RELATIONSHIP BETWEEN OBJECTS

CHAPTER 14

It is possible to create objects for different classes and establish relationship between them. When the objects are related, it is possible to access and use members of one object in another object. This becomes an advantage when an object should start processing data where another object has left. It is also helpful to pass data from one object to another object and from there to another object in a chained form.

There are three ways to relate objects in Java:

- Using references
- Using inner class concept
- Using inheritance

We take up a discussion of the first two in this chapter. The third one will be discussed in the next chapter as it needs an entirely different treatise.

Relating Objects using References

Let us create two classes, 'One' and 'Two'. Suppose we want to access the members of class Two, in class One, we should relate their objects. For this, just declare the reference variable of class Two as an instance variable in class One.

```
class One
{
      Two t;   //t is a reference of class Two
}
```

Now this reference variable 't' can be used to refer to all the members of class Two from class One. For example, to call class Two's `display()` method, we can write:

```
t.display();
```

And to access class Two's instance variable 'y', unless it is private, we can write as:

```
t.y;
```

To understand this, we can take the example of following program.

Program 1: Write a program to take class One and class Two, and create the reference of class Two in class One. Using this reference, let us refer to the instance variables and methods of class Two.

```
//Relating class Two with class One
class One
{
	//instance variables
	int x;
	Two t;   //class Two's reference

	//constructor that receives Two's reference
	One(Two t)
	{
		//copy Two's reference into t
		this.t = t;
		x = 10;
	}

	//method to display class One and class Two vars
	void display()
	{
		System.out.println("One's x= "+x);
		//call class Two's method
		t.display();
		//access class Two's var
		System.out.println("Two's var= "+t.y);
	}
}
class Two
{
	//instance variable
	int y;

	//initialize y
	Two(int y)
	{
		this.y = y;
	}

	//method to display y
	void display()
	{
		System.out.println("Two's y= "+y);

	}
}

class Relate
{
	public static void main(String args[])
	{
		//create class Two object and store 22 there.
		Two obj2 = new Two(22);

		//create class One object and pass class Two object to it
		One obj1 = new One(obj2);

		//call class One's method
		obj1.display();
	}
}
```

Output:

```
C:\> javac Relate.java
C:\> java Relate
One's x= 10
Two's y= 22
Two's var= 22
```

In preceding program, we followed the following steps:

- At the time of creating class One's object, we passed reference of class Two's object, as:

```
One obj1 = new One(obj2);
```

Here, reference of Two's object obj2 is passed to One's object obj1.

- This reference obj2's value is copied into class One's constructor and there, it initializes the reference 't' which is declared at class One, as:

```
this.t = t;
```

Here, `this.t` refers to class Two's reference 't' declared at class One.

- Now, 't' refers to class Two's object. So it can access class Two's members. For example, to call class Two's method, we can use:

```
t.display();
```

And, to access class Two's variable, we can use:

```
t.y;
```

This discussion achieves clarity if you see Figure 14.1.

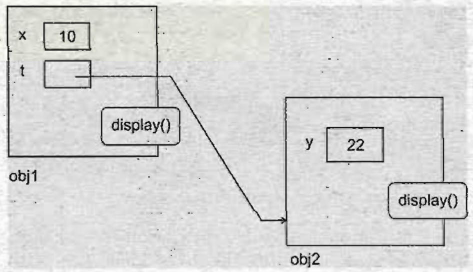


Figure 14.1 't' in obj1 can access members of obj2

In Figure 14.1, we have shown the relation between obj1 and obj2, which are objects of two different classes. Similarly, it is also possible to relate several objects of various classes. This is what, we attempt in the following program.

