CHAPTER

13

METHODS IN JAVA

A method represents a group of statements that performs a task. Here 'task' represents a calculation or processing of data or generating a report etc., For example, take `sqrt()` method. It calculates square root value and returns that value. A method has two parts:

- Method header or Method prototype
- Method body

Method Header or Method Prototype

It contains method name, method parameters and method return data type. Method prototype is written in the form:

```
returndatatype  methodname(parameter1, parameter2,…)
```

Here, method name represents the name given to the method. After the method name, we write some variables in the simple braces. These variables are called parameters. Method parameters are useful to receive data from outside into the method. This data can be used by the method. Return data type is written before the method name to represent what type of result/data the method is returning. Let us examine some method prototypes:

- `double sqrt(double num)` : Here, sqrt is the method name. num is method parameter that represents that this method accepts a double type value. This method calculates square root value of num and returns that result which is again double type. We can call this method something like:

```
double result = obj.sqrt(25.6);
```

- `void sum()`: This method is not accepting any data from outside. So after the method name we write empty braces (). This method does not return any value. So void is written before the method name. 'void' means 'no value'. Probably this method is doing its own calculations and displaying the result (but not returning it). We can call this method as:

```
obj.sum();
```

- `long factorial(int x)`: This method is calculating factorial value of given value x which is an integer. So at the time of calling this method, we should pass x value to the method. It is returning long type value which is factorial value of x. We can call this method as:

```
long result = obj.factorial(10);
```

- `double power(double x, int n)`: This method calculates power value of a number when it is raised to a particular power. When it is called, we should pass two values, the number (x) and its power (n). It then calculates x^n value and returns that result as double value. So double is written before the method name. We can call this method as:

```
double result = obj.power(5.12, 3);
```

- `Employee calculateTax(Employee obj)`: This method is intended to calculate income tax of an employee. It accepts Employee object which contains salary of an employee. By taking the salary, it calculates the tax and stores it again in Employee object which is returned. So while calling this method, we should pass an Employee object e2 and it returns another Employee object e3, as shown here:

```
Employee e3 = e1.calculateTax(Employee e2);
```

- `int[ ][ ] matrixSum(int a[ ][ ], int b[ ][ ])`: This method is taking two integer type 2D arrays which represent two matrices. It calculates sum of these two matrices and returns the sum matrix, which is again a 2D array of integer type. Hence, before this method name, we wrote `int[ ][ ]`. If arr1 and arr2 are two matrices being passed to this method, we can call this method as:

```
arr3 = obj.matrixSum(arr1, arr2);
```

Method Body

Below the method header, we should write the method body. Method body consists of a group of statements which contains logic to perform the task. Method body can be written in the following format:

```
{
     statements;
}
```

For example, we want to write a method that calculates sum of two numbers. It may contain the body, as shown here:

```
{
     double c = a+b;
     System.out.println("Sum of two= "+ c);
}
```

If a method returns some value, then a return statement should be written within the body of the method, as:

```
return x;     //return x value
return (x+y);   //return sum of x and y
return -1;  //return -1
return  obj;  //return object obj
return arr;   //return array arr
```

For example, a method that calculates sum of two numbers can return the result using return statement, as:

```
{
    double c = a+b;
    return c;
}
```

A method can never return more than one value. Hence the following statements will be invalid:

- `return x,y;` //invalid – returning two values
- `return x; return y;` //invalid - two executable return statements

Understanding Methods

To understand how to write methods, let us take an example program. In this program, we write Sample class with two instance variables num1 and num2. To find the sum of these values, we can write a method as:

```
void sum()
{
   double res = num1+ num2;
   System.out.println("Sum= "+ res);
}
```

Observe the method header. Since this method does not accept any values, we did not declare any parameters in the braces ( ). This method does not return any result, so 'void' is written before the method.

Observe the method body. The sum of num1 and num2 is stored in a variable 'res' and that it displayed. Displaying the result does not mean that it is returned. To return the result, a method needs a return statement in the body. To call the preceding method, we can use a statement like:

```
s.sum();
```

Here, we are not passing any values to the method and also not catching any result from the method.

Program 1: Write a program for a method without parameters and without return type.

```
//Understanding the methods
class Sample
{
  //instance variables
  private double num1, num2;

  //parameterized constructor
  Sample(double x, double y)
  {
     num1 = x;
     num2 = y;
  }

  //method to calculate sum of num1, num2
  //this method does not accept any values and
  //does not return result
  void sum()
  {
     double res = num1+ num2;
     System.out.println("Sum= "+ res);
```

```
    }
}

class Methods
{
  public static void main(String args[ ])
  {
    //create the object and pass values 10 and 22.5 to constructor.
    //they will be stored into num1, num2.
    Sample s = new Sample(10, 22.5);

    //call the method and find sum of num1, num2
    s.sum();
  }
}
```

Output:

```
C:\> javac Methods.java
C:\> java Methods
Sum= 32.5
```

In the preceding program, the `sum()` method is acting on num1 and num2 which are instanc variables of the Sample class. Since instance variables are available in the object, while calling th method we created an object to Sample class, and using the object we called the method. Suc methods are called 'instance methods'.

Important Interview Question

What are instance methods?

Instance methods are the methods which act on the instance variables of the class. To call the instance methods, we should use the form: `objectname.methodname()`.

