Object Orientation

Software development can be seen as a modeling activity. The first step in the software development is the modeling of the problem we are trying to solve and building the conceptual model of the problem domain. The next step is to convert this model to the solution domain model which represents the program that will actually solve our problem. In this perspective, programming languages define the solution domain model that can be converted to the running program by means of compilers. In summary, the software development process takes place in two different domains and at least two kinds of modeling techniques are required.

Object Orientation narrows the gap between problem and solution models as it enables us to represent solution domain in terms of problem domain elements. The problem domain elements include real world objects that interact to accomplish a given task. For example, a library system deals with librarians, borrowers, books, and their interactions. While we model the problem, we model each problem domain element with their states and behaviors and identify their interactions. For example, any system to automate the operations in a library should deal with the objects in the problem domain.

We can identify two merits of the object oriented approach in software development process: they can be used for any problem we face with and solution domain model actually resembles the problem domain model. Since the real world can be seen as a bunch of objects collaborating to achieve a task, shaping the solution domain in terms of objects and their interactions potentially gives us an opportunity to solve any problem using object oriented approach. Moreover, as solution domain model reflects the problem domain model, while you read your program, you are actually navigating through the problem domain. This, obviously, will contribute to reduction of complexity, ease of development and ease of maintenance of the programs.

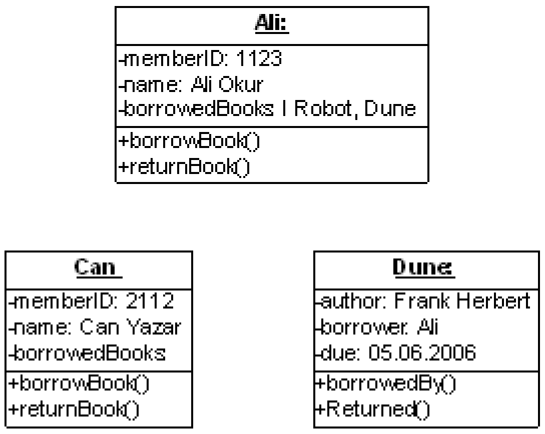
Encapsulation

Objects are the constructing elements of an object oriented program. Like real world objects, each object in a program has an identity, state, and behavior. That is, an object encapsulates state and behavior.

Every object has an identity that distinguishes it from the other objects. That is, every object has a reference that points to memory location where the object resides in the memory. The reference of an object may be considered as the name of an object in a program. Therefore, in order to get services from an object you must know the reference for that object.

Objects provide services to the other objects through their member methods which define the behavior of the object. Objects also has a state which is the data kept in member variables. The state of an object can be changed through calling member methods of that object and behavior of the object may change according to the state of the object.

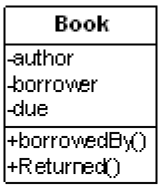
The following diagram shows some of the objects that may be important for a library system (also important for the users of the library system). The notation used here is the Unified Modeling Language (UML) - which is a widely used modeling language used in object oriented analysis design. Objects are shown as rectangles with identities of the objects written in the top (which is underlined and a semicolon appended), state is written in the middle and the services provided are written in the bottom portion. As it can be seen, each object may have different states, and associated behavior.



Classes

A `class` is a data abstraction that represents and defines a set of similar objects with the same kind of states and behavior. That is, a set of objects with the same characteristics belong to the same `class`. A class can be seen as a blueprint of an object that defines its internals, an object is an instance of a class.

A class actualy contains code declaring the member variables and member methods. The class `Book`, for example, would declare member variables `author`, `borrower`, and `due`, and member methods `borrowedBy` and `Returned`. After creating the class, you can create and use objects which are instances of that class. The following UML diagram shows the class `Book`. After the class `Book` is declared, any number of objects may be created as instances of the `Book` class.



In practice, objects brings cohesion and modularity while classes bring reusability to the programs. Each object has a certain well-defined behavior and data and you can modify one object without affecting other objects. By declaring the classes, the same code can be reused to create instances of that class.

