Chapter 4

Information Hiding and Design Principles

In an object-oriented program, objects of different classes interact in many different ways. Good software engineering practice recommends that these interactions be properly controlled; otherwise, the program will be difficult to understand, debug, and maintain. *Information hiding* is the process of hiding all the secrets of an object that do not contribute to its essential characteristics. This chapter explains how information hiding can be achieved in object-oriented programming. It also discusses some design principles that are pertinent to the development of object-oriented software.

4.1 Encapsulation

The previous chapter has shown how to combine the attributes and behaviors of an object into a single unit called a class. Attributes and behaviors are said to be *encapsulated* in the class. Encapsulation provides the first level of information hiding in an object-oriented application since it separates the external aspects of an object (its interface) from the internal implementation details of the object. Another level of information hiding is to control access to the individual attributes and methods of an object by the use of access modifiers. This kind of information hiding is the subject of this section.

4.1.1 Access Modifiers

Access modifiers are used in Java to control access to the individual members of a class such as its attributes and methods. There are three access modifiers that can be used when declaring the attributes and methods of a class:

- `private`
- `protected`
- `public`

↓ Increasing level of access

These access modifiers permit an increasing amount of access going down the list. For example, `private` members are accessible only by objects of the same class. However, `public` members can be accessed by objects of any other class. If none of the above access modifiers is used to declare a member of a class, Java provides a default level of access to the member being declared. The following sub-sections describe how access modifiers are used to control access to the members of a class.

4.1.2 Hiding the Attributes of an Object

It is a good idea to hide the attributes of an object from its clients. When this is done, client code will not be able to directly access the value of an instance variable. Direct access implies that a client can get hold of an object through an object variable and then use the variable name, followed by a dot, followed by the name of the instance variable, to either read or change the variable. For example, suppose that `a` is a reference to an `Account` object. Direct access to the `balance` attribute of an `Account` object means that a client can do things such as the following:

```
double currentBalance = a.balance;   // client can read balance
a.balance = 10000.00;                // client can change balance
```

Giving clients uncontrolled access to the `balance` attribute makes code maintenance very difficult. If there is a change in the `Account` class involving the `balance` attribute, it is necessary to find and change all the clients which may have accessed this attribute directly. Debugging is also more difficult since an error involving the `balance` attribute requires several other classes to be examined. If no client is able to directly access an attribute of another class, changes involving that attribute are isolated to its own class, simplifying program maintenance and debugging. This is the reason why clients should not be able to directly access the values of attributes as in the examples above.

Access modifiers are used to selectively control access to each attribute of an object. An attribute declared as `private` is only visible to the methods of the object. For example, the instance variable `balance` of the `Account` class can be declared `private` as follows:

```
private double balance;
```

Since the `balance` instance variable is declared as `private`, it can only be seen and manipulated by methods of the `Account` class. For example, the `deposit()` method of the `Account` class adds the value of the `amount` parameter to the current `balance` and updates the `balance` attribute to this value:

```
public void deposit(double amount) {
   balance = balance + amount;    // read and update balance
}
```

It is not possible for clients of an object to view an instance variable that has been declared as `private`. Thus, client code such as the following will not compile successfully since the client does not have direct access to the `balance` instance variable:

```
Account a;                           // declaration OK
a = new Account(10, 1000.00);        // creation of Account instance OK

System.out.println("Account balance: " + a.balance);
                                     // cannot access balance directly
```

The access permitted to a `public` instance variable is completely opposite to that which is permitted to a `private` instance variable. A `public` instance variable is visible to objects of every class and thus can be seen and modified directly by these objects. For example, suppose that the `balance` instance variable of `Account` is declared `public` as follows:

```
public double balance;
```

It is now possible for a client object of some other class to create an instance of `Account` and then directly change the value of `balance`. This can be done as follows:

```
Account a;

a = new Account(10, 1000.00);          // create account instance

System.out.println("Old balance: " + a.balance);
                                       // read value of balance

a.balance = 5000.00;                   // change value of balance

System.out.println("New balance: " + a.balance);
                                       // read value of balance
```

Allowing client objects to directly access the `balance` instance variable is not recommended for the reasons given earlier. In addition, it is possible for client code to change the variable in undesirable ways. For example, clients may not be aware of validation checks that have to be performed before changing an instance variable (e.g., ensuring that the `balance` is within a certain range). As part of the practice of information hiding, instance variables are normally declared as `private`.

The third access modifier, `protected`, is discussed later in this chapter after the section on packages. When no access modifier is specified, a default type of access is permitted referred to as *friendly* or *package-private*. Recall that the instance variables of the `Account` class discussed in Chapter 3 were not declared with access modifiers. Thus, the type of access permitted is *friendly*. *Friendly* access is discussed later in the chapter.