Let us rewrite the Program1, this time using `sum()` method with a return type. This method written as:

```
double sum()
{
   double res = num1+ num2;
   return res; //return result
}
```

Observe the return statement where the result '`res`' is returned from the method. This result can caught into a variable x, as:

```
double x = s.sum();
```

Remember a method always returns the result to the calling method. A method call is replaced the returned result. In the following program, we called the `sum()` method of Sample class from t `main()` method of Methods class. So `sum()` method will return the result to `main()` method. In t `main()` method, we can receive the result into x.

Program 2: Write a program for a method without parameters but with return type.

```
//Understanding the methods
class Sample
{
  //instance variables
  private double num1, num2;
```

```
    //parameterized constructor
    Sample(double x, double y)
    {
       num1 = x;
       num2 = y;
    }

    //method to calculate sum of num1, num2
    //this method does not accept any values
    //but returns the result
    double sum()
    {
       double res = num1+ num2;
       return res; //return result
    }
}

class Methods
{
  public static void main(String args[ ])
  {
    //create the object and pass values 10 and 22.5 to constructor.
    //they will be stored into num1, num2.
    Sample s = new Sample(10, 22.5);

    //call the method and store the result in x
    double x = s.sum();
    System.out.println("Sum= "+ x);

  }
}
```

Output:

```
C:\> javac Methods.java
C:\> java Methods
Sum= 32.5
```

Let us rewrite the Program2, where the `sum()` method not only return the result, but also receives the values from outside, as:

```
double sum(double num1, double num2)
  {
     double res = num1+ num2;
     return res; //return result
  }
```

The preceding method has two parameters num1 and num2. They receive double type values from outside. So we should pass double type values to this method at the time of calling it, as:

```
    double x = s.sum(10,22.5);
```

Here, 10 is stored into num1 as 10.0 and 22.5 is stored into num2.

Program 3: Write a program for a method with two parameters and return type.

```
//Understanding the methods
class Sample
{

//method to calculate sum of num1, num2
  //this method accepts two double values,
  //and also returns the double type result
  double sum(double num1, double num2)
```

```
    {
       double res = num1+ num2;
       return res; //return result
    }
}

class Methods
{
  public static void main(String args[ ])
 {
   //create the object to Sample class.
   Sample s = new Sample();

   //call the method and pass two values to
   //the method. Store the returned result in x.
   double x = s.sum(10,22.5);
   System.out.println("Sum= "+ x);

  }
}
```

Output:

```
C:\> javac Methods.java
C:\> java Methods
Sum= 32.5
```

Please observe that in Program3, the `sum()` method is not acting on any instance variables. In fact, there are no instance variables defined in the Sample class. We know instance variables are available in the object. Since `sum()` method is not using any instance variables, there is no need to create an object to call this method. Such methods are called 'static methods'.

Important Interview Question

What are static methods?

Static methods are the methods which do not act upon the instance variables of a class. Static methods are declared as 'static'.

So, to call the static methods, we need not create an object. We can call a static method, as:

```
Classname. methodname();
```

For example, to call the `sum()` method, we can write as, `Sample.sum(10,22.5)`. Let us write the preceding program again, to use the static method.

Program 4: Write a program for a static method that accepts data and returns the result.

```
//Understanding the methods
class Sample
{
//static method should be declared as static
  static double sum(double num1, double num2)
  {
     double res = num1+ num2;
     return res; //return result
  }
}

class Methods
{
  public static void main(String args[ ])
 {
   //call the static method using Classname.methodname().
```

```
        double x = Sample.sum(10,22.5);
        System.out.println("Sum= "+ x);

      }
}
```

Output:

```
C:\> javac Methods.java
C:\> java Methods
Sum= 32.5
```

Static Methods

A static method is a method that does not act upon instance variables of a class. A static method is declared by using the keyword 'static'. Static methods are called using `Classname.methodname()`. The reason why static methods can not act on instance variables is that the JVM first executes the static methods and then only it creates the objects. Since the objects are not available at the time of calling the static methods, the instance variables are also not available.

In the following program, we are trying to read and display the instance variable 'x' of Test class in a static method 'access()'. This gives an error at compile time. See the output.

Program 5: Write a program to test whether a static method can access the instance variable or not.

```
//static method trying to access instance variable
class Test
{
//instance var
  int x;

  //parameterized constructor
  Test(int x)
 {
   this.x = x;
 }

 //static method accessing x value
  static void access()
 {
   System.out.println("x= "+ x);
 }
}

class Demo
{
  public static void main(String args[])
 {
    Test obj = new Test(55);
    Test.access();
 }
}
```

Output:

```
C:\> javac Demo.java
Demo.java:16: non-static variable x cannot be referenced from a static context
   System.out.println("x= "+ x);
                              ^
1 error
```

But a static method can access static variables. Static variables are also declared as 'static'. In th following program, we are declaring x as a static variable. Now it is accessible in the static metho `access()`.

Program 6: Write a program to test whether a static method can access the static variable or not.

```
//static method accessing static variable
class Test
{
  //static var
  static int x = 55;

 //static method accessing x value
  static void access()
 {
   System.out.println("x= "+ x);
 }
}

class Demo
{
public static void main(String args[])
 {
     Test.access();
 }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
x= 55
```

The other name for static variable is 'class variable' and for static method is 'class method'.

Important Interview Question

What is the difference between instance variables and class variables (static variables)?

1. An instance variable is a variable whose separate copy is available to each object. A class variable is a variable whose single copy in memory is shared by all objects.

