import static mypkg.State.*;         // (2) Static import on demand
import static java.lang.System.out;  // (3) Single static import

public class Factory {
  public static void main(String[] args) {
    State[] states = {               // (4) Using type import implied by (1)
        IDLE, BUSY, IDLE, BLOCKED    // (5) Using static import implied by (2)
    };
    for (State s : states)           // (6) Using type import implied by (1)
      out.print(s + " ");            // (7) Using static import implied by (3)
  }
}
```

Output from the program:

```
IDLE BUSY IDLE BLOCKED
```

Identifiers in a class can *shadow* static members that are imported. Example 6.5 illustrates the case where the parameter `out` of the method `writeInfo()` has the same name as the statically imported field `java.lang.System.out`. The type of the parameter `out` is `ShadowImport` and that of the statically imported field `out` is `PrintStream`. Both classes `PrintStream` and `ShadowImport` define the method `println()` that is called in the program. The only way to access the imported field `out` in the method `write-Info()` is to use its fully qualified name.

Example 6.5 *Shadowing Static Import*

Click here to view code image

```
import static java.lang.System.out;       // (1) Static import

public class ShadowImport {

  public static void main(String[] args) {
    out.println("Calling println() in java.lang.System.out");
    ShadowImport sbi = new ShadowImport();
    writeInfo(sbi);
  }

  // Parameter out shadows java.lang.System.out:
  public static void writeInfo(ShadowImport out) {
    out.println("Calling println() in the parameter out");
    System.out.println("Calling println() in java.lang.System.out"); // Qualify
  }

  public void println(String msg) {
    out.println(msg + " of type ShadowImport");
  }
}
```

Output from the program:

Click here to view code image

```
Calling println() in java.lang.System.out
Calling println() in the parameter out of type ShadowImport
Calling println() in java.lang.System.out
```

The next code snippet illustrates a common conflict that occurs when a static field with the same name is imported by *several* static import statements. This conflict is readily resolved by using the fully qualified name of the field. In the case shown here, we can use the simple name of the class in which the field is declared, as the `java.lang` package is implicitly imported by all compilation units.

Click here to view code image

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

public class StaticFieldConflict {
  public static void main(String[] args) {
    System.out.println(MAX_VALUE);          // (1) Ambiguous! Compile-time error!
    System.out.println(Integer.MAX_VALUE);  // OK
    System.out.println(Double.MAX_VALUE);   // OK
  }
}
```

Conflicts can also occur when a static method with the same signature is imported by several static import statements. In Example 6.6, a method named `binarySearch` is imported 21 times by the static import statements. This method is overloaded twice in the `java.util.Collections` class and 18 times in the `java.util.Arrays` class, in addition to one declaration in the `mypkg.Auxiliary` class. The classes `java.util.Arrays` and `mypkg.Auxiliary` have a declaration of this method with the *same signature* (`binarySearch(int[], int`) that matches the method call at (2), resulting in a signature conflict that is flagged as a compile-time error. The conflict can again be resolved by specifying the fully qualified name of the method.

If the static import statement at (1) is removed, there is no conflict, as only the class `java.util.Arrays` has a method that matches the method call at (2). If the declaration of the method `binarySearch()` at (3) is allowed, there is also *no* conflict, as this method declaration will *shadow* the imported method whose signature it matches.

Example 6.6 *Conflict in Importing a Static Method with the Same Signature*

Click here to view code image

```
package mypkg;

public class Auxiliary {
  public static int binarySearch(int[] a, int key) { // Same in java.util.Arrays
    // Implementation is omitted.
    return -1;
  }
}
```

Click here to view code image

```
// File: MultipleStaticImport.java (in unnamed package)
import static java.util.Collections.binarySearch;  //    2 overloaded methods
import static java.util.Arrays.binarySearch;       // + 18 overloaded methods
import static mypkg.Auxiliary.binarySearch; // (1) Causes signature conflict

public class MultipleStaticImport {
  public static void main(String[] args) {
    int index = binarySearch(new int[] {10, 50, 100}, 50); // (2) Ambiguous!
    System.out.println(index);
  }

//public static int binarySearch(int[] a, int key) {       // (3)
//  return -1;
//}
}
```

Compiling Code into Package Directories

Conventions for specifying pathnames vary on different platforms. In this chapter, we will use pathname conventions used on a Unix-based platform. While trying out the examples in this section, attention should be paid to platform dependencies in this regard—especially the fact that the *separator characters* in *file paths* for the Unix-based and Windows platforms are `/` and `\`, respectively.

As mentioned earlier, a package can be mapped on a hierarchical file system. We can think of a package name as a pathname in the file system. Referring to **Example 6.1**, the package name `wizard.pandorasbox` corresponds to the pathname `wizard/ pandorasbox`. The Java bytecode for all types declared in the source files `Clown.java` and `LovePotion.java` will be placed in the *package directory* with the pathname `wizard/pandorasbox`, as these source files have the following `package` declaration:

```
package wizard.pandorasbox;
```

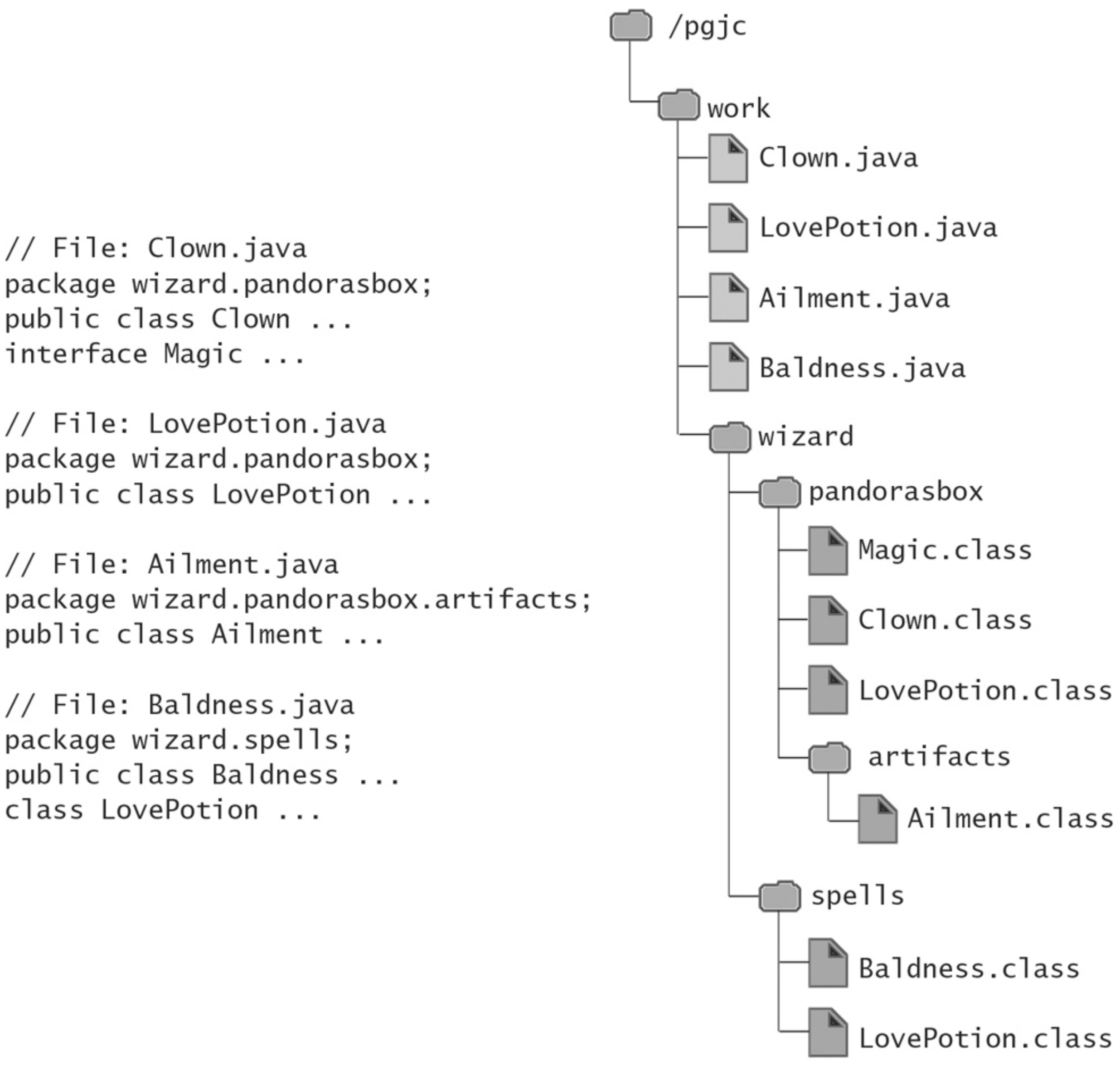
The *location* in the file system where the package directory should be created is specified using the `-d` option (`d` for *destination*) of the `javac` command. The term *destination directory* is a synonym for this location in the file system. The compiler will create the package directory with the pathname `wizard/pandorasbox` (including any subdirectories required) *under* the specified location, and will place the Java bytecode for the types declared in the source files `Clown.java` and `LovePotion.java` inside the package directory.