4.1.3 Hiding the Methods of an Object

The access modifiers `private`, `protected` and `public` can also be used to control access to methods, i.e., the services offered by an object. A `private` method defined in a class can only be invoked by methods of that class. This type of access to a method is normally used if the method provides services that are not required by objects of other classes. This often happens when a method performs a task that is not particularly useful to other classes (e.g., "helper" methods).

A client object can invoke a `public` method by using the name of the object variable which refers to the object, followed by a dot, followed by the method name and list of arguments (if any). We saw numerous examples of this in Chapter 3 in the `BankApplication` client code. For example,

```
Account a;                    // declare object variable

a = new Account();            // let a refer to object created
a.setNumber(10);              // invoke setNumber() method of Account
a.deposit(1000.00);           // invoke deposit() method of Account

System.out.println(a.toString());
                              // invoke toString() method of Account
```

The methods of a class are generally declared as `public`, since their purpose is to provide services to other objects. For example, the methods of the `Account` class in Chapter 3 are all declared as `public`.

`Protected` methods are discussed later in this chapter, after the section dealing with packages. When no access modifier is specified when declaring a method, a default type of access is permitted referred to as *friendly* or *package-private*. Methods that provide *friendly* access are discussed later in the chapter.

It should be noted that clients do not need to be aware of how a method is implemented (i.e., the logic and data structures that are used to achieve the behavior of the method). Clients are only responsible for invoking methods with the appropriate arguments. As a result, when a client object requests a service from another object, the service is provided, yet the client is unaware of what is involved in providing that service. In other words, clients should know the interface to an object (i.e., they should know how to access its services), but they should not be concerned with the implementation of these services. This enables the service provider to alter the way its services are provided without clients being affected.

4.1.4 Effect of Access Modifiers on Instances of the Same Class

Access modifiers determine the extent to which client code can read and/or modify the individual members of a class. However, in Java, access modifiers have no effect when a client object belongs to the same class as the object being manipulated. Thus, an instance of a class is always permitted complete access to the attributes and methods of another object of the *same* class, regardless of the access modifiers used to declare these members. For example, consider the following method of the `Account` class:

```
public boolean isEqual(Account account) {
   if (this.number == account.number)
      return true;
   else
      return false;
}
```

The `isEqual()` method returns `true` if the `Account` parameter has the same `number` as the object on which the `isEqual()` method is acting (the `this` object). Observe how it is possible to refer directly to the `number` variable of the `Account` parameter despite the fact that the `number` variable was declared `private` in `Account`:

```
if (this.number == account.number)
   // direct access to number attribute permitted
```

4.2 Accessing Private Attributes

It is not possible for a client to directly access a `private` attribute of a class. However, in an object-oriented program, a client will often need to view or modify the value of a `private` attribute. Thus, a class must provide a means for client objects to view and/or modify the value of a `private` attribute. This is achieved by writing special methods that allow client objects to read or change the values of instance variables. These methods are called accessor and mutator methods, respectively.

4.2.1 Accessor and Mutator Methods

An *accessor* method simply returns the value of an instance variable. It is sometimes called a *getter*. An example is the `getBalance()` method of the `Account` class which returns the current value of the `balance` attribute. A *mutator* method changes the value of an instance variable. The simplest mutator method is one that accepts a parameter and changes the value of an instance variable to the value of the parameter. This type of mutator is often called a *setter*. An example is the `setNumber()` method of the `Account` class (before constructor methods were introduced). This method changes the value of the `number` attribute of an `Account` object to a new value supplied as an argument. Mutators may be more complex such as the `withdraw()` method of the `Account` class which changes the value of the `balance` attribute under certain conditions.

Getter methods are usually named by preceding the name of the instance variable with the prefix "*get*"; the first letter of the variable name is also capitalized. For example, `getBalance()` returns the value of the `balance` attribute of an `Account` object. A getter method for an instance variable `x` of type `T` is generally written as follows:

```
public T getX() {       // x is the instance variable, T is its type
   return x;            // simply return value of x
}
```

If `x` is of type `boolean`, the getter method is written with an "*is*" prefix instead of "*get*" as follows:

```
public boolean isX() {  // x is an instance variable of type boolean
   return x;            // return value of x
}
```

`isX()` is a more natural name to find out the value of a `boolean` variable. For example, if `empty` is an instance variable of type `boolean`, it is more readable to name the accessor method `isEmpty()` rather than `getEmpty()`.

Setter methods are similarly named, except that the prefix "*set*" is used. For example, `setNumber(n)` changes the value of the `number` attribute of an `Account` object to `n`. The new value is supplied as an argument to the setter method.