2. Instance variables are created in the objects on heap memory. Class variables are stored on method area.

Since, instance variable will have a separate copy in each object, when the value of an instan variable is modified in an object, it does not affect the instance variables in other objects. This proved in Program 7. Also see the Figure 13.1.

Program 7: Let us make a program by taking an instance variable x in the Test class.

```
//instance variables
class Test
{
  //instance var
  int x = 10;

  //display the variable
  void display()
 {
   System.out.println(x);

 }
}
```

```
class InstanceDemo
{
  public static void main(String args[ ])
 {
     //create two references
     Test obj1, obj2;

     //create two objects
     obj1 = new Test();
     obj2 = new Test();

     //increment x in obj1
     ++obj1.x;
     System.out.print("x in obj1: ");
     obj1.display();

     //display x in obj2
     System.out.print("x in obj2: ");
     obj2.display();
 }
}
```

Output:

```
C:\> javac InstanceDemo.java
C:\> java InstanceDemo
x in obj1: 11
x in obj2: 10
```

Here we create two objects obj1, obj2 to Test class. When x value in obj1 is incremented, it does not change the x value in obj2.

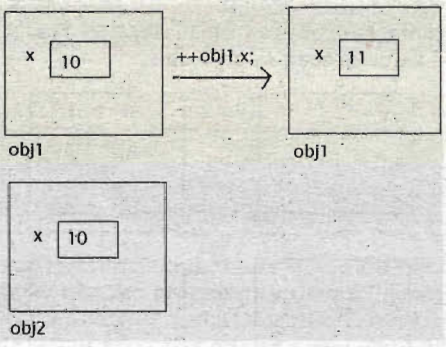

Figure 13.1 Each object gets a separate copy of instance variable

Since a class variable (static variable) will have only one copy in memory and that is shared by all the objects, any modification to it will also affect other objects. This is proved in Program 8. Also see Figure 13.2.

Program 8: Let us make a program by taking a static variable x in the Test class.

```
//class variables
class Test
{
//class var
  static int x = 10;

  //display the variable
  static void display()
 {
```

```
        System.out.println(x);
    }
}
class StaticDemo
{
    public static void main(String args[ ])
    {
        //create two references
        Test obj1, obj2;

        //create two objects
        obj1 = new Test();
        obj2 = new Test();

        //increment x in obj1
        ++obj1.x;
        System.out.print("x in obj1: ");
        obj1.display();

        //display x in obj2
        System.out.print("x in obj2: ");
        obj2.display();
    }
}
```

Output:

```
C:\> javac StaticDemo.java
C:\> java StaticDemo
x in obj1: 11
x in obj2: 11
```

Here in this program, we create two objects obj1, obj2 to Test class. When x value in obj1 is incremented, the incremented value is seen in obj2 also.

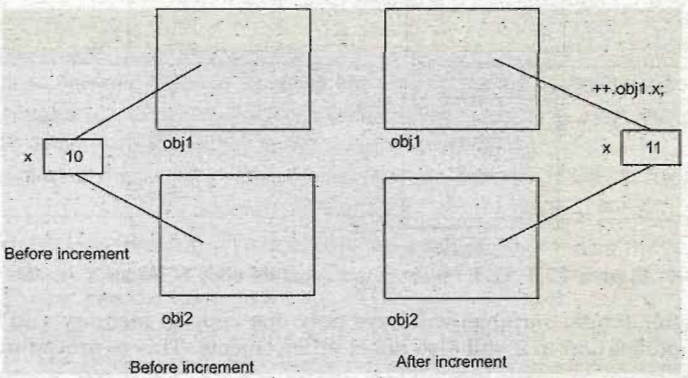

Figure 13.2 All objects share same copy of a static variable

Important Interview Question

Why instance variables are not available to static methods?

After executing static methods, JVM creates the objects. So the instance variables of the objects are not available to static methods.

The execution sequence of JVM is the process where JVM executes first of all any static blocks in the Java program. Then it executes static methods (remember `main()` is a static method) and then it creates any objects needed by the program. Finally, it executes the instance methods.

Static Block

A static block is a block of statements declared as 'static', some thing like this:

```
static {
   statements;
}
```

JVM executes a static block on a highest priority basis. This means JVM first goes to static block even before it looks for the `main()` method in the program. This can be understood from the Program 9.

Program 9: Write a program to test which one is executed first by JVM, the static block or the static method.

```
//Static block or Static method?
class Test
{
  static {
     System.out.println("Static block");
  }

  public static void main(String args[ ])
  {
     System.out.println("Static method");
  }
}
```

Output:

```
C:\> javac Test.java
C:\> java Test
Static block
Static method
```

So far, we thought that the `main()` method is the first one that is given attention by the JVM. This is ok if static block is not present in the program. If a static block is present, then JVM executes it first of all. After that, it searches for the `main()` method. If `main()` method is not found, it will display an error as shown in the output of the program 10.

Program 10: Write a Java program without `main()` method. This program compiles but gives an error at runtime.

```
//No main() method
class Test
{
  static {
     System.out.println("Static block");
  }
}
```

Output:

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

```
Static block
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Now, we can cheat the JVM by terminating it before it realizes that there is no `main()` method in the program, as shown here.

Program 11: Write a Java program without `main()` method. This program compiles and also runs.