Assuming that the current directory (`.`) is the directory `/pgjc/work,` and the four source code files in **Figure 6.3a** (see also **Example 6.1**) are found in this directory, the following command issued in the current directory will create a file hierarchy (**Figure 6.3b**) under this directory that mirrors the package structure in **Figure 6.2**, **p. 327**:

Click here to view code image

```
>javac -d . Clown.java LovePotion.java Ailment.java Baldness.java
```

Note that two of the source code files in Figure 6.3a have multiple type declarations. Note also the subdirectories that are created for a fully qualified package name, and where the class files are located. In this command line, the space between the `-d` option and its argument is mandatory.



```
// File: Clown.java
package wizard.pandorasbox;
public class Clown ...
interface Magic ...

// File: LovePotion.java
package wizard.pandorasbox;
public class LovePotion ...

// File: Ailment.java
package wizard.pandorasbox.artifacts;
public class Ailment ...

// File: Baldness.java
package wizard.spells;
public class Baldness ...
class LovePotion ...
```

(a) Source code files (b) File hierarchy

Figure 6.3 *Compiling Code into Package Directories*

The wildcard `*` can be used to specify all Java source files to be compiled from a directory. It expands to the names of the Java source files in that directory. The two commands below are equivalent to the command above.

```
>javac -d . *.java
>javac -d . ./*.java
```

We can specify any *relative* pathname that designates the destination directory, or its *absolute* pathname:

Click here to view code image

```
>javac -d /pgjc/work Clown.java LovePotion.java Ailment.java Baldness.java
```

We can, of course, specify destinations other than the current directory where the class files with the bytecode should be stored. The following command in the current directory `/pgjc/work` will create the necessary packages with the class files under the destination directory `/pgjc/myapp`:

Click here to view code image

```
>javac -d ../myapp Clown.java LovePotion.java Ailment.java Baldness.java
```

Without the `-d` option, the default behavior of the `javac` compiler is to place all class files directly under the current directory (where the source files are located), rather than in the appropriate subdirectories corresponding to the packages.

The compiler will report an error if there is any problem with the destination directory specified with the `-d` option (e.g., if it does not exist or does not have the right file permissions).

Running Code from Packages

Referring to **Example 6.1**, if the current directory has the absolute pathname `/pgjc/work` and we want to run `Clown.class` in the directory with the pathname `./wizard/ pandorasbox`, the *fully qualified name* of the `Clown` class *must* be specified in the `java` command:

Click here to view code image

```
>java wizard.pandorasbox.Clown
```

This command will load the bytecode of the class `Clown` from the file with the pathname `./wizard/pandorasbox/Clown.class`, and will start the execution of its `main()` method.

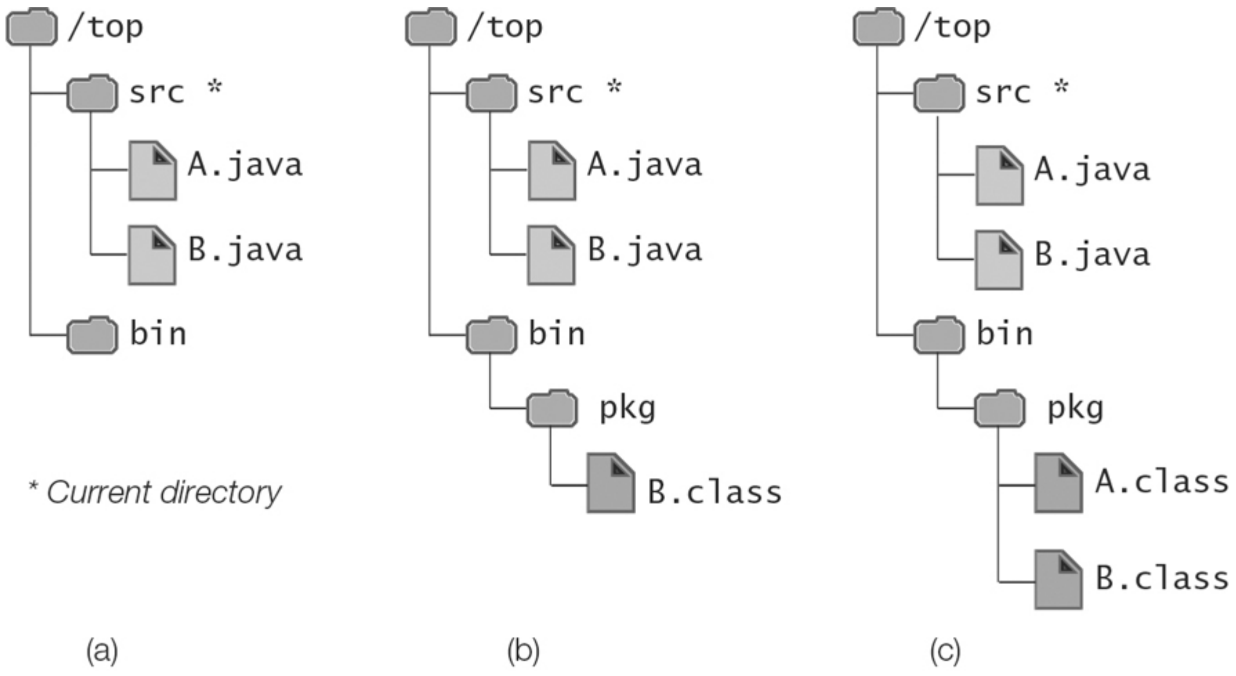
6.4 Searching for Classes on the Class Path

A program typically uses other precompiled classes and libraries, in addition to the ones provided by the Java standard libraries. In order for the JDK tools to find these files efficiently, the `CLASSPATH` environment variable or the `-classpath` option can be used, both of which are explained below.

In particular, the `CLASSPATH` environment variable can be used to specify the *class search path* (usually abbreviated to just *class path*), which is the *pathnames* or *locations* in the file system where JDK tools should look when searching for third-party and user-defined classes. Alternatively, the `-classpath` option (*short form* `-cp`) of the JDK tool commands can be used for the same purpose. The `CLASSPATH` environment variable is not recommended for this purpose, as its class path value affects *all* Java applications on the host platform, and any application can modify it. However, the `-classpath` option can be used to set the class path for each application individually. This way, an application cannot modify the class path for other applications. The class path specified in the `-classpath` option supersedes the path or paths set by the `CLASSPATH` environment variable while the JDK tool command is running. We will not discuss the `CLASSPATH` environment variable here, and will assume it to be undefined.

