Classes and objects in Java 8

As learnt earlier object and class are fundamental parts of object-oriented programming. Java is an object-oriented Language. This chapter explains how to create a class and object in Java programming. After studying this chapter, we will be able to get a clear picture as to what are objects and what are classes in Java.

Introduction

A class contains both data (referred to as attributes), and program code (functions referred to as methods).

In the previous chapter, we wrote programs using a class that contained a single method named 'main'. When the class was executed, the main method was called, and the application ran as a normal program. We have not used any data members in those examples.

While it is possible to use only a single class in a Java project, this is not a good practice for large applications. When designing software, we should divide the entire application into simpler components that perform logically related tasks. For each such component or module we may create a class.

Let us understand the concept of class and object in Java programming through an example of 'Room'. A room of a house, hostel, hotel or school possess common characteristics. Each room can be uniquely identified by its properties like length, width, height, number of windows, number of doors and direction. For simplicity, let us consider here length, width, height and number of windows.

We will now write a Java code as shown in code listing 8.1 to create a class named 'Room' with the attributes length, width, height, nWindows and three methods. The first method will be used to assign values to attributes. The second method will calculate the area, while the third will display its attributes. Thereafter, we will write code to use this class.

```
/* Class Room */
class Room
{
        float length, width, height;
        byte nWindows;

        void setAttr (float l, float w, float h, byte n)
        {
                length = l; width = w; height = h;
                nWindows = n;
        } // end setAttr () method

      double area ( )  // area =  length * width
```

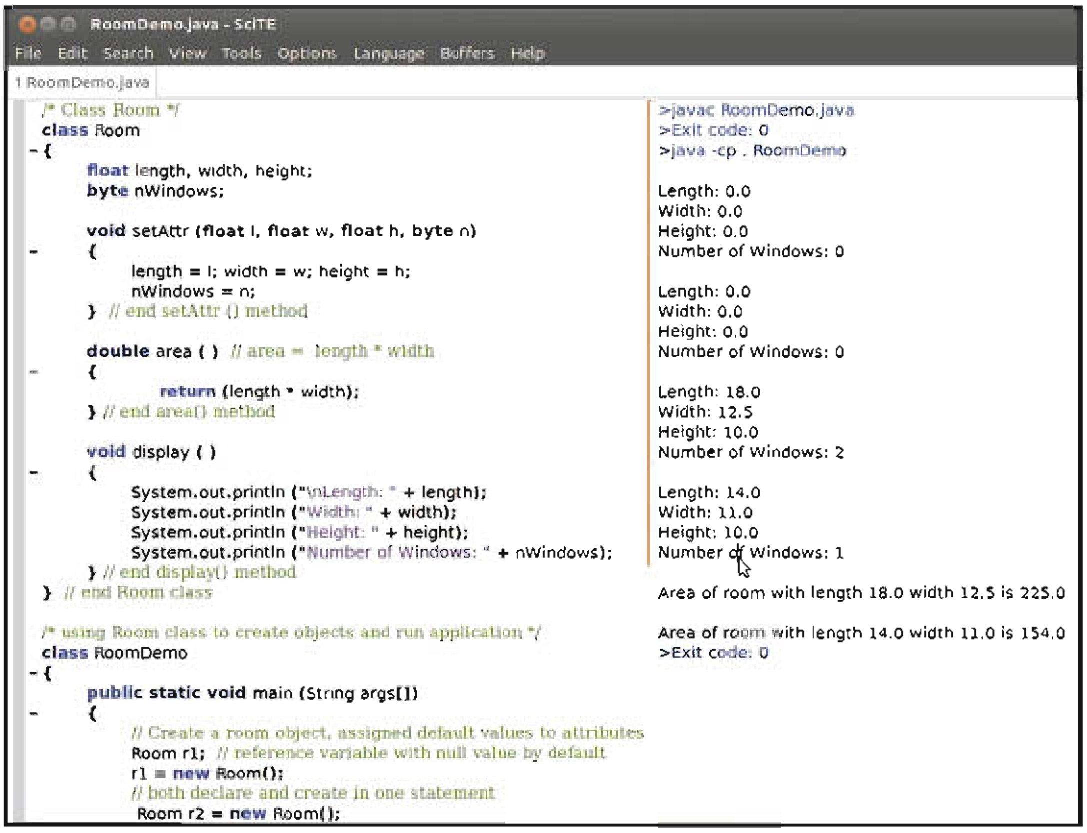
```
            {
                        return (length * width);
            } // end area() method

          void display ( )
            {
                        System.out.println ("\nLength: " + length);
                        System.out.println ("Width: " + width);
                        System.out.println ("Height: " + height);
                        System.out.println ("Number of Windows: " + nWindows);
            } // end display() method
}  // end Room class

/* using Room class to create objects and run application */
class RoomDemo
{
        public static void main (String args[])
            {
                        // Create a room object, assigned default values to attributes
                        Room r1;  // reference variable with null value by default
                        r1 = new Room();
                        // both declare and create in one statement
                         Room r2 = new Room();

                  // Display two room objects with initial default values
                        r1.display();
                        r2.display();

                        // Assign values of attributes of objects
                        r1.setAttr (18, 12.5f, 10, (byte)2);
                        r2.setAttr (14, 11, 10, (byte)1);

                        // Display updated contents
                        r1.display();
                        r2.display();

                        // Display area
                        System.out.println ("\nArea of room with length " + r1.length
                           + " width " + r1.width +  " is " + r1.area());
                        System.out.println ("\nArea of room with length " + r2.length
                           + " width " + r2.width +  " is " + r2.area());
            }  // end main()
}  // end RoomDemo
```

Code Listing 8.1 : Creating and using class and objects

Here, first of all, code is written to create a class named 'Room'. Thereafter, code is written to create objects of class 'Room' and use its methods. To do so, we have created another class named 'RoomDemo' containing main() method. In a source file, these classes may appear in any order.

Here, we have separated creating 'Room' class performing logically related tasks and using this class in application class 'RoomDemo'.

When a program contains two or more classes, only one class can contain the main() method. Figure 8.1 shows the execution of the code in SciTE editor.



```
/* Class Room */
class Room
{
    float length, width, height;
    byte nWindows;

    void setAttr (float l, float w, float h, byte n)
    {
        length = l; width = w; height = h;
        nWindows = n;
    } // end setAttr () method

    double area ( ) // area = length * width
    {
            return (length * width);
    } // end area() method

    void display ( )
    {
        System.out.println ("\nLength: " + length);
        System.out.println ("Width: " + width);
        System.out.println ("Height: " + height);
        System.out.println ("Number of Windows: " + nWindows);
    } // end display() method
} // end Room class

/* using Room class to create objects and run application */
class RoomDemo
{
    public static void main (String args[])
    {
        // Create a room object, assigned default values to attributes
        Room r1; // reference variable with null value by default
        r1 = new Room();
        // both declare and create in one statement
         Room r2 = new Room();
```

```
>javac RoomDemo.java
>Exit code: 0
>java -cp . RoomDemo

Length: 0.0
Width: 0.0
Height: 0.0
Number of Windows: 0

Length: 0.0
Width: 0.0
Height: 0.0
Number of Windows: 0

Length: 18.0
Width: 12.5
Height: 10.0
Number of Windows: 2

Length: 14.0
Width: 11.0
Height: 10.0
Number of Windows: 1

Area of room with length 18.0 width 12.5 is 225.0

Area of room with length 14.0 width 11.0 is 154.0
>Exit code: 0
```

Figure 8.1 : Java Program to demonstrate use of multiple classes

In the given example of RoomDemo application as shown in figure 8.1, following operations are performed.

- Two objects r1 and r2 of class Room are created first. They are initialized with default values (Numeric values are zero by default).
- Contents of these two objects are displayed using display method.
- Invoking setAttr() method with numeric literals as parameters, attributes of room objects are modified for both the objects r1 and r2.
- The updated contents of two objects are displayed.

Finally area of both Room objects is displayed. Here area() method is invoked to get area.

Class in Java

We have already written one program that uses class. Let us now learn the syntax and other details of how to create class, objects and use them in an application.

A class is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects. For example, class 'Room' is a template for all rooms with their common properties.

In Java, a class is defined using **class** keyword as follows :

```
class <ClassName >
{
            <Variables>
            <Methods>
}
```

Every class we write in Java is generally made up of two components: attributes and behaviour. Attributes are defined by variables in a class. Behaviour is defined by methods in a class. Methods are used to access or modify attributes.

In code listing 8.1, we have defined a class named 'Room' with attributes defined by variables named length, width, height and nWindows; and behaviour by methods named setAttr(), display() and area(). Another class is created just to create an application using 'Room' class.

Creating objects

Creating an object from a class requires the following steps :

- **Declaration :** A variable (reference variable) name of type class is declared with syntax <class name> <variable name>
- **Instantiation :** Keyword **new** is used to create the object by allocating memory
- **Initialization :** Constructor (a special type of method) is called to initialize the newly created object

In Java, a class is a type, similar to the built-in types such as int and boolean. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function.

To declare an object of class 'Room', use following statement :

```
Room r1;
```

Declaring a variable does not create an object. This is an important point to be remembered. In Java, no variable can ever store an object. A variable declared using class type can only store a reference to an object. So variables of class type are also referred to as reference variables. Here r1 is a reference variable.

Next step is to create an object. Using new keyword, we can create an object. Operator new allocates the memory for an object and returns the address of the object for later use. Reference is the address of the memory location where the object is stored. In fact, there is a special portion of memory called the heap where the objects live.

When an object is created, in addition to allocating memory, a special method called 'constructor' is executed to perform initial task. We will learn about constructors later in this chapter.

Let us create an object of type Room and assign its address to variable r1 as follows :

r1 = new Room();

Here, parentheses are important; don't leave them off. With empty parentheses without arguments, a default constructor is called. It initializes the attributes (variables) of the object using default values. The parentheses can contain arguments that determine the initial values of variables. This is possible by using user-defined constructor.

When statement "r1=new Room();" is executed, remember that object is not stored in variable r1. Variable r1 contains only address of an object.

Both the above steps used to declare and create an object can be combined into a single statement as follows :

```
Room r2 = new Room();
```

Variable r2 contains a reference or address of memory location where a new object is created.

It is also to be noted that the class determines only the types of the variables. The actual data is contained inside the individual objects and not in the class. Thus, every object has its own set of data. In code listing 8.1, we have created two objects r1 and r2 using new operator. These objects are allocated different memory space to hold their data values as seen in figure 8.2 .

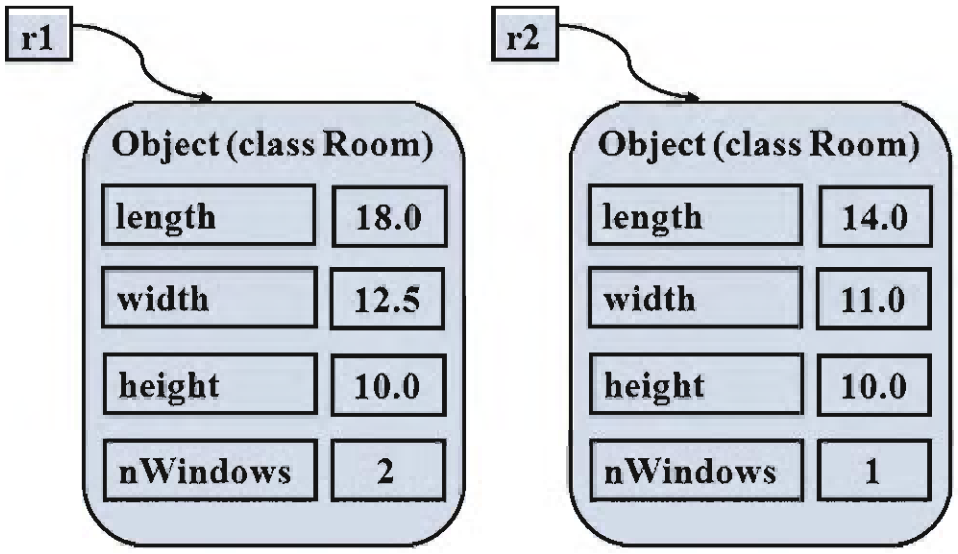


Figure 8.2 : Two instances of class Room

In Java, when objects are no more needed, the memory is claimed back for reuse. Java has a garbage collector that looks for unused objects and reclaims the memory that those objects are using. We do not need to do any explicit freeing of memory.