```
//No main() method
class Test
{
  static {
     System.out.println("Static block");
     System.exit(0);
  }
}
```

Output:

```
C:\> javac Test.java
C:\> java Test
Static block
```

Important Interview Question

Is it possible to compile and run a Java program without writing `main()` method?

Yes, it is possible by using a static block in the Java program.

So far, we discussed about instance variables and static variables. There are other types of variables, called 'local variables'. A local variable is a variable that is declared locally inside method or a constructor and is available only within that method or constructor. It means a local variable cannot be accessed outside the method or constructor. For example,

```
void modify(int a)
{
   x = a;
}
```

In the preceding code, we declared 'a' inside the `modify()` method. It is a local variable. It cannot be accessed outside of `modify()` method. The other variable 'x' is not declared within it. It may be a instance variable. Instance variables are accessible any where within the class. See the Program 12

Program 12: Let us make a program to access an instance variable 'x' and a local variable 'a' from the method `access()`.

```
//local variables
class Sample
{
   //x is instance variable
   private int x;

   //a is local variable
   void modify(int a)
   {
      x = a;
   }

   //we can access x, but not a.
   void access()
   {
     System.out.println("x= "+x);
```

```
    System.out.println("a= "+a);
  }
}

class Local
{
  public static void main(String args[ ])
  {
    Sample s = new Sample();
    s.modify(10);
    s.access();
  }
}
```

Output:

```
C:\> javac Local.java
Local.java:17: cannot find symbol
Symbol  : variable a
Location: class Sample
    System.out.println("a= "+a);
                           ^
1 error
```

By observing the output, we can understand that there is an error while accessing 'a'. This clearly tells that a local variable cannot be accessed outside the method where it is declared. But the instance variable 'x' is accessible by `access()` method.

Sometimes a local variable has the same name as that of an instance variable. This leads to problems regarding their accessibility. For example,

```
  void modify(int x)
  {
    x = x;  //both the x refer only local variable
  }
```

In the preceding code, the local variable is 'x' and the instance variable is also 'x'. Inside the `modify()` method, if we write 'x', it denotes by default the local variable only. Now the question is how to refer to the instance variable. For this purpose, we should use 'this' keyword.

The keyword 'this'

'this' is a keyword that refers to the object of the class where it is used. In other words, 'this' refers to the object of the present class. Generally, we write instance variables, constructors and methods in a class. All these members are referenced by 'this'. When an object is created to a class, a default reference is also created internally to the object, as shown in the Figure 13.3. This default reference is nothing but 'this'. So 'this' can refer to all the things of the present object.

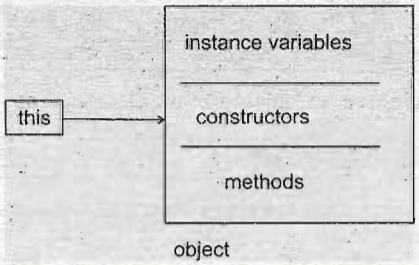

Figure 13.3 The keyword 'this' is a reference to class object

Now `modify()` method can be written as:

```
void modify(int x)
  {
    this. x = x;  //store local variable x into present class instance variable
x.
  }
```

In the preceding code 'this.x' refers to the present class instance variable and just 'x' refers to th local variable. To understand the use of 'this', let us take a sample program.

Program 13: Write a program to use this to refer the current class parameterized constructor a 'this()', its method as 'this.method()' and its instance variable as 'this.variable'.

```
//this - refers to all the members of present class
class Sample
{
   //x is instance variable
   private int x;

   //default constructor
   Sample()
  {
    this(55); //call present class para constructor and send 55
    this.access(); //call present class method
  }

   //parameterized constructor
   Sample(int x)
  {
     this.x = x; //refer present class instance variable
  }

  //method
  void access()
 {
   System.out.println("x= "+x);
 }
}

class ThisDemo
{
  public static void main(String args[])
 {
   Sample s = new Sample();

 }
}
```

Output:

```
C:\> javac ThisDemo.java
C:\> java ThisDemo
x= 55
```

In the preceding program, we created an object to Sample class, as:

```
Sample s = new Sample();
```

Here, two things will happen. First, Sample class object is created and then its default constru is also executed. Hence the code:

```
this(55);
this.access();
```

executed. In the first statement, present class (Sample) constructor is called and 55 is passed to . This value is used by parameterized constructor to initialize the present class instance variable as:

```
this.x = x;
```

Here, 'this.x' represents present class instance variable 'x'. Similarly to call the present class access() method, we can use 'this.access()', which displays the x value 55 as output.

Instance Methods

Instance methods are the methods which act upon the instance variables. To call the instance methods, object is needed, since the instance variables are contained in the object. We call the instance methods by using objectname.methodname(). The specialty of instance methods is that they can access not only instance variables but also static variables directly.

There are two types of instance methods:

- Accessor methods, and
- Mutator methods.

Accessor methods are the methods that simply access or read the instance variables. They do not modify the instance variables. Mutator methods not only access the instance variables but also modify them.

Suppose, we take a 'Person' class with name and age as instance variables. To store data into these instance variables, we can use setName() and setAge() methods. These methods are called mutator methods. Similarly to read and return the instance variables, we can write methods like getName() and getAge(). These are called accessor methods. These are demonstrated in the next program.

Program 14: Write a program to create Person class object.