Basically, the JDK tools first look in the directories where the Java standard libraries are installed. If the class is not found in the standard libraries, the tool searches in the class path. When no class path is defined, the default value of the class path is assumed to be the current directory. If the `-classpath` option is used and the current directory should be searched by the JDK tool, the current directory must be specified as an entry in the class path, just like any other directory that should be searched. This is most conveniently done by including `'.'` as one of the entries in the class path.

We will use the file hierarchies shown in **Figure 6.4** to illustrate some of the intricacies involved when searching for classes. The current directory has the absolute pathname `/top/src`, where the source files are stored. The package `pkg` will be created under the directory with the absolute pathname `/top/bin`. The source code in the two source files `A.java` and `B.java` is also shown in **Figure 6.4**.



```
// File: A.java
package pkg;
class A { B b; } // A uses B
```

```
// File: B.java
package pkg;
class B { }
```

Figure 6.4 *Searching for Classes*

The file hierarchy before any files are compiled is shown in Figure 6.4a. Since the class `B` does not use any other classes, we compile it first with the following command, resulting in the file hierarchy shown in Figure 6.4b:

```
>javac -d ../bin B.java
```

Next, we try to compile the file `A.java`, and we get the following results:

Click here to view code image

```
>javac -d ../bin A.java
A.java:3: cannot find symbol
symbol  : class B
location: class pkg.A
public class A { B b; }
                 ^
1 error
```

The compiler cannot find the class `B`—that is, the file `B.class` containing the Java bytecode for the class `B`. In Figure 6.4b, we can see that it is in the package `pkg` under the directory `bin`, but the compiler cannot find it. This is hardly surprising, as there is no bytecode file for the class `B` in the current directory, which is the default value of the class path. The following command sets the value of the class path to be `/top/bin`, and compilation is successful (Figure 6.4c):

Click here to view code image

```
>javac -classpath /top/bin -d ../bin A.java
```

It is very important to understand that when we want the JDK tool to search in a *named package*, it is the *location* or the *root* of the package that is specified; in other words, the class path indicates the directory that *contains* the first component of the fully qualified package name. In Figure 6.4c, the package `pkg` is contained under the directory whose absolute path is `/top/bin`. The following command will *not* work, as the directory `/top/bin/pkg` does *not* contain a package with the name `pkg` that has a class `B`:

Click here to view code image

```
>javac -classpath /top/bin/pkg -d ../bin A.java
```

Also, the compiler is *not* using the class path to find the source file(s) that are specified in the command line. In the preceding command, the source file has the relative pathname `./A.java`. Consequently, the compiler looks for the source file in the current directory. The class path is used to find the classes used by the class `A`, in this case, to find class `B`.

Given the file hierarchy in Figure 6.3, the following `-classpath` option sets the class path so that *all* packages (`wizard.pandorasbox`, `wizard.pandorasbox.artifacts`, `wizard.spells`) in Figure 6.3 will be searched, as all packages are located under the specified directory:

```
-classpath /pgjc/work
```

However, the following `-classpath` option will not help in finding *any* of the packages in Figure 6.3, as none of the *packages* are located under the specified directory:

Click here to view code image

```
>java -classpath /pgjc/work/wizard pandorasbox.Clown
```

This command also illustrates an important point about package names: The *fully qualified package name* should not be split. The package name for the class `wizard.pandorasbox.Clown` is `wizard.pandorasbox`, and must be specified fully. The following command will search all packages in Figure 6.3 for classes that are used by the class `wizard.pandorasbox.Clown`:

Click here to view code image

```
>java -classpath /pgjc/work wizard.pandorasbox.Clown
```

The class path can specify several *entries* (i.e., several locations), and the JDK tool searches them in the order they are specified, from left to right.

Click here to view code image

```
-classpath /pgjc/work:/top/bin:.
```

We have used the path-separator character `':'` for Unix-based platforms to separate the entries, and also included the current directory (`.`) as an entry. There should be no whitespace on either side of the path-separator character. On the Windows platform, the path-separator character is a semicolon (`;`).

The search in the class path entries stops once the required class file is found. Therefore, the order in which entries are specified can be significant. If a class `B` is found in a package `pkg` located under the directory `/ext/lib1`, and also in a package `pkg` located under the directory `/ext/lib2`, the order in which the entries are specified in the two `-classpath` options shown below is significant. They will result in the class `pkg.B` being found under `/ext/lib1` and `/ext/lib2`, respectively.

Click here to view code image

```
-classpath /ext/lib1:/ext/lib2
-classpath /ext/lib2:/ext/lib1
```

The examples so far have used absolute pathnames for class path entries. We can, of course, use relative pathnames as well. If the current directory has the absolute pathname `/pgjc/work` in **Figure 6.3**, the following command will search the packages under the current directory:

Click here to view code image

```
>java -classpath . wizard.pandorasbox.Clown
```

If the current directory has the absolute pathname `/top/src` in **Figure 6.4**, the following command will compile the file `./A.java`:

Click here to view code image

```
>javac -classpath ../bin -d ../bin A.java
```

If the name of an entry in the class path includes whitespace, the name should be double-quoted so that it will be interpreted correctly:

```
-classpath "../new bin"
```

Table 6.1 summarizes the commands and options to compile and execute non-modular code. The tool documentation from Oracle provides more details on how to use the JDK development tools.

Table 6.1 *Compiling and Executing Non-Modular Java Code*

Operation	Command
Compiling non-modular code:	`javac --class-path` *classpath* `-d` *directory* *sourceFiles* `javac  -classpath` *classpath* `-d` *directory* *sourceFiles* `javac  -cp` *classpath* `-d` *directory* *sourceFiles*
Executing non-modular code:	`java --class-path` *classpath qualifiedClassName* `java  -classpath` *classpath qualifiedClassName* `java  -cp` *classpath qualifiedClassName*

Review Questions

6.1 Which of the following statements are true about the following code in a source file?

Click here to view code image

```
package net.alphabet;
import java.util.ArrayList;
public class A {}
class B {}
```

Select the two correct answers.

a. Both class `A` and class `B` will be placed in the package `net.alphabet`.

b. Only class `A` will be placed in the package `net.alphabet`. Class `B` will be placed in the default package.

c. Both class `A` and class `B` can access the imported class `java.util.ArrayList` by its simple name.

d. Only class `A` can access the imported class `java.util.ArrayList` by its simple name.

6.2 Given the following code:

Click here to view code image

```
package app;
public class Window {
  final static String frame = "Top-frame";
}
```

Click here to view code image

```
package app;
// (1) INSERT IMPORT STATEMENT HERE
public class Canvas {
  private String str = frame;
}
```

Which import statement, when inserted at (1), will make the code compile?

Select the one correct answer.

a. `import app.*;`

b. `import app.Window;`

c. `import java.lang.*;`

d. `import java.lang.String;`

e. `import static app.Window.frame;`

6.3 Given the following code:

Click here to view code image

```
package mainpkg.subpkg1;
public class Window {}
```

Click here to view code image

```
package mainpkg.subpkg2;
public class Window {}
```

Click here to view code image

```
package mainpkg;
// (1) INSERT IMPORT STATEMENTS HERE
public class Screen {
  private Window win;
}
```

Which import statement, when inserted independently at (1), will make the code compile?

Select the four correct answers.

a. `import mainpkg.*;`

b. `import mainpkg.subpkg1.*;`

c. `import mainpkg.subpkg2.*;`

d.

Click here to view code image

```
import mainpkg.subpkg1.*;
import mainpkg.subpkg2.Window;
```

e.

Click here to view code image

```
import mainpkg.subpkg1.Window;
import mainpkg.subpkg2.*;
```

f.

Click here to view code image

```
import mainpkg.subpkg1.*;
import mainpkg.subpkg2.*;
```

g.

Click here to view code image

```
import mainpkg.subpkg1.Window;
import mainpkg.subpkg2.Window;
```

6.4 Given the following code:

Click here to view code image

```
// (1) INSERT ONE IMPORT STATEMENT HERE
public class RQ700A20 {
  public static void main(String[] args) {

    System.out.println(sqrt(49));
  }
}
```

Which import statement, when inserted independently at (1), will make the program print `7` when the program is compiled and run?

