- **Method Overloading:** When two or more methods have same name within the same class but the parameter declarations are different is known as **method overloading**. Method overloading is one of the ways that java implements polymorphism.

- Java uses the type and number of arguments to determine which version of the overloaded method will be called when an overloaded method is called. Therefore, overloaded methods must differ in the type and 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.

- When java call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is simple example that shows the method overloading.

```
// Demonstrate method overloading.
class OverloadDemo
{
        void test()
         {
                System.out.println("No parameters");
        }
 // Overload test for one integer parameter.
         void test(int a)
        {
                System.out.println("a: " + a);
        }
 // Overload test for two integer parameters.
        void test(int a, int b)
         {
                System.out.println("a and b: " + a + " " + b);
        }
 // overload test for a double parameter
        double test(double a)
         {
                System.out.println("double a: " + a);
                return a*a;
        }
}
class Overload {
 public static void main(String args[]) {
  OverloadDemo ob = new OverloadDemo();
  double result;
  // call all versions of test()
  ob.test();
  ob.test(10);
  ob.test(10, 20);
  result = ob.test(123.25);
  System.out.println("Result of ob.test(123.25): " + result);
 }}
```

- When an overloaded method is called, java looks for a match between the arguments used to call the method and method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution. For example, consider the following program.

/ Automatic type conversions apply to overloading.

```
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // overload test for a double parameter and return type
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i);           // this will invoke test(double)
    ob.test(123.2);       // this will invoke test(double)
  }
}
```

- **How methods overloading support polymorphism in java?**

Polymorphism means "one interface with multiple methods". The language that does not support method overloading must be given a unique name. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of t his function each with a slightly different name.

For example in C, the function **abs()** returns the absolute value of an integer.

labs() returns absolute value of a long integer and
fabs() returns the absolute value of floating point value.

C language does not support overloading, so each function has its own name even though all three functions do essentially the same thing.

This situation does not possible in Java, because each absolute value method can use the same name. Java's standard class library includes an absolute value method called **abs()**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs()** to call based upon the type of argument.

❖ Overloading Constructors:

As like to overload methods we can also overload constructor. To understand how it is done consider the following example.

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.

*/
class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
  }
}
```

❖ **Using Objects as Parameters:**

We can also pass object as parameters in methods similar to pass simple type as parameters.

```
// Objects may be passed to methods.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }
  // return true if o is equal to the invoking object

  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}
class PassOb {
  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));
  }
}
```

❖ Using Objects as Parameters in constructors

One of the most common uses of object parameters involves constructors. You can construct new object by passing argument as object in the constructor of existing object.

```
// Here, Box allows one object to initialize another.

class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

 // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
 // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }
 // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }
 // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    Box myclone = new Box(mybox1);
    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

    // get volume of clone
    vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
  }
}
```

❖ **What do you mean by call by value & call by reference?**

- **Call by value:** When simple type is passed to a method, this method is done by use of call by value. This method copies the value of an argument into the formal parameter of the method. Therefore, changes made to the parameter of the method have no effect on the argument.

- **Call by reference:** When objects are passed to methods, this method is done by use of call by reference. In this method, a reference to an argument is passed to the parameter.

Inside the method, 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 method.

Consider the following program,

```
// Simple Types are passed by value.
class Test {
  void meth(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.meth(a, b);

    System.out.println("a and b after call: " +
                       a + " " + b);
  }
}
```

The output from this program is shown here:

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

When you pass an object to a method, the situation is totally changed because objects are passed by reference. This effectively means that objects are passed to methods by use of **call-by-reference**.

For example consider the following program.

```
// Objects are passed by reference.

class Test {
  int a, b;

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

  // pass an object
  void meth(Test o) {
    o.a *=  2;
    o.b /= 2;
  }
}
class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a and ob.b before call: " +
                       ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a and ob.b after call: " +
                       ob.a + " " + ob.b);
  }
}
```

The output from this program is shown here:

```
Ob.a  and ob.b before call:     15      20
Ob.a  and ob.b after   call:    30      10
```

❖ **Returning Objects:**
A method can return any type of data including class types that you create.

For example consider the following program.

```
// Returning an object.
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
  public static void main(String args[]) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "
                       + ob2.a);
  }
```

❖ **Recursion:** Recursion means a function which call itself. 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:

```
// A simple example of recursion.
class Factorial {
  // this is a recusive function
  int fact(int n) {
    int result;

    if(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));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

❖ **How factorial function does works?**

When **fact()** is called with an argument of 1, the function returns 1; otherwise it returns the product of **fact(n-1)*n**. To evaluate this expression, **fact()** is called with n-1. This process repeats until **n** equals **1** and the calls to the method begin returning.

New local variables and parameters of a method are storage on the stack and the method code is executed with the new variables from the beginning. A recursive call does not make a new copy of the method. Only the arguments are new.