```
//Accessor and mutator methods
class Person
{
   //instance variables
   private String name;
   private int age;

   //mutator methods to store data
   public void setName(String name)
   {
     this.name = name;
   }
   public void setAge(int age)
   {
      this.age = age;
   }

   //accessor methods to read data
   public String getName()
   {
      return name;
   }
   public int getAge()
   {
      return age;
```

```
    }
}

class Methods
{
  public static void main(String args[ ])
 {
   //create an empty Person class object
   Person p1 = new Person();

   //store some data into the object
   p1.setName("Raju");
   p1.setAge(20);

   //access data from object
   System.out.println("Name= "+ p1.getName());
   System.out.println("Age= "+ p1.getAge());
  }
}
```

Output:

```
C:\> javac Methods.java
C:\> java Methods
Name= Raju
Age= 20
```

In this program, we use setter methods to set some values into the instance variables. We also us getter methods to return the same.

Passing Primitive Data Types to Methods

Primitive data types or fundamental data types represent single entities or single values. Fo example, char, byte, short, int, long, float, double and boolean are called primitive data types because they store single values. They are passed to methods *by value*. This means when we pas primitive data types to methods, a copy of those will be passed to methods. Therefore, any change made to them inside the method will not affect them outside the method.

Let us take an example. There is a method '`swap()`' to interchange two values, num1 and num2. The logic for this can be taken as:

- Take a temporary variable temp.
- Preserve a copy of num1 into temp.
- Now store num2 into num1.
- Store the previous copy of num1 from temp into num1.

In the following program, we pass two integer numbers 10 and 20 to the swap method. Let display the output to know whether they are interchanged or not.

Program 15: Let us try to interchange two integers 10 and 20 by passing them to `swap()` method.

```
//Primitive data types are passed to methods by value
class Check
{
  //to interchange num1 and num2 values
  void swap(int num1, int num2)
 {
   int temp; //take a temporary variable
   temp = num1;
   num1 = num2;
   num2 = temp;
 }
```

```
}
class PassPrimitive
{
 public static void main(String args[ ])
 {
    //take two primitive data types
    int num1 = 10, num2 = 20;

    //create Check class object
    Check obj = new Check();

    //display data before calling
    System.out.println(num1+"\t"+num2);

    //call swap and pass primitive data types
    obj.swap(num1, num2);

    //display data after calling
    System.out.println(num1+"\t"+num2);
  }
}
```

Output:

```
C:\> javac PassPrimitive.java
C:\> java PassPrimitive
10    20
10    20
```

The output indicates that the two integers are not interchanged. From `main()` method, we passed 10 and 20 to `swap()` method. Inside the method, they are interchanged. But when we come out of `swap()` method, again we find the previous values as they are, not interchanged.

When we send primitive data types like int, float, char, etc., to a method, a copy of their values will be sent to the method. So any modifications to them inside the method will not affect their original copy. This is also called pass *by value* or call *by value*.

Passing Objects to Methods

We can also pass class objects to methods, and return objects from the methods. For example,

```
Employee myMethod(Employee obj)
{
      statements;
      return obj;
}
```

Here, `myMethod()` is a method that accepts Employee class object. So reference of Employee class is declared as parameter in `myMethod()`. After doing some processing on the object, if it returns the same object, we can write a statement like:

```
return obj;
```

Even the objects are also passed to methods *by value*. This means, when we send an object to a method, its bit by bit copy will be sent to the method. Any modifications to the object inside the method will not affect the original copy outside the method. So when we come out of the method, we find the original value unchanged.

Output:

```
C:\> javac PassObjects.java
C:\> java PassObjects
10   20
10   20
```

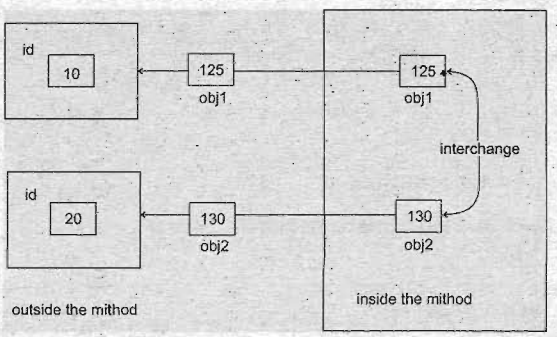

Figure 13.4 Interchanging the objects will interchange the references

Some authors have written out of confusion that primitive data types are passed to methods *by value* and objects are passed to methods *by reference*. This confusion primarily stems from applying C/C++ concept of 'pass by reference' in Java where the memory address (pointer) of the variables is passed to the methods. Since Java does not support creation and use of pointers, there is no question of passing memory addresses. So 'pass by reference' is not valid in Java.

Important Interview Question

How are objects are passed to methods in Java?

Primitive data types, objects, even object references – every thing is passed to methods using 'pass by value' or 'call by value' concept. This means their bit by bit copy is passed to the methods.

So, how can we interchange the 'id' values of Employee class objects by using `swap()` method? One possible solution is to use only one object and take two variables id1 and id2, and interchange their values in the object. This is shown in Program 17. In this program, we call the `swap()` method as:

```
obj.swap(obj1);
```

Here, a copy of obj1 is sent to the method and from there still, it can refer to the same object. So using it, if we modify the instance variables, we can have the data really modified in the original object. This is shown in Figure 13.5.

Program 17: Write a program to interchange the values inside an object, since the same object data is modified, we can see the data has been interchanged.