A setter method for an instance variable `x` of type `T` is generally written as follows:

```
public void setX (T newValue) {
                      // x is the instance variable, T is its type

   x = newValue;      // change x to value of parameter
}
```

Consider Figure 4.1 which is a UML diagram of a `Customer` class:

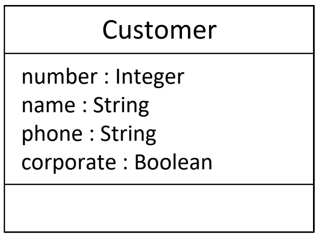

Figure 4.1: UML Diagram of a `Customer` Class

Getter methods for all the instance variables of the `Customer` class are given below:

```
public int getNumber() {
   return number;
}

public String getName() {
   return name;
}

public String getPhone() {
   return phone;
}
```

```
public boolean isCorporate() {
   return corporate;
}
```

The corresponding setter methods for the `Customer` class are given below:

```
public void setNumber(int number) {
   this.number = number;
}

public void setName(String name) {
   this.name = name;
}

public void setPhone(String phone) {
   this.phone = phone;
}

public void setCorporate(boolean corporate) {
   this.corporate = corporate;
}
```

A setter method can apply validation checks before modifying the value of an instance variable. For example, suppose there is a requirement that the `name` attribute of `Customer` should be at most fifteen characters long. This constraint can be implemented in the `setName()` method as follows:

```
public void setName(String name) {
   if (name.length() > 15)
      this.name = name.substring(0, 15);
                // take first 15 characters & ignore the rest

   else
      this.name = name;
                // take all the characters
}
```

It should be noted that accessor and mutator methods are not normally shown in a UML diagram. Also, some integrated development environments (IDEs) and tools that generate code from UML diagrams automatically generate getter and setter methods for all the attributes of a class. This feature should be used with caution, as the next section explains.

4.2.2 Problem with Accessor and Mutator Methods

It is not necessary to have getter and setter methods for all the attributes of a class. Getter and setter methods should only be provided for attributes when it is expected that client objects will need to view and/or modify the values of these attributes. If a client object only needs to view the value of a particular attribute, only a getter method should be provided for that attribute. Care must also be taken in providing a setter method for an attribute which acts as a primary key. If this attribute is modified, it will become disconnected from other objects which are still referring to the old primary key.

It should be noted that getter and setter methods violate information hiding to a certain degree. Consider the `number` instance variable of the `Customer` class. This variable is of type `int`. So, the getter method returns a value of type `int` and the setter method accepts a parameter of type `int`. Suppose a decision is made later to change the type of the variable to `String`. This requires changes in the getter and setter methods to accommodate the new type. More importantly, every client object which uses the getter and setter methods must change to deal with the new type as well. This happens because the getter and setter methods did not completely encapsulate all the details of the `number` attribute so client objects were not shielded from modifications to this setter methods should be used with caution.

4.3 Immutable Classes

If a class has at least one mutator (either a setter method or a more complex method that modifies the value of an instance variable) it is said to be *mutable*. If a class has no mutators and all its instance variables are `private`, it is impossible for clients to modify objects of that class after they have been created. Such a class is called *immutable*. Objects of an immutable class can be freely shared with other client objects without the danger of the objects being modified. Some authors suggest that you should make a class immutable whenever you can do so (Horstmann, 2002).

4.3.1 Achieving Immutability

To ensure that an object is immutable, its attributes should be declared as `final` in the class definition. For example, the attributes of the `Customer` class can be declared `final` as follows:

```
private final int number;
private final String name;
private final String phone;
private final boolean corporate;
```

However, when this is done, the attributes can only be modified in the constructor. Thus, the four setter methods must be removed from the `Customer` class.

It is possible to assign a value to a `final` attribute in the declaration statement itself. For example,

```
   private final int number = 100;   // assigns 100 to number
```

Once this is done, `number` cannot be subsequently modified, not even in a constructor method. Clearly this approach for setting the value of a `final` attribute has very limited use. So, the recommended approach for creating an immutable class is to declare all the instance variables as `final` and use a constructor to set the instance variables. Since the attributes are `final`, Java ensures that no subsequent modification of the instance variables takes place.

4.3.2 Immutable Classes in Java

There are several classes in Java that are immutable. For example, the `String` class is immutable. Consider the following code which creates a `String` object:

```
String greeting = "Hello There!";
```

An equivalent `String` object can be created with the following code:

```
String greeting = new String("Hello There!");
```

The object variable `greeting` refers to a `String` object which contains twelve characters. It is impossible to modify the characters in the `String` object. However, the object variable can be re-assigned to another `String` object as follows:

```
greeting = "Bye!";
```

Eventually, the `String` object corresponding to the string "Hello There!" will be removed from the memory space of the application, if no other variable refers to it.

Consider the following code:

Line 1:
```
greeting = "John";
```

Line 2:
```
greeting = "Hello there " + greeting + "!";
```

It might seem as if the `String` object referred to by `greeting` is being modified in Line 2. However, what is really happening is that a new `String` object is created after applying the concatenation operator on the right hand side of the equals sign. A reference to this new `String` object is assigned to `greeting` in Line 2. Thus, the original `String` object containing “John” remains unmodified in memory.