Program 2: Write a program to call `cube()` method in class One.

```
//Relating the objects of three classes.
class One
{
      //obj2 is class Two's reference
      Two obj2;

      //initialize obj2
```

```
        One(Two obj2)
        {
                this.obj2 = obj2;
        }

        double cube(double x)
        {
                //call class Two's method using obj2
                double result = x*obj2.square(x);
                //return result to Relate class
                return result;
        }
}

class Two
{
        //obj3 is class Three's reference
        Three obj3;

        //initialize obj3
        Two(Three obj3)
        {
                this.obj3 = obj3;
        }

        double square(double x)
        {
                //call class Three's method using obj3
                double result = x*obj3.get(x);
                //return result to class One
                return result;
        }
}

class Three
{
        double get(double x)
        {
                //just return x value to class Two
                return x;
        }
}

class Relate
{
        public static void main(String args[ ])
        {
                //create class Three's object obj3
                Three obj3 = new Three();

                //create class Two's object obj2 and pass obj3
                Two obj2 = new Two(obj3);

                //create class One's object obj1 and pass obj2
                One obj1 = new One(obj2);

                //call cube() method of class One
                double result1 = obj1.cube(5);
                System.out.println("Cube of 5 = "+ result1);

                //call square() method of class Two
                double result2 = obj2.square(6);
                System.out.println("Square of 6 = "+ result2);
        }
}
```

Output:

```
C:\> javac Relate.java
C:\> java Relate
Cube of 5 = 125.0
Square of 6 = 36.0
```

In preceding program, cube() method of class One calls `square()` method of class Two, which in turn calls `get()` method of class Three. Let us discuss this program in detail, we create three classes, One, Two and Three. Class Three's reference obj3 is declared as variable in class Two. Similarly, class Two's reference obj2 is taken as a variable in class One. So from class One, we can access class Two's members. Similarly from class Two, we can access class Three's members.

Please observe the `main()` method in Relate.class. In this method, first we created class Three object obj3. This obj3 is passed to class Two's object obj2. Then obj2 is passed to class One's object obj1. For this, we write:

```
Three obj3 = new Three();
Two obj2 = new Two(obj3);
One obj1 = new One(obj2);
```

The preceding three statements can be combined and written as:

```
One obj1 = new One(new Two(new Three()));
```

Please compare this with the statement for creating the `BufferedReader` object to accept data from keyboard, which we have been already using, as:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Here, `System.in` gives `InputStream` object which represents the keyboard. This is passed to `InputStreamReader` object which is again passed to `BufferedReader` object (`br`). So the data from the keyboard can be received by the `InputStreamReader` object, and from there it can be received into `br`. This is the reason `BufferedReader` is able to read the data coming from the keyboard.

The relationship between the obj1, obj2 and obj3 in memory is shown in Figure 14.2. From this figure, we can understand that obj1 can call the method of obj2 as: `obj2.square()`   //in class One

And obj2 can call the method of ob3, as:

- `obj3.get()`   // in class Two

In this way, relationship between objects will be advantageous to access the members of the related objects. The relationship diagram shown in Figure 14.2 is called 'object graph'.

Important Interview Question

What is object graph?

Object graph is a graph showing relationship between different objects in memory.

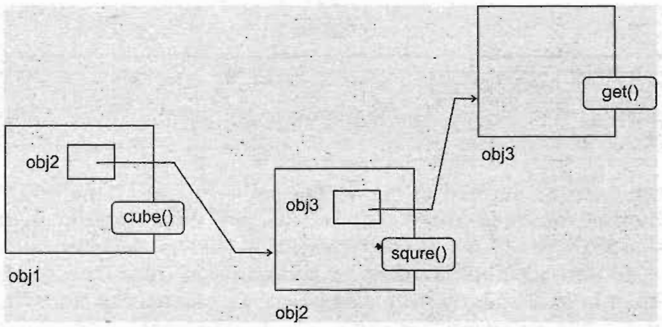


Figure 14.2 Object graph of three objects of classes One, Two and Three

Inner Class

Inner class is a class written within another class. Inner class is basically a safety mechan
since it is hidden from other classes in its outer class.

To make instance variables not available outside the class, we use 'private' access specifier be
the variables. This is how we provide the security mechanism to variables. Similarly, in some cas
we want to provide security for the entire class. In this case, can we use 'private' access spec
before the class? The problem is, if we use private access specifier before a class, the class is
available to the Java compiler or JVM. So it is illegal to use 'private' before a class name in Java.

But, private specifier is allowed before an inner class and thus it is useful to provide security for
entire inner class. Once private is used before the inner class, it is not available to other clas
This means an object to inner class cannot be created in any other class.