It means as each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the call inside the method.

❖ **What are the main advantages of recursive method?**

1. It can be used to create clearer and simpler versions of several algorithms than to use iterative relatives. For example, quicksort algorithm is quite difficult to implement in an iterative way.
2. Some problems especially related to AI-related ones can be solve themselves to recursive problem.

❖ **What are the main advantages of recursive method?**

1. Recursive versions of many methods may execute a bit more slowly than the iterative statement because of the added overhead of the additional function calls.

2. Many recursive calls to a method could reason a stack overrun. Because storage for arguments and local variables is on the stack and each new call creates a new copy of these variables, therefore it is possible that the stack could be tired. If this occurs, the java run-time system will generate an exception.

// Another example that uses recursion.

```
class RecTest {
  int values[];

  RecTest(int i) {
    values = new int[i];
  }

  // display arrary -- recursively
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}

class Recursion2 {
  public static void main(String args[]) {
    RecTest ob = new RecTest(10);
    int i;

    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
  }
}
```

❖ **Introducing Access Control:**

Encapsulation links data with the code that manipulates it. Access control provides way to make control over the accessibility of a class data member. So you can prevent misuse of your data member.

For example consider the stack class in which the method **push()** and **pop()** do provide restricted boundary to the stack, this interface is not compulsory.

Java's access specifies are **public, private** and **protected**. Java also defines a **default access** level. **Protected** applies only when inheritance is involved.

Let's begin by defining **public and private**.

- **Public:** When a member of a class is modified by the public specifier, then that member can only be accessed by any other code.
- **Private:** When member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- **Default access:** when no access specifier is used then by default the member of a class is public within its own package, but can't be accessed outside its package.

Now you can understand why **main()** has always been preceded by the **public** specifier. It is called by code that is outside the program – that is by the java run-time system.

```
/* This program demonstrates the difference between
   public and private.
*/

class Test {
  int a;                        // default access
  public int b;                 // public access
  private int c;                 // private access

  // methods to access c
  void setc(int i) {            // set c's value
    c = i;
  }
  int getc() {                  // get c's value
    return c;
  }
}

class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

// These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

// This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK

    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

To see how access control can be applied to more practical example, consider the following improved version of the **stack** class

```
// This class defines an integer stack that can hold 10 values.
class Stack {

  /* Now, both stck and tos are private.  This means
     that they cannot be accidentally or maliciously
     altered in a way that would be harmful to the stack.
  */
  private int stck[] = new int[10];
  private int tos;
// Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
class TestStack {
public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

// push some numbers onto the stack

    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

// pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());

// these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

❖ **What do you mean by static in java?**

- Sometimes there will be situation generates when we want to access class data member without using the reference of class object. It is possible to create a class member that can be used by itself without reference of class object. To create such a member, precede its declaration with the keyword **static.**

- When a member is declared as **static**, it can be accessed before any objects of its class are created and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is **main()** method. **main()** is declared as **static** because it must be called before any object exists.

- Instance variables declared as **static** are, essentially global variables. When objects of its class are declared, no copy of a **static** variable is made. All instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions.

1. They can only call other **static** methods.
2. They must only access **static** data.
3. They cannot refer to **this** or **super** in any way.

If you want to initialize your **static** variable, you have to declare a static block and initialize static variable inside that block.

The following example shows a class that has a **static** method, some **static** variables and **static** initialization block.

// Demonstrate static variables, methods, and blocks.

```
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(48);
  }
}
```

Here is the output of the program:

```
Static block initialized
x=42
a=3
b=12
```

You can access the **static variables** and **methods** from the outside class by specifying the name of their class followed by the dot operator. Consider the following example.

Static method:	**classname.method()**
Static variable:	**classname.static_variable.**

Here is an example. Inside **main()** the static method **callme()** and the **static** variable **b** are accessed outside of their class.

```
class StaticDemo {
  static int a = 48;                          //static variable a
  static int b = 99;                          //static variable b
  static void callme() {                      //static method
    System.out.println("a = " + a);
  }
}
class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();                             //static method
    System.out.println("b = " + StaticDemo.b);       //static variable
  }
}
```

❖ Final keyword for variable:

A variable can be declared as **final**. When you declare variable as final it means you cannot change contents of variable so you must initialize a final variable when it is declared. (final is similar to cost in c/c++/c#).