Member Variables and Their Initialization

Each object's state is kept in member variables. Member variables can be declared in a class body (not in method implementations - variables declared within methods act as local variables) as ordinary variable declarations like:

```
        DataType variableName;
```

Every member variable, if it is not explicitly initialized, are initialized to their default values (`0` for integer, floating point, and character primitive data types, `false` for boolean data types, and `null` for reference data types such as `String`, array or any other object reference) when an object is created. However, it is possible to initialize member variables explicitly in their declarations as:

```
        DataType variableName = initialValue;
```

If you like, you may initialize the member variables within constructors. It is also possible to create explicit initialization blocks as:

```
        class Circle{
                int originX;
                int originY;
                float radius;
                {
                        originX = 0;
                        originY = 0;
                        radius = 1;
                }
        ...
        }
```

Note that, for any of the above initialization methods, in order to initialize a variable you may provide a literal directly in code or write an expression producing a value (including function calls and

object creation expressions).
Member variable declarations and initializations may be placed anywhere within a class between method declarations, and the order of initialization is determined by the order they are initialized. But the variables are guaranteed to be initialized at least to their default values before any method even the constructor is called. Therefore, initialization within constructors takes place after default initializations, initializations at declaration, and explicit initialization within initialization block are done.

Methods and Overloading

In the Java Programming Language, a member method by simply declared by specifying a name, a set of arguments, and a return value type. Here, argument names and method name must be valid identifiers (valid identifiers are explained in the previous chapter). There may be zero or more arguments of primitive or reference type and the return value type may be a primitive type, reference type or `void` which means nothing is returned. When the method execution completes you may exit from the method using the keyword `return` followed by the return value if the return value type is not `void` anywhere within the method implementation.
In some cases, you may want to provide different implementations for the same method. Overloading enables you to give the same name for such methods. To overload the method, you may declare two or more methods with the same name but with different parameter data types. Here, either number of parameters or data types of parameters (order of parameter declaration is important) must be different in different methods with the same name. Therefore, the compiler knows which method to call in run-time.

Class Methods and Class Variables

Member variables may keep different values in different instances of a class as every instance may have a different state. However, it is possible to declare a special kind of member variable called class variable, whose value is the same across all instances of that class, using the `static` keyword as:

```
        static VariableType  variableName;
```

A class variable can be accessed through instances like any other member variable, or it can be accessed using the class name as `ClassName.variableName`. This actually means that you can access class variables even if no instances has been created yet.
Like member variables, class variables take their default values even if you don’t initialize them explicitly. However, it is also possible to assign initial values while declaring or within explicit initialization blocks as:

```
        class MyClass{
                static int i;
                // i is initialized to 0 by default
                static int j = 10;
                static int k;
                static int r:
                static{
                // this is explicit initialization block
                        k=20;
                        r=20;
                }
        }
```

Similar to class variables, you may declare member methods as class methods using the `static` keyword as:

```
        class ClassName{
                public static ReturnType
MethodName(parameters){
                        ...
                }
                ...
        }
```

Like class variables, class methods can be called through a reference variable or directly using the class name without requiring an instance. That is, class methods may be used without creating an instance. Since they can be called without an instance, they can not modify member variables except the class variables.

Creation, Initialization, and Cleanup

As a class is a blueprint, the instances of that class are created from it. For this purpose the `new` keyword of the Java Programming Language is used as explained in previous chapter. The creation of an object also takes the responsibility of initialization of the object. For this purpose, a special member method of the object, namely constructor, is used. They are special methods in that they have the same name with the class and they have no return values.
Since, the created objects needs to be used later by other objects, we should keep its identity in variables of reference type. Since variables may contain any values, an object may be referred by many variables.
Suppose that we have the following `Rectangle` class declaration:

```
class Rectangle{
        int positionX;
        int positionY;
        int width;
        int height;

        Rectangle(int X, int Y, int aWidth, int aHeight){
                positionX = X;
                positionY = Y;
                width = aWidth;
                height = aHeight;
        }
}
```

A reference variable of type `Rectangle` can be declared and initialized simply as:

```
Rectangle aRectangle = new Rectangle(5, 5, 15, 10);
```

It is also possible to declare more than one constructors for a class as long as overloading rules are ensured. Moreover, you may call other constructors within constructors using the keyword `this` as `this(parameters if there is any)`. In practice, compiler passes a hidden variable, `this`, to each method that points to the current instance of the class. The `this` keyword can be used in some special cases when you explicity need to use the reference of the current object (for example, while calling constructors within constructors, and returning a reference to the current object from methods).
If you do not declare a constructor for a class, the compiler will automatically create a constructor with no arguments for the class. The constructor with no arguments is also known as default constructor. For example, if you declare a basic class as:

```
class IntegerHolder{
  int i;
}
```