In object-oriented programming (OOP) languages, creating an object is also called object instantiation. An instance for an object is created by allocating memory to store the data for that object. An object that belongs to a class is said to be an instance of that class.

Thus, an instance of a class is another word for an actual object. Class is an abstract representation of an object; whereas an instance is its concrete representation. In fact, the terms instance and object are often used interchangeably in OOP language.

Each instance of a class can hold different values for its attributes in variables declared in a class. Such variables are referred to as instance variables. Instance variables are created at the time of creating an object and stay throughout the life of the object.

Instance variables define the attributes of an object. The class defines the kind of attribute. Each instance stores its own value for that attribute.

To define an object's behaviour, we create methods. In Java, methods can be defined inside a class only. These methods can be invoked using the objects to access or modify the instance variables. Such methods are known as instance methods.

Instance methods are used to define behaviour of an object. Invoking a method is to ask the object to perform some task.

In code listing 8.1, we have defined a class named Room with instance variables length, width, height and nWindows; and instance methods setAttr(), display() and area().

To summarize,

- Objects are created with the new keyword.
- The new keyword returns a reference to an object that represents an instance of the class.
- All instances of class are allocated memory in data structure called heap.
- Each object instance has its own set of data.

Accessing instance variables and calling instance methods

Instance variables and instance methods are accessed via objects. They can be referred by using dot (.) operator as follows :

<object refernce>.<instance variable or method>

For example, we can refer to length of room r1 using r1.length in a program and invoke method r1.display() that displays attribute values of room r1 as can be seen in code listing 8.1. Note that here dot (.) is an operator and associativity of dot operator is from left to right.

Remember that data should be protected from such direct access from anywhere in the program. Such protection is possible with the use of access modifiers that we will see later.

When the instance variables are referred within the methods of the same class, there is no need to use dot (.) operator. For example, in code listing 8.1, in method display, instance variables are accessed without using dot operator. This is possible because method is invoked using reference variable r1 and thus referred instance variables are considered to be of the corresponding object stored at r1.

As studied, when we only declare an object using class, object is not created. In this case, reference variable does not refer to any object and its initial value is null by default. Use of such null reference or null pointer is illegal and may raise an exception. So, referring instance variable or invoking method with null reference will give an error. Try following code:

```
Room r1; // null value assigned to reference variable by default

System.out.println (r1.length); // illegal

r1.display();  // illegal
```

Class variables and class methods

As discussed earlier, when an object is created using new keyword, memory is allocated from heap area to store the value of its attributes. Thus every object has its own instance variables occupying different space in memory.

Now suppose we want to have a total number of windows of all Room objects created so far. To store this value, it requires only one variable per class. It is meaningless to keep this variable as an attribute of each object. Actually, it is not an attribute of an object; it is an attribute of a class.

Thus, when there is a need to have some variable that is shared by all object instances of the same class, the variable should be allocated the memory only once per class. It means that variable belongs to a class and not to an object. Such variables can be declared within a class using static keyword before data type and these static variables are called class variables.

Values of instance variables are stored in instance (memory allocated for the object); whereas values of class variables are stored in class itself.

Let us declare class variable totWindows by adding following statement in class with instance variables:

```
static int totWindows;
```

Static variables can be accessed without creating an instance of a class. For example, if we try to display the value of totWindows without creating any object of the class 'Room', it will display 0.

Just like class variable, class method can be defined using static keyword in front of the method definition. Try following method in 'Room' class to display total windows.

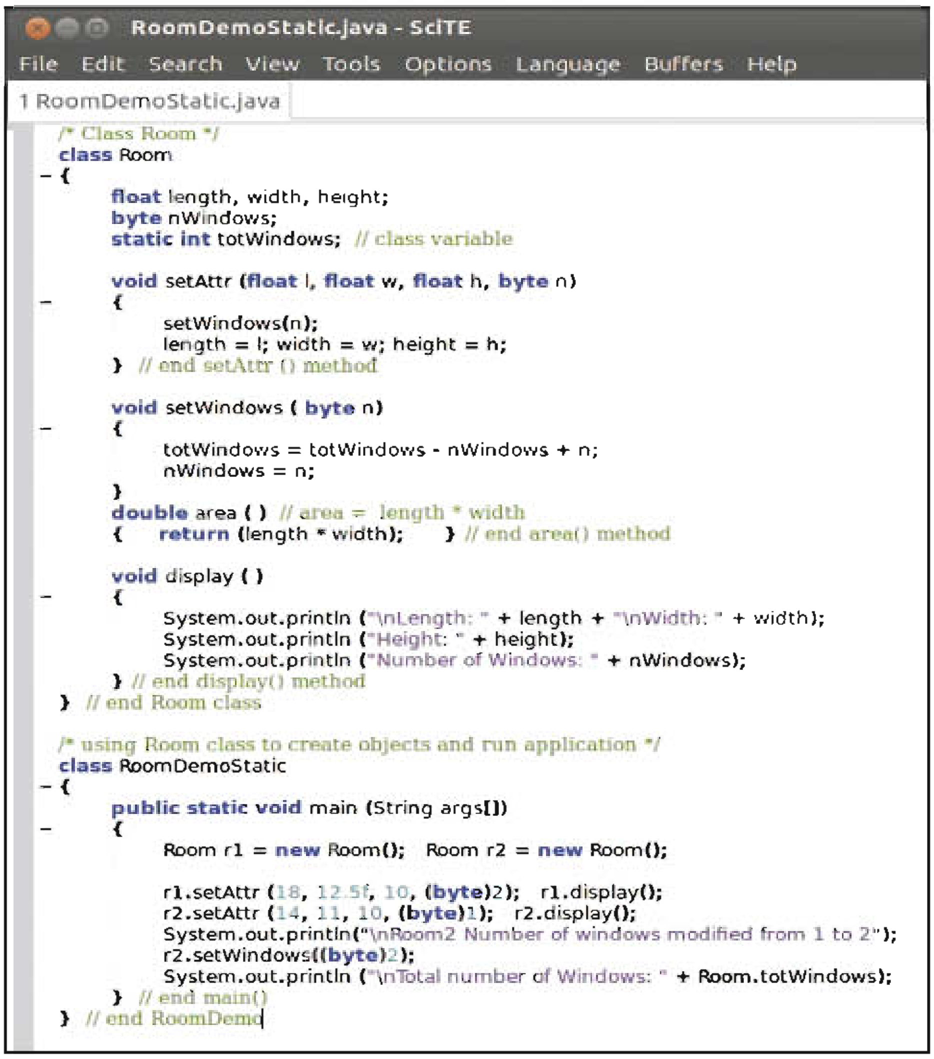
```
static void displayTotalWindows()
{
    System.out.println("Total Windows: " +totWindows);
}
```

Class variables and class methods can be accessed outside the class using

```
        <classname>.< class variable/method name>
```

For example : Room.totWindows, Room.displayTotalWindows()

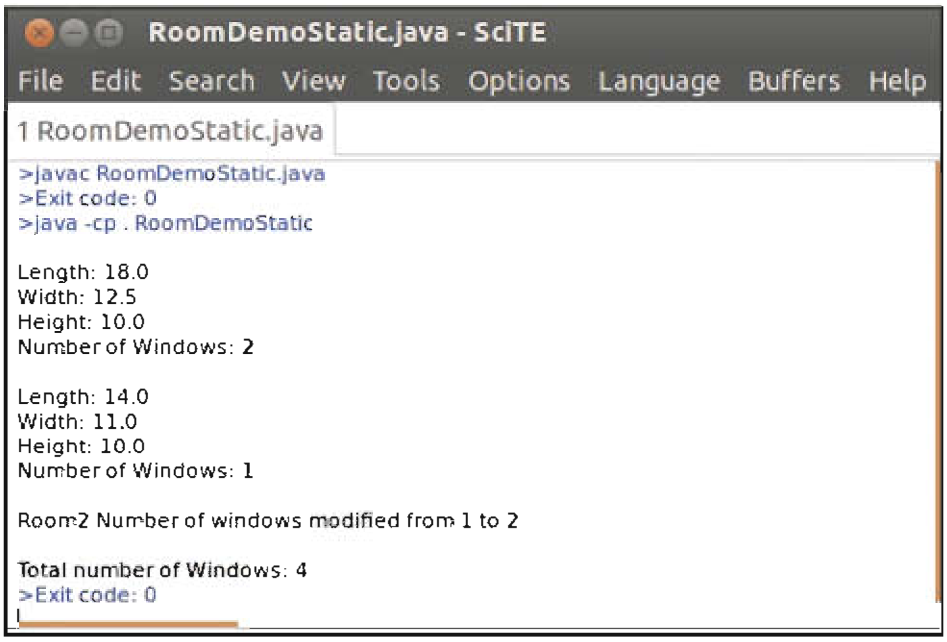
Note that class members (variables and methods) can be referred without class name by the methods of the same class as shown in the code listing of figure 8.3.


RoomDemoStatic.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 RoomDemoStatic.java

```java
/* Class Room */
class Room
{
    float length, width, height;
    byte nWindows;
    static int totWindows;  // class variable

    void setAttr (float l, float w, float h, byte n)
    {
        setWindows(n);
        length = l; width = w; height = h;
    }  // end setAttr () method

    void setWindows ( byte n)
    {
        totWindows = totWindows - nWindows + n;
        nWindows = n;
    }
    double area ( )  // area =  length * width
    {    return (length * width);      }  // end area() method

    void display ( )
    {
        System.out.println ("\nLength: " + length + "\nWidth: " + width);
        System.out.println ("Height: " + height);
        System.out.println ("Number of Windows: " + nWindows);
    }  // end display() method
}  // end Room class

/* using Room class to create objects and run application */
class RoomDemoStatic
{
    public static void main (String args[])
    {
        Room r1 = new Room();   Room r2 = new Room();

        r1.setAttr (18, 12.5f, 10, (byte)2);   r1.display();
        r2.setAttr (14, 11, 10, (byte)1);   r2.display();
        System.out.println("\nRoom2 Number of windows modified from 1 to 2");
        r2.setWindows((byte)2);
        System.out.println ("\nTotal number of Windows: " + Room.totWindows);
    }  // end main()
}  // end RoomDemo
```

Figure 8.3 : Use of class variable totWindows

The output of the code shown in figure 8.3 is given in figure 8.4.


