Access Control

4

Exam Objectives

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of `package` and `import` statements (including static imports).
- ❍ *The package and import statements are covered in this chapter.*
- ❍ *For class declarations, see Section 3.1, p. 40.*
- ❍ *For abstract classes, see Section 4.8, p. 135.*
- ❍ *For nested classes, see Chapter 8, p. 351.*
- ❍ *For interfaces, see Section 7.6, p. 309.*
- ❍ *For enums, see Section 3.5, p. 54.*

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

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

Supplementary Objectives

- Creating JAR files.
- Using system properties.

4.1 Java Source File Structure

The structure of a skeletal Java source file is depicted in Figure 4.1. 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 Section 4.2.
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` statements are discussed in Section 4.2.
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 declaration, but that is hardly useful.

   The JDK imposes the restriction that at the 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. If the `public` class name is `NewApp`, the file name must be `NewApp.java`.

   Classes are discussed in Section 3.1, p. 40, and interfaces are discussed in Section 7.6, p. 309.

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

Figure 4.1 *Java Source File Structure*

```
// Filename: 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 class and interface declarations
public class NewApp { }

class A { }

interface IX { }

class B { }

interface IY { }

enum C { FIRST, SECOND, THIRD }

// end of file
```

4.2 Packages

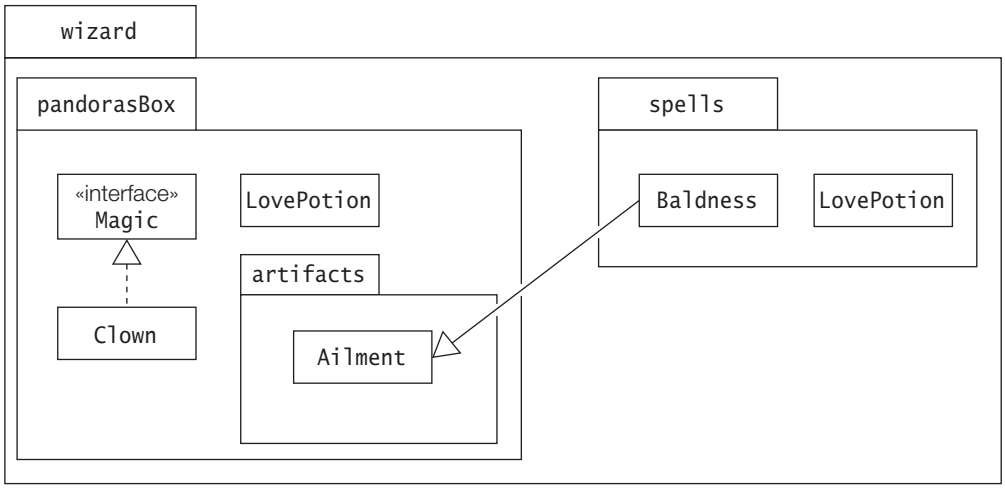
A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces, enums, and subpackages.

Figure 4.2 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` is different from the class `wizard.spells.LovePotion`. The `Ailment` class can be easily identified by the name `wizard.pandorasBox.artifacts.Ailment`. This is called 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`. Analogously, the fully qualified name of a *subpackage* comprises the fully qualified name of the parent package and the simple name of the subpackage.

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 4.2 *Package Hierarchy*



Conventionally, a global naming scheme based on the Internet domain names is used to uniquely identify packages. If the above 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.

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 rather than a language feature. Accessibility of members in a package is discussed in Section 4.6. Accessibility of members defined in type declarations is discussed in Section 4.9.

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 byte code for each of them. A type declaration can indicate that its Java byte code be placed in a particular package, using a `package` declaration.

The `package` statement has the following syntax:

```
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 byte code for the types contained in the package.

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 byte code 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 4.1 illustrates how the packages in Figure 4.2 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 4.10 on page 133.

Example 4.1 *Defining Packages and Using Type Import*

```
//File: Clown.java
package wizard.pandorasBox;                  // (1) Package declaration

import wizard.pandorasBox.artifacts.Ailment; // (2) Importing class

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

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

```
//File: LovePotion.java
package wizard.pandorasBox;                  // (1) Package declaration

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

```
//File: Ailment.java
package wizard.pandorasBox.artifacts;        // (1) Package declaration

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

```
//File: Baldness.java
package wizard.spells;                       // (1)Package declaration

import wizard.pandorasBox.*;                 // (2) Type-import-on-demand
import wizard.pandorasBox.artifacts.*;       // (3) Import from subpackage

public class Baldness extends Ailment {      // (4) Abbreviated name for Ailment
  wizard.pandorasBox.LovePotion tlcOne;      // (5) Fully qualified name
  LovePotion tlcTwo;                         // (6) 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 and interfaces) 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:

```
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 (that is, its identifier) can now be used to access this particular type. Given the following `import` declaration:

```
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:

```
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.

An `import` declaration does not recursively import subpackages. The declaration also does not result in inclusion of the source code of the types. The declaration only imports type names (that is, it makes type names available to the code in a compilation unit).

All compilation units implicitly import the `java.lang` package (see Section 10.1, p. 424). 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.

Example 4.1 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`, respectively. 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 4.1 at (5). Such name conflicts can usually be resolved by using variations of the `import` statement 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 (3) 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` as shorthand for the `java.awt.List` type as expected:

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

Given the following two `import` declarations:

```
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 as both the types `java.util.List` and `java.awt.List` match.

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

```
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*.

Static import allows accessible static members (static fields, static methods, static member classes, enum, and interfaces) declared in a type to be imported, so that they can be used by their simple name, and therefore need not be qualified. The import applies to the whole compilation unit, and importing from the unnamed package is not permissible.

The two forms of static import are shown below:

```
// 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 `static`. 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 4.2. 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 qualifying 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 4.2 by the following `import` statement:

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

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

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

Example 4.2 *Single Static Import*

```
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 CalculateI {
  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:

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

Using static import avoids the *interface constant antipattern,* as illustrated in Example 4.3. 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 in order to access the constants by their simple name:

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

Example 4.3 *Avoiding the Interface Constant Antipattern*

```
package mypkg;

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

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

public class MyFactory {
  public static void main(String[] args) {
    int[] states = { IDLE, BUSY, IDLE, BLOCKED };
    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. Example 4.4 combines type and static import. The enum constants can be accessed at (4) 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 (5).

Example 4.4 *Importing Enum Constants*

```
package mypkg;

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

```
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 = {
        IDLE, BUSY, IDLE, BLOCKED    // (4) Using static import implied by (2).
    };
    for (State s : states)           // (5) Using type import implied by (1).
      out.print(s + " ");            // (6) 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 4.5 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 is `PrintWriter`, and that of the statically imported field is `PrintStream`. Both classes `PrintStream` and `PrintWriter` define the method `println()` that is called in the program. The only way to access the imported field in the method `writeInfo()` is to use its fully qualified name.

Example 4.5 *Shadowing by Importing*

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

import java.io.FileNotFoundException;
import java.io.PrintWriter;               // (2) Single type import

public class ShadowingByImporting {

  public static void main(String[] args) throws FileNotFoundException {
    out.println("Calling println() in java.lang.System.out");
    PrintWriter pw = new PrintWriter("log.txt");
    writeInfo(pw);
    pw.flush();
    pw.close();
  }

  public static void writeInfo(PrintWriter out) { // Shadows java.lang.System.out
    out.println("Calling println() in the parameter out");
    System.out.println("Calling println() in java.lang.System.out"); // Qualify
  }
}
```

Output from the program:

```
Calling println() in java.lang.System.out
Calling println() in java.lang.System.out
```

Contents of the file `log.txt`:

```
Calling println() in the parameter out
```

Conflicts can also occur when a static method with the same signature is imported by *several* static import statements. In Example 4.6, 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* that matches the method call at (2), resulting in a signature conflict. 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 4.6 *Conflict in Importing Static Method with the Same Signature*

```
package mypkg;

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

```
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.

class MultipleStaticImport {
  public static void main(String[] args) {
    int index = binarySearch(new int[] {10, 50, 100}, 50); // (2) Not ok!
    System.out.println(index);
  }

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

Example 4.6 illustrates importing nested static types (Section 8.2, p. 355). The class `yap.Machine` declares three static members, which all are *types*. Since these nested members are types that are static, they can be imported both as types *and* as static members. The class `MachineClient` uses the static types declared in the `yap.Machine` class. The program shows how the import statements influence which types and members are accessible. The following statement in the `main()` method declared at (10) does not compile:

```
String s1 = IDLE;                          // Ambiguous because of (3) and (6)
```

because the constant `IDLE` is imported from both the static class `StateConstant` and the enum type `MachineState` by the following import statements:

```
import static yap.Machine.StateConstant.*;    // (3)
...
import static yap.Machine.MachineState.*;     // (6)
```

Similarly, the following statement in the `main()` method is also not permitted:

```
MachineState ms1 = BLOCKED;                // Ambiguous because of (3) and (6)
```

The conflicts are resolved by qualifying the member just enough to make the names unambiguous.