Select the two correct answers.

a. `import static Math.*;`

b. `import static Math.sqrt;`

c. `import static java.lang.Math.sqrt;`

d. `import static java.lang.Math.sqrt();`

e. `import static java.lang.Math.*;`

6.5 Given the following directory structure:

Click here to view code image

```
/top
  |--- wrk
        |--- pkg
              |--- A.java
              |--- B.java
```

assume that the two files `A.java` and `B.java` contain the following code, respectively:

```
package pkg;
class A { B b; }
```

```
package pkg;
class B {}
```

For which combinations of current directory and command is the compilation successful?

Select the two correct answers.

a.

Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp .:pkg A.java
```

b.

Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp . pkg/A.java
```

c.

Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp pkg A.java
```

d.

Click here to view code image

```
Current directory: /top/wrk
Command: javac -cp .:pkg pkg/A.java
```

e.

Click here to view code image

```
Current directory: /top/wrk/pkg
Command: javac A.java
```

f.

Click here to view code image

```
Current directory: /top/wrk/pkg
Command: javac -cp . A.java
```

6.6 Given the following code:

Click here to view code image

```
package a.b;
public class X {
  public static int y = 100;
}
```

Click here to view code image

```
package a.b.c;
// (1) INSERT IMPORT STATEMENT HERE
public class Z {
  public void xyz() {

    int v = y;
    }
  }
```

Which import statement, when inserted individually at (1), will allow class `Z` to compile?

Select the two correct answers.

a. `import a.b.*;`

b. `import a.b.X.*;`

c. `import static a.b.*;`

d. `import static a.b.X.*;`

e. `import static a.b.X.y;`

6.7 Given the following code:

Click here to view code image

```
package life.animals;
public class Cat { }
```

Click here to view code image

```
package life.animals;
public class Cow { }
```

Click here to view code image

```
package life.animals;
public class Dog { }
```

Click here to view code image

```
package habitat;
// (1) INSERT IMPORT STATEMENTS HERE
public class Farm {
  private Cat cat;
```

```
    private Cow cow;
  }
```

Which import statements, when inserted individually at (1), will allow class `Farm` to compile?

Select the two correct answers.

a. `import life.animals.*;`

b. `import static life.animals.*;`

Click here to view code image

```
import static life.animals.Cat;
import static life.animals.Cow;
```

c. `import life.animals.Cat;`

d. `import life.animals.Cow;`

6.8 Which statement is true about `import` statements?

Select the one correct answer.

a. Import of a package also imports all subpackages.

b. Import of a package with a wildcard also imports all subpackages.

c. Import statements are required to access code in other packages.

d. All import statements are removed by the compiler.

6.5 Access Modifiers

In this section, we discuss accessibility of top-level type declarations that can be encapsulated into packages and accessibility of members that can be encapsulated in a top-level type declaration. A top-level reference type is a reference type (class, interface, enum, record) that is not declared inside another reference type.

Access modifiers are sometimes also called *visibility modifiers*.

Access Modifiers for Top-Level Type Declarations

The access modifier `public` can be used to declare top-level reference types that are accessible from everywhere, both from inside their own package and from inside other packages. If the access modifier is omitted, the reference types can be accessed only in their own package and not in any other packages—that is, they have *package access*, also called *package-private* or *default access*.

The packages shown in **Figure 6.2**, **p. 327**, are implemented by the code in **Example 6.7**. Class files with Java bytecode for top-level type declarations are placed in designated packages using the `package` statement. A top-level type declaration from one package can be accessed in another packages either by using the *fully qualified name* of the type or by using an `import` statement to import the type so that it can be accessed by its *simple name*.

Example 6.7 *Access Modifiers for Top-Level Reference Types*

Click here to view code image

```
// File: Clown.java
package wizard.pandorasbox;                     // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing class Ailment

public class Clown implements Magic {          // (1)
  LovePotion tlc;                              // Class in same package
  Ailment problem;                             // Simple class name
  Clown() {
    tlc = new LovePotion("passion");
    problem = new Ailment("flu");              // Simple class name
  }
  @Override public void levitate()  {          // (2)
    System.out.println("Levitating");
  }
  public void mixPotion()   { System.out.println("Mixing " + tlc); }
  public void healAilment() { System.out.println("Healing " + problem); }

  public static void main(String[] args) {
    Clown joker = new Clown();
    joker.levitate();
    joker.mixPotion();
    joker.healAilment();
  }
}
```

```
interface Magic { void levitate(); }          // (3)
```

Click here to view code image

```
// File: LovePotion.java
package wizard.pandorasbox;                   // Package declaration

public class LovePotion {                     // (4) Accessible outside package
  String potionName;
  public LovePotion(String name) { potionName = name; }
  public String toString()       { return potionName; }
}
```

Click here to view code image

```
// File: Ailment.java
package wizard.pandorasbox.artifacts;         // Package declaration

public class Ailment {                        // Accessible outside package
  String ailmentName;
  public Ailment(String name) { ailmentName = name; }
  public String toString() { return ailmentName; }
}
```

Click here to view code image

```
// File: Baldness.java
package wizard.spells;                        // Package declaration

import wizard.pandorasbox.*;                  // Redundant
import wizard.pandorasbox.artifacts.*;        // Import of subpackage

public class Baldness extends Ailment {       // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;       // Fully qualified name
  LovePotion tlcTwo;                          // Class in same package
  Baldness(String name) {
    super(name);
    tlcOne = new wizard.pandorasbox.          // Fully qualified name
                 LovePotion("romance");
    tlcTwo = new LovePotion();                // Class in same package
  }
}

class LovePotion /* implements Magic */ {     // (5) Magic is not accessible
```

```
  // @Override public void levitate() {}     // (6) Cannot override method
}
```

Compiling and running the program from the current directory gives the following results:

Click here to view code image

