3. More about Methods and Classes

❖ Overloading Methods

- Class have multiple methods(functions) by same name but different parameters,it is known as **method overloading**.
- Method overloading is one of the ways that Java supports polymorphism.
- But as you will see, method overloading is one of Java's most exciting and useful features.
- When an overloaded method is called, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- Here is a simple example that illustrates method overloading:
- **EXAMPLE:** Sum of two numbers using method overloading.

```
//Demonstrate method overloading.
class sumoftwo
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }
    void sum(int a,int b,int c)
    {
        System.out.println(a+b+c);
    }
}
class sum
{
    public static void main(String args[])
    {
        sumoftwo ob=new sumoftwo();
        ob.sum(10,20,30);
        ob.sum(20,50);
    }
}
```

- **EXAMPLE:**

```
// Demonstrate method overloading.
class one
{
    void test()
    {
        System.out.println("No parameters");
    }
    void test(int a)
    {
        System.out.println("a: " + a);
```

```
    }
    void test(int a, int b)
    {
        System.out.println("a and b: " + a + " " + b);
    }
}
class two
{
    public static void main(String args[])
    {
        one ob = new one();
        ob.test();
        ob.test(10);
        ob.test(10, 20);
    }
}
```

OUTPUT:

No parameters
a: 10
a and b: 10 20

❖ Overloading Constructors

- Constructor is special type of method which called at the time of object creation.
- In addition to overloading normal methods, you can also overload constructor.
- There is two types of constructor:
  (1) Default constructor (Constructor have no-parameter)
  (2) Parameterized constructor (Constructor have parameter)

- **EXAMPLE:**

//Demonstrate constructor(default constructor) overloading.

```
class display
{
        display()
        {
                System.out.println("Hello world");
        }
}
class read_display
{
        public static void main(String args[])
        {
                display ob=new display();
        }
}
```

- /* Here, Box defines three constructors to initialize the dimensions of a box various ways.*/
  //Demonstrate constructor overloading.

```
class Box
```

```
{
    double width;
    double height;
    double depth;
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    Box()
    {
        width = -1;  // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1;  // box
    }
    double volume()
    {
        return width * height * depth;
    }
}
class Demobox
{
    public static void main(String args[])
    {
        Box b1 = new Box(10, 20, 15);
        Box b2 = new Box();
        double vol;
        vol = b1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        vol = b2.volume();
        System.out.println("Volume of mybox2 is " + vol);
    }
}
```

OUTPUT:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

❖ Using Objects as Parameters

- Using simple types as parameters to methods.
- However, it is both correct and common to pass objects to methods.
- For example, consider the following short program:

```
class Test
{
    int a, b;
    Test(int i, int j)
    {
        a = i;
        b = j;
```

```
    }
    boolean equals(Test o)
    {
        if(o.a == a && o.b == b)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
class Demopass
{
    public static void main(String args[])
    {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: " + ob1.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}
```

OUTPUT:

```
ob1 == ob2: true
ob1 == ob3: false
```

- As you can see, the equals( ) method inside Test compares two objects for equality and returns the result.
- That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns true .
- Otherwise, it returns false.
- Notice that the parameter object in equals( ) specifies Test as its type. Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.

❖ Call-By-Value

- If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.
- This approach copies the value of an argument into the formal parameter of the subroutine.
- Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- Inside the subroutine, this reference is used to access the actual argument specified in the call.
- This means that changes made to the parameter will affect the argument used to call the subroutine.
- In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

- **EXAMPLE:**

// Primitive types are passed by value.

```
class Test
{
    void abc(int i, int j)
    {
        i = 2;
        j = 2;
    }
}
class CallByValue
{
    public static void main(String args[])
    {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a and b before call: " + a + " " + b);
        ob.abc(a, b);
        System.out.println("a and b after call: " + a + " " + b);
    }
}
```

OUTPUT:

a and b before call: 15 20
a and b after call: 15 20

❖ Call-By-Reference

- This concept is not correct for java.
- In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects are passed to methods by use of call-by-reference.
- Changes to the object inside the method do affect the object used as an argument.

- **EXAMPLE**

```
// Objects are passed by reference.
class Test
{
    int a, b;
    Test(int i, int j)
    {
        a = i;
        b = j;
    }
    // pass an object
    void abc(Test o)
    {
        o.a =  2;
        o.b = 4;
    }
```

```
}
class CallByRef
{
    public static void main(String args[])
    {
        Test ob = new Test(15, 20);
        System.out.println("Value Of ob.a Before Call " + ob.a );
        System.out.println("Value Of ob.b Before Call " + ob.b );
        ob.abc(ob);
        System.out.println("Value Of ob.a After Call " + ob.a );
        System.out.println("Value Of ob.b After Call " + ob.b );
    }
}
```

OUTPUT:

Value Of ob.a Before Call 15
Value Of ob.a Before Call 20
Value Of ob.a After Call 2
Value Of ob.b After Call 4

❖ Returning an object

- A method can return any type of data, including class types that you create.
- For example, in the following program, the abc( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

EXAMPLE:

```
class Test
{
    int a;
    Test(int i)
    {
        a = i;
    }
    Test abc()
    {
        Test temp = new Test(a+10);
        return temp;
    }
}
class two
{
    public static void main(String args[])
    {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.abc();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.abc();
        System.out.println("ob2.a after second increase: "+ ob2.a);
    }
```

```
    }
```

OUTPUT:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

❖ Recursion

- *Recursion* is a basic programming technique you can use in Java, in which a method calls itself
- A method that calls itself is said to be recursive.
- The classic example of recursion is the computation of the factorial of a number.
- The factorial of a number N is the product of all the whole numbers between 1 and N.For
- example, 3 factorial is 1 × 2 × 3 , or 6.
- Here is how a factorial can be computed by use of a recursive method:
- **EXAMPLE:-**

```
class Factorial
{
    // this is a recursive method
    int fact(int n)
    {
        int result;
        if(n==0 || n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}
class Recursion
{
        public static void main(String args[])
        {
                Factorial f = new Factorial();
                System.out.println("Factorial of 3 is " + f.fact(3));
        }
}
```

OUTPUT:

```
Factorial of 3 is 6
```

❖ Static and Final Keyword

- The static keyword is used in java mainly for memory management.
- We may apply static keyword with variables, methods and blocks.
- The static can be:
  1. variable (also known as class variable)
  2. method (also known as class method)
  3. block
- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an object of a class.
- static method can access static data member and can change the value of it.