Suppose we are writing BankAcct class with bank account details like 'balance' and 'rate' of inte
as instance variables and calculateInterest() method to calculate interest amount and add
balance amount, as shown here:

```
class BankAcct
{
        //balance and rate of interest
        private double bal;
        private double rate;

        //calculate interest and update balance
        void calculateInterest()
        {
                double interest= bal*rate/100;
                bal += interest;
                System.out.println("Balance= "+bal);
        }
}
```

There is no security for this code. Since we did not use any access specifier before BankAcct c
it comes under default access specifier. default indicates that the class is available to any o
class outside. So any other programmer can easily create an object to this class and access
members of the class. Hence, there is no security for the BankAcct class. For example
programmer can write another class, where he can create an object to BankAcct class and cal
calculateInterest() method, as:

```
class Myclass
{
        public static void main(String args[ ])
        {
                BankAcct account = new BankAcct(10000);
                account.calculateInterest(9.5);
        }
    }
```

The calculateInterest() method is a very sensitive method. It should be protected from outsiders. If it is available to other programmers, they may call it as shown earlier and the balance amounts in the bank will be updated. Only authorized people should be able to update the balance amounts, and hence calculateInterest() method should not be available to others. The way to provide security to calculateInterest() method and the rate of interest is to write them inside a class Interest and make it an private inner class in BankAcct class, as shown here:

```
    private class Interest
    {
        private double rate;

        void calculateInterest()
        {
                double interest= bal*rate/100;
                bal += interest;
                System.out.println("Balance= "+bal);
        }
    }
```

Since Interest class is declared as private, an object to it cannot be created in any other class. Then how to use the inner class? We can create an object to the inner class in its outer class. For this purpose, a method contact() can be written in outer class where the inner class object is created. Any programmer should interact with the inner class by calling contact(). Some authentication procedure to verify the user can be implemented in contact() method. When the user calls this method, he is verified, and if he is a legitimate user, then only the inner class object is created. Then the user will be able to use the inner class. The contact() method can be designed some thing like this:

```
      //in this method, inner class object is created after validating.
      //the authenticity of the user. r is the rate of interest.
      void contact(double r) throws IOException
      {
          //accept the password from keyboard and verify
          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
          System.out.print("Enter password: ");
          String passwd = br.readLine();
          if(passwd.equals("xyz123"))
          {
                  //if password is correct then calculate interest
                  Interest in = new Interest(r);
                  in.calculateInterest();
          }
          else {
                  System.out.println("Sorry, you are not an authorized person");
                  return;
          }
      }
```

Now, see the complete program to create an inner class Interest in the outer class BankAcct. Also, in this program, we are showing that a user can use inner class by calling the contact() method of the outer class, where authentication is checked. Thus, outer class is acting like a fire wall (implementer of security mechanism) between the user and the inner class.

Program 3: Let us make a program to create the outer class BankAcct and the inner class Interes
in it.

```
//Inner class example
//this is the outer class
import java.io.*;
class BankAcct
{
        //balance amount is the variable
        private double bal;

        //initialize the balance
        BankAcct(double b)
        {
                bal = b;
        }

        //in this method, inner class object is created after verifying
        //the authentication of user. r is rate of interest
        //this method accepts rate of interest r
        void contact(double r) throws IOException
        {

        //accept the password from keyboard and verify
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in))
        System.out.print("Enter password: ");
        String passwd = br.readLine();

        if(passwd.equals("xyz123"))
        {
                //if password is correct then calculate interest
                Interest in = new Interest(r);
                in.calculateInterest();
        }
        else {
                System.out.println("Sorry, you are not an authorized person");
                return;
        }
        }
        //inner class
        private class Interest
        {
                //rate of interest
                private double rate;

                //initialize the rate
                Interest(double r)
                {
                        rate = r;
                }

                //calculate interest amount and update balance
                void calculateInterest()
                {
                        double interest= bal*rate/100;
                        bal += interest;
                        System.out.println("Updated Balance= "+bal);
                }
        }
}

 //Using inner class
 class InnerClass
{
        public static void main(String args[]) throws IOException
        {
                //bank account is holding a balance of 10000
                BankAcct account = new BankAcct(10000);

                //update balance amount by adding interest at 9.5%
                account.contact(9.5);
        }
}
```

Output:

```
C:\> javac InnerClass.java
C:\> java InnerClass
Enter password: xyz123
Updated Balance= 10950.0
```

When the objects for `BankAcct` and `Interest` classes are created, they are created in memory separately, but they will have some relation between them, as shown in the Figure 14.3.