Java also has a set of *wrapper* classes for each of the eight primitive types. A wrapper class is used in situations where an object is required instead of a primitive value. Objects of the wrapper classes are immutable.

Suppose that `x` is a variable of type `int`. To create an *object* which contains the value of `x`, the `Integer` wrapper class is used. For example,

```
int x = 10;
Integer xWrapper = new Integer(x);
```

Once the instance `xWrapper` is created, its contents cannot be modified. The only way to change `xWrapper` is to create another instance of `Integer`. For example,

```
x = 25;
xWrapper = new Integer(x);
```

Methods such as `intValue()` can be used to obtain the value of the `int` stored in `xWrapper`. For example,

```
System.out.println("Value of x: " + xWrapper.intValue());
```

The wrapper classes for the eight primitive types are listed in Table 4.1. A constructor method for each class is also given, together with the method used to retrieve the primitive value stored in a wrapper object.

Primitive Type	*Wrapper Class*	*Constructor*	*Method to Retrieve Primitive Value*
byte	Byte	Byte(byte b)	byteValue()
short	Short	Short(short s)	shortValue()
int	Integer	Integer(int i)	intValue()
long	Long	Long(long l)	longValue()
float	Float	Float(float f)	floatValue()
double	Double	Double(double d)	doubleValue()
char	Character	Character(char c)	charValue()
boolean	Boolean	Boolean(boolean b)	booleanValue()

Table 4.1: Wrapper Classes in Java

In passing, it should be mentioned that the wrapper classes in Table 4.1 have class methods which can be used to convert a primitive value stored as a string to a value of the corresponding primitive type. For example, the `parseInt()` method of the `Integer` class can be used to convert the string "25" to an `int` value:

```
String numberString = "25";
int accountNumber = Integer.parseInt(numberString);
```

The methods of the other wrapper classes have similar names. These conversion methods will often be used in the book to obtain the primitive value stored in a string.

4.4 Object-Oriented Design Guidelines

This section discusses some design guidelines for building an object-oriented application. It explains how coupling and cohesion, two fundamental concepts in software design, are applicable to the design of an object-oriented application. It also describes the Law of Demeter, a guideline for reducing the coupling among objects.

4.4.1 Coupling and Cohesion

An object-oriented program consists of a set of objects that collaborate to achieve some goal. Two objects collaborate when one object requests a service from the other. As mentioned at the beginning of this chapter, objects of different classes interact in many different ways and it is important to control these interactions to create high quality programs. Two notions that affect the quality of object-oriented programs are *coupling* and *cohesion*.

Object-oriented coupling refers to the degree or strength of interconnection among classes. Loosely coupled classes (as opposed to tightly coupled classes) are desirable since each class can be handled in a relatively independent manner. A class can become coupled to another class, `C`, if it knows about `C` in some way. For example, it may create an object of `C` or one of its methods may accept an object of class `C` as a parameter. An object-oriented program with loosely coupled classes facilitates:

- The replacement of one class by another so that only a few classes are affected by the change
- The speedy debugging of errors since it is easier to track down an error and isolate the defective class causing the error

One extreme in coupling is to have the classes in an application totally uncoupled. When this is done, there is no way for objects to collaborate with each other. This defeats the fundamental notion of an object-oriented program. The other extreme is to have all the classes coupled to each other so there is a high degree of dependence between each pair of classes. This makes it extremely difficult to replace classes and to debug and maintain programs. Between these two extremes, there are many degrees of coupling. A desirable goal in object-oriented programming is to reduce any excess or unnecessary coupling among classes. This reduces maintenance and debugging costs and promotes reusability of classes.

The notion of *cohesion* concerns the internal strength of a class, i.e., how strongly related the parts of a class are. A class whose parts are strongly related to each other and to the concept being represented is said to be *strongly cohesive*. In contrast, a class whose parts are hardly related to each other is said to be *weakly cohesive*. Strongly cohesive classes are easier to understand, debug, and maintain.

An example of a weakly cohesive class is one where the class embodies more than one unrelated concept and thus takes on responsibilities that could be best handled by another class. For example, if customer information such as customer name and telephone number are stored in an `Account` class, then the `Account` class is weakly cohesive.

4.4.2 Law of Demeter

The *Law of Demeter* is a guideline for reducing the amount of coupling between objects in an object-oriented application. Consider again the `Account` class. The `Account` class has two attributes, `number` and `balance`. Suppose we wanted an `Account` object to store information on the customer who owns the account. An additional attribute of `Account` can be used to store a reference to the corresponding `Customer` object. This attribute is called `customer` and is of type `Customer`. Figure 4.2 is a UML diagram showing the enhanced `Account` class.