You can create an instance of `IntegerHolder` by calling default constructer as:

```
IntegerHolder anIntHolder = new IntegerHolder();
```

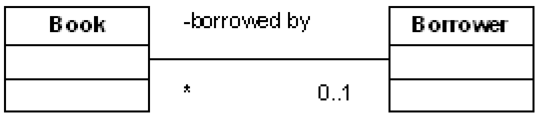
Note that, if you do not declare any default constructor and you declare a constructor with arguments, the compiler will not create a default constructor, so you cannot create the instance simply by calling the default constructor.
An object is accessible within its scope using its reference variable and whenever, the object will not be used anymore, it is garbage collected. Any object is garbage collected when no variables referring to an object remains in the program. Since gargabe collection is automatic, there is nothing to do to clean-up the object. However, in some cases, you may want a special cleanup method to release some resources allocated for the object. In this case you may declare a special method, `finalize()`, and put clean-up code inside this method. However, while using `finalize()`, you must be careful because, the `finalize()` is called only when an object is garbage collected and your objects may not be garbage collected immediately. So, if you really want a special clean-up method, you may declare a member method for this purpose, and you may call this method by yourself when you finish

with the object.

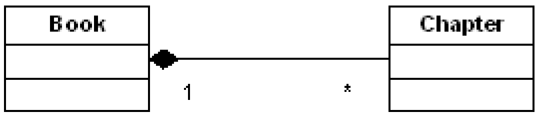
Reuse

When you declare a class, you may reuse its declaration primarily in two dimensions: the class declaration may be used to create instances of that class, or class declaration may be used to create a new class. The first one is the straight forward way of reuse of class declaration and the latter one is called inheritance which will be discussed later.

When you declare an instance of a class within another object, this reuse is called composition or association depending on the relation between the creating object and created object. The ordinary association between two classes describes a relationship that will exist between instances at run-time. That is, one object is used by another object in the program by means of reference variables. On the other hand, compositions are special associations that represent whole-part relationships. If there is a composition relationship between two classes, the instances of the parts is really a part of instances of the aggregate. The following diagram shows a simple association and a simple composition relationship between two classes in UML. In general each association is labeled to indicate the nature of association and symbols indicating multiplicity (how many instances of the class at this end of the association) at each end of association. Note that, * means zero or more.



An association



A composition

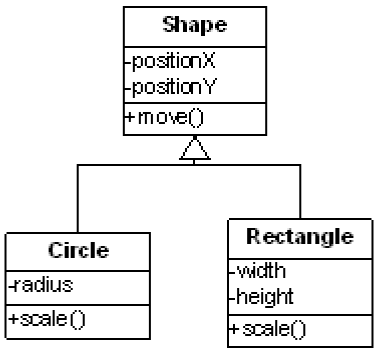
INHERİTANCE

The second kind of reuse is the reuse of class structure through inheritance. Inheritance is one of the key elements of object oriented programming languages.

If different classes have similar states and provides similar service, inheritance allows us to reuse those similar parts from a generalized class declaration. This is accomplished through creating an abstract base class and reusing its state and behavior in more specialized classes. If a class is derived from a base class, all member variables and member methods of the base class are inherited by (or made available to) the derived class. The base class is also called `superclass` and the derived class is also called `subclass` of that `superclass`. By inheritance, you are actually creating a new class which is also compatible with the superclass. That is, subclass can also be used as if it is of type superclass. This makes possible casting the instances of subclass to superclass, which is another key feature of object oriented programming languages, polymorphism. Polymorphism will be discussed later in this chapter.

Inheritance relationship is also referred as `isa` relationship. For example, if you create a `Shape` class, and create `Rectangle` and `Circle` classes by inheriting from the `Shape` class, we can say that every `Rectangle` *is a* `Shape`, and every `Circle` *is a* `Shape`. This feature also helps to make generalizations using inheritance. If there is an inheritance relationship between two classes, `isa` rule should hold. Otherwise, you should not use inheritance. For example, it is not good to make an inheritance relationship between `Circle` and `Rectangle`, since we cannot say 'every `Rectangle` *is a* `Circle`'.