```
>javac -d . Clown.java LovePotion.java Ailment.java Baldness.java
>java wizard.pandorasbox.Clown
Levitating
Mixing passion
Healing flu
```

In Example 6.7, `t`he class `Clown` at (1) and the interface `Magic` at (3) are placed in a package called `wizard.pandorasbox`. The `public` class `Clown` is accessible from everywhere. The `Magic` interface has package accessibility, and can only be accessed within the package `wizard.pandorasbox`. It is not accessible from other packages, not even from subpackages.

The class `LovePotion` at (4) is also placed in the package called `wizard.pandorasbox`. The class has `public` accessibility, and is therefore accessible from other packages. The two files `Clown.java` and `LovePotion.java` demonstrate how several compilation units can be used to group classes in the same package, as the type declarations in these two source files are placed in the package `wizard.pandorasbox`.

In the file `Clown.java`, the class `Clown` at (1) implements the interface `Magic` at (3) from the same package. We have used the annotation `@Override` in front of the declaration of the `levitate()` method at (2) so that the compiler can aid in checking that this method is declared correctly as required by the interface `Magic`.

In the file `Baldness.java`, the class `LovePotion` at (5) wishes to implement the interface `Magic` at (3) from the package `wizard.pandorasbox`, but this is not possible, although the source file imports from this package. The reason is that the interface `Magic` has package accessibility, and can therefore only be accessed within the package `wizard.pandorasbox`. The method `levitate()` of the `Magic` interface therefore cannot be overridden in class `LovePotion` at (6).

Table 6.2 summarizes accessibility of top-level reference types in a package. Just because a reference type is accessible does not necessarily mean that members of the type are also accessible. Accessibility of members is governed separately from type accessibility, as explained in the next subsection.

Table 6.2 *Access Modifiers for Top-Level Reference Types (Non-Modular)*

Modifiers	Top-level types
No modifier	Accessible in its own package (*package accessibility*)
`public`	Accessible anywhere

Access Modifiers for Class Members

By specifying member access modifiers, a class can control which information is accessible to clients (i.e., other classes). These modifiers help a class define a *contract* so that clients know exactly which services are offered by the class.

The accessibility of a member in a class can be any one of the following:

- `public`
- `protected`
- *package access* (also known as *package-private* and *default access*), when no access modifier is specified
- `private`

In the following discussion of access modifiers for members of a class, keep in mind that the member access modifier has meaning only if the class (or one of its subclasses) is accessible to the client. Also, note that only one access modifier can be specified for a member.

The discussion in this subsection applies to both instance and static members of top-level classes.

In UML notation, when applied to member names the prefixes `+`, `#`, and `-` indicate `public`, `protected`, and `private` member access, respectively. No access modifier indicates package access for class members.

The package hierarchy shown in Figure 6.5 is implemented by the code in Example 6.8. The class `Superclass1` at (1) in `pkg1` has two subclasses: `Subclass1` at (3) in `pkg1` and `Subclass2` at (5) in `pkg2`. The class `Superclass1` in `pkg1` is used by the other classes (designated as Client 1 to Client 4) in Figure 6.5.

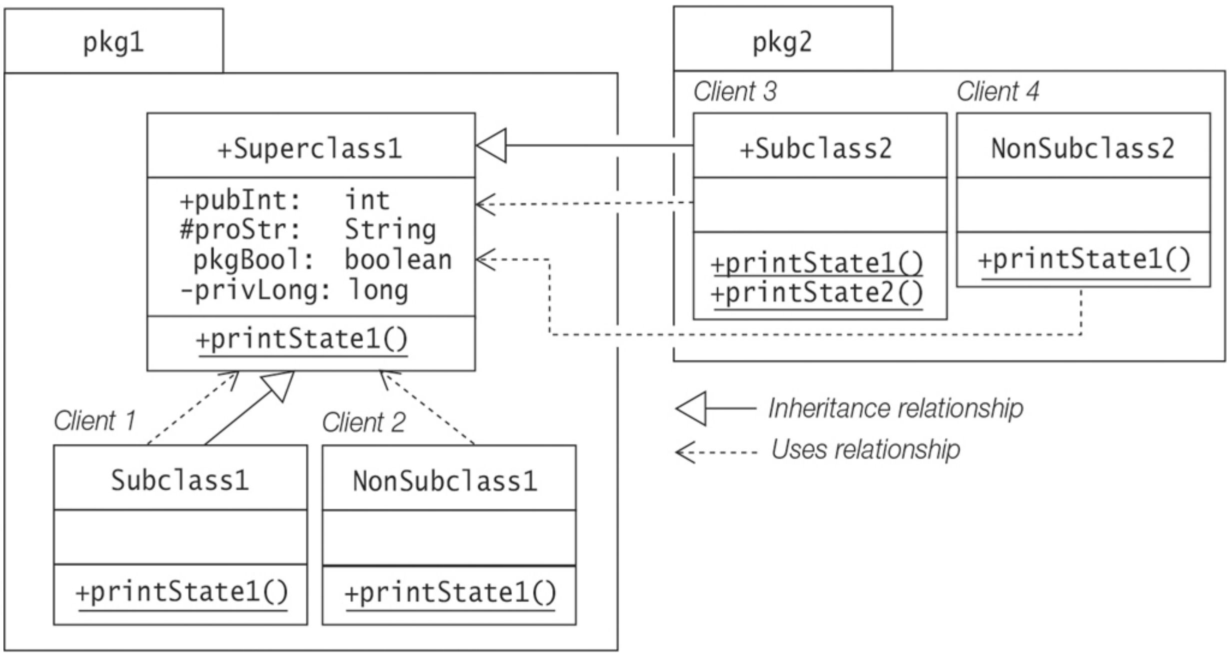


Figure 6.5 *Accessibility of Class Members*

Accessibility of a member is illustrated in Example 6.8 by the four instance fields defined in the class `Superclass1` in `pkg1`, where each instance field has a different accessibility. These four instance fields are accessed in a `Superclass1` object created in the `static` method `printState1()` declared in five different contexts:

- Defining class: The class in which the member is declared—that is, `pkg1.Superclass1` in which the four instance fields being accessed are declared
- Client 1: From a subclass in the same package—that is, `pkg1.Subclass1`
- Client 2: From a non-subclass in the same package—that is, `pkg1.NonSubclass1`
- Client 3: From a subclass in another package—that is, `pkg2.Subclass2`
- Client 4: From a non-subclass in another package—that is, `pkg2.NonSubclass2`

Example 6.8 *Accessibility of Class Members*

Click here to view code image

```
// File: Superclass1.java
package pkg1;

import static java.lang.System.out;

public class Superclass1 {                          // (1)

  // Instance fields with different accessibility:
  public    int     pubInt   = 2017;
  protected String  proStr   = "SuperDude";
            boolean pgkBool  = true;
  private   long    privLong = 0x7777;
  public static void printState1() {              // (2)
    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);
  }
}

// Client 1
class Subclass1 extends Superclass1 {            // (3)
  public static void printState1() {

    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);  // Compile-time error! Private access.
  }
}

// Client 2
class NonSubclass1 {                             // (4)
  public static void printState1() {

    Superclass1 obj1 = new Superclass1();
    out.println(obj1.pubInt);
    out.println(obj1.proStr);
    out.println(obj1.pgkBool);
    out.println(obj1.privLong);  // Compile-time error! Private access.
  }
}
```

Click here to view code image

```
// File: Subclass2.java
package pkg2;
import pkg1.Superclass1;

import static java.lang.System.out;

// Client 3
public class Subclass2 extends Superclass1 {     // (5)

  public static void printState1() {              // (6)
    Superclass1 obj1 = new Superclass1();        // Object of Superclass1
    out.println(obj1.pubInt);
```

```
      out.println(obj1.proStr);   // (7) Compile-time error! Protected access.
      out.println(obj1.pgkBool);  // Compile-time error! Package access.
      out.println(obj1.privLong); // Compile-time error! Private access.
    }

    public static void printState2() {             // (8)
      Subclass2 obj2 = new Subclass2();            // (9) Object of Subclass2
      out.println(obj2.pubInt);
      out.println(obj2.proStr);   // (10) OK! Protected access.
      out.println(obj2.pgkBool);  // Compile-time error! Package access.
      out.println(obj2.privLong); // Compile-time error! Private access.
    }
  }