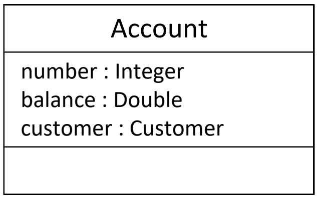

Figure 4.2 UML Diagram of the Enhanced `Account` Class

A getter method, `getCustomer()`, can be written in `Account` that returns the `Customer` instance:

```
public Customer getCustomer() {
   return customer;
}
```

Now, suppose that a client object obtains a reference to an `Account` object. Let this reference be `a`. The client object can then invoke the `getCustomer()` method on `a` as follows:

```
Customer accountHolder;
accountHolder = a.getCustomer();
```

The client object now has a reference to a `Customer` object in its hands. Since `Customer` is a mutable class, the client object is free to invoke any of its setter methods and modify the `Customer` object as it pleases. If this is done, it means that an attribute of `Account` (i.e., its `Customer` object) can be modified without the `Account` object knowing about it. This is a violation of encapsulation. The `getCustomer()` method was designed to give information about the `Customer` object, not to permit modification of it.

The problem just described can be reduced or eliminated by following the *Law of Demeter*. This law states that a method `m()` of an object `o` should only invoke methods from the following kinds of objects:

- Methods from `o` itself
- Methods from objects passed to `m()` as arguments
- Methods of objects created within `m()`

A client object that follows the Law of Demeter does not invoke methods on objects that are part of another object. Thus, the client object should not operate on the `Customer` instance returned by the `getCustomer()` method since this violates the Law of Demeter. The Law of Demeter is sometimes called the *Principle of Least Knowledge* since its purpose is to make objects know as little as possible about other types of objects.

The Law of Demeter is not a natural or mathematical law. It is simply a guideline to be considered when designing an object-oriented program. Following the guideline often requires client objects to request services from server objects instead of directly manipulating objects managed by the server objects (e.g., the `Customer` instance above). Server objects must be enlarged to provide these services. The next section gives a few more guidelines for developing an object-oriented application.

4.5 Organizing the Classes of an Application

An object-oriented application usually consists of several layers of software where each layer is composed of several classes. As the complexity of the application increases, it is convenient to organize the classes into different units. These units are known as packages in Java. This section describes the concept of a package and explains how a package can be created and manipulated. The section also presents a three-layered architecture of an object-oriented application in which the layers consist of a set of packages which interact with each other.

4.5.1 The Concept of a Package

A *package* is a way of grouping related classes into a single unit. So, instead of an object-oriented program consisting all of its classes lumped together in one place, it can consist of a group of packages each containing a set of related classes.

So far in this book, all the classes that have been created reside in a single unit. This unit is referred to as the *default package* and corresponds to the folder on your computer where the classes are stored and compiled. Since no package was specified when the classes were created, they were automatically placed in the default package. However, as the number of classes in an application increases, it is more convenient to group the classes into packages.

Suppose in a banking application it is decided to keep all the classes related to accounts in a package called `CustomerAccount`. `CustomerAccount` may contain classes such as `Account`, `SavingsAccount`, `ChequingAccount`, `MutualFundAccount`, `FixedDepositAccount`, etc. Figure 4.3 shows how the `CustomerAccount` package can be drawn using the UML. The package diagram is drawn as a tabbed folder where the name of the package is specified in the tab and the folder specifies the names of the classes which belong to the package".

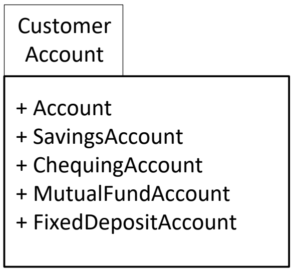

Figure 4.3: UML Diagram of `CustomerAccount` Package

The `public` parts of a package are called its exports (i.e., its class members declared with the `public` keyword). The '+' symbol preceding each class name indicates that the class is exported from `CustomerAccount` and is thus available to classes in other packages.

Packages promote information hiding since the members of the classes in one package (instance variables and methods) can be selectively exported to other packages. Thus, there may be elements in one package that are not accessible by other packages.

The notions of coupling and cohesion are also applicable to the decomposition of an object-oriented application into packages. A desirable goal is to have loosely coupled packages. Thus, packages should be independent as far as possible. The classes within a package should also be highly cohesive. Thus, the classes in a package should be related to each other as much as possible.

4.5.2 Three-tier Architecture for Object-Oriented Software

Except for small applications, an object-oriented program does not usually consist of one layer of software grouped together in one unit or package. It is common to employ a three-layered architecture when designing object-oriented programs. This is known as the *three-tiered architecture* and consists of the following three vertical layers:

1. User Services—visual interface for presenting information and gathering data
2. Business Services—tasks and rules that govern application processing
3. Data Services—persistent storage mechanism to maintain, access, and update data

The unique quality of the three-tier architecture is the separation of the application (business) logic into a distinct logical middle tier of software. The *User Services* tier is relatively free of application processing and forwards task requests to the *Business Services* tier. The *Business Services* tier communicates with the back-end *Data Services* tier. The classes in the *Business Services* tier will often be referred to as *domain classes* in the book

The three-tier architecture for an object-oriented application can be designed as a set of packages as shown in Figure 4.4 below. Depending on the complexity of different applications, the packages may be further decomposed into sub-packages representing finer layers in the architecture.