Example 4.7 *Importing Nested Static Types*

```
package yap;                                    // yet another package

public class Machine {                          // Class with 3 nested types

  public static class StateConstant {           // A static member class
    public static final String BUSY = "Busy";
    public static final String IDLE = "Idle";
    public static final String BLOCKED = "Blocked";
  }

  public enum MachineState {                    // A nested enum is static.
    BUSY, IDLE, BLOCKED
  }

  public enum AuxMachineState {                 // Another static enum
    UNDER_REPAIR, WRITE_OFF, HIRED, AVAILABLE;
  }
}
```

```
import yap.Machine;                             // (0)

import yap.Machine.StateConstant;               // (1)
import static yap.Machine.StateConstant;        // (2) Superfluous because of (1)
import static yap.Machine.StateConstant.*;      // (3)

import yap.Machine.MachineState;                // (4)
import static yap.Machine.MachineState;         // (5) Superfluous because of (4)
import static yap.Machine.MachineState.*;       // (6)

import yap.Machine.AuxMachineState;             // (7)
import static yap.Machine.AuxMachineState;      // (8) Superfluous because of (7)
import static yap.Machine.AuxMachineState.*;    // (9)
import static yap.Machine.AuxMachineState.WRITE_OFF;  // (10)

public class MachineClient {
  public static void main(String[] args) {     // (10)

    StateConstant msc = new StateConstant();    // Requires (1) or (2)
  //String s1 = IDLE;                           // Ambiguous because of (3) and (6)
    String s2 = StateConstant.IDLE;           // Explicit disambiguation necessary.

  //MachineState ms1 = BLOCKED;                 // Ambiguous because of (3) and (6)
    MachineState ms2 = MachineState.BLOCKED;    // Requires (4) or (5)
    MachineState ms3 = MachineState.IDLE;     // Explicit disambiguation necessary.

    AuxMachineState[] states = {                // Requires (7) or (8)
        AVAILABLE, HIRED, UNDER_REPAIR,         // Requires (9)
        WRITE_OFF,                              // Requires (9) or (10)
        AuxMachineState.WRITE_OFF,              // Requires (7) or (8)
        Machine.AuxMachineState.WRITE_OFF,      // Requires (0)
        yap.Machine.AuxMachineState.WRITE_OFF // Does not require any import
    };
```

```
      for (AuxMachineState s : states)
        System.out.print(s + " ");
    }
  }
```

Output from the program:

```
AVAILABLE HIRED UNDER_REPAIR WRITE_OFF WRITE_OFF WRITE_OFF WRITE_OFF
```

Compiling Code into Packages

In this chapter, we will use pathname conventions used on a Unix platform. See Section 11.2, p. 468, for a discussion on pathnames and conventions for specifying pathnames on different platforms. While trying out the examples in this section, attention should be paid to platform-dependencies in this regard. Particularly, the fact that the *separator character* in a *file path* for the Unix and Windows platform is `'/'` 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 4.1, the package name `wizard.pandorasBox` corresponds to the pathname `wizard/pandorasBox`. The Java byte code 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 place the Java byte code 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 files in Example 4.1 are to be found in this directory, the command

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

issued in the current directory will create a file hierarchy under this directory, that mirrors the package hierarchy in Figure 4.2 (see also Figure 4.3). Note the subdirectories that are created for a fully qualified package name, and where the class files are located. In the command line above, space between the `-d` option and its argument is mandatory.

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

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

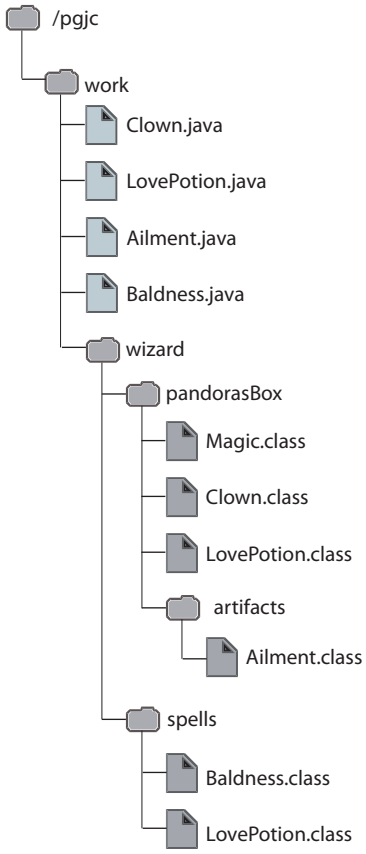
We can, of course, specify other destinations than the current directory where the class files with the byte code should be stored. The following command

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

in the current directory `/pgjc/work` will create the necessary packages with the class files under the destination directory `/pgjc/myapp`.

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.

Figure 4.3 *File Hierarchy*



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 4.1, 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

```
>java wizard.pandorasBox.Clown
```

This will load the class `Clown` from the byte code in the file with the pathname `./wizard/pandorasBox/Clown.class`, and start the execution of its `main()` method.

4.3 Searching for Classes

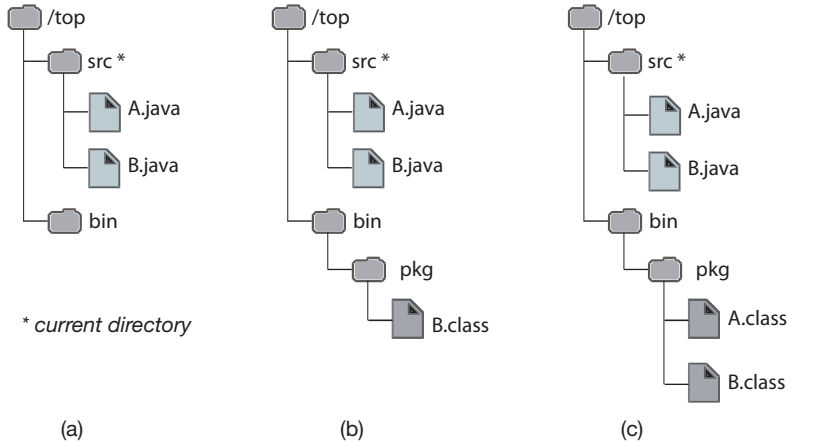
The documentation for the JDK tools explains how to organize packages in more elaborate schemes. In particular, the `CLASSPATH` environment variable can be used to specify the *class search path* (usually abbreviated to just *class path*), which are *pathnames* or *locations* in the file system where JDK tools should look when searching for classes and other resource files. Alternatively, the `-classpath` option (often abbreviated to `-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 `-cp` 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 `-cp` 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 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 `-cp` 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 4.4 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` is stored 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 4.4.

The file hierarchy before any files are compiled is shown in Figure 4.4a. 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 4.4b:

Figure 4.4 *Searching for Classes*




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

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

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

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

```
>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`, i.e., the file `B.class` containing the Java byte code for the class `B`. From Figure 4.4b 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 byte code file for the class `B` in the current directory, which is the default value of the class path. The command below sets the value of the class path to be `/top/bin`, and compilation is successful (see Figure 4.4c):

```
>javac -cp /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* of the package that is specified, i.e., the class path indicates the directory that *contains* the first element of the fully qualified package name. In Figure 4.4c, 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`:

```
>javac -cp /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 command above, the source file has the relative pathname `./A.java`. So the compiler looks for the source file in the current directory. The class path is used to find classes used by the class `A`.

Given the file hierarchy in Figure 4.3, the following `-cp` option sets the class path so that *all* packages (`wizard.pandorasBox`, `wizard.pandorasBox.artifacts`, `wizard.spells`) in Figure 4.3 will be searched, as all packages are located under the specified directory:

```
-cp /pgjc/work
```

However, the following `-cp` option will not help in finding *any* of the packages in Figure 4.3, as none of the *packages* are located under the specified directory:

```
>java -cp /pgjc/work/wizard pandorasBox.Clown
```

The command above 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 4.3 for classes that are used by the class `wizard.pandorasBox.Clown`:

```
>java -cp /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.

```
-cp /pgjc/work:/top/bin/pkg:.
```

We have used the path-separator character `':'` for Unix platforms to separate the entries, and also included the current directory (`.`) as an entry. There should be no white space on either side of the path-separator character.

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 `-cp` options shown below is significant. They will result in the class `pkg.B` being found under `/ext/lib1` and `/ext/lib2`, respectively.

```
-cp /ext/lib1:/ext/lib2
-cp /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 4.3, the following command will search the packages under the current directory:

```
>java -cp . wizard.pandorasBox.Clown
```

If the current directory has the absolute pathname `/top/src` in Figure 4.4, the following command will compile the file `./A.java`:

```
>javac -cp ../bin/pkg -d ../bin A.java
```

If the name of an entry in the class path includes white space, the name should be double quoted in order to be interpreted correctly:

```
-cp "../new bin/large pkg"
```

4.4 The JAR Utility

The JAR (Java ARchive) utility provides a convenient way of bundling and deploying Java programs. A JAR file is created by using the `jar` tool. A typical JAR file for an application will contain the class files and any other resources needed by the application (for example image and audio files). In addition, a special *manifest file* is also created and included in the archive. The manifest file can contain pertinent information, such as which class contains the `main()` method for starting the application.

The `jar` command has many options (akin to the Unix `tar` command). A typical command for making a JAR file for an application (for example, Example 4.10) has the following syntax:

```
>jar cmf whereismain.txt bundledApp.jar wizard
```

Option `c` tells the `jar` tool to create an archive. Option `m` is used to create and include a manifest file. Information to be included in the manifest file comes from a text file specified on the command line (`whereismain.txt`). Option `f` specifies the name of the archive to be created (`bundledApp.jar`). The JAR file name can be any valid file name. Files to be included in the archive are listed on the command line after the JAR file name. In the command line above, the contents under the `wizard` directory will be archived. If the order of the options `m` and `f` is switched in the command line, the order of the respective file names for these options must also be switched.

Information to be included in the manifest file is specified as name-value pairs. In Example 4.10, program execution should start in the `main()` method of the `wizard.pandorasBox.Clown` class. The file `whereismain.txt` has the following single text line:

```
Main-Class: wizard.pandorasBox.Clown
```

The value of the predefined header named `Main-Class` specifies the execution entry point of the application. The last text line in the file must be terminated by a newline as well, in order to be processed by the `jar` tool. This is also true even if the file only has a single line.

The application in an archive can be run by issuing the following command:

```
>java -jar bundledApp.jar
```

Program arguments can be specified after the JAR file name.

Another typical use of a JAR file is bundling packages as libraries so that other Java programs can use them. Such JAR files can be made available centrally, e.g., in the `jre/lib/ext` directory under Unix, where the `jre` directory contains the Java runtime environment. The *pathname* of such a JAR file can also be specified in the CLASSPATH environment variable. Clients can also use the `-cp` option to specify the pathname of the JAR file in order to utilize its contents. In all cases, the Java tools will be able to find the packages contained in the JAR file. The compiler can search the JAR file for classes when compiling the program, and the JVM can search the JAR file for classes to load in order to run the program.

As an example, we consider the file organization in Figure 4.5, where the class `MyApp` uses the class `org.graphics.draw4d.Menu`, and also classes from packages in the JAR file `gui.jar` in the directory `/top/lib`. We can compile the file `MyApp.java` in the current directory `/top/src` with the following command:

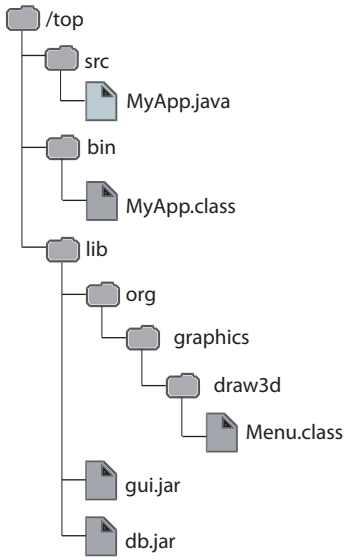
```
>javac -cp /top/lib/gui.jar:/top/lib -d /top/bin MyApp.java
```

Note that we need to specify *pathnames* of JAR files, but we specify *locations* where to search for particular packages.

We can also use the class path wildcard `*` to include all JAR files contained in a directory. Referring to Figure 4.5, the following `-cp` option will set the class path to include both the JAR files `gui.jar` and `db.jar`:

```
>javac -cp /top/lib/*:/top/lib -d /top/bin MyApp.java
```

Figure 4.5 *Searching in JAR files*



It may be necessary to quote the wildcard, depending on the configuration of the command line environment:

```
>javac -cp "/top/lib/*":/top/lib -d /top/bin MyApp.java
```

The wildcard * only expands to *JAR* files under the directory designated by the class path entry. It does *not* expand to any *class* files. Neither does it expand recursively to any JAR files contained in any subdirectories under the directory designated by the class path entry. The order in which the JAR files are searched depends on how the wildcard is expanded, and should not be relied upon when using the JDK tools.

4.5 System Properties

The Java runtime environment maintains persistent information like the operating system (OS) name, the JDK version, and various platform-dependent conventions (e.g., file separator, path separator, line terminator). This information is stored as a collection of *properties* on the platform on which the Java runtime environment is installed. Each property is defined as a *name-value* pair. For example, the name of the OS is stored as a property with the name `"os.name"` and the value `"Windows Vista"` on a platform running this OS. Properties are stored in a hash table, and applications can access them through the class `java.util.Properties`, which is a subclass of the `java.util.Hashtable` class (Section 15.8, p. 821).

Example 4.8 provides a basic introduction to using system properties. The `System.getProperties()` method returns a `Properties` hashtable containing all the properties stored on the host platform, (1). An application-defined property can be added to the `Properties` hashtable by calling the `setProperty()` method, with the appropriate name and value of the property. At (2), a property with the name `"appName"` and the value `"BigKahuna"` is put into the `Properties` hashtable. A property with a particular name can be retrieved from the `Properties` hashtable by calling the `getProperties()` method with the property name as argument, (3). Note that the type of both property name and value is `String`.

The program in Example 4.8 is run with the following command line:

```
>java SysProp os.name java.version appName FontSize
```

The program arguments are property names. The program looks them up in the `Properties` hashtable, and prints their values. We see that the value of the application-defined property with the name `"appNam"` is retrieved correctly. However, no property with the name `"FontSize"` is found, there `null` is printed as its value.

Another way of adding a property is by specifying it with the `-D` option (`D` for *Define*) in the `java` command. Running the program with the following command line

```
>java SysProp -DFontSize=18 os.name java.version appName FontSize
```

produces the following result:

```
os.name=Windows Vista
java.version=1.6.0_05
appName=BigKahuna
FontSize=18
```

The name and the value of the property are separated by the character = when specified using the -D option. The property is added by the JVM, and made available to the application.

There is also no white space on either side of the separator = in the -D option syntax, and the value can be double quoted, if necessary.

Example 4.8 *Using Properties*

```
import java.util.Properties;

public class SysProp {
  public static void main(String[] args) {
    Properties props = System.getProperties();      // (1)
    props.setProperty("appName", "BigKahuna");      // (2)
    for (String prop : args) {
      String value = props.getProperty(prop);       // (3)
      System.out.printf("%s=%s%n", prop, value);
    }
  }
}
```

Output from the program:

```
>javac SysProp.java
>java SysProp os.name java.version appName FontSize
os.name=Windows Vista
java.version=1.6.0_05
appName=BigKahuna
FontSize=null
```

Review Questions

4.1 What will be the result of attempting to compile this code?

```
import java.util.*;

package com.acme.toolkit;

public class AClass {
  public Other anInstance;
}

class Other {
  int value;
}
```

Select the one correct answer.

(a) The code will fail to compile, since the class `Other` has not yet been declared when referenced in the class `AClass`.
(b) The code will fail to compile, since an `import` statement cannot occur as the first statement in a source file.
(c) The code will fail to compile, since the `package` declaration cannot occur after an `import` statement.
(d) The code will fail to compile, since the class `Other` must be defined in a file called `Other.java`.
(e) The code will fail to compile, since the class `Other` must be declared `public`.
(f) The class will compile without errors.

4.2 Given the following code:

```
// (1) INSERT ONE IMPORT STATEMENT HERE
public class RQ700_20 {
  public static void main(String[] args) {
    System.out.println(sqrt(49));
  }
}
```

Which statements, when inserted at (1), will result in a program that prints `7`, when 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.*;`

4.3 Given the following code:

```
// (1) INSERT ONE IMPORT STATEMENT HERE
public class RQ700_10 {
  public static void main(String[] args) {
    System.out.println(Locale.UK);     // Locale string for UK is "en_GB".
  }
}
```

Which statements, when inserted at (1), will result in a program that prints `en_GB`, when compiled and run?

Select the two correct answers.

(a) `import java.util.*;`
(b) `import java.util.Locale;`
(c) `import java.util.Locale.UK;`
(d) `import java.util.Locale.*;`
(e) `import static java.util.*;`
(f) `import static java.util.Locale;`
(g) `import static java.util.Locale.UK;`
(h) `import static java.util.Locale.*;`

4.4 Given the following code:

```
package p1;
enum Signal {
  GET_SET, ON_YOUR_MARKS, GO;
}
-------------------------------------------------------------
package p2;
// (1) INSERT IMPORT STATEMENT(S) HERE
public class RQ700_50 {
  public static void main(String[] args) {
    for(Signal sign : Signal.values()) {
      System.out.println(sign);
    }
  }
}
```

Which import statement(s), when inserted at (1), will result in a program that prints the constants of the enum type `Signal`, when compiled and run?

Select the one correct answer.

(a) `import static p1.Signal.*;`
(b) `import p1.Signal;`
(c) `import p1.*;`
(d) `import p1.Signal;`
`import static p1.Signal.*;`
(e) `import p1.*;`
`import static p1.*;`
(f) None of the above.

4.5 Given the following code:

```
package p3;
public class Util {
  public enum Format {
    JPEG { public String toString() {return "Jpeggy"; }},
    GIF  { public String toString() {return "Giffy"; }},
    TIFF { public String toString() {return "Tiffy"; }};
  }
  public static <T> void print(T t) {
    System.out.print("|" + t + "|");
  }
}
-------------------------------------------------------------
// (1) INSERT IMPORT STATEMENTS HERE
public class NestedImportsA {
  public static void main(String[] args) {
    Util u = new Util();
    Format[] formats = {
        GIF, TIFF,
        JPEG,
        Format.JPEG,
        Util.Format.JPEG,
        p3.Util.Format.JPEG
```

```
      };
      for (Format fmt : formats)
        print(fmt);
    }
  }
```

Which sequence of import statements, when inserted at (1), will result in the code compiling, and the execution of the `main()` method printing:

```
|Giffy||Tiffy||Jpeggy||Jpeggy||Jpeggy||Jpeggy|
```

Select the three correct answers.

(a)
```
import p3.Util;
import p3.Util.Format;
import static p3.Util.print;
import static p3.Util.Format.*;
```

(b)
```
import p3.Util;
import static p3.Util.Format;
import static p3.Util.print;
import static p3.Util.Format.*;
```

(c)
```
import p3.*;
import static p3.Util.*;
import static p3.Util.Format.*;
```

(d)
```
import p3.*;
import p3.Util.*;
import static p3.Util.Format.*;
```

4.6 Which statements are true about the import statement?

Select the two correct answers.

(a) Static import from a class automatically imports names of static members of any nested types declared in that class.
(b) Static members of the default package cannot be imported.
(c) Static import statements must be specified after any type import statements.
(d) In the case of a name conflict, the name in the last static import statement is chosen.
(e) A declaration of a name in a compilation unit can shadow a name that is imported.

4.7 Given the source file `A.java`:

```
package top.sub;
public class A {}
```

And the following directory hierarchy:

```
/proj
  |--- src
  |     |--- top
  |           |--- sub
  |                 |--- A.java
  |--- bin
```

Assuming that the current directory is `/proj/src`, which of the following statements are true?

Select the three correct answers.

(a) The following command will compile, and place the file `A.class` under `/proj/bin`:
```
javac -d . top/sub/A.java
```

(b) The following command will compile, and place the file `A.class` under `/proj/bin`:
```
javac -d /proj/bin top/sub/A.java
```

(c) The following command will compile, and place the file `A.class` under `/proj/bin`:
```
javac -D /proj/bin ./top/sub/A.java
```

(d) The following command will compile, and place the file `A.class` under `/proj/bin`:
```
javac -d ../bin top/sub/A.java
```

(e) After successful compilation, the absolute pathname of the file `A.class` will be:
```
/proj/bin/A.class
```

(f) After successful compilation, the absolute pathname of the file `A.class` will be:
```
/proj/bin/top/sub/A.class
```

4.8 Given the following directory structure:

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

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

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

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

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

Select the two correct answers.

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

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

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

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

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

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

4.9 Given the following directory structure:

```
/proj
  |--- src
  |     |--- A.class
  |
  |
  |--- bin
        |--- top
              |--- sub
                    |--- A.class
```

Assume that the current directory is `/proj/src`. Which classpath specifications will find the file `A.class` for the class `top.sub.A`?

Select the two correct answers.

(a) `-cp /top/bin/top`

(b) `-cp /top/bin/top/sub`

(c) `-cp /top/bin/top/sub/A.class`

(d) `-cp ../bin;.`

(e) `-cp /top`

(f) `-cp /top/bin`

4.10 Given that the name of the class `MyClass` is specified correctly, which commands are syntactically valid:

Select the two correct answers.

(a) `java -Ddebug=true MyClass`

(b) `java -ddebug=true MyClass`

(c) `java -Ddebug="true" MyClass`

(d) `java -D debug=true MyClass`

4.11 Which statement is true?

Select the one correct answer.

(a) A JAR file can only contain one package.

(b) A JAR file can only be specified for use with the `java` command, in order to run a program.

(c) The classpath definition of the platform overrides any entries specified in the

classpath option.

(d) The `-d` option is used with the `java` command, and the `-D` is used with the `javac` command.

(e) None of the above statements are true.

4.6 Scope Rules

Java provides explicit accessibility modifiers to control the accessibility of members in a class by external clients (see Section 4.9, p. 138), 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 4.1 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 4.1 assumes the following declarations:

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

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

class ClassName extends SuperName {
  int instanceVar;
  static int staticVar;

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

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, therefore the references `this` and `super` are not available. An object has knowledge of its class, therefore, 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.

Some factors that can influence the scope of a member declaration are:

- shadowing of a field declaration, either by local variables (see Section 4.6, p. 131) or by declarations in the subclass (see Section 7.3, p. 294)
- initializers preceding the field declaration (see Section 9.7, p. 406)
- overriding an instance method from a superclass (see Section 7.2, p. 288)
- hiding a static method declared in a superclass (see Section 7.3, p. 294)

Accessing members within nested classes is discussed in Chapter 8.

Table 4.1 *Accessing Members within a Class*

Member declarations	**Non-static Code in the Class `ClassName` Can Refer to the Member as**	**Static Code in the Class `ClassName` 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`
	`ClassName.staticVar`
	`staticVarInSuper`
	`this.staticVarInSuper`
	`super.staticVarInSuper`
	`ClassName.staticVarInSuper`
	`SuperName.staticVarInSuper`
		`staticVar`
		
		`ClassName.staticVar`
		`staticVarInSuper`
		
		`ClassName.staticVarInSuper`
		`SuperName.staticVarInSuper`

Static methods	`staticMethod()`
	`this.staticMethod()`
	`ClassName.staticMethod()`
	`staticMethodInSuper()`
	`this.staticMethodInSuper()`
	`super.staticMethodInSuper()`
	`ClassName.staticMethodInSuper()`
	`SuperName.staticMethodInSuper()`
		`staticMethod()`
		
		`ClassName.staticMethod()`
		`staticMethodInSuper()`
		
		`ClassName.staticMethodInSuper()`
		`SuperName.staticMethodInSuper()`

Within a class `C`, references of type `C` can be used to access *all* members in the class `C`, regardless of their accessibility modifiers. In Example 4.9, the method `duplicate-`

`Light` 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 4.9 *Class Scope*

```
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 braces, `{}`. 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* inside the block in which it is declared, but it is not accessible outside of 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 (see Section 6.5, p. 235).

Figure 4.6 illustrates block scope for local variables. A method body is a block. 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.

Figure 4.6 *Block Scope*

```
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.
      } // 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.
      } //if
    } // for
    int index;          // (7) OK
} // main
```

4.7 Accessibility Modifiers for Top-Level Type Declarations

The accessibility modifier `public` can be used to declare top-level types (that is, classes, enums, and interfaces) in a package to be accessible from everywhere, both inside their own package and other packages. If the accessibility modifier is omitted, they are only accessible in their own package and not in any other packages or subpackages. This is called *package* or *default accessibility*.

Accessibility modifiers for nested reference types are discussed in Section 8.1 on page 352.

Example 4.10 *Accessibility Modifiers for Classes and Interfaces*

```
//File: Clown.java
package wizard.pandorasBox;                       // (1) Package declaration

import wizard.pandorasBox.artifacts.Ailment;      // (2) Importing class

public class Clown implements Magic {
  LovePotion tlc;                                 // (3) Class in same package
  wizard.pandorasBox.artifacts.Ailment problem;   // (4) Fully qualified class name
  Clown() {
    tlc = new LovePotion("passion");
    problem = new Ailment("flu");                 // (5) Simple class name
  }
  public void levitate()  { 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) {        // (6)
    Clown joker = new Clown();
    joker.levitate();
    joker.mixPotion();
    joker.healAilment();
  }
}

interface Magic { void levitate(); }              // (7)
```

```
//File: LovePotion.java
package wizard.pandorasBox;                 // (1) Package declaration

public class LovePotion {                   // (2) Accessible outside package
  String potionName;
  public LovePotion(String name) { potionName = name; }
  public String toString() { return potionName; }
}
```

```
//File: Ailment.java
package wizard.pandorasBox.artifacts;       // (1) Package declaration

public class Ailment {                      // (2) Accessible outside package
  String ailmentName;
  public Ailment(String name) { ailmentName = name; }
  public String toString() { return ailmentName; }
}
```

```
//File: Baldness.java
package wizard.spells;                        // (1)Package declaration

import wizard.pandorasBox.*;                  // (2) Type import on demand
import wizard.pandorasBox.artifacts.*;        // (3) Import of subpackage

public class Baldness extends Ailment {       // (4) Simple name for Ailment
```

```
  wizard.pandorasBox.LovePotion tlcOne;      // (5) Fully qualified name
  LovePotion tlcTwo;                         // (6) Class in same package
  Baldness(String name) {
    super(name);
    tlcOne = new wizard.pandorasBox.         // (7) Fully qualified name
    LovePotion("romance");
    tlcTwo = new LovePotion();               // (8) Class in same package
  }
}

class LovePotion // implements Magic         // (9) Not accessible
{ public void levitate(){} }
```

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

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

In Example 4.10, the class `Clown` and the interface `Magic` are placed in a package called `wizard.pandorasBox`. The `public` class `Clown` is accessible from everywhere. The `Magic` interface has default accessibility, and can only be accessed within the package `wizard.pandorasBox`. It is not accessible from other packages, not even from its subpackages.

The class `LovePotion` 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.

The class `Clown`, from the file `Clown.java`, uses the class `Ailment`. The example shows two ways in which a class can access classes from other packages:

1. Denote the class by its fully qualified class name, as shown at (4) (`wizard.pandorasBox.artifacts.Ailment`).
2. Import the class explicitly from the package `wizard.pandorasBox.artifacts` as shown at (2), and use the simple class name `Ailment`, as shown at (5).

In the file `Baldness.java` at (9), the class `LovePotion` wishes to implement the interface `Magic` from the package `wizard.pandorasBox`, but cannot do so, although the source file imports from this package. The reason is that the interface `Magic` has default accessibility and can, therefore, only be accessed within the package `wizard.pandorasBox`.

Just because a type is accessible does not necessarily mean that members of the type are also accessible. Member accessibility is governed separately from type accessibility, as explained in Section 4.6.

Table 4.2 *Summary of Accessibility Modifiers for Top-Level Types*

Modifiers	Top-Level Types
default (no modifier)	Accessible in its own package (*package accessibility*)
`public`	Accessible anywhere

4.8 Other Modifiers for Classes

The modifiers `abstract` and `final` can be applied to top-level and nested classes.