RoomDemoStatic.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 RoomDemoStatic.java

```
>javac RoomDemoStatic.java
>Exit code: 0
>java -cp . RoomDemoStatic

Length: 18.0
Width: 12.5
Height: 10.0
Number of Windows: 2

Length: 14.0
Width: 11.0
Height: 10.0
Number of Windows: 1

Room2 Number of windows modified from 1 to 2

Total number of Windows: 4
>Exit code: 0
```

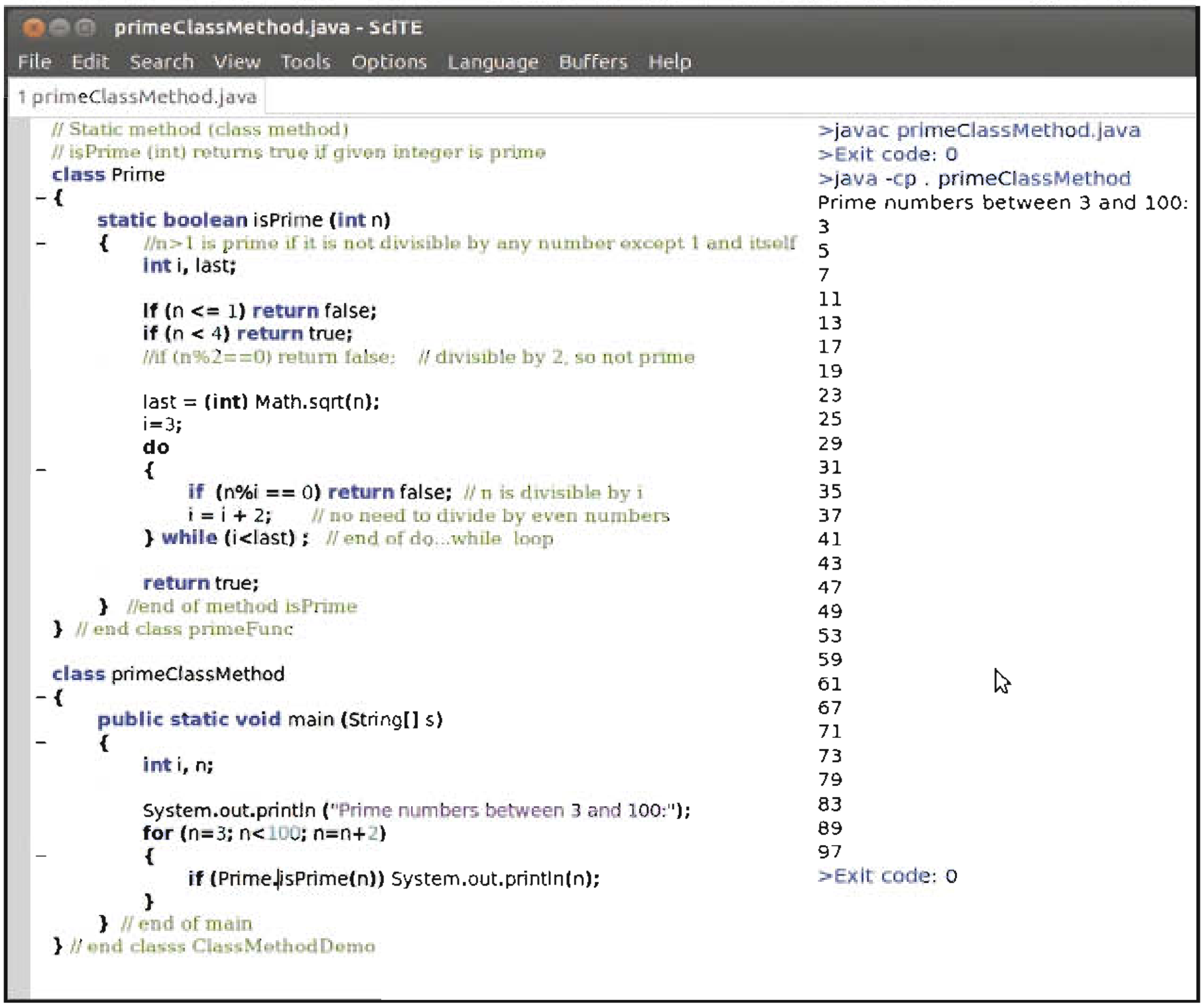
Figure 8.4 : Output of code listing given in figure 8.3

Here, we have written an additional instance method setWindows() that updates total windows by subtracting the old number of windows and adding new number of windows in a room. Method setWindows() is defined in the same class. So it can access class variable totWindows without class name. In main() method (defined in RoomDemoStatic class) that is defined outside 'Room' class, class variable totWindows is to be accessed as Room.totWindows using class name.

Class methods are global to the class itself and available to any other classes or objects. Therefore, class methods can be used anywhere regardless of whether an instance of the class exists or not.

Now, when should we use class methods ? The methods that operate on a particular object, or affect that object, should be defined as instance methods. Methods that provide some general utility but do not directly affect an instance of the class are better declared as class methods.

For example, consider a function that determines whether a given number is prime or not. This function should be defined as class method. The code listing and its output is shown in figure 8.5.


primeClassMethod.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 primeClassMethod.java

```java
// Static method (class method)
// isPrime (int) returns true if given integer is prime
class Prime
{
    static boolean isPrime (int n)
    {     //n>1 is prime if it is not divisible by any number except 1 and itself
          int i, last;

          If (n <= 1) return false;
          if (n < 4) return true;
          //if (n%2==0) return false;    // divisible by 2, so not prime

          last = (int) Math.sqrt(n);
          i=3;
          do
          {
               if  (n%i == 0) return false;  // n is divisible by i
               i = i + 2;       // no need to divide by even numbers
          } while (i<last) ;   // end of do...while  loop

          return true;
    }  //end of method isPrime
} // end class primeFunc

class primeClassMethod
{
    public static void main (String[] s)
    {
          int i, n;

          System.out.println ("Prime numbers between 3 and 100:");
          for (n=3; n<100; n=n+2)
          {
               if (Prime.isPrime(n)) System.out.println(n);
          }
    } // end of main
} // end classs ClassMethodDemo
```

```
>javac primeClassMethod.java
>Exit code: 0
>java -cp . primeClassMethod
Prime numbers between 3 and 100:
3
5
7
11
13
17
19
23
25
29
31
35
37
41
43
47
49
53
59
61
67
71
73
79
83
89
97
>Exit code: 0
```

Figure 8.5 : Using static method to determine whether a given integer is prime or not

The designers of Java have already provided a large number of built-in classes with such class methods. In chapter 7 we have used static method 'sqrt()' without creating any object of class Math.

It is to be noted here that the main() method that we have defined till now is also a class method. This is the reason we use keyword static while defining main() method.

As seen here, the static and the non-static portions of a class serve different purposes. The static definitions in the source code specify the things that are part of the class itself; whereas the non-static definitions in the source code specify the things that will become part of every instance object belonging to a class.

Points to Remember :

- Class variables and class methods can be accessed using a class name or reference variable. To increase readability, it is advised to access with class name.
- Class variables and class methods can be accessed from instance methods also.
- Instance variables and instance methods can't be accessed from class methods, as class methods do not belong to any object.

Classification of variables declared in a class

- **Local variables :** Variables defined inside methods or blocks are called local variables. Formal parameters of the methods are also local variables. They are created when the method or block is started and destroyed when the method or block has completed. Local variables are not initialized by default values.
- **Instance variables :** Instance variables are variables defined within a class but outside any method. These variables are allocated memory from heap area when an object is instantiated (created). Instance variables are initialized by default values.
- **Class variables :** Class variables are variables defined with in a class, outside any method, with the static keyword. These variables are allocated memory only once per class and is shared by all its objects. Class variables are initialized with default values.

Polymorphism (Method Overloading)

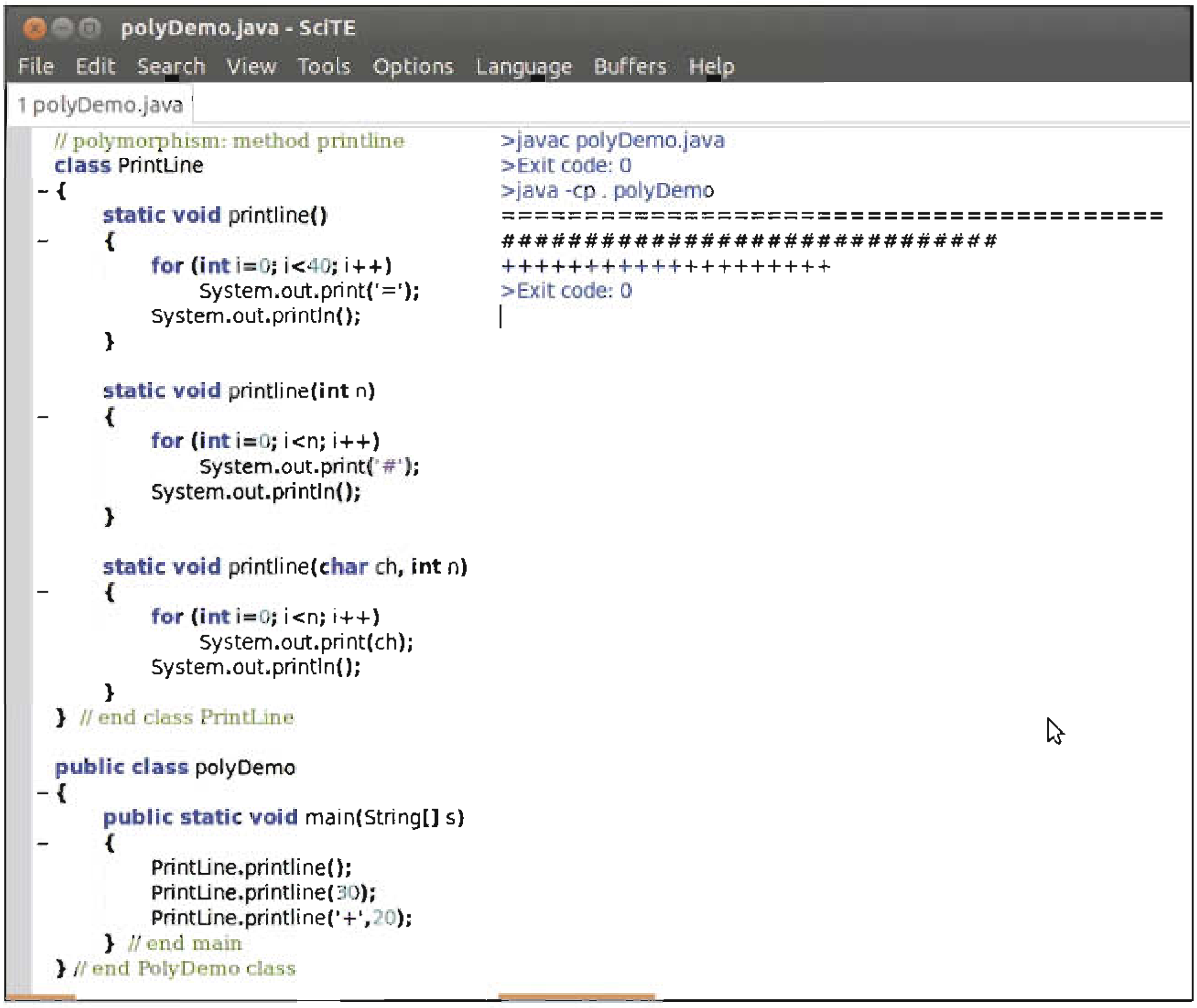
We have seen first two steps of creating objects: Declaration and Instantiation. The third step is Initialization. It requires the use of constructors. Before we learn about constructors, let us see the way to implement polymorphism in Java.

The word polymorphism means "many forms"; different forms of methods with same name. In Java, we can have different methods that have the same name but a different signature. This is called 'method overloading'. The method's signature is a combination of the method name, the type of return value (object or base type), a list of parameters.

For example, to find maximum of two integers, maximum of three integers, maximum of three double precision real numbers and so on one requires to perform similar task but on different set of numbers. In such scenario, Java facilitates to create methods with same name but different parameters. Finding maximum does not need creation of any object, so it can be defined as static class method. For example :

```
static int max(int x, int y) {...}

static int max(int x, int y, int z) {...}

static double max(double x, double y, double z) {...}
```

You may try to define above forms of max methods and use in the application after looking the example of code listing given in figure 8.6. Here it uses polymorphism to print a line as per specified parameters. Call printline() without any parameter prints 40 times '=' character, printline(int n) prints n times '#' character, whereas printline(int n, char ch) prints n times specified character ch.



```java
// polymorphism: method printline
class PrintLine
{
    static void printline()
    {
        for (int i=0; i<40; i++)
            System.out.print('=');
        System.out.println();
    }

    static void printline(int n)
    {
        for (int i=0; i<n; i++)
            System.out.print('#');
        System.out.println();
    }

    static void printline(char ch, int n)
    {
        for (int i=0; i<n; i++)
            System.out.print(ch);
        System.out.println();
    }
} // end class PrintLine

public class polyDemo
{
    public static void main(String[] s)
    {
        PrintLine.printline();
        PrintLine.printline(30);
        PrintLine.printline('+',20);
    } // end main
} // end PolyDemo class
```

```
>javac polyDemo.java
>Exit code: 0
>java -cp . polyDemo
========================================
##############################
++++++++++++++++++++
>Exit code: 0
```

Figure 8.6 : Method Overloading

Constructors

Constructor is a special kind of method that is invoked when a new object is created. Constructor can perform any action; but it is mainly designed to perform initializing actions.