  // Client 4
  class NonSubclass2 {                               // (11)
    public static void printState1() {
      Superclass1 obj1 = new Superclass1();
      out.println(obj1.pubInt);
      out.println(obj1.proStr);   // Compile-time error! Protected access.
      out.println(obj1.pgkBool);  // Compile-time error! Package access.
      out.println(obj1.privLong); // Compile-time error! Private access.
    }
  }
```

`public` *Members*

Public access is the least restrictive of all the access modifiers. A `public` member is accessible from anywhere, both in the package containing its class and by other packages where its class is accessible.

In **Example 6.8**, the `public` instance field `pubInt` in an instance of the class `Superclass1` is accessible by all four clients. Subclasses can access their inherited `public` members by their simple names, and all clients can access `public` members in an instance of the class `Superclass1`.

`protected` *Members*

A `protected` member is accessible in all classes in the same package, and by all subclasses of its class in any package where its class is accessible.

In other words, a `protected` member cannot be accessed by non-subclasses in other packages. This kind of access is more restrictive than `public` member access.

In Example 6.8, the `protected` instance field `proStr` in an instance of the class `Superclass1` is accessible within `pkg1` by Client 1 and Client 2. Also as expected, Client 4—the class `NonSubclass2` in `pkg2`—cannot access this `protected` member of the class `Superclass1`.

However, the compiler reports an error at (7) in the method `printState1()` of the class `Subclass2` where the `protected` instance field `proStr` in an instance of the class `Superclass1` *cannot* be accessed by the reference `obj1`.

In contrast, the method `printState2()` at (8) in the class `Subclass2` uses the reference `obj2` that refers to an instance of the class `Subclass2` to access the instance fields declared in the class `Superclass1`, and now the `protected` instance field `proStr` in the superclass is accessible, as shown at (10).

This apparent anomaly is explained by the fact that a subclass in another package can only access `protected` instance members in the superclass via references of its own type—that is, `protected` instance members that are *inherited* by objects of the subclass. No inheritance from the superclass is involved in a subclass in another package when an object of the superclass is used.

Note that the above anomaly would *not* arise if a subclass in another package were to access any `protected static` members in the superclass, as such members are not part of any object of the superclass.

Members with Package Access

No access modifier implies package accessibility in this context. When no member access modifier is specified, the member is accessible only by other classes in the same package in which its class is declared. Even if its class is accessible in another package, the member is not accessible elsewhere. Package member access is more restrictive than `protected` member access.

In Example 6.8, the instance field `pkgBool` in an instance of the class `Superclass1` has package access and is only accessible within `pkg1`, but not in any other packages — that is to say, it is accessible only by Client 1 and Client 2. Client 3 and Client 4 in `pkg2` cannot access this field.

`private` *Members*

The `private` modifier is the most restrictive of all the access modifiers. Private members are not accessible by any other classes. This also applies to subclasses, whether they are in the same package or not. Since they are not accessible by their simple names in a subclass, they are also not inherited by the subclass. A standard design strategy for a class is to make all instance fields `private` and provide `public` get methods for such fields. Auxiliary methods are often declared as `private`, as they do not concern any client.

None of the clients in **Figure 6.5** can access the `private` instance field `privLong` in an instance of the class `Superclass1`. This instance field is only accessible in the defining class—that is, in the class `Superclass1`.

Table 6.3 provides a summary of access modifiers for members in a class. References in parentheses refer to clients in **Figure 6.5**.

Table 6.3 *Accessibility of Members in a Class (Non-Modular)*

Member access	In the defining class Superclass1	In a subclass in the same package (Client 1)	In a non-subclass in the same package (Client 2)	In a subclass in a different package (Client 3)	In a non-subclass in a different package (Client 4)
`public`	Yes	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	Yes	No
`package`	Yes	Yes	Yes	No	No
`private`	Yes	No	No	No	No

Additional Remarks on Accessibility

Access modifiers that can be specified for a class member apply equally to *constructors*. However, when no constructor is specified, the default constructor inserted by the compiler implicitly follows the accessibility of the class.

Accessibility of members declared in an enum type is analogous to those declared in a class, except for enum constants that are always `public` and constructors that are always `private`. A `protected` member in an enum type is only accessible in its own package, since an enum type is implicitly `final`. Member declarations in an enum type are discussed in **§5.13**, **p. 290**.

In contrast, the accessibility of members declared in an interface is always implicitly `public` (§5.6, p. 238). Omission of the `public` access modifier in this context does *not* imply package accessibility.

6.6 Scope Rules

Java provides explicit access modifiers to control the accessibility of members in a class by external clients, but in two areas access is governed by specific scope rules:

- Class scope for members: how member declarations are accessed within the class
- Block scope for local variables: how local variable declarations are accessed within a block

Class Scope for Members

Class scope concerns accessing members (including inherited ones) from code within a class. **Table 6.4** gives an overview of how static and non-static code in a class can access members of the class, including those that are inherited. **Table 6.4** assumes the following declarations:

Click here to view code image

```
class SuperClass {
  int instanceVarInSuper;
  static int staticVarInSuper;

  void instanceMethodInSuper()      { /* ... */ }
  static void staticMethodInSuper() { /* ... */ }
  // ...
}

class MyClass extends SuperClass {
  int instanceVar;
  static int staticVar;

  void instanceMethod()      { /* ... */ }
  static void staticMethod() { /* ... */ }
  // ...
}
```

Table 6.4 *Accessing Members within a Class*

Member declarations	Non-static code in the class `MyClass` can refer to the member as	Static code in the class `MyClass` can refer to the member as

Instance variables	`instanceVar`
	`this.instanceVar`
	`instanceVarInSuper`
	`this.instanceVarInSuper`
	`super.instanceVarInSuper`
		Not possible

Instance methods	`instanceMethod()`
	`this.instanceMethod()`
	`instanceMethodInSuper()`
	`this.instanceMethodInSuper()`
	`super.instanceMethodInSuper()`
		Not possible

Static variables	`staticVar`
	`this.staticVar`
	`MyClass.staticVar`
	`staticVarInSuper`
	`this.staticVarInSuper`
	`super.staticVarInSuper`
	`MyClass.staticVarInSuper`
	`SuperClass.staticVarInSuper`
		`staticVar`
		`MyClass.staticVar`
		`staticVarInSuper`
		`MyClass.staticVarInSuper`
		`SuperClass.staticVarInSuper`

Static methods	`staticMethod()`
	`this.staticMethod()`
	`MyClass.staticMethod()`
	`staticMethodInSuper()`
	`this.staticMethodInSuper()`
	`super.staticMethodInSuper()`
	`MyClass.staticMethodInSuper()`
	`SuperClass.staticMethodInSuper()`
		`staticMethod()`
		`MyClass.staticMethod()`
		`staticMethodInSuper()`
		`MyClass.staticMethodInSuper()`
		`SuperClass.staticMethodInSuper()`

The golden rule is that static code can only access other static members by their simple names. Static code is not executed in the context of an object, so the references `this` and `super` are not available. An object has knowledge of its class, so static members are always accessible in a non-static context.

Note that using the class name to access static members within the class is no different from how external clients access these static members.

The following factors can all influence the scope of a member declaration:

- Shadowing of a field declaration, either by local variables (p. 354) or by declarations in the subclass (§5.1, p. 203)
- Overriding an instance method from a superclass (§5.1, p. 196)
- Hiding a static method declared in a superclass (§5.1, p. 203)

Within a class, references of the class can be declared and used to access *all* members in the class, regardless of their access modifiers. In **Example 6.9**, the method `duplicateLight` at (1) in the class `Light` has the parameter `oldLight` and the local variable `newLight` that are references of the class `Light`. Even though the fields of the class are `private,` they are accessible through the two references (`oldLight` and `newLight`) in the method `duplicateLight()`, as shown at (2), (3), and (4).

Example 6.9 *Class Scope*

Click here to view code image

```
class Light {
  // Instance variables:
  private int     noOfWatts;       // Wattage
  private boolean indicator;       // On or off
  private String  location;        // Placement