The simple inheritance allows you to reuse the superclass's member methods and variables, and define new member variables and methods in the subclasses. The following diagram shows a simple inheritance hierarchy for geometric shapes in UML. To indicate the inheritance relationship between classes, we use a triangle points to superclass.



Note that, inheritence can be of any depth. That is, a class may inherit from a subclass of an superclass, and the new class inherit all methods and variables in its superclass while some of these may have been inherited from superclass's superclass.

In the Java programming language in order to inherit from a class, while creating your class you can use `extends` keyword in class declaration as:

```
class Shape{
        int positionX;
        int positionY;
        void move(int newX, int newY){
                positionX = newX;
                positionY = newY;
        }
}
class Circle extends Shape{
        int radius;
        void scale(int scaleFactor){
                radius *= scaleFactor;
        }
}
class Rectangle extends Shape{
        int radius;
        void scale(int scaleFactor){
                width *= scaleFactor;
                height *= scaleFactor;
        }
}
```

The creation and use of the subclasses is the same with ordinary classes. Inheritance also allow you to call superclass member methods and variables as well as subclass member methods and variables from the subclass instances as:

```
Circle c = new Circle();
c.positionX = 10;
```

```
c.positionY = 20;
c.radius = 3;
c.move(11,11);
c.scale(5);
```

Even if you don't inherit a class from another class, the compiler automatically inherit the class from `Object` class. Every class you declare is inherited directly or indirectly from the `Object` class. The `Object` class is the top class in the inheritence hierarchy and it is possible to cast any object within the program to `Object` class (this is useful when you store, retrive, and manipulate instances of different classes). `Object` class provides many useful services that are inherited by all classes in the program. Some of these are, `toString()`, `equals()`, `clone()`, `getClass()`. For complete listing of these services you may refer to Java Documentation.

METHOD OVERRIDING

While inheriting from a superclass, you inherit member methods and variables from superclass. However, it is possible to change the behavior of the subclass by overriding the superclass methods. For this purpose, you may exactly redeclare the member method of the superclass in the subclass, and provide an alternative implementation. In this case, all calls to this method will execute the newly provided alternative implementation.
The following example illustrates overriding a method:

```
class Shape{
int positionX;
int positionY;
...
        String toString(){
                return "A Shape at (" + positionX + "," +
positionY + ")";
        }
...
}
class Circle extends Shape{
        int radius;
...
        String toString(){
                return "A Circle of radius " + radius +
                        " at (" + positionX + "," + positionY
+ ")";
        }
...
}
```

It is also possible to call superclass version of the overridden method by using the keyword `super`. As you call a method of a subclass, compiler passes a hidden parameter `super` which points to the superclass's implementation. Therefore in order to call a method implemented in superclass you may simply use `super.methodName(parameters)` expression. The following example illustrates the use of `super` keyword within a subclass.

```
class Shape{
int positionX;
int positionY;
...
        String toString(){
                return "A Shape at (" + positionX + "," +
positionY + ")";
        }
...
}
class Circle extends Shape{
        int radius;
...
        String toString(){
                String s = super.toString();
                s += " which is circle of radius " + radius.
return s;
        }
...
}
```

Initializing Base Classes

A subclass may initialize itself by means of `constructor`. While initializing itself, the super class must also be initialized, i.e., its constructor must also be called. If the superclass has a default constructor (a constructor with no parameters - it may be declared explicity or compiler automatically generates a default constructor if there is no constructor defined for a class), the compiler calls the base class's constructor automatically. However, if there is no default constructor of the superclass, the subclass must call its superclass's constructor inside its constructor explicitly by providing necessary parameters and using the keyword `super`. You may also explicitly call the superclass's non-default constructor although it has a default constructor.
The superclass's constructor must be called in the first statement in the subclass's constructor. That is, superclasses are initialized before subclass is initialized.

Controlling Access

Hiding implementation details is one of the most important features of object oriented languages. Users of a class do not need to know all the details of a class. By choosing different access levels to each member method and variable, a class may specify necessary interface required by the users of the class. This contributes to simplicity and more importantly robustness of the system. Since, a class can be used only in a specified way, it is possible to ensure proper state and use of the class.
In the Java Programming Language, a class can protect its member variables and methods from access by other objects by the use of four kinds of access specifiers. These access specifiers enable you

to provide different interface to different users of the system. The access specifiers available in the Java Programming Language and their meanings are as follows:

> `private`: The private methods and private variables can only be accessed by the class declaring them.
> `protected`: The protected methods and protected variables can be accessed by the class declaring them, the subclasses of that class, and the package the class is declared (packages will be explained later).
> `public`: The public methods and public variables can be accessed anywhere within a program.