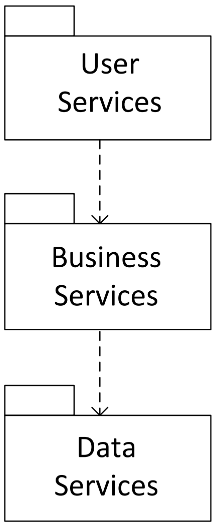

Figure 4.4: An Object-Oriented Program as a Set of Packages

Note that the dashed lines between packages indicate a dependency relationship between the packages (this type of relationship is described further in the next chapter). So, the `UserServices` package depends on the `BusinessServices` package which, in turn, depends on the `DataServices` package.

4.5.3 Creating a Package and its Elements

Suppose that the `Account` class must be part of the `BusinessServices` package. This must be indicated at the top of the source file containing `Account`. The keyword `package` is used followed by the name of the package in lower case:

```
package business;        // 'services' has been omitted from the name
```

The folder structure corresponding to the packages in an application is shown in Figure 4.5. Notice that all the folders are sub-folders of the top level `application` folder.

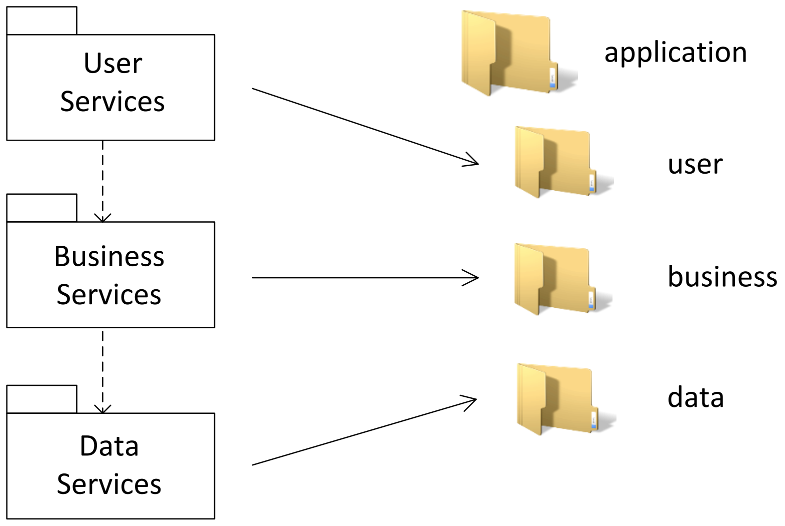

Figure 4.5: Correspondence between Packages and Folders

After the `Account` class has been compiled, the class file must be placed in the `business` folder. Since the `BankApplication` class from Chapter 3 belongs to the `UserServices` package, it must be placed in the `user` folder. The `package` statement must be placed at the top of the source file to indicate that the class belongs to the `UserServices` package:

```
package user;            // 'services' has been omitted from the name
```

The next section discusses the changes that are necessary in `BankApplication` so that it can access the `Account` class in the `BusinessServices` package.

4.5.4 Using the Elements of a Package

The `Account` class is now in the `BusinessServices` package. Classes in the same package can access each other depending on the levels of access permitted. For example, suppose that a `Bank` class belongs to the `BusinessServices` package. An instance of `Bank` can create an instance of `Account` in the normal manner:

```
Account a;
a = new Account(10, 1000.00);
```

However, consider the `BankApplication` class which has been placed in the `UserServices` package. The `BankApplication` class needs to create and manipulate instances of the `Account` class which now resides in the `business` folder. To do so, it needs to specify the full name and path of the package containing `Account`, relative to the root `application` folder. Thus, the `BankApplication` class requires the following code in order to create an instance of `Account`:

```
business.Account a;
a = new business.Account(10, 1000.00);
```

To access the `Account` class, the `BusinessServices` package (abbreviated to `business`) must be specified, followed by a dot, followed by the name of the class. This can be rather cumbersome. The code can be considerably simplified by placing the path to the `Account` class in an `import` statement. The `import` statement that is required to use the `Account` class is written as follows:

```
import business.Account;
```

If there are other classes from the `BusinessServices` package that will be needed by the `BankApplication` class (or some other client), each one can be listed in a separate `import` statement. Alternatively, the following statement can be used to import all the classes from the `BusinessServices` package:

```
import business.*;   // '*' specifies all the classes in the package
```

The `import` statement must be placed at the top of the source file, before the client class is declared. Now, whenever the client class wishes to access the `Account` class, it simply uses the name of the class. For example,

```
Account a;
a = new Account(10, 1000.00);
```