Till now, we have used class without user-defined constructors. Every class is having its default constructor; sometimes referred to as no-argument constructor. Default constructor does not take any argument. It initializes the attributes of a newly created object using default values based on their data types.

Constructor differs from general methods in the following ways :

- Constructor must have the same name as class name.
- Constructor does not have return type.

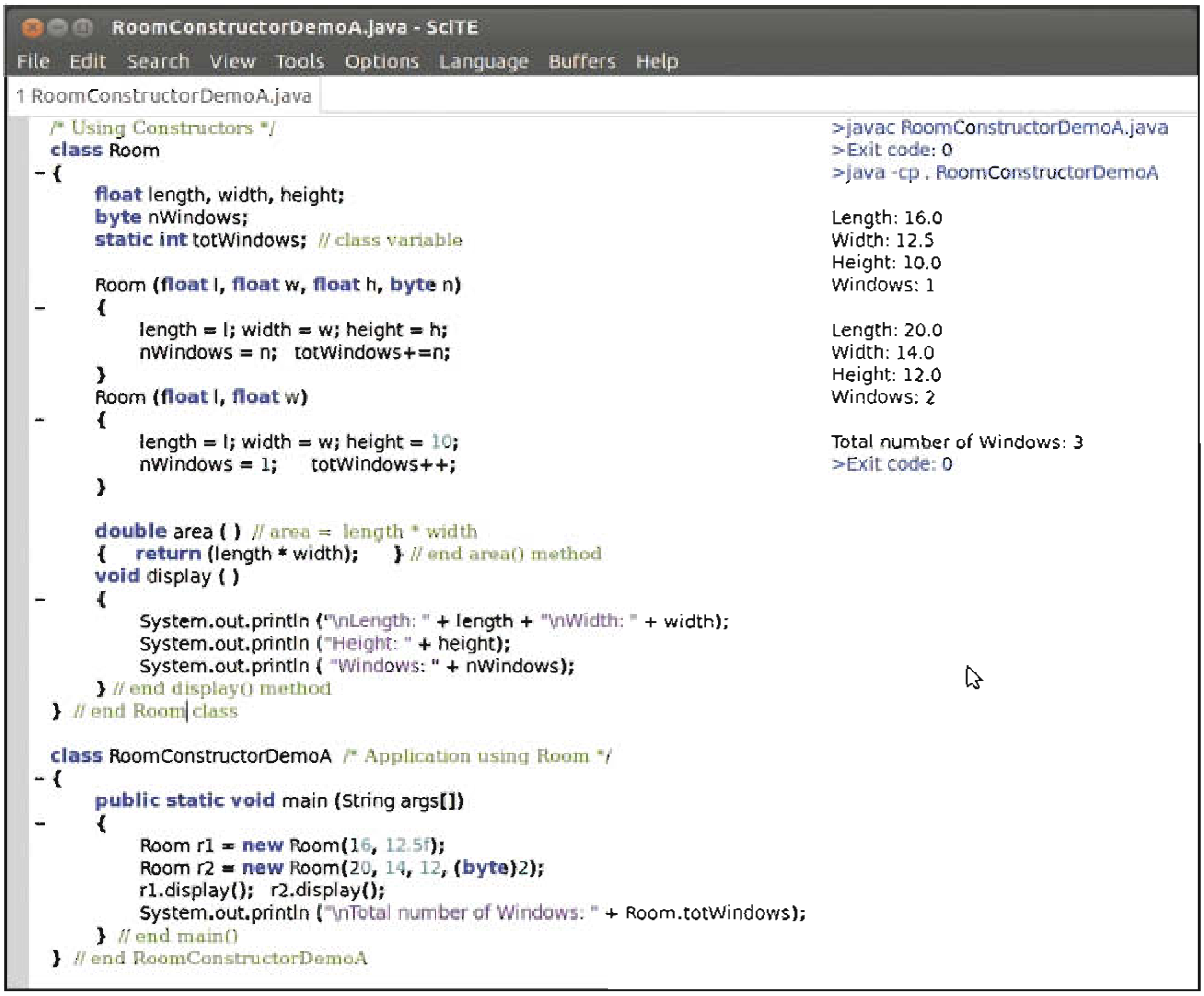
- Constructor is invoked implicitly only when an object is constructed using new operator.
- Constructor cannot be invoked explicitly elsewhere in the program.

Like other methods, constructor can also be overloaded. It is possible with varying list of parameters. For example, we may write our own constructors to initialize the attributes of 'Room' object with different parameters as follows:

Room(float l, float w, float h, byte n) : To initialize length to l, width to w, height to h and nWindows to n

Room(float l, float w) : To initialize length to l, width to w, height to 10, nWindows to 1

Program seen in figure 8.7 shows the use of constructors.


RoomConstructorDemoA.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 RoomConstructorDemoA.java

```java
/* Using Constructors */
class Room
{
    float length, width, height;
    byte nWindows;
    static int totWindows;  // class variable

    Room (float l, float w, float h, byte n)
    {
        length = l; width = w; height = h;
        nWindows = n;  totWindows+=n;
    }
    Room (float l, float w)
    {
        length = l; width = w; height = 10;
        nWindows = 1;     totWindows++;
    }

    double area ( ) // area = length * width
    {   return (length * width);    } // end area() method
    void display ( )
    {
        System.out.println ("\nLength: " + length + "\nWidth: " + width);
        System.out.println ("Height: " + height);
        System.out.println ( "Windows: " + nWindows);
    } // end display() method
} // end Room class

class RoomConstructorDemoA /* Application using Room */
{
    public static void main (String args[])
    {
        Room r1 = new Room(16, 12.5f);
        Room r2 = new Room(20, 14, 12, (byte)2);
        r1.display();  r2.display();
        System.out.println ("\nTotal number of Windows: " + Room.totWindows);
    } // end main()
} // end RoomConstructorDemoA
```

```
>javac RoomConstructorDemoA.java
>Exit code: 0
>java -cp . RoomConstructorDemoA

Length: 16.0
Width: 12.5
Height: 10.0
Windows: 1

Length: 20.0
Width: 14.0
Height: 12.0
Windows: 2

Total number of Windows: 3
>Exit code: 0
```

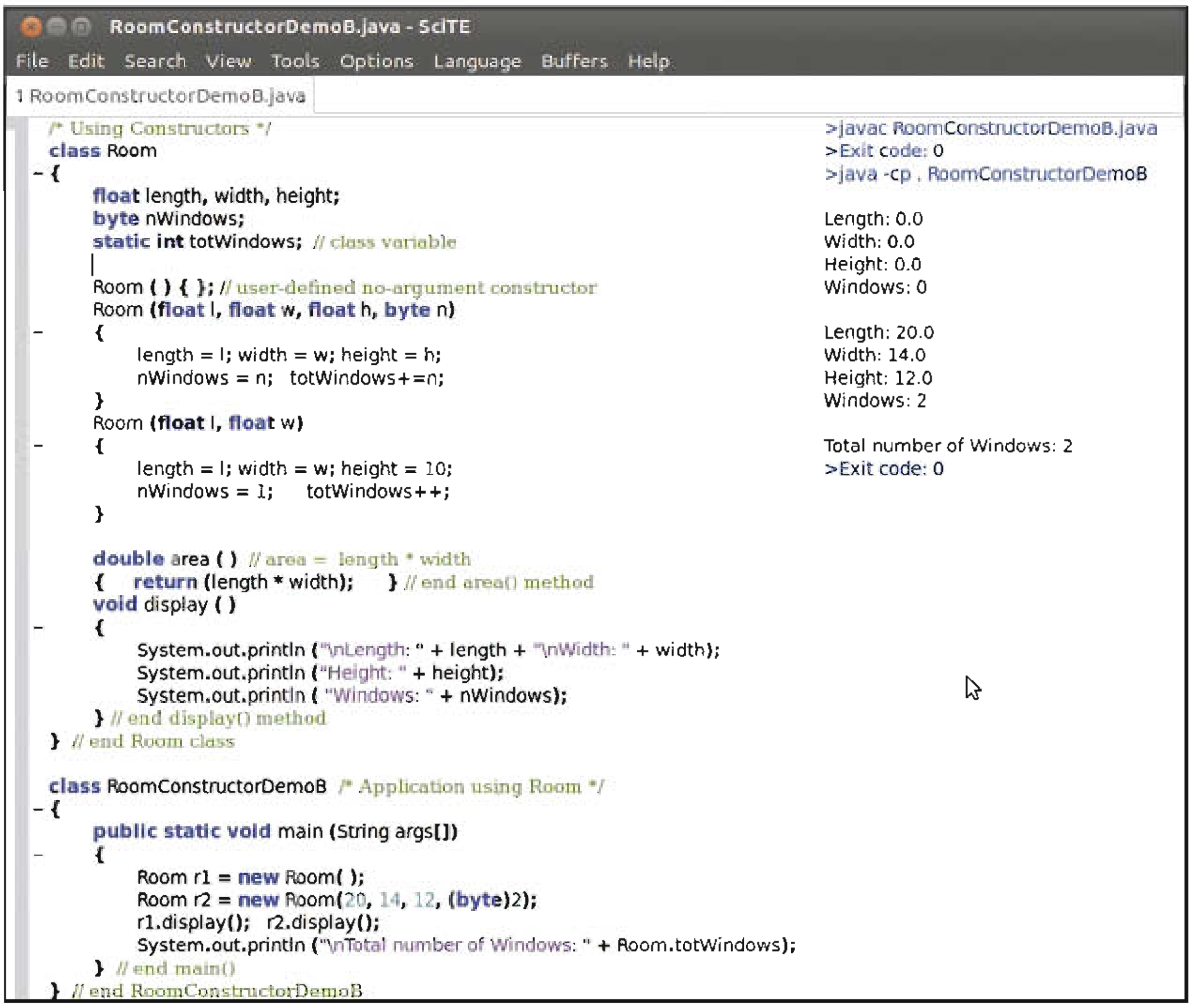
Figure 8.7 : Using constructors

In absence of user-defined constructors in a class, objects are constructed using default no-argument constructor. It initializes the attributes using default values.

In presence of user-defined constructors in a class, default constructor is no more available. An attempt to create an object using constructor without arguments, compiler returns an error. To solve this problem, we need to provide a user-defined no-argument constructor as follows: <classname> ( ) { };

In code listing given in figure 8.7, try to create a Room object in main method as follows and observe an error occurred during compilation: Room r3 = new Room();

To solve this error, add user-defined no-argument constructor 'Room () {};' and execute. See the successful execution as given in figure 8.8.


RoomConstructorDemoB.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 RoomConstructorDemoB.java

```
/* Using Constructors */
class Room
{
    float length, width, height;
    byte nWindows;
    static int totWindows; // class variable

    Room ( ) { }; // user-defined no-argument constructor
    Room (float l, float w, float h, byte n)
    {
        length = l; width = w; height = h;
        nWindows = n;   totWindows+=n;
    }
    Room (float l, float w)
    {
        length = l; width = w; height = 10;
        nWindows = 1;     totWindows++;
    }

    double area ( )  // area = length * width
    {   return (length * width);     } // end area() method
    void display ( )
    {
        System.out.println ("\nLength: " + length + "\nWidth: " + width);
        System.out.println ("Height: " + height);
        System.out.println ( "Windows: " + nWindows);
    } // end display() method
} // end Room class

class RoomConstructorDemoB  /* Application using Room */
{
    public static void main (String args[])
    {
        Room r1 = new Room( );
        Room r2 = new Room(20, 14, 12, (byte)2);
        r1.display();  r2.display();
        System.out.println ("\nTotal number of Windows: " + Room.totWindows);
    } // end main()
} // end RoomConstructorDemoB
```

```
>javac RoomConstructorDemoB.java
>Exit code: 0
>java -cp . RoomConstructorDemoB

Length: 0.0
Width: 0.0
Height: 0.0
Windows: 0

Length: 20.0
Width: 14.0
Height: 12.0
Windows: 2

Total number of Windows: 2
>Exit code: 0
```

Figure 8.8 : Using user-defined no-argument constructor

Visibility Modifiers for Access Control

Access control is about controlling visibility. So, access modifiers are also known as visibility modifiers. If a method or variable is visible to another class, then only it can be referred in another class. To protect a method or variable from such references, we use the four levels of visibility to provide necessary protection.