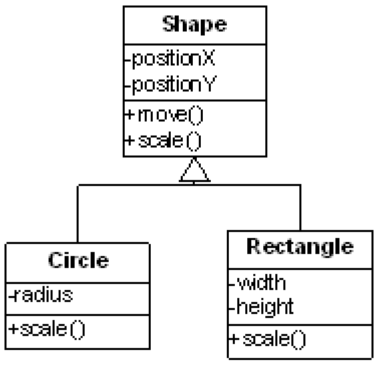
In addition to these, there is one more access level, `package`, in the Java Programming Language. In this access level, member methods and member variables can only be accessed anywhere within the package in which the class is declared (it obviously includes the class itself). If you do not use any access specifier explicitly, the package access level is chosen automatically.
These access specifiers can directly used at member function and member variable declarations as:

```
class MyClass{
        public int i;
        protected int j;
        private int k;
        int p; // package access
        public int getPublic(){…}
        protected int getProtected(){…}
        private int getPrivate(){…}
        int getPackage(){…} // package access
}
```

POLYMORPHİSM

Polymorphism is one of the most powerful features of object oriented languages. By polymorphism, every object in the program may be seen in different forms according to its inheritance hierarchy while its behavior does not change.
As mentioned before, an instance of a subclass may be casted to one of its superclasses automatically by the compiler. Casting an object to one of the superclasses is called upcasting. By upcasting objects of different subclasses to a common superclass, the same code may be used to manipulate them. More importantly, if you create new subclasses, you may continue to use the same code manipulating the objects without changing anything. This brings extensibility to your program, and simplifies the code by using the same code for objects of similar classes. That is you do not need to write statements for each distinct class to manipulate them.

Inheritance, method overriding, together with dynamic binding enables polymorphic objects to behave correctly even if they are upcasted. Dynamic binding is a method calling mechanism which determines the correct method to call when a member method of an object is called. Consider, for example, our previous shape classes are modified a little bit to have the following inheritance diagram in UML.



In this inheritance diagram, the `scale` method of `Shape` class is overridden in `Circle` and `Rectangle` classes. Any call made to scale method of an instance of `Circle` class has two alternative implementations: `scale` method of `Shape` and `scale` method of `Circle`. Since the `scale` method of an instance of `Circle` is called, the compiler will call the `scale` method of the `Circle` (not the `scale` method of the `Shape`). Upcasting the instance of `Circle` object does not effect this behavior. This property leads to many benefits. For example, suppose we have an array of instances of classes `Circle` and `Rectangle` as follows:

```
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle();
        shapes[1] = new Rectangle();
        shapes[2] = new Circle();
```

In order to `scale` all instances, the following code will be sufficient:

```
for(int i=0; i<shapes.length; i++) shapes[i].scale(4);
```

In above example, we make use of polymorphism to store instances of different classes in a common storage, and we call scale method of instances without worry of from which class they are created. After some time, you may create another subclass of the `Shape` class, `Triangle`, and you may use the above code without changing anything to store and scale the instances of `Circle`, `Rectangle`, and newly created `Triangle`.

Downcasting and RTTI

By upcasting, you actually loose the type information of an object. In some cases, you may want to cast an object to its actual type, which is called downcasting. Since you cast an object to a more specific class, downcasting must be made explicity as:

```
Shape s = new Circle();
Circle c = (Circle) s;
```

While downcasting you should guarantee that casting is correct, so that you will not cast an object to an irrelevant class. For example, you cannot cast `s` in above example to a `Triangle` class. Actually, in Java, every downcast is type checked in run-time to ensure correct casting is done. If you try to cast an object to an incorrect class you will get a run-time exception. For this purpose run-time type identification (RTTI) feature is employed. Please refer to Java documentation for more information on RTTI.
If you have to check whether an object can be downcasted to a type or not, the `instanceof` operator may be used. For example, if you want to identify and modify only `Circles` in the `shapes` array in the previous example, you may use:

```
for(int i=0; i<shapes.length; i++)
  if(shapes[i] instanceof Circle){
        Circle c = (Circle) shapes[i];
        ... // work with c here
  }
```