4.5.5 Nesting Packages

It is possible for packages to be nested. For example, the `BusinessServices` package may contain three sub-packages called `CustomerAccount`, `Loan`, and `Mortgage`. Each of the sub-packages contains classes that manage the business operations for each category of business in a bank. This is shown in the UML diagram in Figure 4.6:

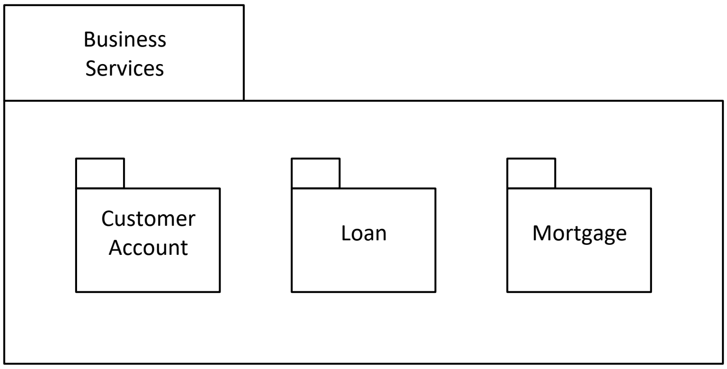

Figure 4.6 Nested Packages

Nested packages correspond to a nested folder structure. So, the three sub-packages would be stored as sub-folders of the `business` folder. Figure 4.7 shows the folder structure corresponding to the nested packages:

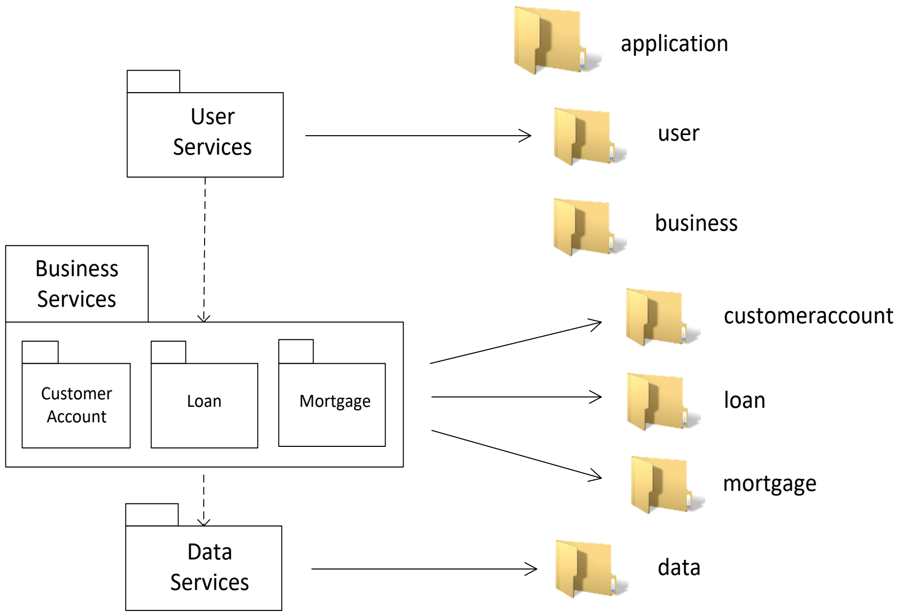

Figure 4.7: Folder Structure Corresponding to Nested Packages

Suppose that the `Account` class must now belong to the `CustomerAccount` sub-package. The `package` statement in `Account` must change to reflect this:

```
package business.customeraccount;
```

After compiling the `Account` class, it must be placed in the `customeraccount` sub-folder of `business`. A similar procedure must be followed for all the classes belonging to a particular package.

If the `BankApplication` class needs to create an instance of the `Account` class, the following code is required:

```
business.customeraccount.Account a;
a = new business.customeraccount.Account(10, 1000.00);
```

Again, the code in `BankApplication` can be simplified by placing the path to the `customeraccount` package in an `import` statement. The `import` statement required to use the `Account` class is written as follows:

```
import business.customeraccount.Account;
```

Now, whenever the `BankApplication` class wishes to access the `Account` class, it simply uses the name of the class, as before.

4.6 Accessing the Attributes and Methods of a Class in a Package

Within a package, the `private` and `public` access modifiers permit the same level of access to the attributes and methods of a class as previously described. However, if no access modifier is used to declare an instance variable of a class, client objects in the *same* package are automatically granted *friendly* (*package-private*) access to this variable. For example, the following statement declares *name* to be a *friendly* instance variable of the `Customer` class:

```
String name;                    // no access modifier specified
```

Granting *friendly* access to an instance variable is equivalent to granting `public` access to all objects in the same package. So, declaring an instance variable with no access modifier is equivalent to declaring that instance variable as `public`; however, `public` access is restricted to classes in the same package. The instance variable is `private` to all classes outside the package.