`abstract` Classes

A class can be declared with the keyword `abstract` to indicate that it cannot be instantiated. A class might choose to do this if the abstraction it represents is so general that it needs to be specialized in order to be of practical use. The class `Vehicle` might be specified as `abstract` to represent the general abstraction of a vehicle, as creating instances of the class would not make much sense. Creating instances of non-`abstract` subclasses, like `Car` and `Bus`, would make more sense, as this would make the abstraction more concrete.

Any *normal class* (that is, a class declared with the keyword `class`) can be declared `abstract`. However, if such a class that has one or more `abstract` methods (see Section 4.10, p. 150), it must be declared `abstract`. Obviously such classes cannot be instantiated, as their implementation might only be partial. A class might choose this strategy to dictate certain behavior, but allow its subclasses the freedom to provide the relevant implementation. In other words, subclasses of the `abstract` class have to take a stand and provide implementations of any inherited `abstract` methods before instances can be created. A subclass that does not provide an implementation of its inherited `abstract` methods, must also be declared `abstract`.

In Example 4.11, the declaration of the `abstract` class `Light` has an `abstract` method named `kwhPrice` at (1). This forces its subclasses to provide an implementation for this method. The subclass `TubeLight` provides an implementation for the method `kwhPrice()` at (2). The class `Factory` creates an instance of the class `TubeLight` at (3). References of an `abstract` class can be declared, as shown at (4), but an `abstract` class cannot be instantiated, as shown at (5). References of an `abstract` class can refer to objects of the subclasses, as shown at (6).

Interfaces just specify `abstract` methods and not any implementation; they are, by their nature, implicitly `abstract` (that is, they cannot be instantiated). Though it is

legal, it is redundant to declare an interface with the keyword `abstract` (see Section 7.6, p. 309).

Enum types *cannot* be declared `abstract`, because of the way they are implemented in Java (see Section 3.5, p. 54).

Example 4.11 *Abstract Classes*

```
abstract class Light {
  // Fields:
  int     noOfWatts;       // wattage
  boolean indicator;       // on or off
  String  location;        // placement

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

  // Abstract instance method
  abstract public double kwhPrice();              // (1) No method body
}
//_____________________________________________________________________________
class TubeLight extends Light {
  // Field
  int tubeLength;

  // Implementation of inherited abstract method.
  public double kwhPrice() { return 2.75; }       // (2)
}
//_____________________________________________________________________________
public class Factory {
  public static void main(String[] args) {
    TubeLight cellarLight = new TubeLight();      // (3) OK
    Light nightLight;                             // (4) OK
//  Light tableLight = new Light();               // (5) Compile time error
    nightLight = cellarLight;                     // (6) OK
    System.out.println("KWH price: " + nightLight.kwhPrice());
  }
}
```

Output from the program:

```
KWH price: 2.75
```

`final` Classes

A class can be declared `final` to indicate that it cannot be extended; that is, one cannot declare subclasses of a `final` class. This implies that one cannot override any methods declared in such a class. In other words, the class behavior cannot be

changed by extending the class. A `final` class marks the lower boundary of its *implementation inheritance hierarchy* (see Section 7.1, p. 284). Only a class whose definition is *complete* (that is, provides implementations of all its methods) can be declared `final`.

A `final` class must be complete, whereas an `abstract` class is considered incomplete. Classes, therefore, cannot be both `final` and `abstract` at the same time. Interfaces are inherently `abstract`, as they can only specify methods that are abstract, and therefore cannot be declared `final`. A `final` class and an interface represent two extremes when it comes to providing an implementation. An `abstract` class represents a compromise between these two extremes. An enum type is also implicitly `final`, and cannot be explicitly declared with the keyword `final`.

The Java standard library includes many `final` classes; for example, the `java.lang.String` class and the wrapper classes for primitive values.

If it is decided that the class `TubeLight` in Example 4.11 may not be extended, it can be declared `final`:

```
final class TubeLight extends Light {
  // ...
}
```

Discussion of `final` methods, fields, and local variables can be found in Section 4.10, p. 148.

Table 4.3 *Summary of Other Modifiers for Types*

Modifiers	Classes	Interfaces	Enum types
`abstract`	A non-`final` class can be declared `abstract`. A class with an `abstract` method must be declared `abstract`. An `abstract` class cannot be instantiated.	Permitted, but redundant.	Not permitted.
`final`	A non-`abstract` class can be declared `final`. A class with a `final` method need not be declared `final`. A `final` class cannot be extended.	Not permitted.	Not permitted.

Review Questions

4.12 Given the following class, which of these alternatives are valid ways of referring to the class from outside of the package `net.basemaster`?

```
package net.basemaster;

public class Base {
  // ...
}
```

Select the two correct answers.

(a) By simply referring to the class as `Base`.
(b) By simply referring to the class as `basemaster.Base`.
(c) By simply referring to the class as `net.basemaster.Base`.
(d) By importing with `net.basemaster.*`, and referring to the class as `Base`.
(e) By importing with `net.*`, and referring to the class as `basemaster.Base`.

4.13 Which one of the following class declarations is a valid declaration of a class that cannot be instantiated?

Select the one correct answer.

(a) `class Ghost { abstract void haunt(); }`
(b) `abstract class Ghost { void haunt(); }`
(c) `abstract class Ghost { void haunt() {}; }`
(d) `abstract Ghost { abstract void haunt(); }`
(e) `static class Ghost { abstract haunt(); }`

4.14 Which one of the following class declarations is a valid declaration of a class that cannot be extended?

Select the one correct answer.

(a) `class Link { }`
(b) `abstract class Link { }`
(c) `native class Link { }`
(d) `static class Link { }`
(e) `final class Link { }`
(f) `private class Link { }`
(g) `abstract final class Link { }`

4.9 Member Accessibility Modifiers

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

The accessibility of members can be one of the following:

- `public`
- `protected`
- default (also called *package accessibility*)
- `private`

If an accessibility modifier is not specified, the member has package or default accessibility.

In the following discussion on accessibility modifiers for members of a class, keep in mind that the member accessibility modifier only has meaning if the class (or one of its subclasses) is accessible to the client. Also, note that only one accessibility modifier can be specified for a member. The discussion in this section applies to both instance and static members of top-level classes. It applies equally to *constructors* as well. Discussion of member accessibility for nested classes is deferred to Chapter 8.

In UML notation, the prefixes `+`, `#`, and `-`, when applied to a member name, indicate `public`, `protected`, and `private` member accessibility, respectively. No prefix indicates default or package accessibility.

`public` Members

Public accessibility is the least restrictive of all the accessibility modifiers. A `public` member is accessible from anywhere, both in the package containing its class and in other packages where this class is visible. This is true for both instance and static members.

Example 4.12 contains two source files, shown at (1) and (6). The package hierarchy defined by the source files is depicted in Figure 4.7, showing the two packages, `packageA` and `packageB`, containing their respective classes. The classes in `packageB` use classes from `packageA`. The class `SuperclassA` in `packageA` has two subclasses: `SubclassA` in `packageA` and `SubclassB` in `packageB`.

Example 4.12 *Public Accessibility of Members*

```
//Filename: SuperclassA.java                              (1)
package packageA;

public class SuperclassA {
  public int superclassVarA;                          // (2)
  public void superclassMethodA() {/*...*/}           // (3)
}

class SubclassA extends SuperclassA {
  void subclassMethodA() { superclassVarA = 10; }   // (4) OK.
}
```

```
class AnyClassA {
  SuperclassA obj = new SuperclassA();
  void anyClassMethodA() {
    obj.superclassMethodA();                          // (5) OK.
  }
}
```

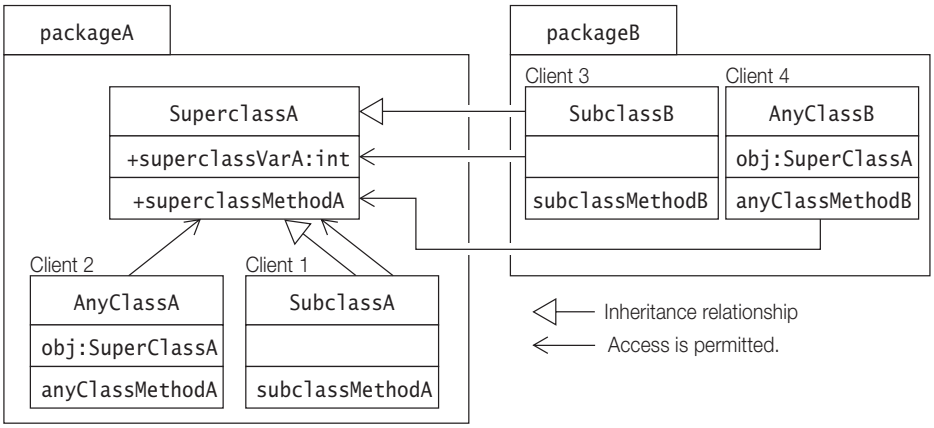
```
//Filename: SubclassB.java                                (6)
package packageB;
import packageA.*;

public class SubclassB extends SuperclassA {
  void subclassMethodB() { superclassMethodA(); }   // (7) OK.
}

class AnyClassB {
  SuperclassA obj = new SuperclassA();
  void anyClassMethodB() {
    obj.superclassVarA = 20;                          // (8) OK.
  }
}
```

Accessibility is illustrated in Example 4.12 by the accessibility modifiers for the field `superclassVarA` and the method `superclassMethodA()` at (2) and (3), respectively, defined in the class `SuperclassA`. These members are accessed from four different clients in Example 4.12.