The Four P's of Protection are public, package (default protection), protected, and private. Access modifiers public, protected and private are used before the type of variable or method. When no modifier is used, it is the default one having visibility only within a package that contains the class.

Package is used to organize classes. To do so, package statement should be added as the first non-comment or non-blank line in the source file. When a file does not have package statement, the classes defined in the file are placed in default package. Package statement has following syntax: package <packageName>;

In this book, we will use default package. In all our programs, we have used the default access modifier till now. Table 8.1 shows the type of access modifier and its visibility.

		Type		
Access Modifier	public	(default: package)	protected	private
Visibility	widest	→ → →	→ →	narrowest

Table 8.1 : Type of access modifier and its visibility

We will now see examples of default and private modifier.

public

Any method or variable is visible to the class in which it is defined. If we want to make it visible to all the classes outside this class, declare the method or variable to have public access. This is the widest possible access. It provides visibility to classes defined in other package also. To provide public access, use access modifier public before type of variable or method. For example,

```
public float length
```

```
public double area ( )
```

Note that public variables and methods are visible anywhere and thus can be accessed from other source files and packages also.

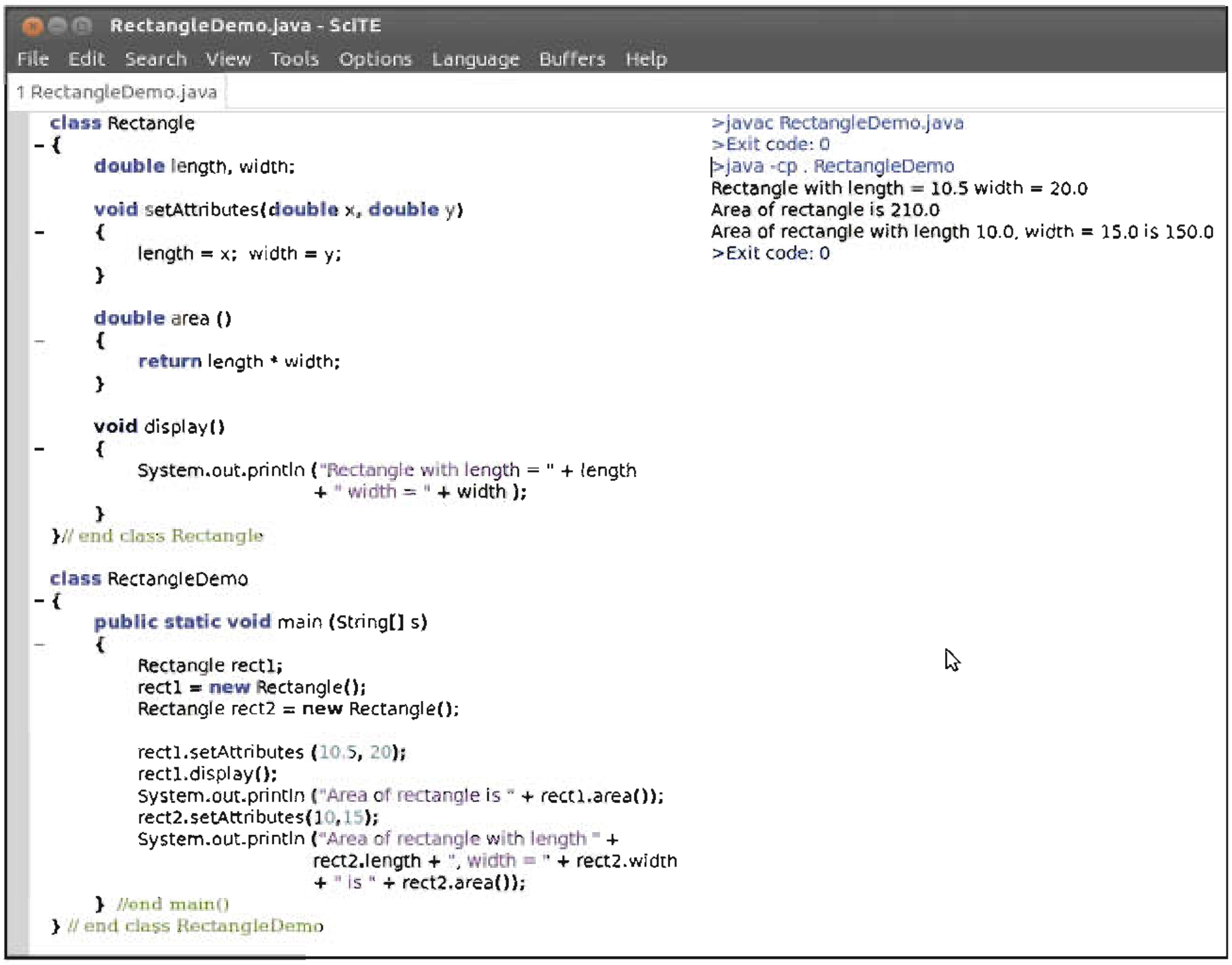
We have used public keyword with main() method to make it available to everyone.

```
public static void main(String[] args) { ... }
```

Package (without any modifier)

This is the next level of access that has no precise name. It is indicated by the lack of any access modifier keyword in a declaration. This is the default level of protection. The scope is narrower than public variables. The variable or method can be accessed from anywhere in the package that contains the class, but not from outside that package. Note that a source file without package statement is considered as a package by default. So, in our programs till now, it is as good as public.

Refer the program shown in figure 8.9. Here 'Rectangle' class has two attributes: length and width. It also has various methods. We have not used any modifier keyword, so they have package protection by default. Due to this reason, they are directly accessible in another class 'RectangleDemo' defined in the same source file (default package).


RectangleDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 RectangleDemo.java

```
class Rectangle
{
    double length, width;

    void setAttributes(double x, double y)
    {
        length = x;  width = y;
    }

    double area ()
    {
        return length * width;
    }

    void display()
    {
        System.out.println ("Rectangle with length = " + length
                                + " width = " + width );
    }
}// end class Rectangle

class RectangleDemo
{
    public static void main (String[] s)
    {
        Rectangle rect1;
        rect1 = new Rectangle();
        Rectangle rect2 = new Rectangle();

        rect1.setAttributes (10.5, 20);
        rect1.display();
        System.out.println ("Area of rectangle is " + rect1.area());
        rect2.setAttributes(10,15);
        System.out.println ("Area of rectangle with length " +
                                rect2.length + ", width = " + rect2.width
                                + " is " + rect2.area());
    } //end main()
} // end class RectangleDemo
```

```
>javac RectangleDemo.java
>Exit code: 0
>java -cp . RectangleDemo
Rectangle with length = 10.5 width = 20.0
Area of rectangle is 210.0
Area of rectangle with length 10.0, width = 15.0 is 150.0
>Exit code: 0
```

Figure 8.9 : Default visibility modifier, available everywhere in a package

protected

This level of protection is used to allow the access only to subclasses or to share with the methods declared as "friend". Thus the visibility is narrower than previous two levels; but wider than full privacy provided by fourth level "private".

Use of protected protection will become more relevant when we use inheritance concept of object-oriented programming.

private

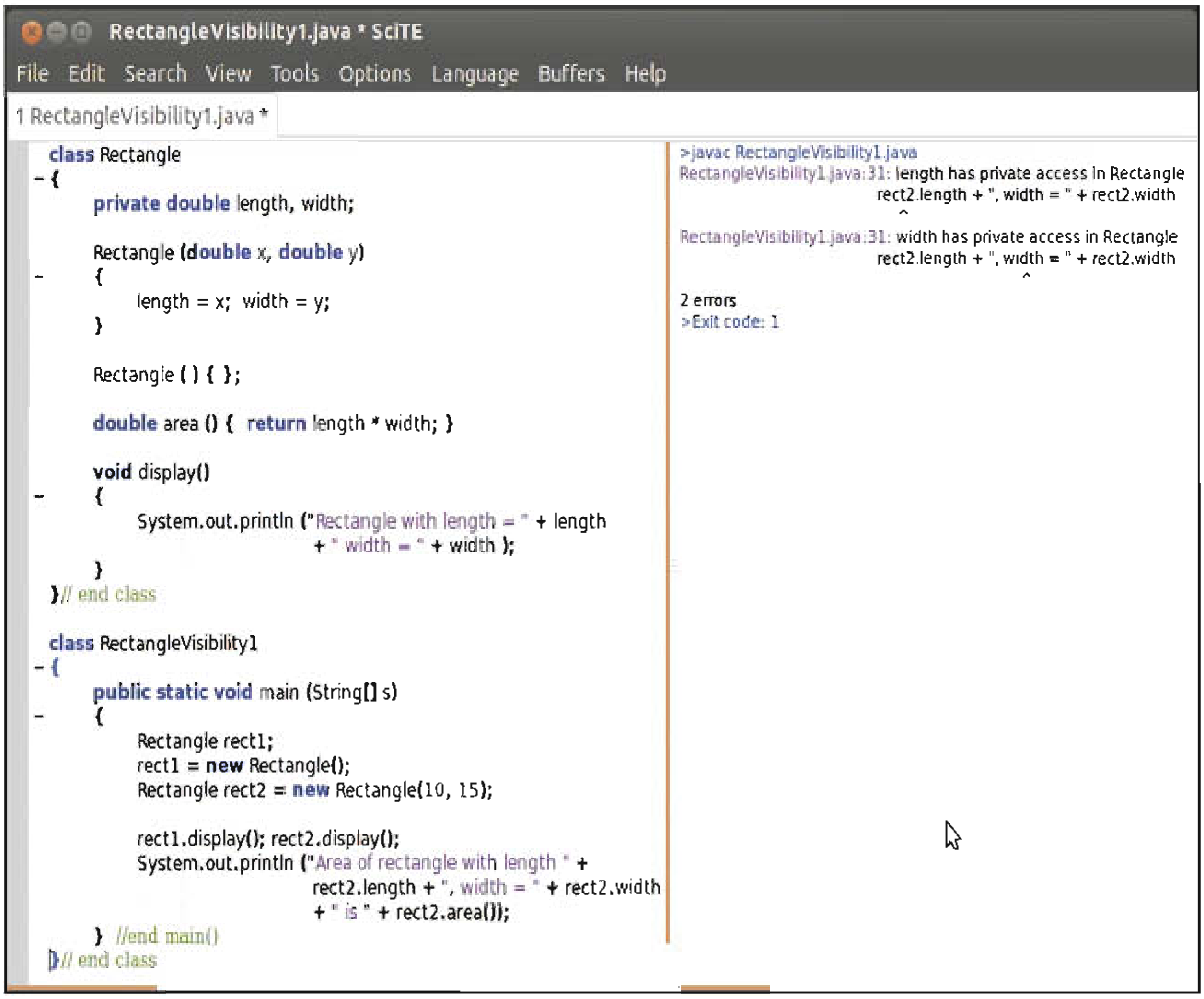
Highest level of protection can be achieved by using "private" protection level. This provides the narrowest visibility. The private methods and variables are directly accessible only by the methods defined within a class. They cannot be seen by any other class.

This may seem extremely restrictive, but it is, in fact, a commonly used level of protection. It provides data encapsulation; hiding data from the world's sight and limiting its manipulation. Anything that shouldn't be directly shared with anyone including its subclass is private.

The best way to provide data encapsulation is to make as much data as private as possible. It separates design from implementation, minimizes the amount of information one class needs to know about another to get its job done.

Let us modify code listing given in figure 8.9 and declare length and width as private. As private members are directly available only within the same class, it does not give any error when accessed in area() or display() method. When they are accesses in main method() of RectangleDemo class, it will show an error.

The modified code is shown in figure 8.10. Here, we have used constructors in addition to private instance variables. Note the error in the output pane that says 'length has private access in Rectangle'. This means that it is not accessible in class 'RectangleVisibility1'.