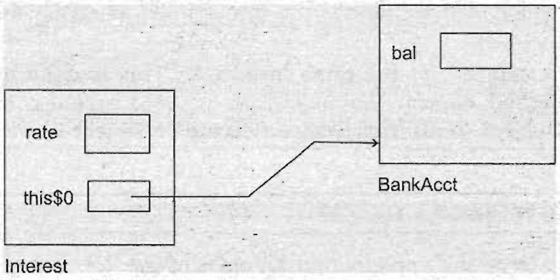


Figure 14.3 Relation between Interest and BankAcct objects

The inner class object will contain an additional field by the name `this$0` which stores the reference number of the outer class object in memory. `this$0` is an invisible field and created in the object as an additional field. Because reference of outer class object is available to inner class object, now inner class is able to refer to all the members of the outer class. For example, in our Program 3, the outer class balance is directly available to inner class.

It is also possible to use same names for the members of both the outer class and inner classes. In that case, there may be confusion while referring to the outer and inner class members separately. Then it is convenient to use the following notation to refer to the members in the inner class:

- Outer class members can be referred as `Outerclassname.this.member`. For example, to refer to `BanckAcct` class's bal in the inner class, we can write: `BankAcct.this.bal`.
- Inner class members can be referred as `this.member`. For example, to refer to `Inner` class interest, we can write: `this.interest`.

Let us now summarize the points regarding the inner classes:

- An inner class is a class that is defined inside another class.
- Inner class is a safety mechanism.
- Inner class is hidden from other classes in its outer class.
- An object to inner class can not be created in other classes.
- An object to inner class can be created only in its outer class.
- Inner class can access the members of outer class directly.
- Inner class object and outer class objects are created in separate memory locations.
- Inner class object contains an additional invisible field 'this$0' that refers to its outer class object.
- When same names are used, we can refer to outer class members in the inner class, as:

```
outerclassname.this.member;
```

and, the inner class members can be referred in the inner class, as:

```
this.member;
```

- Inner classes decrease readability (understanding) of a program. This is against to the design principle of Java to be a simple programming language.

Anonymous Inner Class

It is an inner class without a name and for which only a single object is created. Anonymous inner classes are very useful in writing implementation classes for listener interfaces in graphics programming. To understand the use of anonymous inner class, let us take a program, where we display a push button, which will terminate the application when clicked. For this purpose, we should follow the steps:

- Add ActionListener interface to the push button 'b'. This is done by addActionListener() method. But this method expects an object of ActionListener interface. Since it is not possible to create an object to an interface, we create an object to the implementation class of the interface.

```
b.addActionListener(new Myclass());
```

Here, Myclass is assumed to be the implementation class of the ActionListener interface.

- Now we write the Myclass as:

```
class Myclass implements ActionListener
{
	//this method is executed when button is clicked
	public void actionPerformed(ActionEvent ae)
	{
		//exit the application
		System.exit(0);
	}
}
```

Since, Myclass is an implementation class of ActionListener interface, the method actionPerformed() should be implemented in this class. This method code is executed when the push button is clicked by the user.