Final Methods and Final Classes

If you do not want inheritors of your class to override some of your methods, you can specify those methods can not be overridden by using `final` keyword as:

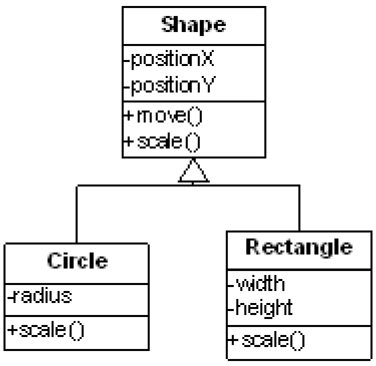
```
class MyClass{
  …
  final int aFinalMethod(){
    …
  }
  …
}
```

Like `final` methods, you may also declare a `final` class. If you declare a class as `final`, no class can inherit from it by any means.

```
final class MyFinalClass{
  …
}
```

Abstract Classes

Consider our previous shapes example shown below:



In this scenario, the superclass `Shape` provides a common interface to its subclasses, so that polymorphism applies. However, the `scale` method in the `Shape` class does nothing and its actual implementation is leaved to subclasses. Another important fact about the `Shape` class is that, no meaninful instances of it can be created. In the Java Programming Language, some methods of a class may be declared as `abstract` and if a class has `abstract` methods, the class itself becomes an `abstract` class. If a method is declared as `abtract`, there is no need to implement that method, and you force subclasses to implement that method. Another important feature is that, you cannot create an instance of an `abstract` base class which solves above mentioned problem. It is also possible to declare a class as `abstract` although it does not contain any `abstract` methods. This also ensures that meaningless instances of that class cannot be created.
A method or a class can be declared as `abstract` by using the `abstract` keyword as:

```
abstract class anAbstractClass{
        ...
}

class aClass {
// this class is also abstract
// since it contains an abstract method
        ...
        abstract int anAbstractMethod(int Parameter);
        ...
}
```

INTERFACES

An `interface` is generalization of `abstract` classes. The main difference between an `abstract` class and an `interface` is that, all methods defined in the `interface` are `abstract` whereas an `abstract` class may declare and implement some methods. The methods declared in an interface must be implemented in classes implementing (or you may say inheriting) the `interface`.
Another important distinction is that, a class can have only one superclass whereas it may implement any number of interfaces as long as it implements all methods of those interfaces. Therefore,

it can be said that, interfaces provide some kind of multiple inheritance which is not possible with classes.
An interface can be declared in java simply by using the `interface` keyword. The following code demonstrates declaration of a simple interface `Drawable`:

```
interface Drawable{
        void draw();
};
```

Any class may implement the interface by specifying the interface implemented using `implements` keyword, and providing implementation of all methods of the interface. In the following example, `Circle` class implements the `Drawable` interface:

```
class Rectangle extends Shape implements Drawable{
        int radius;
        void scale(int scaleFactor){
                width *= scaleFactor;
                height *= scaleFactor;
        }
        void draw(){
                ...
        }
}
```

Like classes interfaces may also be inherited by other interfaces using the keyword `extends` in interface declaration in the same way the classes extend other classes. An interface may also declare same member variables. All member variables declared in an interface are inherited by the classes implementing that interface. However, all member variables declared in an interface are automatically `static` and `final`.

PACKAGES

It is possible to organize your classes within `package`s. By putting related classes into a `package`, you may ease finding classes, avoid naming conflicts, and control access to individual classes.
In the java platform, classes are located in various packages. In order to use a specific class in a specific package in your program, the first statement in the program must `import` the class as:

```
import java.util.ArrayList;
```

It is also possible to import all classes in a package using `*` symbol as:

```
import java.util.*;
```

You may also create your own packages. In order to create a package, you create a directory whose name is the same with your package, and all files in this directory must start with the following statement:

```
package <package name>;
```

Note: If you don’t specify a package name in a file, the classes in that file will be in the `default package`.
Each file within the package declares a class as the `public` class as:

```
public class <class name>{
...
}
```

By this way, that class becomes available to the package users. Since there may be more than one file in the directory, a package may include more than one class available for the package users. (Some compilers allow having more than one public class in each file. However, it is best to have only one public class in each file.)