- Client 1: From a subclass in the same package, which accesses an inherited field. `SubclassA` is such a client, and does this at (4).
- Client 2: From a non-subclass in the same package, which invokes a method on an instance of the class. `AnyClassA` is such a client, and does this at (5).
- Client 3: From a subclass in another package, which invokes an inherited method. `SubclassB` is such a client, and does this at (7).
- Client 4: From a non-subclass in another package, which accesses a field in an instance of the class. `AnyClassB` is such a client, and does this at (8).

In Example 4.12, the field `superclassVarA` and the method `superclassMethodA()` have public accessibility, and are accessible by all four clients listed above. Subclasses can access their inherited public members by their simple name, and all clients can access public members through an instance of the class. Public accessibility is depicted in Figure 4.7.

Figure 4.7 *Public Accessibility*



`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 this class is visible. In other words, non-subclasses in other packages cannot access protected members from other packages. It is more restrictive than public member accessibility.

In Example 4.12, if the field `superclassVarA` and the method `superclassMethodA()` have protected accessibility, they are accessible within `packageA`, and only accessible by subclasses in any other packages.

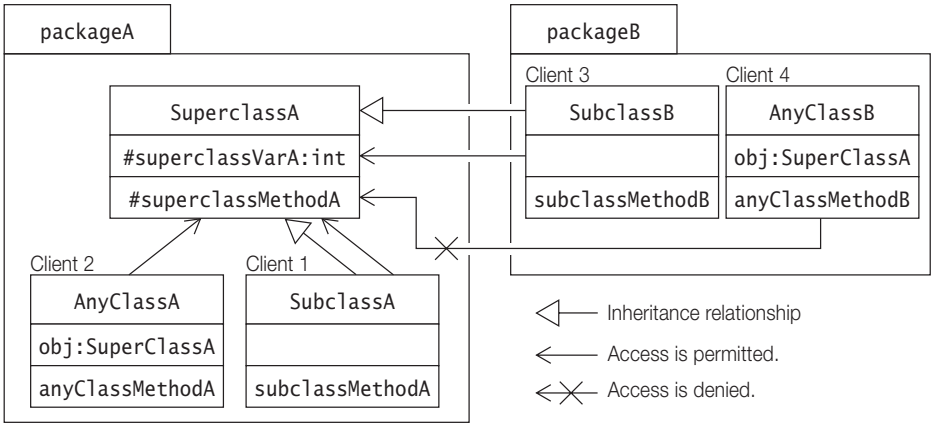
```
public class SuperclassA {
  protected int superclassVarA;                    // (2) Protected member
  protected void superclassMethodA() {/*...*/}     // (3) Protected member
}
```

Client 4 in `packageB` cannot access these members, as shown in Figure 4.8.

A subclass in another package can only access protected members in the superclass via references of its own type or its subtypes. The following new declaration of `SubclassB` in `packageB` from Example 4.12 illustrates the point:

```
// Filename: SubclassB.java
package packageB;
import packageA.*;
public class SubclassB extends SuperclassA {     // In packageB.
  SuperclassA objRefA = new SuperclassA();        // (1)
  void subclassMethodB(SubclassB objRefB) {
    objRefB.superclassMethodA();                  // (2) OK.
    objRefB.superclassVarA = 5;                   // (3) OK.
    objRefA.superclassMethodA();                  // (4) Not OK.
    objRefA.superclassVarA = 10;                  // (5) Not OK.
  }
}
```

Figure 4.8 *Protected Accessibility*



The class `SubclassB` declares the field `objRefA` of type `SuperclassA` at (1). The method `subclassMethodB()` has the formal parameter `objRefB` of type `SubclassB`. Access is permitted to a protected member of `SuperclassA` in `packageA` by a reference of the subclass, as shown at (2) and (3), but not by a reference of its superclass, as shown at (4) and (5). Access to the field `superclassVarA` and the call to the method `superclassMethodA()` occur in `SubclassB`. These members are declared in `SuperclassA`. `SubclassB` is not involved in the implementation of `SuperclassA`, which is the type of the reference `objRefA`. Hence, access to protected members at (4) and (5) is not permitted as these are not members of an object that can be guaranteed to be implemented by the code accessing them.

Accessibility to protected members of the superclass would also be permitted via any reference whose type is a subclass of `SubclassB`. The above restriction helps to ensure that subclasses in packages different from their superclass can only access protected members of the superclass in their part of the implementation inheritance hierarchy. In other words, a protected member of a superclass is only accessible in a subclass that is in another package if the member is inherited by an object of the subclass (or by an object of a subclass of this subclass).

Default Accessibility for Members

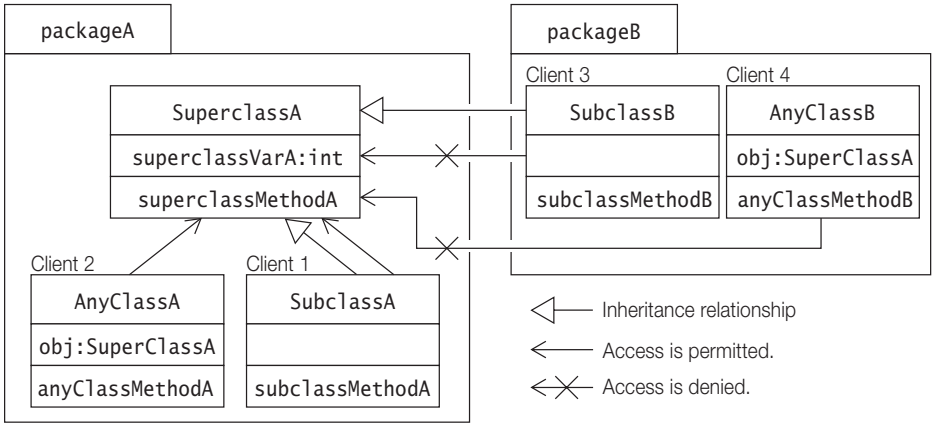
When no member accessibility modifier is specified, the member is only accessible by other classes in its own class's package. Even if its class is visible in another (possibly nested) package, the member is not accessible elsewhere. Default member accessibility is more restrictive than protected member accessibility.

In Example 4.12, if the field `superclassVarA` and the method `superclassMethodA()` are defined with no accessibility modifier, they are only accessible within `packageA`, but not in any other packages.

```
public class SuperclassA {
  int superclassVarA;                              // (2)
  void superclassMethodA() {/*...*/}               // (3)
}
```

The clients in `packageB` (that is, Clients 3 and 4) cannot access these members. This situation is depicted in Figure 4.9.

Figure 4.9 *Default Accessibility*



`private` Members

This is the most restrictive of all the accessibility modifiers. Private members are not accessible from 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 name in a subclass, they are also not inherited by the subclass. This is not to be confused with the existence of such a member in the state of an object of the subclass (see Section 9.11, p. 416). A standard design strategy for JavaBeans is to make all fields private and provide public accessor methods for them. Auxiliary methods are often declared private, as they do not concern any client.

In Example 4.12, if the field `superclassVarA` and the method `superclassMethodA()` have private accessibility, they are not accessible by any other clients.

```
public class SuperclassA {
  private int superclassVarA;                      // (2) Private member
  private void superclassMethodA() {/*...*/}       // (3) Private member
}
```

None of the clients in Figure 4.10 can access these members.

Figure 4.10 *Private Accessibility*

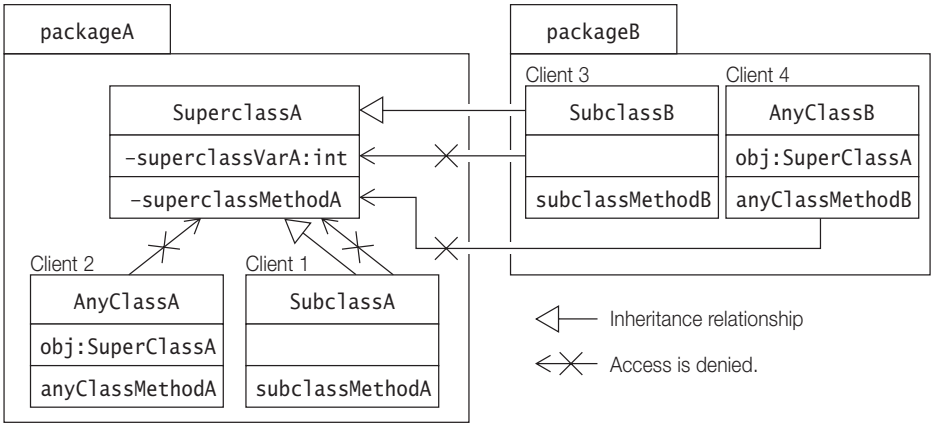


Table 4.4 *Summary of Accessibility Modifiers for Members*

Modifiers	Members
`public`	Accessible everywhere.
`protected`	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
`private`	Only accessible in its own class and not anywhere else.

Review Questions

4.15 Given the following declaration of a class, which fields are accessible from outside the package `com.corporation.project`?

```
package com.corporation.project;

public class MyClass {
            int i;
  public    int j;
  protected int k;
  private   int l;
}
```

Select the two correct answers.

(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.

4.16 How restrictive is the default accessibility compared to `public`, `protected`, and `private` accessibility?

Select the one correct answer.