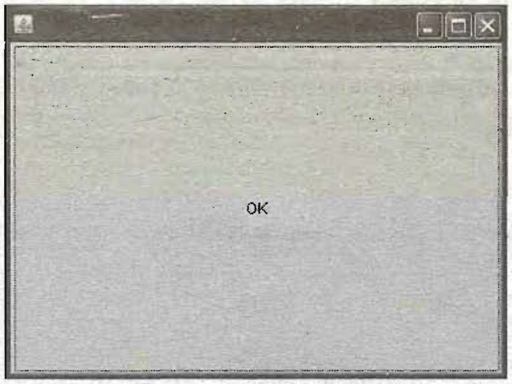
Program 4: Write a program to create a push button and add it to a frame.

```
//Creating a push button and providing action to it.
import java.awt.*;  //for Button
import java.awt.event.*; //for ActionListener
class But extends Frame
{
	But ()
	{
		//create a push button b
		Button b = new Button("OK");

		//add push button to frame
		add(b);

		//add action listener to button.
		//Myclass is implementation class of ActionListener interface
		b.addActionListener(new Myclass());
```

```
        }
        public static void main(String args[ ])
        {
                //create a frame by creating But class object
                But obj = new But();

                //set the size of frame to width: 400 px and height: 300 px
                obj.setSize(400,300);

                //display the frame
                obj.setVisible(true);

        }
}
//Myclass should implement the methods of ActionListener.
class Myclass implements ActionListener
{
        //this method is executed when button is clicked
        public void actionPerformed(ActionEvent ae)
        {
                //exit the application
                System.exit(0);
        }
}
```

Output:

```
C:\> javac But.java
C:\> java But
```



preceding program, we add ActionListener interface to provide some action to the button. Here, ActionListener needs an implementation class. We write Myclass as the implementation class of ActionListener interface.

As done in Program 4, we need not write a separate Myclass and pass its object to addActionListener() method, as:

```
b.addActionListener(new Myclass());
```

Instead, we can directly copy the code of Myclass into this method parameter, as shown here:

```
b.addActionListener(new ActionListener()
{
        //this method is executed when button is clicked
        public void actionPerformed(ActionEvent ae)
        {
                //exit the application
                System.exit(0);
        }
    } );
```

Please do not think that we are creating an object to `ActionListener` in the preceding method are, in fact, creating an object of Myclass and copying the entire class code into the me parameter. This is possible only with anonymous inner class. Here, we did not write the name o class in the method parameter. Such a class is called 'anonymous inner class' so, we call 'Myc as an anonymous inner class.

Let us understand the following points from the preceding discussion:

- Myclass is inner class inside But class.
- The name 'Myclass' is not written in its outer class, i.e., But class.
- Myclass object is created only once.
- Myclass code is directly copied into the method parameter.

Important Interview Question

What is anonymous inner class?

It is an inner class whose name is not written in the outer class and for which only one object created.

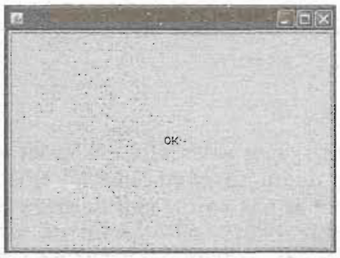
Please see Program 5, which is the anonymous inner class version of the Program 4.

Program 5: Write a program by taking Myclass as an anonymous inner class whose name is mentioned in the But class's `addActionListener()` method.

```
//Creating a push button and providing action to it using inner class.
import java.awt.*;
import java.awt.event.*;
class But extends Frame
{
        But ()
        {
                //create a push button b
                Button b = new Button("OK");

                //add push button to frame
                add(b);

                //add action listener to button.
                //Myclass is hidden inner class of ActionListener interface,
                //whose name is not written but an object to it created.
                b.addActionListener(new ActionListener()
                {
                        //this method is executed when button is clicked
                        public void actionPerformed(ActionEvent ae)
                        {
                                //exit the application
                                System.exit(0);
                        }
                } );
        }

        public static void main(String args[ ])
        {
                //create a frame by creating But class object
                But obj = new But();
                //set the size of frame to width: 400 px and height: 300 px
                obj.setSize(400,300);

                //display the frame
                obj.setVisible(true);

        }
}
```

Output:

```
C:\> javac But.java
C:\> java But
```



Conclusion

To relate the objects of two classes, we declare the reference of one class as an instance variable in another class. This is useful to access the members of that class whose object is referenced. Another way of relating the objects is by using inner class concept. When the programmer wants to restrict the access of entire code of a class, he creates an inner class as a private class. The way to access the inner class is through its outer class only. So any authentication mechanism is implemented in the outer class. Finally, anonymous inner class is helpful when we want to pass entire class code to a method, where the class name is not written but an object is created to it.

Another way of relating objects is using inheritance, which we discuss in the next chapter.