Similarly, if no access modifier is used to declare a method of a class, client objects in the same package are automatically granted *friendly* access to this method. For example, if the `setName()` method of the `Customer` class is defined as follows, then `setName()` is a *friendly* method:

```
void setName (String name) {    // no access modifier specified
   this.name = name;
}
```

Declaring a method with no access modifier is equivalent to declaring that method as `public`; however, `public` access is restricted to classes in the same package. The method is `private` to all classes outside the package.

If an instance variable is declared as `protected`, it is accessible to all the classes in the same package. Thus, a `protected` instance variable permits the same type of access as a *friendly* instance variable. Similarly, a method declared as `protected` provides the same type of access as a *friendly* method. An example of a `protected` instance variable is the `number` attribute of the `Customer` class which is declared as follows:

```
protected int number;
```

`protected` instance variables are different from *friendly* instance variables since they can also be accessed by objects of another kind of class. This type of class is referred to as a *subclass* and is discussed further in Chapter 9.

4.7 Controlling Access to a Class

This chapter has shown how access modifiers permit different degrees of access to the individual members of a class such as its attributes and methods. It is also possible to control access to the class as a whole using the same set of access modifiers. In addition, it is possible to declare a class as an inner class to restrict its visibility to client objects.

4.7.1 Using Access Modifiers to Declare a Class

A class is declared using the `public` access modifier. This makes the class accessible to objects of any other class. It is not possible to declare a class as `private` or `protected` (except for inner classes which are discussed in the next sub-section). If no access modifier is used to declare a class, *friendly* (package-private) access is permitted. *Friendly* access means that objects of the same package can access the class. Two examples of a class declaration are as follows:

```
public class Account { … }            // public access permitted
class Account { … }                   // friendly access permitted
```

4.7.2 Inner Classes

An *inner class* is a class that is written inside another class. An inner class is written the same way as any other class. However, an inner class is normally declared as `private` so that it can only be accessed by the class in which it is contained. Inner classes provide a level of encapsulation that is similar to `private` methods which are written for the exclusive use of the class in which they are contained. Thus, a `private` inner class is written for the sole benefit of the containing class.

The only time an inner class is used in this book is in Chapter 13, where a `LinkedList` class contains a `private` inner class called `Node`. The following code is part of the `LinkedList` class:

```
public class LinkedList
{
   private Node head;         // first node of the linked list
   private int count;         // amount of nodes in the linked list
   public LinkedList() {
      head = null;
      count = 0;
   }

   public void addFirst(int element) {
      Node newNode = new Node(element);
      newNode.next = head;    // can access next attribute of Node
      head = newNode;
      count++;
   }

   // Other linked list methods

   // Node inner class

   private class Node {
      private int element;
      private Node next;

      public Node(int element) {
         this.element = element;
      }
   }
}
```

As can be seen from the `addFirst()` method in the code above, the class containing the inner class (`LinkedList`) can access the attributes and methods of the inner class (`Node`) regardless of the access modifiers used to declare these members. Similarly, an inner class can access the attributes and methods of the containing class regardless of the access modifiers used to declare these members (not shown in the code above).

To reduce coupling, it is better for the containing class to be coupled to the inner class only rather than for both classes to be coupled to each other. However, this is unavoidable at times if the inner class needs to access data from its containing class.

An inner class can be declared using the `protected` or `public` access modifiers. This permits other objects to access the inner class using the name of the outer class, followed by a dot, followed by the name of the inner class. For example, assume that the `Node` inner class was declared `protected`. This means that a client in the same package can declare an object variable to access a `Node` instance as follows:

```
LinkedList.Node node;
```

The client can then obtain a reference to an instance of `Node` or create a `Node` object on its own. However, the coding is not only cumbersome and error-prone, but it also assumes that a client knows what to do with a `Node` object. If it is intended that an inner class should be accessible to other clients, it is probably better to make it an independent class.

Exercises

1. Explain why the attributes of a class are usually `private` and the methods are usually `public`. Of what use is a `private` method? What is an appropriate use of a `public` instance variable?

2. Why should a class provide accessor and mutator methods? Is it necessary to provide accessor and mutator methods for *all* the attributes of a class? If not, which attributes should be omitted?

3. What is an inner class? What are the implications of declaring an inner class as `public`?

4. What do you think will happen if all the constructors of a class are declared with the `private` access modifier?

5. The attributes and methods of a certain class are declared without any access modifiers. Explain the type of access that will be permitted to objects in the same package and to objects in other packages.

6. What are the advantages of having low coupling among classes? What are the disadvantages of having weakly cohesive classes?

7. Describe the three-tiered architecture for an object-oriented application and explain how the classes of the application can be partitioned into packages.

8. Suppose that an attribute, `a`, of a class `C` is declared `private`. Is it possible in Java for an instance of `C` to view and/or modify attribute `a` of another instance of `C`?