(a) Less restrictive than `public`.
(b) More restrictive than `public`, but less restrictive than `protected`.
(c) More restrictive than `protected`, but less restrictive than `private`.
(d) More restrictive than `private`.
(e) Less restrictive than `protected` from within a package, and more restrictive than `protected` from outside a package.

4.17 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 only be accessed within the class of the member.
(c) A member with default accessibility 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/default accessibility for a member can be declared using the keyword `default`.

4.18 Which lines that are marked will compile in the following code?

```
//Filename: A.java
package packageA;

public class A {
  protected int pf;
}
```

```
//Filename: B.java
package packageB;
import packageA.A;

public class B extends A {
  void action(A obj1, B obj2, C obj3) {
    pf = 10;                // (1)
    obj1.pf = 10;           // (2)
    obj2.pf = 10;           // (3)
    obj3.pf = 10;           // (4)
```

```
  }
}

class C extends B {
  void action(A obj1, B obj2, C obj3) {
    pf = 10;                // (5)
    obj1.pf = 10;           // (6)
    obj2.pf = 10;           // (7)
    obj3.pf = 10;           // (8)
  }
}

class D {
  void action(A obj1, B obj2, C obj3) {
    pf = 10;                // (9)
    obj1.pf = 10;           // (10)
    obj2.pf = 10;           // (11)
    obj3.pf = 10;           // (12)
  }
}
```

Select the five correct answers.

(a) `(1)`
(b) `(2)`
(c) `(3)`
(d) `(4)`
(e) `(5)`
(f) `(6)`
(g) `(7)`
(h) `(8)`
(i) `(9)`
(j) `(10)`
(k) `(11)`
(l) `(12)`

4.10 Other Modifiers for Members

The following keywords can be used to specify certain characteristics of members in a type declaration:

- `static`
- `final`
- `abstract`
- `synchronized`
- `native`
- `transient`
- `volatile`

`static` Members

Static members belong to the class in which they are declared and are not part of any instance of the class. The declaration of static members is prefixed by the keyword `static` to distinguish them from instance members. Depending on the accessibility modifiers of the static members in a class, clients can access these by using the class name or through object references of the class. The class need not be instantiated to access its static members.

Static variables (also called *class variables*) exist in the class they are defined in only. They are not instantiated when an instance of the class is created. In other words, the values of these variables are not a part of the state of any object. When the class is loaded, static variables are initialized to their default values if no explicit initialization expression is specified (see Section 9.9, p. 410).

Static methods are also known as *class methods*. A static method in a class can directly access other static members in the class. It cannot access instance (i.e., non-static) members of the class, as there is no notion of an object associated with a static method.

A typical static method might perform some task on behalf of the whole class and/or for objects of the class. In Example 4.13, the static variable `counter` keeps track of the number of instances of the `Light` class that have been created. The example shows that the static method `writeCount` can only access static members directly, as shown at (2), but not non-static members, as shown at (3). The static variable `counter` will be initialized to the value `0` when the class is loaded at runtime. The `main()` method at (4) in the class `Warehouse` shows how static members of the class `Light` can be accessed using the class name and via object references of the type `Light`.

A summary of how static members are accessed in static and non-static code is given in Table 4.1, p. 130.

Example 4.13 *Accessing Static Members*

```
class Light {
  // Fields:
  int     noOfWatts;      // wattage
  boolean indicator;      // on or off
  String  location;       // placement

  // Static variable
  static int counter;     // No. of Light objects created.         (1)

  // Explicit default constructor
  Light(int noOfWatts, boolean indicator, String location) {
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
    ++counter;          // Increment counter.
  }
```

```
  // Static method
  public static void writeCount() {
    System.out.println("Number of lights: " + counter);         // (2)
    // Compile-time error. Field noOfWatts is not accessible:
    // System.out.println("Number of Watts: " + noOfWatts);     // (3)
  }
}
//_____________________________________________________________________________
public class Warehouse {
  public static void main(String[] args) {                      // (4)

    Light.writeCount();                                 // Invoked using class name
    Light light1 = new Light(100, true, "basement"); // Create an object
    System.out.println(
        "Value of counter: " + Light.counter         // Accessed via class name
    );
    Light light2 = new Light(200, false, "garage");  // Create another object
    light2.writeCount();                                // Invoked using reference
    Light light3 = new Light(300, true, "kitchen");  // Create another object
    System.out.println(
        "Value of counter: " + light3.counter        // Accessed via reference
    );
    final int i;
  }
}
```

Output from the program:

```
Number of lights: 0
Value of counter: 1
Number of lights: 2
Value of counter: 3
```

`final` Members

A `final` variable is a constant despite being called a variable. Its value cannot be changed once it has been initialized. Instance and static variables can be declared `final`. Note that the keyword `final` can also be applied to local variables, including method parameters. Declaring a variable `final` has the following implications:

- A `final` variable of a primitive data type cannot change its value once it has been initialized.
- A `final` variable of a reference type cannot change its reference value once it has been initialized. This effectively means that a `final` reference will always refer to the same object. However, the keyword `final` has no bearing on whether the *state of the object* denoted by the reference can be changed or not.

Final static variables are commonly used to define *manifest constants* (also called *named constants*), e.g., `Integer.MAX_VALUE`, which is the maximum `int` value. Variables defined in an interface are implicitly `final` (see Section 7.6, p. 309). Note that a `final` variable need not be initialized in its declaration, but it must be initialized in

the code once before it is used. These variables are also known as *blank final variables*. For a discussion on `final` parameters, see Section 3.7, p. 89.

A `final` method in a class is *complete* (that is, has an implementation) and cannot be overridden in any subclass (see Section 7.2, p. 288).

Final variables ensure that values cannot be changed and `final` methods ensure that behavior cannot be changed. Final classes are discussed in Section 4.8, p. 136.

The compiler may be able to perform code optimizations for `final` members, because certain assumptions can be made about such members.

In Example 4.14, the class `Light` defines a `final` static variable at (1) and a `final` method at (2). An attempt to change the value of the `final` variable at (3) results in a compile-time error. The subclass `TubeLight` attempts to override the `final` method `setWatts()` from the superclass `Light` at (4), which is not permitted. The class `Warehouse` defines a `final` local reference `aLight` at (5). The state of the object denoted by the reference `tableLight` is changed at (6), but its reference value cannot be changed as attempted at (7). Another `final` local reference `streetLight` is declared at (8), but it is not initialized. The compiler reports an error when an attempt is made to use this reference at (9).

Example 4.14 *Accessing Final Members*

```
class Light {
  // Final static variable                  (1)
  final public static double KWH_PRICE = 3.25;

  int noOfWatts;

  // Final instance method                  (2)
  final public void setWatts(int watt) {
    noOfWatts = watt;
  }

  public void setKWH() {
    // KWH_PRICE = 4.10;                 // (3) Not OK. Cannot be changed.
  }
}
//_____________________________________________________________________________
class TubeLight extends Light {
  // Final method in superclass cannot be overridden.
  // This method will not compile.
  /*
    public void setWatts(int watt) {     // (4) Attempt to override.
         noOfWatts = 2*watt;
    }
   */
}
//_____________________________________________________________________________
```

```
public class Warehouse {
  public static void main(String[] args) {

    final Light tableLight = new Light();// (5) Final local variable.
    tableLight.noOfWatts = 100;          // (6) OK. Changing object state.
//  tableLight = new Light();            // (7) Not OK. Changing final reference.

    final Light streetLight;             // (8) Not initialized.
//  streetLight.noOfWatts = 2000;        // (9) Not OK.
  }
}
```

`abstract` Methods

An `abstract` method has the following syntax:

`abstract` <*accessibility modifier*> <*return type*> <*method name*> (<*parameter list*>) <*throws clause*>;

An `abstract` method does not have an implementation; i.e., no method body is defined for an `abstract` method, only the *method header* is provided in the class declaration. The keyword `abstract` is mandatory in the header of an `abstract` method declared in a class. Its class is then incomplete and must be explicitly declared `abstract` (see Section 4.8, p. 135). Subclasses of an `abstract` class must then provide the method implementation; otherwise, they must also be declared `abstract`. The accessibility of an `abstract` method declared in a class cannot be `private`, as subclasses would not be able to override the method and provide an implementation. See Section 4.8, where Example 4.11 also illustrates the use of `abstract` methods.

Only an instance method can be declared `abstract`. Since static methods cannot be overridden, declaring an `abstract` static method makes no sense. A `final` method cannot be `abstract` (i.e., cannot be incomplete) and vice versa. The keyword `abstract` can only be combined with accessibility modifiers `public` or `private`.

Methods specified in an interface are implicitly `abstract` (see Section 7.6, p. 309), and the keyword `abstract` is seldom specified in their method headers. These methods can only have `public` or package accessibility.

`synchronized` Methods

Several threads can be executing in a program (see Section 13.5, p. 626). They might try to execute several methods on the same object simultaneously. Methods can be declared `synchronized` if it is desirable that only one thread at a time can execute a method of the object. Their execution is then mutually exclusive among all threads. At any given time, at most one thread can be executing a `synchronized` method on an object. This discussion also applies to static `synchronized` methods of a class.

In Example 4.15, both the `push()` method, declared at (1), and the `pop()` method, declared at (2), are `synchronized` in the class `StackImpl`. Only one thread at a time can

execute a synchronized method in an object of the class StackImpl. This means that it is not possible for the state of an object of the class StackImpl to be corrupted, for example, while one thread is pushing an element and another is attempting to pop the stack.