Consider the following example,

```
final int a=107;
final int b=201;
final float f1=3.5f;
```

❖ New version of Array:

We can get the length of an array by using the length instance variable. All arrays have this variable and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.

class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }}
```

You can put the **length** member to good use in many situations.

❖ Introducing Nested and Inner classes:

Nested class: To define a class within another class; such classes are known as nested classes.

The scope of a nested class is limited upto the scope of its enclosing class. Therefore if class **B** is defined within class **A**, then **B** is known to **A** but not outside of **A**.

It means class **B** can access all the members including private members of class in which it is nested. (Here for class **A**). However, the enclosing class (**class A**) does not have access to the members of the nested class (**class B**).

There are two types of nested classes: **static and non-static.**

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.

Non-static or 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 do. Thus, an inner class is fully within the scope of its enclosing class.

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }
```

```
 // this is an innner 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 from this application shown here:

```
        Display:        outer_x=100
```

In the above program, an inner class named Inner is defined within the scope of class **outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**.

display() method is defined inside the **Inner** class. This method displays value of **outer_x** on the screen. The main() method of InnerClassDemo create an object of class **outer** and call its **test()** method. This method creates an object of class **Inner** and **display()** method is called.

Inner class is known only within the scope of class **outer**. The java compiler generates an error message if any code outside of **class outer** tries to create an object of class Inner. It means inner class is only known within its enclosing scope.

For example

```
// This program will not compile.
class Outer
{
 int outer_x = 100;

        void test()
        {
                Inner inner = new Inner();
                inner.display();
        }
        // this is an innner class
        class Inner
        {
                int y = 10;                         // y is local to Inner
                void display()
                 {
                        System.out.println("display: outer_x = " + outer_x);
                 }
        }
        void showy()
        {
                System.out.println(y);          // error, y not known here!
        }
}
class InnerClassDemo
{
 public static void main(String args[])
 {
   Outer outer = new Outer();
   outer.test();
  }
}
```

❖ **Define Inner Class within a for loop.**

You can define a nested class within the block defined by a method or even within body of **a for loop**, as this next program shows.

```
// Define an inner class within a for loop.

class Outer
{
        int outer_x = 100;

        void test()
        {
                for(int i=0; i<10; i++)
                {
                        class Inner
                        {
                                void display()
                                {
                                System.out.println("display: outer_x = " + outer_x);
                                }
                        }
                                Inner inner = new Inner();
                                inner.display();
                }
        }
}
class InnerClassDemo
{
        public static void main(String args[])
        {
                Outer outer = new Outer();
                outer.test();
        }
}
```

❖ **String class:**

String is probably the most commonly used class in Java's class library.

There are two important thing about string in java.

1. **String** you create is an **object** of type **String**. Even string constants are actually **String** objects. For example, in the following statement

   System.out.println("This is a string, too");

the string **"This is a String, too"** is a **string constant**. Fortunately, java handles String constants in the same way that other computer languages handle "normal" strings, so you don't have to worry about this.

2. The objects of type **String** is created; its contents cannot be changed. There are two reasons behind this to make restrictions.

   - If you need to change a string, you can always create a new one that contains the modifications.
   - Java defines a class of string called **StringBuffer** which allows strings to be changed.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

String mystring = "this is a test";

You can use it anywhere to display value of **mystring** variable on the screen.

System.out.println(mystring);

Java defines one operator for **String** objects: **+.** It is used to concatenate two strings. For example, this statement

String mystring = "I" + " like " +"Java." ;

results in **mystring** containing "I like Java."

The given program demonstrates the **String** concept.

// Demonstrating Strings.

```
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

The **String** class contains several methods that you can use. Here are a few. The general syntax of these three methods are shown here:

```
        boolean equals(String object)
        int length()
        char charAt(int index)
```

// Demonstrating some String methods.

```
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

This program generates the following output:

```
Length of strob1:12
Char at index 3 in strob1: s

strob1 != strob2
strob1==strob3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example;

```
// Demonstrate String arrays.

class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

❖ **How to use command line arguments?**

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in java programs by using what are known as **command line arguments**.

Command line arguments are parameters that are supplied to the application program at the time of calling it for execution.

We can write Java programs that can receive and use the arguments provided in the command line argument. Use the signature of the **main()** method used in our java programs.

Public static void main(String args[])

As pointed out earlier, **args** is declared as an array of strings **(known as string objects)**. Any arguments provided in the command line (at the time of execution) are passed to the array **args** as its elements.

We can simply access the array elements and use them in the program as we wish.

For example consider the following command line argument

```
java test BASIC FORTRAN C++ JAVA
```

This command line contains four arguments. These are assigned to the array **args** as follows:

BASIC	-→	args[0]
FORTRAN	-→	args[1]
C++	-→	args[2]
JAVA	-→	args[3]

```
/*
        *       This program uses command line
        *       Arguments as input
/*
class comlinetest
{
        public static void main(String args[])
        {
                int count, i=0;
                String string;
                Count=args.length;
                System.out.println("Number of arguments=" +count);
                while( i<count)
                {
                        string=args[i];
                        i=i+1;
                        System.out.println(I +":" +"Java is" +string +"!");
                }
        }
}
```

Run the above program with the command line argument as follows:

```
java comlinetest BASIC FORTRAN C++ JAVA
```