  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  public static Light duplicateLight(Light oldLight) {     // (1)
    Light newLight = new Light();
    newLight.noOfWatts = oldLight.noOfWatts;               // (2)
    newLight.indicator = oldLight.indicator;               // (3)
    newLight.location  = oldLight.location;                // (4)
    return newLight;
  }
}
```

Block Scope for Local Variables

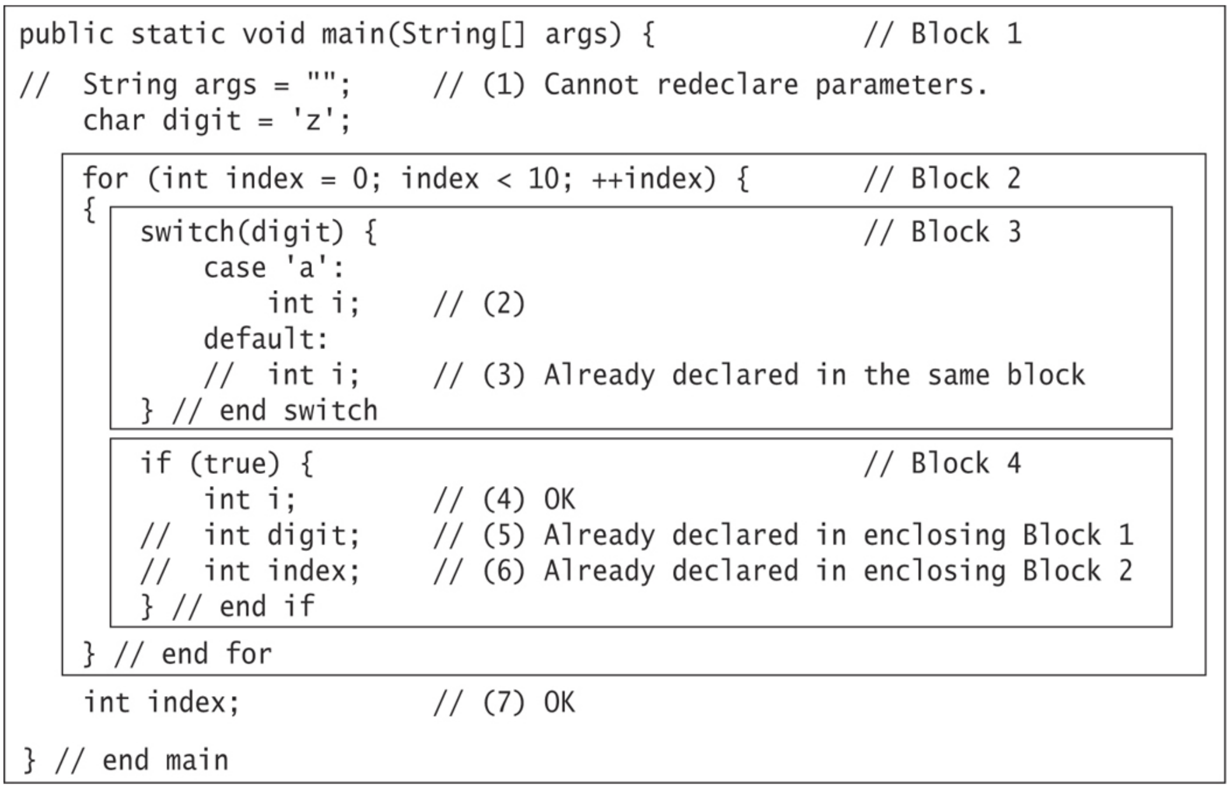
Declarations and statements can be grouped into a *block* using curly brackets, `{}`. Blocks can be nested, and scope rules apply to local variable declarations in such blocks. A local declaration can appear anywhere in a block. The general rule is that a variable declared in a block is *in scope* in the block in which it is declared, but it is not accessible outside this block. It is not possible to redeclare a variable if a local variable of the same name is already declared in the current scope.

Local variables of a method include the formal parameters of the method and variables that are declared in the method body. The local variables in a method are created each time the method is invoked, and are therefore distinct from local variables in other invocations of the same method that might be executing (§7.1, p. 365).

Figure 6.6 illustrates *block scope* (also known as *lexical scope*) for local variables. It shows four blocks: Block 1 is the body of the method `main()`, Block 2 is the body of the `for(;;)` loop, Block 3 is the body of a `switch` statement, and Block 4 is the body of an `if` statement.

- Parameters cannot be redeclared in the method body, as shown at (1) in Block 1.
- A local variable—already declared in an enclosing block, and therefore visible in a nested block—cannot be redeclared in the nested block. These cases are shown at (3), (5), and (6).
- A local variable in a block can be redeclared in another block if the blocks are *disjoint*—that is, they do not overlap. This is the case for variable `i` at (2) in Block 3 and at (4) in Block 4, as these two blocks are disjoint.

The scope of a local variable declaration begins from where it is declared in the block and ends where this block terminates. The scope of the loop variable `index` is the entire Block 2. Even though Block 2 is nested in Block 1, the declaration of the variable `index` at (7) in Block 1 is valid. The scope of the variable `index` at (7) spans from its declaration to the end of Block 1, and it does not overlap with that of the loop variable `index` in Block 2.



```
public static void main(String[] args) {                // Block 1
//  String args = "";     // (1) Cannot redeclare parameters.
    char digit = 'z';

    for (int index = 0; index < 10; ++index) {          // Block 2
    {
        switch(digit) {                                 // Block 3
            case 'a':
                int i;     // (2)
            default:
            //  int i;     // (3) Already declared in the same block
        } // end switch

        if (true) {                                     // Block 4
            int i;         // (4) OK
        //  int digit;     // (5) Already declared in enclosing Block 1
        //  int index;     // (6) Already declared in enclosing Block 2
        } // end if
    } // end for

    int index;             // (7) OK

} // end main
```

Figure 6.6 *Block Scope*

6.7 Implementing Immutability

Shared resources are typically implemented using `synchronized` code in order to guarantee thread safety of the shared resource (§22.4, p. 1387). However, if the shared resource is an immutable object, thread safety comes for free.

An object is *immutable* if its state cannot be changed once it has been constructed. Since its state can only be read, there can be no thread interference and the state is always consistent.

Some examples of immutable classes from the Java SE Platform API are listed in Table 6.5. Any method that seemingly modifies the state of an immutable object is in fact returning a new immutable object with the state modified appropriately based on the original object. Primitive values are of course always immutable.

Table 6.5 *Examples of Immutable Classes*

`java.lang.String`	This class implements immutable strings (§8.4, p. 439).
Wrapper classes in the `java.lang` package: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`	Wrapper classes create immutable objects that wrap a value of their re-

	spective primitive type (§8.3, p. 429).
`java.nio.file.Path`	Objects of classes that implement this interface are immutable and represent a system-dependent file path (§21.2, p. 1289).
Temporal classes in the `java.time` package: `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration`, `ZonedDateTime`	These temporal classes create immutable objects that represent date-/time-based values (Chapter 17, p. 1023).
`java.time.format.DateTimeFormatter`	This class creates immutable objects with customized formatting and parsing capabilities for date-/time-based values (§18.6, p. 1134).
`java.util.Locale`	Immutable objects of this class represent a specific geographical, political, or cultural region (§18.1, p. 1096).

Example 6.10 *Implementing an Immutable Class*

Click here to view code image

```
import java.util.Arrays;

public final class WeeklyStats {                 // (1) Class is final.

  private final String description;              // (2) Immutable string
  private final int weekNumber;                  // (3) Immutable primitive value
  private final int[] stats;                     // (4) Mutable int array

  public WeeklyStats(String description, int weekNumber, int[] stats) {   // (5)
    if (weekNumber <= 0 || weekNumber > 52) {
      throw new IllegalArgumentException("Invalid week number: " + weekNumber);
    }
    if (stats.length != 7) {
      throw new IllegalArgumentException("Stats not for whole week: " +
                                          Arrays.toString(stats));
    }
    this.description = description;
    this.weekNumber = weekNumber;
    this.stats = Arrays.copyOf(stats, stats.length);   // Create a private copy.
  }

  public int getWeekNumber() {                    // (6) Returns immutable primitive.
    return weekNumber;
  }

  public String getDescription() {                // (7) Returns immutable string.
    return description;
  }

  public int getDayStats(int dayNumber) {         // (8) Returns stats for given day.
    return (0 <= dayNumber && dayNumber < 7) ? stats[dayNumber] : -1;
  }

  public int[] getStats() {                       // (9) Returns a copy of the stats.
    return Arrays.copyOf(this.stats, this.stats.length);
  }

  @Override
  public String toString() {
    return description + "(week " + weekNumber + "):" + Arrays.toString(stats);
  }
}
```

Click here to view code image

```
public class StatsClient {
  public static void main(String[] args) {
    WeeklyStats ws1
        = new WeeklyStats("Appointments", 45, new int[] {5, 3, 8, 10, 7, 8, 9});
    System.out.println(ws1);
    WeeklyStats ws2
        = new WeeklyStats("E-mails", 47, new int[] {10, 5, 20, 7});
    System.out.println(ws2);
  }
}
```

Output from the program:

Click here to view code image