```
//Interchanging the values should be done in a single object
class Employee
{
  //instance variables
  int id1, id2;

  //to initialize id values
  Employee(int id1, int id2)
  {
    this.id1 = id1;
    this.id2 = id2;
```

```
  }
}

class Check
{
  //to interchange id values in the same Employee object
  void swap(Employee obj)
  {
    int temp; //take a temporary variable
    temp = obj.id1;
    obj.id1 = obj.id2;
    obj.id2 = temp;

  }
}
class PassObjects
{
 public static void main(String args[ ])
 {
    //take Employee class object with id values
    Employee obj1 = new Employee(10, 20);

    //create Check class object
    Check obj = new Check();

    //display data before calling
    System.out.println(obj1.id1+"\t"+obj1.id2);

    //call swap and pass Employee class object
    obj.swap(obj1);

    //display data after calling
    System.out.println(obj1.id1+"\t"+obj1.id2);
  }
}
```

Output:

```
C:\> javac PassObjects.java
C:\> java PassObjects
10    20
20    10
```

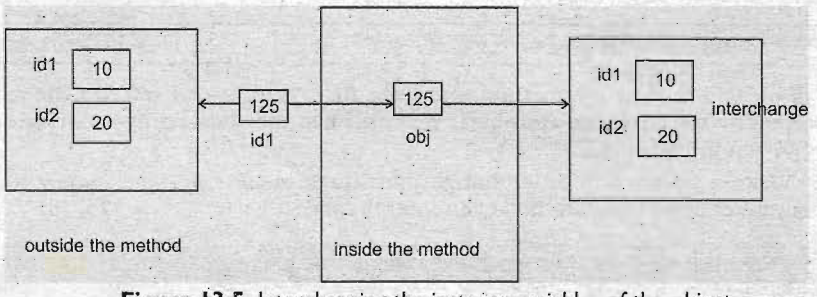

Figure 13.5 Interchanging the instance variables of the object

Passing Arrays to Methods

Just like the objects are passed to methods, it is also possible to pass arrays to methods and re arrays from methods. In this case, an array name should be understood as an object reference example,

```
int[ ] myMethod(int arr[ ])
```

This is the way, we can pass a one dimensional array 'arr' to 'myMethod()'. We can also return a one dimensional array of int type as shown in the preceding statement.

```
int[ ][ ] myMethod(int arr[ ][ ])
```

Here, we are passing and returning a two dimensional array of int type. Let us write a Java program to understand this concept further. In this program, we want to find the sum of two matrices. We take two 2D arrays of int type which represent the two matrices. We add them and get another 2D array of int type which represents the sum matrix. This sum matrix is finally displayed. We want to perform these tasks by using the following methods:

- Accept array (matrix) elements from keyboard by using `getMatrix()` method.
- Accept second array (matrix) elements from keyboard. For this also, same `getMatrix()` method can be used.
- Add the two matrices by using `findSum()` method.
- Finally, display the sum matrix by using `displayMatrix()` method.

Program 18: Write a program to add two matrices and display sum matrix.

```
//Matrix addition using methods.
//Passing arrays to methods and returning them.
import java.io.*;
import java.util.*;
class Matrix
{
  //take a 2D array for matrix and rows,cols
  int arr[ ][ ];
  int r,c;

  //Initialize r,c and allot memory for array
  Matrix(int r,int c)
  {
    this.r = r;
    this.c = c;
    arr = new int[r][c];
  }

  //accept 2D array from keyboard and return it
  int[ ][ ] getMatrix() throws IOException
  {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    //StringTokenizer is useful to accept each row as a single string and then
    // divide it into tokens. Each token represents an array element.
    StringTokenizer st;

      for(int i=0; i<r; i++)
      {
          String s = br.readLine();
          st = new StringTokenizer(s," ");
          for(int j=0; j<c; j++)
          arr[i][j] = Integer.parseInt(st.nextToken());
      }

      return arr;
  }

  //accept two 2D arrays and find sum matrix and return it.
  int[ ][ ] findSum(int a[ ][ ], int b[ ][ ])
  {
    int temp[ ][ ] = new int[r][c];
```

```
        for(int i=0; i<r; i++)
          for(int j=0; j<c; j++)
            temp[i][j]= a[i][j]+b[i][j];

        return temp;
    }
    //display the resultant 2D array as a matrix.
    void displayMatrix(int res[ ][ ])
    {
        for(int i=0; i<r; i++)
        {
          for(int j=0; j<c; j++)
          {
            System.out.print(res[i][j]+"\t");
          }
          System.out.println();
        }
    }
}

class MatrixSum
{
  public static void main(String args[ ])
  throws IOException
  {
    //create 2 objects to Matrix class, since each object
    //contains an array.
    Matrix obj1 = new Matrix(3,3);  //3X3 matrix
    Matrix obj2 = new Matrix(3,3); //3X3 matrix

    //take 3 references for 2D arrays.
    int x[ ][ ],y[ ][ ],z[ ][ ];

    System.out.println("\nEnter elements for first matrix: ");
    x = obj1.getMatrix();

    System.out.println("\nEnter elements for second matrix: ");
    y = obj2.getMatrix();

    //add the matrices and return sum matrix into z
    z = obj1.findSum(x,y);

    System.out.println("\nThe sum matrix is: ");
    obj2.displayMatrix(z);
  }
}
```

Output:

```
C:\> javac MatrixSum.java
C:\> java MatrixSum
Enter elements for first matrix:
1  2  3
4  5  6
7  8  9