RectangleVisibility1.java * SciTE
File Edit Search View Tools Options Language Buffers Help
1 RectangleVisibility1.java *

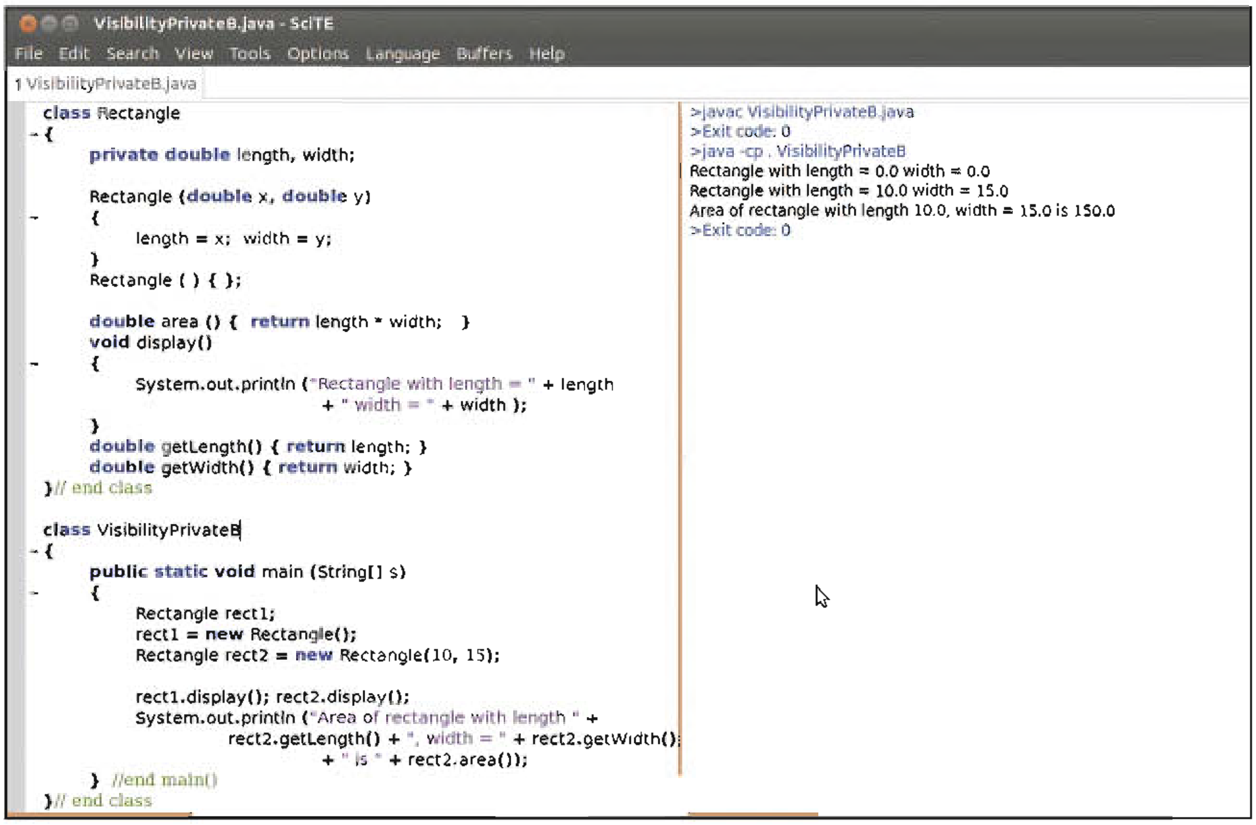
```
class Rectangle
{
      private double length, width;

      Rectangle (double x, double y)
      {
            length = x;  width = y;
      }

      Rectangle ( ) { };

      double area () {  return length * width; }

      void display()
      {
            System.out.println ("Rectangle with length = " + length
                                 + " width = " + width );
      }
}// end class

class RectangleVisibility1
{
      public static void main (String[] s)
      {
            Rectangle rect1;
            rect1 = new Rectangle();
            Rectangle rect2 = new Rectangle(10, 15);

            rect1.display(); rect2.display();
            System.out.println ("Area of rectangle with length " +
                                 rect2.length + ", width = " + rect2.width
                                 + " is " + rect2.area());
      } //end main()
}// end class
```

```
>javac RectangleVisibility1.java
RectangleVisibility1.java:31: length has private access in Rectangle
                            rect2.length + ", width = " + rect2.width
                                 ^
RectangleVisibility1.java:31: width has private access in Rectangle
                            rect2.length + ", width = " + rect2.width
                                                                ^
2 errors
>Exit code: 1
```

Figure 8.10 : Error while accessing private instance variable from another class

The problem now is how to access private variables from another class? It can be made available indirectly by the methods that are accessible by another class. See code listing given in figure 8.11.

Here we have added two methods getLength() and getWidth(). As no modifier is used, they have package visibility and so directly available in another class, 'VisibilityPrivateB' here. Through these methods, we can get the values of private 'length' and 'width' data fields (instance variables). See the use of getLength() and getWidth() method calls in a last output statement in main() method of class 'VisibilityPrivateB'.


VisibilityPrivateB.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 VisibilityPrivateB.java

```
class Rectangle
{
      private double length, width;

      Rectangle (double x, double y)
      {
            length = x;  width = y;
      }
      Rectangle ( ) { };

      double area () {  return length * width;   }
      void display()
      {
            System.out.println ("Rectangle with length = " + length
                                 + " width = " + width );
      }
      double getLength() { return length; }
      double getWidth() { return width; }
}// end class

class VisibilityPrivateB
{
      public static void main (String[] s)
      {
            Rectangle rect1;
            rect1 = new Rectangle();
            Rectangle rect2 = new Rectangle(10, 15);

            rect1.display(); rect2.display();
            System.out.println ("Area of rectangle with length " +
                    rect2.getLength() + ", width = " + rect2.getWidth()
                                 + " is " + rect2.area());
      } //end main()
}// end class
```

```
>javac VisibilityPrivateB.java
>Exit code: 0
>java -cp . VisibilityPrivateB
Rectangle with length = 0.0 width = 0.0
Rectangle with length = 10.0 width = 15.0
Area of rectangle with length 10.0, width = 15.0 is 150.0
>Exit code: 0
```

Figure 8.11 : Accessing private variables through public or package methods

Accessor and Mutator Methods

When we restrict access to data by declaring them as private, our purpose is to protect them from getting directly accessed or modified by methods of other class. If we want to allow such data to be used by others, then we write "accessor" methods. If we want to allow such data to be modified by others, then we write "mutator" methods.

Conventionally, naming of accessor and mutator methods is to capitalize the first letter of variable name and then prepend the variable name with the prefixes get and set respectively. Due to this convention, accessor methods are also known as "getter" and mutator methods as "setter".

Observe that in code listing shown in figure 8.11 we have used "getter" methods getLength() and getWidth() methods. If we change the type of variable 'length', it will be hidden from other users. It affects only the implementation of accessor method 'getLength()'.

If we want to allow other methods to read only the data value, we should use "getter" methods.

If we want to allow other methods to modify the data value, we should use "setter" methods. For example, 'setLength()' method can be defined to set the value of 'length' attribute using passed argument as follows:

```
        void setLength(float l) { length = l; }
```

Use of accessor and mutator methods will prevent the variables from getting directly accessed and modified by other users of the class. It may seem a little difficult to get used to this as we need to write get and set method for each and every instance variable as per need. But, this minor inconvenience will reward us with an ease of reusability and maintenance.

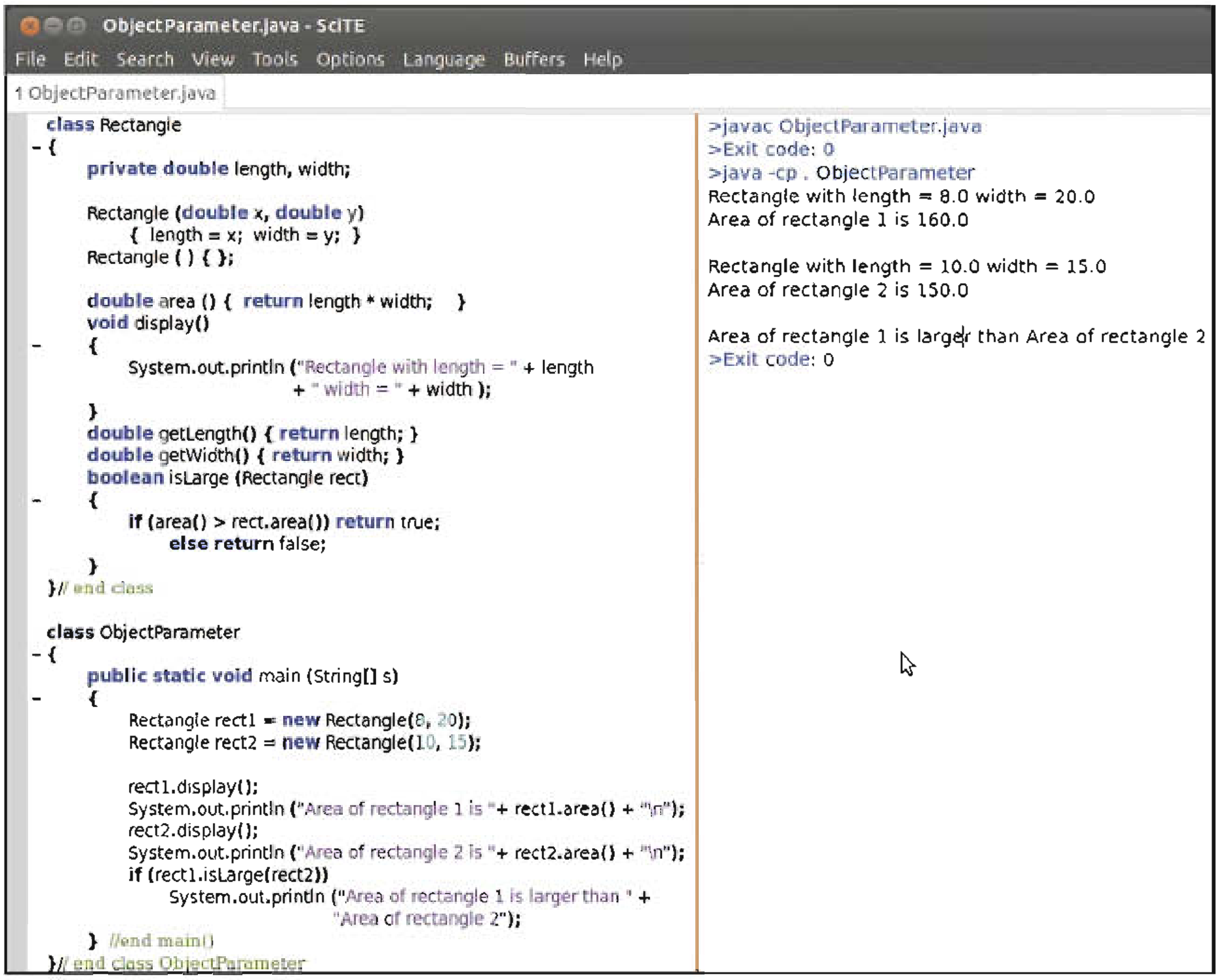
Passing object as a parameter in a method

Just like variables of primitive data types and available built-in data types, objects can also be passed as parameters in a method.

For example, we want to determine whether the area of invoking rectangle object is larger than another rectangle or not. For this, we may write a method where another rectangle object is passed as an argument or parameter. Let us add method 'isLarge' in 'Rectangle' class as shown in figure 8.12 and use it in main() method.

```
boolean isLarge (Rectangle rect)
{ if (area() > rect.area() ) return true; else return false; }
```

See the method call in if statement in the main() method as shown in figure 8.12: rect1.isLarge(rect2). Here 'rect1' is a calling or invoking object and 'rect2' is an object passed as parameter. In the isLarge() method, area() refers to area of calling object 'rect1' and rect.area() refers to area of an object passed as argument.