```
Appointments(week 45):[5, 3, 8, 10, 7, 8, 9]
Exception in thread "main" java.lang.IllegalArgumentException: Stats not for whole
week: [10, 5, 20, 7]
      at WeeklyStats.<init>(WeeklyStats.java:14)
      at StatsClient.main(StatsClient.java:7)
```

There are certain guidelines that can help to avoid common pitfalls when implementing immutable classes. We will illustrate implementing an immutable class called `WeeklyStats` in **Example 6.10**, whose instances, once created, cannot be modified. The class `WeeklyStats` creates an object with weekly statistics of a specified entity.

- *It should not be possible to extend the class.*
  Caution should be exercised in extending an immutable class to prevent any subclass from subverting the immutable nature of the superclass.
  A straightforward approach is to declare the class as `final`, as was done in **Example 6.10** at (1) for the class `WeeklyStats`. Another approach is to declare the constructor as `private` and provide static factory methods to construct instances (discussed below). A static factory method is a static method whose sole purpose is to construct and return a new instance of the class—an alternative to calling the constructor directly.
- *All fields should be declared* `final` *and* `private`.
  Declaring the fields as `private` makes them accessible only inside the class, and other clients cannot access and modify them. This is the case for the fields in the `WeeklyStats` class at (2), (3), and (4).
  Declaring a field as `final` means the value stored in the field cannot be changed once initialized. However, if the `final` field is a reference to an object, the state of this object can be changed by other clients who might be sharing this object, unless the object is also immutable. See the last guideline on how to safeguard the state of an mutable object referenced by a field.
- *Check the consistency of the object state at the time the object is created.*
  Since it is not possible to change the state of an immutable object, the state should be checked for consistency when the object is created. If all relevant information to initialize the object is available when it is created, the state can be checked for consistency and any necessary measures taken. For example, a suitable exception can be thrown to signal illegal arguments.
  In the class `WeeklyStats`, the constructor at (5) is passed all the necessary values to initialize the object, and it checks whether they will result in a legal and consistent state for the object.
- *No set methods (a.k.a. setter or mutator methods) should be provided.*

Set methods that change values in fields or objects referenced by fields should not be permitted. The class `WeeklyStats` does not have any set methods, and only provides get methods (a.k.a. *getter* or *assessor methods*).

If a setter method is necessary, then the method should create a new instance of the class based on the modified state, and return that to the client, leaving the original instance unmodified. This approach has to be weighed against the cost of creating new instances, but is usually offset by other advantages associated with using immutable classes, like thread safety without synchronized code. Caching frequently used objects can alleviate some overhead of creating new objects, as exemplified by the immutable wrapper classes for primitive types. For example, the `Boolean` class has a static factory method `valueOf()` that always returns one of two objects, `Boolean.TRUE` or `Boolean.FALSE`, depending on whether its `boolean` argument was `true` or `false`, respectively. The `Integer` class interns values between –128 and 127 for efficiency so that there is only one `Integer` object to represent each `int` value in this range.

- *A client should not be able to access mutable objects referred to by any fields in the class.*

  The class should not provide any methods that can modify its mutable objects. The class `WeeklyStats` complies with this requirement.

  A class should also not share references to its mutable objects. The field at (4) has the type array of `int` that is mutable. An `int` array is passed as a parameter to the constructor at (5). The constructor in this case makes its own copy of this `int` array, so as not to share the array passed as an argument by the client. The `getWeeklyStats()` method at (8) does not return the reference value of the `int` array stored in the field `stats`. It creates and returns a new `int` array with values copied from its private `int` array. This technique is known as *defensive copying*. This way, the class avoids sharing references of its mutable objects with clients.

The class declaration below illustrates another approach to prevent a class from being extended. The class `WeeklyStats` is no longer declared `final` at (1), but now has a `private` constructor. This constructor at (5a) cannot be called by any client of the class to create an object. Instead, the class provides a static factory method at (5b) that creates an object by calling the `private` constructor. No subclass can be instantiated, as the superclass `private` constructor cannot be called, neither directly nor implicitly, in a subclass constructor.

Click here to view code image

```
public class WeeklyStatsV2 {                      // (1) Class is not final.
  ...
  private WeeklyStatsV2(String description,
      int weekNumber, int[] stats) {              // (5a) Private constructor
    this.description = description;
    this.weekNumber = weekNumber;
    this.stats = Arrays.copyOf(stats, stats.length); // Create a private copy.
  }

  // (5b) Static factory method to construct objects.
  public static WeeklyStatsV2 getNewWeeklyStats(String description,
                                                int weekNumber, int[] stats) {
    if (weekNumber <= 0 || weekNumber > 52) {
      throw new IllegalArgumentException("Invalid week number: " + weekNumber);
    }
    if (stats.length != 7) {
      throw new IllegalArgumentException("Stats not for whole week: " +
                                         Arrays.toString(stats));
    }
    return new WeeklyStatsV2(description, weekNumber, stats);
  }
  ...
}
```

A class having just static methods is referred to as a *utility class*. Such a class cannot be instantiated and has no state, and is thus immutable. Examples of such classes in the Java API include the following: the `java.lang.Math` class, the `java.util.Collections` class, the `java.util.Arrays` class, and the `java.util.concurrent.Executors` class.

Apart from being thread safe, immutable objects have many other advantages. Once created, their state is guaranteed to be consistent throughout their lifetime. That makes them easy to reason about. Immutable classes are relatively simple to construct, amenable to testing, and easy to use compared to mutable classes. There is hardly any need to make or provide provisions for making copies of such objects. Their hash code value, once computed, can be cached for later use, as it will never change. Because of their immutable state, they are ideal candidates for keys in maps, and as elements in sets. They also are ideal building blocks for new and more complex objects.

A downside of using an immutable object is that if a value must be changed in its state, then a new object must be created, which can be costly if object construction is expensive.

Review Questions

6.9 Which of the following statements are true about the use of modifiers? Select the two correct answers.

a. If no access modifier (`public`, `protected`, or `private`) is specified for a member declaration, the member is accessible only by classes in the package of its class and by subclasses of its class in any package.

b. You cannot specify accessibility of local variables. They are not accessible outside the block in which they are declared.

c. Subclasses of a class must reside in the same package as the class they extend.

d. Local variables can be declared as `static`.

e. The objects themselves do not have any access modifiers; only field references do.

6.10 Given the following declaration of a class, which field is accessible from outside the package `com.corporation.project`?

Click here to view code image

```
package com.corporation.project;
public class MyClass {
            int i;
  public    int j;
  protected int k;
  private   int l;
}
```

Select the one correct answer.

a. Field `i` is accessible in all classes in other packages.

b. Field `j` is accessible in all classes in other packages.

c. Field `k` is accessible in all classes in other packages.

d. Field `k` is accessible in subclasses only in other packages.

e. Field `l` is accessible in all classes in other packages.

f. Field `l` is accessible in subclasses only in other packages.

6.11 Which statement is true about the accessibility of members?

Select the one correct answer.

a. A private member is always accessible within the same package.

b. A private member can be accessed only in the class in which it is declared.

c. A member with package access can be accessed by any subclass of the class in which it is declared.

d. A private member cannot be accessed at all.

e. Package accessibility for a member can be declared using the keyword `default`.

6.12 Which statement is true about immutability?

Select the one correct answer.

a. Instances of a `final` class are immutable.

b. Instances of a `static` class are immutable.

c. All members of an immutable class are also immutable.

d. None of the above

6.13 Which code modifications would make class `Dog` immutable?

Click here to view code image

```
public class Dog {
  String name;
  public Dog(String name) {
    this.name = name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}
```

Select the one correct answer.

a. Mark the field `name` as `private` and remove the `setName()` method.

b. Mark the field `name` as `final` and make the `setName()` method `private`.

c. Mark the field `name` as `final` and initialize it in the declaration, then remove the `setName()` method.

d. None of the above