Enter elements for second matrix:
1  1  1
2  2  3
0  0  -2

The sum matrix is:
2  3  4
6  7  9
7  8  7
```

Here in preceding program, we take a matrix as a 2D array of int type. This program assumes 3X3 matrices. But the same logic can be applied to add mXn matrices also.

Let us write another program by using methods to generate prime number series. A prime number is a number that is not divisible by any other number except by 1 and itself. For example, the prime numbers may start from 2 and proceed as: `2,3,5,7,11,13,...`

In this program, we write a static method `prime()` to test if a number is prime or not. Suppose 5 is a prime number, then it should not be divisible by any number from 2 to 4 (except 1 and itself). This logic is used in `prime()` method. `prime()` method returns true if a number passed to it is a prime, else it returns false. Another static method `generate()` generates a sequence of numbers and passes each number to `prime()` method for testing. If it is prime, it is displayed other wise, next number is tested.

Program 19: Write a program to generate required number of primes using methods.

```
//Prime number series
import java.io.*;
class Primes
{
  //to test and return true if num is prime
  static boolean prime(long num)
  {
    //initially isPrime is true, it becomes false if
    //num is not prime
    boolean isPrime = true;

    //from 2 to num-1, if any number divides num, it is not prime
    for(int i=2; i<=num-1; i++)
       if(num % i ==0) isPrime = false;

    return isPrime;
  }

  //accept how many primes required into max.
  // c is counter for no. of primes generated.
  static void generate(long max)
 {
    long c=1, num=2;

    while(c<= max)
   {
     if(prime(num))   //call prime() method directly
     {
        System.out.println(num);
        ++c;
     }
     ++num;
   }
 }
}
class PrimeDemo
{
  public static void main(String args[ ]) throws IOException
 {
     //accept the number of primes are needed
     BufferedReader br= new BufferedReader(new InputStreamReader(System.in));

     System.out.print("How many primes? ");
     int max = Integer.parseInt(br.readLine());

      //generate max number of primes
      Primes.generate(max);
  }
}
```

Output:

```
C:\> javac PrimeDemo.java
C:\> java PrimeDemo
How many primes? 10
2
3
5
7
11
13
17
19
23
29
```

Recursion

A method calling itself is known as 'recursive method', and that phenomenon is called 'recursion' is possible to write recursive methods in Java. Let us take an example to find factorial value o given number. Factorial value for a number num is defined as: num* (num-1)* (num-2)…..* For example, factorial of 5 will be 5*4*3*2*1 = 120. As long as num value is >0, we sho decrement num value and take the products into some other variable say 'fact'. This can be writ as:

```
long fact = 1;
while(num>0)
{
    fact = fact * num ;
    num = num -1;
}
```

This logic is represented in Program 20.

Program 20: Write a program to find factorial value without using recursion.

```
//Factorial without Recursion
class NoRecursion
{
   static long factorial(int num)
  {
    long fact = 1;

     while(num>0)
        fact *= num--;

     return fact;
  }

  public static void main(String args[])
 {
    System.out.println("Factorial of 5: ");
    System.out.println(NoRecursion.factorial(5));

 }
}
```

Output:

```
C:\> javac Recursion.java
C:\> java Recursion
Factorial of 5:
120
```

In the next program, we calculate the factorial value by using recursion. The logic is to call the factorial method from within the same factorial method, as:

```
result = factorial(num-1)*num;
```

this is because, factorial of 5 = factorial(4) * 5

and factorial of 4 = factorial(3) *4

and factorial of 3 = factorial(2) *3

and factorial of 2 = factorial(1) *2

and factorial of 1 = 1.

Hence, substituting in the first statement, we get:

```
factorial of 5 = 1*2*3*4*5 = 120.
```

So, we write the logic something like this:

```
if(num==1) return 1;
    result =factorial(num-1)*num;
```

Program 21: Write a program to calculate factorial value by using Recursion.

```
//Factorial using Recursion
class Recursion
{
   static long factorial(int num)
  {
    long result;

    if(num==1) return 1;
    result =factorial(num-1)*num;

    return result;

  }

  public static void main(String args[ ])
 {
    System.out.println("Factorial of 5: ");
    System.out.println(Recursion.factorial(5));
 }
}
```

Output:

```
C:\> javac Recursion.java
C:\> java Recursion
Factorial of 5:
120
```

Factory Methods

Factory methods are static methods only. But their intension is to create an object depending on the user choice. Precisely, a factory method is a method that returns an object to the class, to which it belongs. For example, `getNumberInstance()` is a factory method. Why? Because it belongs to `NumberFormat` class and returns an object to `NumberFormat` class.

At the time of creating objects, if the user has several types of options (values) to be passed to the object, then several overloaded constructors should be written in the class. For example, there are 10 different types of values to be passed, then 10 constructors are needed to accept those 10 types of options. This can be eliminated by using factory methods. A single factory method gives provision to pass any type of value through a parameter. Generally, the parameter is used to pass different types of values. Based on the value passed, the object is created by the factory method.

Important Interview Question

What are factory methods?

A factory method is a method that creates and returns an object to the class to which it belongs. A single factory method replaces several constructors in the class by accepting different options from the user, while creating the object.

To understand how to use a factory method, let us take an example program to calculate the area o a circle.

Program 22: Write a program for calculating and displaying area of a circle. The area is no formatted and displayed as it is.