ObjectParameter.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 ObjectParameter.java

```
class Rectangle
{
    private double length, width;

    Rectangle (double x, double y)
        { length = x; width = y; }
    Rectangle ( ) { };

    double area () { return length * width;   }
    void display()
    {
        System.out.println ("Rectangle with length = " + length
                            + " width = " + width );
    }
    double getLength() { return length; }
    double getWidth() { return width; }
    boolean isLarge (Rectangle rect)
    {
        if (area() > rect.area()) return true;
            else return false;
    }
}// end class

class ObjectParameter
{
    public static void main (String[] s)
    {
        Rectangle rect1 = new Rectangle(8, 20);
        Rectangle rect2 = new Rectangle(10, 15);

        rect1.display();
        System.out.println ("Area of rectangle 1 is "+ rect1.area() + "\n");
        rect2.display();
        System.out.println ("Area of rectangle 2 is "+ rect2.area() + "\n");
        if (rect1.isLarge(rect2))
            System.out.println ("Area of rectangle 1 is larger than " +
                                "Area of rectangle 2");
    } //end main()
}// end class ObjectParameter
```

```
>javac ObjectParameter.java
>Exit code: 0
>java -cp . ObjectParameter
Rectangle with length = 8.0 width = 20.0
Area of rectangle 1 is 160.0

Rectangle with length = 10.0 width = 15.0
Area of rectangle 2 is 150.0

Area of rectangle 1 is larger than Area of rectangle 2
>Exit code: 0
```

Figure 8.12 : Passing an object as a parameter

Remember that parameters of primitive types are passed by value. The values of actual parameters are copied to formal parameters and then function is executed. Changes made to formal parameters are not affecting actual parameters.

It is to be noted that object parameters are passed by reference. So, whatever modifications are performed to the object inside the method, the original object is affected as well. Here, an address (and not value) of the actual parameter is copied to formal parameter.

Inheritance

Object-oriented programming languages provide reusability feature using inheritance. Inheritance allows us to build new class with added capabilities by extending existing class.

Inheritance models 'is-a' relationship between two classes. For example, classroom is a room, student is a person. Here, room and person are called parent class; classroom and student are called child classes. Parent class is also referred to as superclass or base class. In the same way; child class is also referred to as subclass, derived class or extended class.

Whenever two classes have 'is-a' relationship, we use inheritance. Common features are kept in superclass. A subclass inherits all instance variables and methods from superclass and it may have its own added variables and methods. Note that constructors are not inherited in subclass. For example, like room, classroom also has variables length, width, height, number of windows. In addition, it has number of benches and capacity of each bench to accommodate students. Similarly, subclass inherits all methods of superclass and it may have additional methods. Here, subclass Classroom has additional methods: show(), display(), getSeats() and its constructor methods.

Figure 8.13 shows a class diagram that has a class named 'Classroom' derived from its parent class named 'Room'. See that the arrow points from subclass towards superclass. In subclass, only additional attributes and methods are to be shown.

Note that a subclass is not a subset of superclass. In fact, subclass usually contains more information and methods than its superclass.

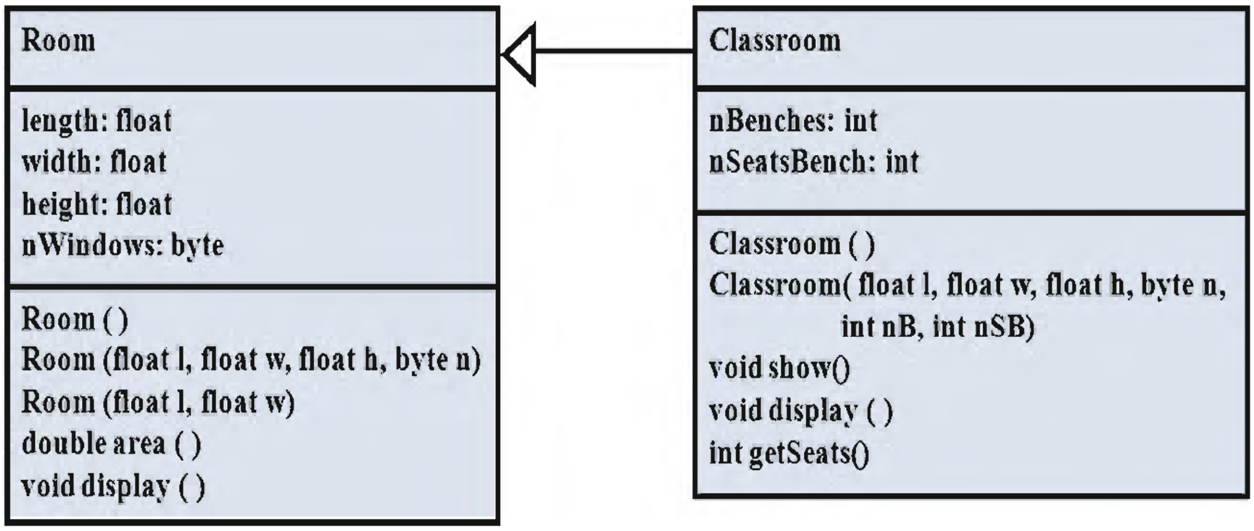


Figure 8.13 : Inheritance Class Diagram

When an object of subclass is instantiated, memory is allocated for all its attributes including inherited ones. Figure 8.14 shows the instances of superclass Room and subclass Classroom.

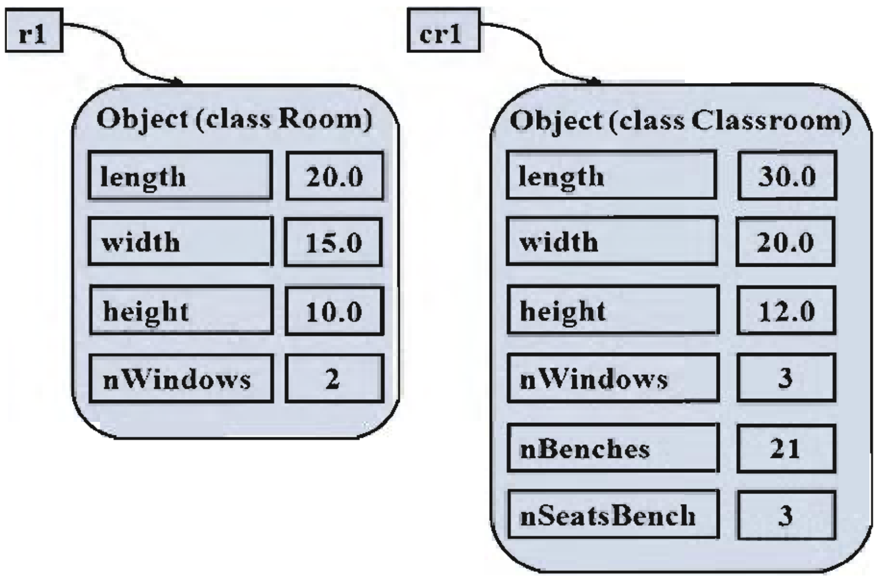


Figure 8.14 : Instances of superclass and subclass

In Java, to create a subclass, keyword 'extends' is used in the class definition. Let us create a subclass 'Classroom' using existing class 'Room' as shown in code listing 8.2.

```
class Room
{
        float length, width, height;
        byte nWindows;
        static int totWindows;  // class variable

        Room ( ) { }; // user-defined no-argument constructor
        Room (float l, float w, float h, byte n)
        {
                length = l; width = w; height = h;
                nWindows = n;   totWindows+=n;
        }
        Room (float l, float w)
        {
                length = l; width = w; height = 10;
                nWindows = 1;       totWindows++;
        }

    double area ( )  // area =  length * width
        {       return (length * width);        } // end area() method

         void display ( )
        {
                System.out.println ("\nLength: " + length + "\nWidth: " + width);
                System.out.println ("Height: " + height);
                System.out.println ( "Windows: " + nWindows);
        } // end display() method
}  // end Room class
```

Code Listing 8.2 : Example of using Inheritance

Now, let us add code to create a subclass 'Classroom' by extending superclass 'Room' as shown in code listing 8.3. Subclass has two additional instance variables nBenches and nSeatsBench. 8Variable nSeatsBench denotes the number of students that can seat on one bench. It has its own additional constructors and three methods: show(), display() and getSeats().

```
class Classroom extends Room
{
        int nBenches, nSeatsBench;
        Classroom( ) {};
        Classroom( float l, float w, float h, byte n, int nB, int nSB)
        {
              super (l,w,h,n);
              nBenches = nB; nSeatsBench = nSB;
        }
        void show()
        {
              super.display();
              System.out.println ("Benches: " + nBenches );
              System.out.println ("Seats per Bench: " + nSeatsBench );
              System.out.println ("Total Seats in a class: " + getSeats() );
        }
        void display()
        {
              System.out.println("\nClassroom with length " + length + " feet, width "
                  + width + " feet\nhas " + nBenches + " Benches, each to accomodate "
                  + nSeatsBench + " Students\nSo, Total seats in a class is  "
                  + getSeats());
        }
        int getSeats() {return nBenches * nSeatsBench; }
}  // end class Classroom
```

Code Listing 8.3 : Code for subclass named Classroom

In sub class 'Classroom', user defined constructor and method show() have reused the code written in superclass 'Room'.

As constructors of a superclass are not inherited in the subclass, keyword 'super' is used to call the constructor of superclass in the constructor of subclass. This call must be the first statement in the constructor. When there is no explicit call to constructor of superclass, no-argument constructor of superclass is implicitly called as the first statement 'super()'.

Method show() is used to display attributes of Classroom object. To display first four attributes inherited from superclass 'Room', we want to use existing code in display() method of superclass.

Now display() method is available in subclass also. In show(), our intention is call display() method of superclass. To do so, we have used super.display().

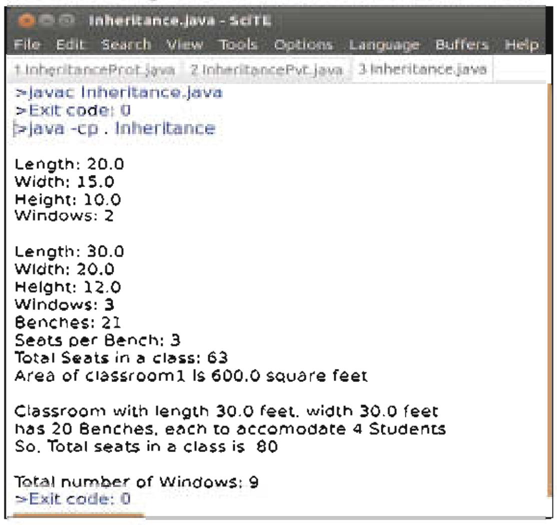
When superclass and subclass have methods with same signature, a superclass method is said to be overridden in the subclass. Method display() in subclass is used to override display() method of super-class. It means that we want to display the details in a different way without reusing the display() method of superclass. When such method of superclass is to be referred, we need to use keyword 'super' with dot operator and method name. Here we have used super.display() in show() method to invoke display() method of superclass. Method getSeats() is used to compute the seating capacity of a classroom.

Let us use these classes in an application by adding code as given in code listing 8.4. Here, we have created objects of both superclass and subclass.

```
class Inheritance  /* Application using Room, Classroom */
{
        public static void main (String args[])
        {
                Room r1 = new Room(20, 15, 10, (byte)2);
                r1.display();
                Classroom cr1 = new Classroom (30, 20, 12, (byte)3, 21, 3);
                cr1.show();
                System.out.println("Area of classroom1 is " + cr1.area() + " square feet");
                Classroom cr2 = new Classroom (30,30,10, (byte)4, 20, 4);
                cr2.display();
                System.out.println ("\nTotal number of Windows: " + Room.totWindows);
        }  // end main()
}  // end Inheritance
```

Code Listing 8.4 : Application using Room, Classroom

All instance variables and methods are inherited from super class to subclass. Thus method area() of superclass can be invoked using an object of subclass also using cr1.area(). When overridden method is referred in an application using subclass object, it calls the method of subclass as cr2.display(). The output of the code created by combining code 8.2, 8.3 and 8.4 is shown in figure 8.15.