- To create such a member, precede its declaration with the keyword static.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- The most common example of a static member is main( ).
- main( )is declared as static because it must be called before any objects exist.
- Methods declared as static have several **restrictions**:
  - They can only call other static methods.
  - They must only access static data.
  - They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)
- **EXAMPLE**:-Demonstrate static variables, methods, and blocks.

```
class UseStatic
 {
        static int a = 3;
        static void meth(int x)
        {
                System.out.println("x = " + x);
                System.out.println("a = " + a);
        }
        static
        {
                System.out.println("Static block initialized.");
        }
        public static void main(String args[])
        {
                meth(42);
        }
}
```

OUTPUT:

```
Static block initialized.
x = 42
a = 3
```

- Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.
- For example, if you wish to call a static method from outside its class, you can do so using the following general form.

  classname.method( )
- Here,classname is the name of the class in which the static method is declared.
- As you can see, this format is similar to that used to call non- static methods through object-reference variables.
- A static variable can be accessed in the same way—by use of the dot operator on the name of the class.

```
class Demo
{
        static int a = 42;
        static int b = 99;
```

```
        static void callme()
        {
                System.out.println("a = " + a);
        }
}
class test
{
        public static void main(String args[])
        {
                Demo.callme();
                System.out.println("b = " + Demo.b);
        }
}
```

OUTPUT:

a = 42

b = 99

❖ final KEYWORD:

- A variable can be declared as final.
- The Java final keyword is used to indicate that something "cannot change".
- Use of Final keyword:
  - It is used to indicate that a class cannot be extended.
  - It is used to indicate that a method cannot be overridden.
  - It is used to indicate that a local variable cannot be changed once its value is set.
  - It is used to indicate that a static variable cannot be changed once set, in effect implementing "constants".
- For example:

  final int I= 1;
- Subsequent parts of your program can now use i, etc., as if they were constants, without fear that a value has been changed.
- **EXAMPLE:**

```
public class Demo
{
        final int I=10;
        void abc()
        {
                System.out.println("Final variable value : "+i);
        }
        public static void main(String[] args)
        {
                Demo f=new Demo();
                f.abc();
        }
}
```

OUTPUT:-

Final variable value :10

❖ Nested Classes

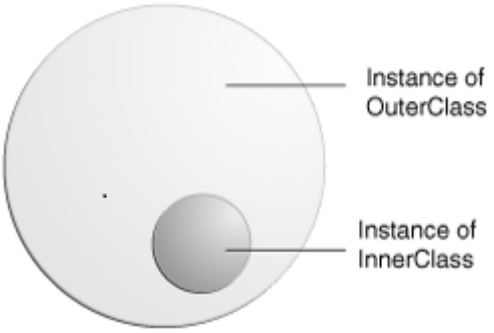
- Class within another class such classes is known as **nested classes**.
- A nested class has access to the members, including private members, of the class in which it is nested.
- However, the enclosing class does not have access to the members of the nested class.
- There are two types of nested classes: **static and non-static**.
- A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- The most important type of nested class is the **inner class**.
- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class

❖ Inner Classes

- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
- Also, because an inner class is associated with an instance, it cannot define any static members itself.
- Objects that are instances of an inner class exist within an instance of the outer class.
- Consider the following classes:

```
class OuterClass
{
    ...
    class InnerClass
    {
        ...
    }
}
```

- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.
- The next figure illustrates this idea.



- An Instance of Inner Class Exists Within an Instance of OuterClass.

- **Example:**

```
class Outer
{
        int outer_x = 100;
```

```
        void test()
        {
                Inner inner = new Inner();
                inner.display();
        }
    // this is an inner class
        class Inner
        {
                void display()
                {
                        System.out.println("display: outer_x = " + outer_x);
                }
        }
    }
    class InnerClassDemo
    {
            public static void main(String args[])
            {
                Outer outer = new Outer();
                outer.test();
            }
     }
```

OUTPUT:

display: outer_x = 100

❖ Command Line Arguments

- The user enters command-line arguments at the time of application run and specifies them after the name of the class to be run.
- For example, suppose a Java application called Sort sorts lines in a file. To sort the data in a file named friends.txt, a user would enter:

  java Sort friends.txt
- When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings.
- In the previous example, the command-line arguments passed to the Sort application in an array that contains a single String: "friends.txt".
- **EXAMPLE:**

```
public class Test2
{
    public static void main(String[] args)
    {
        int num=args.length;
        String s[]=new String[num];
        if(num>0)
        {
            System.out.println("The values enter at argument command line are:");
            for(int i=0;i<num;i++)
            {
             System.out.println("Argument"+(i+1)+"="+args[i]);
```

```
            }
        }
        else
        {
            System.out.println("No values has been entered at the command line.");
        }
    }
    }
```

OUTPUT:

```
C:\JavaPro>java Test2 programming with java
        The values enter at argument command line are:
        Argument 1 = programming
        Argument 2 = with
        Argument 3 = java
```