Example 4.15 *Synchronized Methods*

```
class StackImpl {                // Non-generic partial implementation
  private Object[] stackArray;
  private int topOfStack;
  // ...
  synchronized public void push(Object elem) { // (1)
    stackArray[++topOfStack] = elem;
  }

  synchronized public Object pop() {           // (2)
    Object obj = stackArray[topOfStack];
    stackArray[topOfStack] = null;
    topOfStack--;
    return obj;
  }

  // Other methods, etc.
  public Object peek() { return stackArray[topOfStack]; }
}
```

native Methods

Native methods are methods whose implementation is not defined in Java but in another programming language, for example, C or C++. Such a method can be declared as a member in a Java class declaration. Since its implementation appears elsewhere, only the method header is specified in the class declaration. The keyword native is mandatory in the method header. A native method can also specify checked exceptions in a throws clause (Section 6.9, p. 257), but the compiler cannot check them, since the method is not implemented in Java.

In the following example, a native method in the class Native is declared at (2). The class also uses a static initializer block (see Section 9.9, p. 410) at (1) to load the native library when the class is loaded. Clients of the Native class can call the native method like any another method, as at (3).

```
class Native {

  /*
   * The static block ensures that the native method library
   * is loaded before the native method is called.
   */
  static {
    System.loadLibrary("NativeMethodLib");  // (1) Load native library.
  }
```

```
  native void nativeMethod();               // (2) Native method header.
  // ...

}

class Client {
  //...
  public static void main(String[] args) {
    Native trueNative = new Native();
    trueNative.nativeMethod();              // (3) Native method call.
  }
  //...
}
```

The Java Native Interface (JNI) is a special API that allows Java methods to invoke native functions implemented in C.

`transient` Fields

Often it is desirable to save the state of an object. Such objects are said to be *persistent*. In Java, the state of an object can be stored using serialization (see Section 11.6, p. 510). Serialization transforms objects into an output format that is conducive for storing objects. Objects can later be retrieved in the same state as when they were serialized, meaning that all fields included in the serialization will have the same values as at the time of serialization.

Sometimes the value of a field in an object should not be saved, in which case, the field can be specified as `transient` in the class declaration. This implies that its value should not be saved when objects of the class are written to persistent storage. In the following example, the field `currentTemperature` is declared `transient` at (1), because the current temperature is most likely to have changed when the object is restored at a later date. However, the value of the field `mass`, declared at (2), is likely to remain unchanged. When objects of the class `Experiment` are serialized, the value of the field `currentTemperature` will not be saved, but that of the field `mass` will be, as part of the state of the serialized object.

```
class Experiment implements Serializable {
  // ...

  // The value of currentTemperature will not persist.
  transient int currentTemperature;     // (1) Transient value.

  double mass;                          // (2) Persistent value.

}
```

Specifying the `transient` modifier for static variables is redundant and, therefore, discouraged. Static variables are not part of the persistent state of a serialized object.

`volatile` Fields

During execution, compiled code might cache the values of fields for efficiency reasons. Since multiple threads can access the same field, it is vital that caching is not allowed to cause inconsistencies when reading and writing the value in the field. The `volatile` modifier can be used to inform the compiler that it should not attempt to perform optimizations on the field, which could cause unpredictable results when the field is accessed by multiple threads (see also Example 13.5, p. 644).

In the simple example below, the value of the field `clockReading` might be changed unexpectedly by another thread while one thread is performing a task that involves always using the current value of the field `clockReading`. Declaring the field as `volatile` ensures that a write operation will always be performed on the master field variable, and a read operation will always return the correct current value.

```
class VitalControl {
  // ...
  volatile long clockReading;
  // Two successive reads might give different results.
}
```

Table 4.5 *Summary of Other Modifiers for Members*

Modifiers	Fields	Methods
`static`	Defines a class variable.	Defines a class method.
`final`	Defines a constant.	The method cannot be overridden.
`abstract`	Not applicable.	No method body is defined. Its class must also be designated `abstract`.
`synchronized`	Not applicable.	Only one thread at a time can execute the method.
`native`	Not applicable.	Declares that the method is implemented in another language.
`transient`	The value in the field will not be included when the object is serialized.	Not applicable.
`volatile`	The compiler will not attempt to optimize access to the value in the field.	Not applicable.

Review Questions

4.19 Which statements about the use of modifiers are true?

Select the two correct answers.

(a) If no accessibility modifier (`public`, `protected`, or `private`) is specified for a member declaration, the member is only accessible 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 only accessible within 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 `static`.
(e) The objects themselves do not have any accessibility modifiers, only the object references do.

4.20 Given the following source code, which comment line can be uncommented without introducing errors?

```
abstract class MyClass {
  abstract void f();
  final    void g() {}
//final    void h() {}                          // (1)

  protected static int i;
  private          int j;
}

final class MyOtherClass extends MyClass {
//MyOtherClass(int n) { m = n; }                // (2)

  public static void main(String[] args) {
    MyClass mc = new MyOtherClass();
  }

  void f() {}
  void h() {}
//void k() { i++; }                             // (3)
//void l() { j++; }                             // (4)

  int m;
}
```

Select the one correct answer.

(a) (1)
(b) (2)
(c) (3)
(d) (4)

4.21 What would be the result of compiling and running the following program?

```
class MyClass {
  static MyClass ref;
  String[] arguments;

  public static void main(String[] args) {
    ref = new MyClass();
    ref.func(args);
  }

  public void func(String[] args) {
    ref.arguments = args;
  }
}
```

Select the one correct answer.

(a) The program will fail to compile, since the static method `main()` cannot have a call to the non-static method `func()`.
(b) The program will fail to compile, since the non-static method `func()` cannot access the static variable `ref`.
(c) The program will fail to compile, since the argument `args` passed to the static method `main()` cannot be passed to the non-static method `func()`.
(d) The program will compile, but will throw an exception when run.
(e) The program will compile and run successfully.

4.22 Given the following member declarations, which statement is true?

```
int a;                          // (1)
static int a;                   // (2)
int f() { return a; }           // (3)
static int f() { return a; }    // (4)
```

Select the one correct answer.

(a) Declarations (1) and (3) cannot occur in the same class declaration.
(b) Declarations (2) and (4) cannot occur in the same class declaration.
(c) Declarations (1) and (4) cannot occur in the same class declaration.
(d) Declarations (2) and (3) cannot occur in the same class declaration.

4.23 Which statement is true?

Select the one correct answer.

(a) A static method can call other non-static methods in the same class by using the `this` keyword.
(b) A class may contain both static and non-static variables, and both static and non-static methods.
(c) Each object of a class has its own instance of the static variables declared in the class.
(d) Instance methods may access local variables of static methods.
(e) All methods in a class are implicitly passed the `this` reference as argument, when invoked.

4.24 What, if anything, is wrong with the following code?

```
abstract class MyClass {
  transient int j;
  synchronized int k;

  final void MyClass() {}

  static void f() {}
}
```

Select the one correct answer.

(a) The class `MyClass` cannot be declared `abstract`.
(b) The field `j` cannot be declared `transient`.
(c) The field `k` cannot be declared `synchronized`.
(d) The method `MyClass()` cannot be declared `final`.
(e) The method `f()` cannot be declared `static`.
(f) Nothing is wrong with the code; it will compile successfully.

4.25 Which one of these is not a legal member declaration within a class?

Select the one correct answer.

(a) `static int a;`
(b) `final Object[] fudge = { null };`
(c) `abstract int t;`
(d) `native void sneeze();`
(e) `final static private double PI = 3.14159265358979323846;`

4.26 Which statements about modifiers are true?

Select the two correct answers.

(a) Abstract classes can declare `final` methods.
(b) Fields can be declared `native`.
(c) Non-`abstract` methods can be declared in `abstract` classes.
(d) Classes can be declared `native`.
(e) Abstract classes can be declared `final`.

4.27 Which statement is true?

Select the one correct answer.

(a) The values of `transient` fields will not be saved during serialization.
(b) Constructors can be declared `abstract`.
(c) The initial state of an array object constructed with the statement `int[] a = new int[10]` will depend on whether the array variable a is a local variable or a field.
(d) A subclass of a class with an `abstract` method must provide an implementation for the `abstract` method.
(e) Only static methods can access static members.

Chapter Summary

The following information was included in this chapter:

- the structure of a Java source file
- defining, using, and deploying packages
- explanation of class scope for members, and block scope for local variables
- discussion of accessibility (default, `public`) and other modifiers (`abstract`, `final`) for reference types
- applicability of member accessibility (default, `public`, `protected`, `private`) and other member modifiers (`static`, `final`, `abstract`, `synchronized`, `native`, `transient`, `volatile`)

Programming Exercise

4.1 Design a class for a bank database. The database should support the following operations:

- deposit a certain amount into an account
- withdraw a certain amount from an account
- get the balance (i.e., the current amount) in an account
- transfer an amount from one account to another

The amount in the transactions is a value of type `double`. The accounts are identified by instances of the class `Account` that is in the package `com.megabankcorp.records`. The database class should be placed in a package called `com.megabankcorp.system`.

The deposit, withdraw, and balance operations should not have any implementation, but allow subclasses to provide the implementation. The transfer operation should use the deposit and withdraw operations to implement the transfer. It should not be possible to alter this operation in any subclass, and only classes within the package `com.megabankcorp.system` should be allowed to use this operation. The deposit and withdraw operations should be accessible in all packages. The balance operation should only be accessible in subclasses and classes within the package `com.megabankcorp.system`.