Inheritance.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 InheritanceProt.java 2 InheritancePvt.java 3 Inheritance.java

```
>javac Inheritance.java
>Exit code: 0
>java -cp . Inheritance

Length: 20.0
Width: 15.0
Height: 10.0
Windows: 2

Length: 30.0
Width: 20.0
Height: 12.0
Windows: 3
Benches: 21
Seats per Bench: 3
Total Seats in a class: 63
Area of classroom1 is 600.0 square feet

Classroom with length 30.0 feet, width 30.0 feet
has 20 Benches, each to accomodate 4 Students
So, Total seats in a class is  80

Total number of Windows: 9
>Exit code: 0
```

Figure 8.15 : Output of Inheritance using classes Classroom and Room

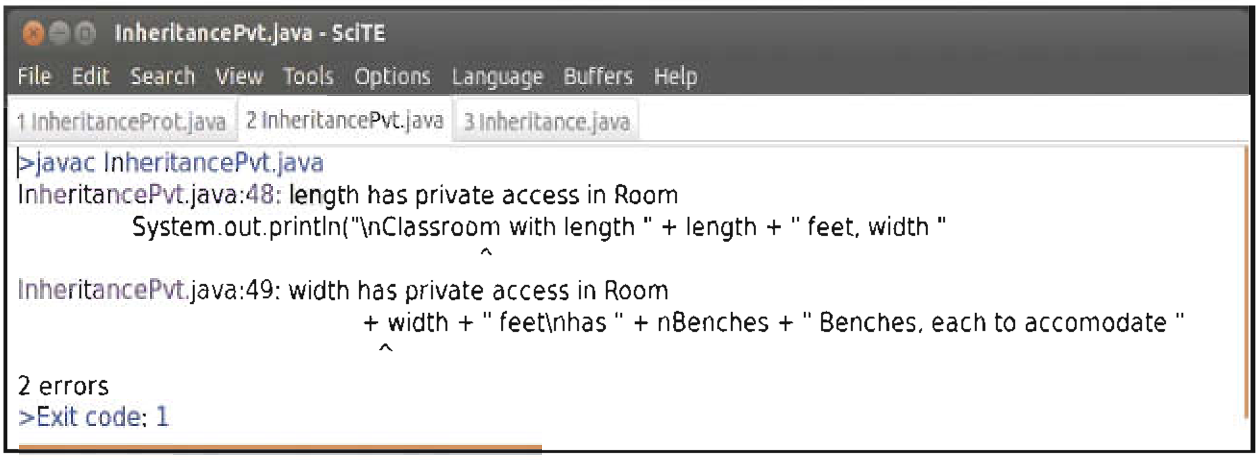
Private members of superclass

In the previous example, all instances have package visibility. So they are available for direct use in the entire package. If we try to access r1.length or cr1.width directly in main() method, there will not be any error.

If we change the visibility of instance variables of superclass to private, these variables are not directly accessible outside the class. Remember that these attributes belongs to subclass also, but still private instance variables or methods are not visible even in its subclass.

Modify declaration of instance variables in code listing 8.2 to make them private as follows and observe an error in output as shown in figure 8.16.

```
private float length, width, height;

private byte nWindows;
```


InheritancePvt.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 InheritanceProt.java
2 InheritancePvt.java
3 Inheritance.java

```
>javac InheritancePvt.java
InheritancePvt.java:48: length has private access in Room
            System.out.println("\nClassroom with length " + length + " feet, width "
                                                           ^
InheritancePvt.java:49: width has private access in Room
                                   + width + " feet\nhas " + nBenches + " Benches, each to accomodate "
                                     ^
2 errors
>Exit code: 1
```

Figure 8.16 : Private members of a class are not accessible outside its class

Remember that private members are directly available only in the class in which they are defined and nowhere else. To make them available elsewhere, use public accessor and mutator methods; 'getter' and 'setter' methods.

Protected members of superclass

As studied before in the topic about 'Visibility Modifiers', 'protected' members are available as 'private' members in the inherited subclass. Modify the instance variables of class 'Room' to protected as follows.

```
protected float length, width, height;

protected byte nWindows;
```

When we execute the modified code we will get the same result as shown in figure 8.15. It is to be noted that Java does not support multiple inheritance. A subclass can be derived from only one superclass.

Composition and Aggregation

Composition and aggregation are the construction of classes that incorporate other objects. They establish a "has-a" relationship between classes.

For example, if we define class 'Library', it has a reading room. Here reading room is an object of the class 'Room'. Thus Library has a Room. When a class includes objects of other class, it is also referred to as container class.

Other examples are :

- Person has a name, address. Here, name is of the class Name with three attributes first name, middle name and last name; address is of class Address with attributes house number, apartment/ society name, area, city, state, country, pin code.
- Car has a steering, wheels and engine. Here steering is of the class Steering, wheel is of the class Wheel and engine is of the class Engine. All the three attributes of class Car are components or parts of a car.

Let us create class 'Library' that contains following attributes :

- nBooks: int, number of books in library
- nMagazines: int, number of magazines subscribed in library
- nNewspapers: int, number of newspapers subscribed in library
- readingRoom: Room

See that readingRoom is not of the basic data types. It is of the type 'Room' class.

Code listing 8.5 shows the code to create class named Room and Library and use them in application named 'Container.java'.

```
/* Using objects as data members in a container class */
class Room
{
        protected float length, width, height;
        protected byte nWindows;
        static int totWindows;  // class variable

        Room ( ) { }; // user-defined no-argument constructor
        Room (float l, float w, float h, byte n)
        {
                length = l; width = w; height = h;
                nWindows = n;   totWindows+=n;
        }
        Room (float l, float w)
        {
                length = l; width = w; height = 10;
                nWindows = 1;       totWindows++;
        }
```

```
            double area ( )  // area =  length * width
          {      return (length * width);       } // end area() method
            void display ( )
          {
                 System.out.println ("Length: " + length + "\nWidth: " + width);
                 System.out.println ("Height: " + height);
                 System.out.println ( "Windows: " + nWindows);
          } // end display() method
}  // end Room class

class Library
{
          int nBooks, nMagazines, nNewspapers;
          Room readingRoom;
          Library( ) {};
          Library( int nB, int nM, int nN, Room r)
          {
                 nBooks = nB; nMagazines=nM; nNewspapers=nN;
                 readingRoom=r;
          }
          void display()
          {
                 System.out.println ("\nLibrary Details:\nNumber of books: " + nBooks );
                 System.out.println ("Number of subscribed magazines: " + nMagazines );
                 System.out.println ("Number of subscribed newspapers : " + nNewspapers);
                 System.out.println ("Reading Room:");
                 readingRoom.display();
          }
}  // end class Library

class Container  /* Application using Room, Library */
{
           public static void main (String args[])
          {
                 Room r1 = new Room(20, 15, 10, (byte)2);
                 r1.display();
                 Library lib = new Library (300, 20, 5, r1);
                 lib.display();
          }  // end main()
}  // end class Container
```

Code Listing 8.5 : Example of Composition and aggregation

Observe the following points in code listing 8.5,

- In main() method of class 'Container',
  - An object lib of class 'Library' is created using a constructor with four arguments. Last argument is r1 of class 'Room'.
  - Method display() of class 'Library' is invoked using lib object.
- In 'Library' class,
  - An object readingRoom is an attribute of class 'Room'
  - Constructor with four arguments uses last argument 'r' of the class 'Room', assigns the values of Room attributes using an assignment statement 'readingRoom = r;'
  - Defines display() method which invokes display() method of 'Room' class using 'readingRoom.display();'. Note that display() method is not overridden; we have not used inheritance. Readers may use another name for this method, say show() in class 'Library' and use 'lib.show()' in main() method instead of 'lib.display()'.

In this scenario, reader may think: Why not to use inheritance? Why not to inherit 'Library' class from 'Room' class and add three additional attributes?

Note that 'Library' is not of the type of 'Room'. There is no 'is-a' relationship between 'Library' and 'Room'. There is 'has-a' relationship; 'Library' has a 'Room' used as reading room. So, we have used readingRoom of type 'Room' as an attribute of class 'Library'.

Summary

We have seen how to create class and object, how to access instance variables of class and invoke instance methods using objects. Constructors are special methods with the same name as class name, no argument and no return type. Constructors are called implicitly when an object is instantiated using new operator. Instance variables belong to each object. Class variables and methods are defined using 'static' keyword. Static members are allocated memory only once per class and shared by all objects of the class. They belong to class and not to object. We also studied about access modifiers that determine visibility of instance members. Private members are visible only within a class where they are defined, protected members are visible only in inherited subclasses, package members are visible anywhere within a package, public members are visible anywhere. For protection purpose, it is advisable to use private instance variables and supply 'getters' and 'setters' methods as public. In the last, we studied how to inherit new class using existing class; re-use, extend and override the methods. We also see the use of an object as an instance variable in class. This enables creating classes as composition and aggregation of objects.

EXERCISE

1. What do you mean by instantiation of an object ?
2. Give an example for the need of class variable.
3. Write the differences between methods and constructors.

4. Write about accessor and mutator methods.

5. How can a superclass constructor be invoked from the subclass ?

6. How can an overridden method of superclass be invoked from the subclass ?

7. Write a short note on 'Access modifiers'.

8. Explain the use of inheritance and composition or aggregation based on type of relationship between classes.

9. Explain the difference between method overloading and method overriding.

10. Choose the correct option from the following :

(1) Which of the following defines attributes and methods ?

(a) Class (b) Object (c) Instance (d) Variable

(2) Which of the following keyword is used to declare Class variables and class methods ?

(a) static (b) private (c) public (d) package

(3) Which of the following operator creates an object and returns its reference ?

(a) dot (.) (b) new (c) colon (:) (d) assignment (=)

(4) Which of the following method can be called without creating an instance of a class ?

(a) Instance method (b) Class method

(c) Constructor method (d) All of the above

(5) Which of the following refers more than one method having same name but different parameters ?

(a) Overloaded methods (b) Overridden methods

(c) Duplicate methods (d) All of the above

(6) Which method is invoked automatically with creation of an object ?

(a) Instance method (b) Constructor

(c) Class method (d) All of the above

(7) Which of the following is the keyword used to refer a superclass constructor in subclass constructor ?

(a) extends (b) super

(c) name of the superclass (d) new

(8) Which of the following is used to invoke an instance method in Java ?

(a) The name of the object, colon(:) and the name of the method

(b) The name of the object, dot(.) and the name of the method

(c) The name of the class, colon(:) and the name of the method

(d) The name of the class, dot(.) and the name of the method

(9) Which of the following is accessible by instance methods ?

(a) Only instance variables (b) Only class variables

(c) Both instance variables and class variables (d) All of the above

(10) When methods in the superclass and subclass have same name and signature, what are they called ?

(a) Overloaded methods (b) Overridden methods

(c) Inherited methods (d) All of the above

LABORATORY EXERCISE

Write Java Programs for the following :

1. Create a class named 'FixedDeposit' that contains three attributes (principal amount, annual interest rate and period (years) of deposit) and a method that returns maturity amount using compound interest. Create another class named 'FixedDepositDemo' with main() method. In main() method, create two objects, assign the values to their attributes and display them with maturity amount.

2. Add following constructors in 'FixedDeposit' class and use them to create objects.

   a. No-argument constructor that initializes principal amount with 1000, annual interest rate with 5% and period of deposit as 3 years.

   b. Parameterized constructor with 3 arguments to initialize 3 attributes.

3. Modify the visibility of instance variables in class 'FixedDeposit' as 'private' in lab exercise 1 and try to execute. Will it give an error ? If so, add display() method in the class to display the value of the instance variables.

4. Add accessor and mutator methods to get and set three attributes of the class 'FixedDeposit'. Using these methods, display the value of private instance variables in main() method.

5. Add class variable 'totDeposit' in a class that contains total amount of deposited principal amount. Modify the constructors and setter methods so as to get the total deposit amount. Write a method to display the value of 'totDeposit' variable and show its use.

6. Using 'Rectangle' class, derive a subclass 'Box' having additional attribute 'height' and method 'volume'. (Volume = height x width x length = height x area)