```
//Area of a circle
class Circle
{
  public static void main(String args[])
 {
    final double PI = (double)22/7;  //constant
    double r = 15.5;  //radius

    double area = PI*r*r;
    System.out.println("Area= "+ area);
 }
}
```

Output:

```
C:\> javac Circle.java
C:\> java Circle
Area= 755.0714285714286
```

The digits before the decimal point are called 'integer digits' and the digits after the decimal poi are called 'fraction digits'. In the output of the preceding program, there are 13 digits displayed aft the decimal point. For most of the general purposes, these many digits are not needed. For exampl in an electricity bill, we need not display more than 2 digits after the decimal point to indicate pais So the question is how to format the output (area) of the preceding program to display as ma digits as we want in the integer and fraction parts. This is achieved by `NumberFormat` clas `NumberFormat` class of `java.text` package is useful to format the numerical values. The way to this is:

- Create `NumberFormat` object. For this, we should use the factory meth `getNumberInstance()`.

- Decide how to format the area value. Depending on this, we should use any of the `NumberFormat` class methods:

```
setMaximumIntegerDigits();
setMinimumIntegerDigits();
setMaximumFractionDigits();
setMinimumFractionDigits();
```

These methods specify how many integer digits or fraction digits to be displayed in the output.

- Apply the format to the area value using `format()` method. This method returns a string that contains formatted area value.

These steps can be seen in the following program.

Program 23: Write a program for calculating and displaying area of a circle. The area is formatted to have 7 maximum integer digits and 2 minimum fraction digits.

```
//Area of a circle
import java.text.*;
class Circle
{
  public static void main(String args[])
 {
    final double PI = (double)22/7;
    double r = 15.5;

    double area = PI*r*r;
    System.out.println("Area= "+ area);

    //create NumberFormat class object
    NumberFormat obj = NumberFormat.getNumberInstance();

    //store the format information into obj
    obj.setMaximumFractionDigits(2);
    obj.setMinimumIntegerDigits(7);

    //apply the format to area value
    String str = obj.format(area);

    //display formatted area value
    System.out.println("Formatted area= "+ str);

 }
}
```

Output:

```
C:\> javac Circle.java
C:\> java Circle
Area= 755.0714285714286
Formatted area= 0,000,755.07
```

Please observe the formatted area value in the output. It has 7 integer digits separated properly by commas and 2 fraction digits as specified by the methods:

```
obj.setMaximumFractionDigits(2);
obj.setMinimumIntegerDigits(7);
```

Also observe using the `getNumberInstance()` method to create an object to `NumberFormat` class. We used the default format of this method. There is another way of using `getNumberInstance()` method as:

```
getNumberInstance(Locale constant);
```

Here, the parameter represents a constant that represents the name of a country (locality) as Locale.CANADA, Locale.CHINA, Locale.FRANCE, Locale.GERMANY, Locale.ITALY, Locale.JAPAN, Locale.US, Locale.UK, etc., For example, to format a number according to the numerical format of United States, we can pass that constant to the NumberFormat object, as:

```
NumberFormat obj  = NumberFormat.getNumberInstance(Locale.US);
```

In the place of `getNumberInstance()` method, if constructors are provided in the `NumberFormat` class, they should be created something like: `NumberFormat(Locale.CANADA)`, `NumberFormat(Locale.CHINA)`,...

In this way, several constructors should be created. This awkwardness is avoided by using a single factory method `getNumberInstance()` and passing the required name of the country to it.

We can consider factory methods as alternative way to create objects, other than the new operator. So, new operator is not the only way to create objects in Java. There are other ways for that, which are discussed here in the important interview questions.

Important Interview Question

In how may ways can you create an object in Java?

There are four ways of creating objects in Java:

1. Using new operator.

```
Employee obj = new Employee();
```

Here, we are creating Employee class object 'obj' using new operator.

2. Using factory methods:

```
NumberFormat obj  = NumberFormat.getNumberInstance();
```

Here, we are creating `NumberFormat` *object using the factory method* `getNumberInstance()`.

3. Using `newInstance()` *method. Here, we should follow two steps, as:*

(a) First, store the class name 'Employee' as a string into an object. For this purpose, factory method `forName()` *of the class 'Class' will be useful:*

```
Class c = Class.forName("Employee");
```

We should note that there is a class with the name 'Class' in `java.lang` *package.*

(b) Next, create another object to the class whose name is in the object c. For this purpose, we need `newInstance()` *method of the class 'Class', as:*

```
Employee obj = (Employee)c.newInstance();
```

4. By cloning an already available object, we can create another object. Creating exact copy of an existing object is called 'cloning':

```
Employee obj1 = new Employee();
Employee obj2 = (Employee)obj1.clone();
```

Earlier, we created obj2 by cloning the Employee object obj1. `clone()` *method of Object class is used to clone an object. We should note that there is a class by the name 'Object' in* `java.lang` *package.*

Conclusion

Methods are very important components of a program. The programmer should divide the main task into sub tasks and represent each sub task in the form of a method. In Java, we got Static methods, Instance methods and Factory methods. Again, Instance methods are divided into two methods and these are Accessor and Mutator methods.

We also discussed different variables available in Java. An instance variable is directly declared in the class and available anywhere in the class. A separate copy of an instance variable is available to each object. A static variable is a variable whose single copy is shared by all the objects. A local variable is a variable whose scope is limited only to that method or constructor, where it is declared.

Also, we examined the key word 'this', which is a reference to the class object, where it is used, and hence it can refer to all the members of that class object.