Controlling Access to Class Members, Pass Objects to Methods, How Arguments are passed, Returning Objects, Method Overloading, Overloading Constructors, Recursion, Understanding Static, Introducing Nested and Inner Classes, Varargs: Variable-Length Arguments.

Controlling Access to Class Members

- Restricting access to a class's members is a fundamental part of object-oriented programming because it helps prevent the misuse of an object.

- There are two types of class members
  - public – It can be accessed by code defined outside of its class.
  - private – It can be accessed only by other members of its class.

1.1.1 Java's Access Modifiers

- Members access control is achieved through the use of three access modifiers
  - public
  - private
  - protected
  - default
- If no access modifiers is used, the default access setting is assumed.

Modifier	Same package			Other package	
	Same class	Other class	Sub class	Sub class	Other class
Private	Yes	No	No	No	No
Default	Yes	Yes	Yes	No	No
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	Yes	Yes

Pass objects to methods

- As we pass simple types as parameter to method, it is also possible to pass objects to methods.
- If we want to compare all members of two objects for equality, then passing object to method is the best example.
- Example:

```java
class Test {
      int data;
      Test (int d){
            data=d;
      }
      void show(){
            System.out.println("Data="+data);
      }
}
class Myprogram{

      static void increase(Test p){
            p.data = p.data+1;
      }
      public static void main(String[] args){
            Test p =new Test(5);
            p.show(); // Data = 5
            Myprogram.increase(p);
            p.show(); //Data = 6
      }}
```

How arguments are passed

- There are two ways of passing argument to method
  1. Call-by-value
     - This approach copies the value of an argument into the formal parameter of the subroutine.
     - Changes made to the parameter of the subroutine have no effect on the argument.
  2. Call-by-reference
     - In this approach, a reference to an argument instead the value is passed to the parameter.
     - The passed reference is used to access the actual argument specified in the call.
     - Objects are passed by default as call-by-reference.
     - When you pass reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
     - Thus Changes to the object inside the method do affect the object used as an argument.

- Example: *call-by-value and call-by-reference*

Call-By-Value	Call-By-Reference

`class Program{`
` static void increase(int p)`
` {`
`   p =p+1;`
`  }`
` public static void main(String[]args){`
`  int a=5;`
`  increase(5);`
` System.out.println("A = "+a)`
` }`
`}`
	`class Test {`
	` int data;`
	` Test (int d){`
	`    data=d;`
	`  }`
	` void show(){`
	`      System.out.println("Data="+data);`
	` }`
	`}`
	`class Myprogram{`
	` static void increase(Test p){`
	`            p.data = p.data+1;`
	`      }`
	`public static void main(String[] args){`
	`      Test p =new Test(5);`
	`      p.show(); // Data = 5`
	`      Myprogram.increase(p);`
	`      p.show(); //Data = 6`
	` }`
	`}`

`Output:`
`A =5`
	`Output:`
	`Data = 5`
	`Data = 6`

2. Example for call-by-value and call-by-reference

```
class Number{
   int num;

   Number(int a){
         num=a;
   }
   void change(int a){
         a+=5;
   }
   void change(Number ob){
         ob.num +=5;
   }
   intgerNum() {
         return num;
   }
}
class DemoArgumentPassing {
   public static void main(String[] args){
     Number n1 =new Number(10);
     int x=20;
     n1.change(x);   // call-by-value

     System.out.println("Value of x  change x ="+x);

     Number n2 =new Number(10);
     n1.change(n2);  //call-by -reference

    System.out.println("Value of object after change num="+p2.getNum());
   }

}
```

Output:

```
Value of x change x = 20
Value of object after change num = 15
```

Note: In Java, when you pass a primitive type to a method, it is passed by value and Objects are implicitly passed by using call-by-reference.

Returning Objects

- In Java a method can return any type of data, including class types that you create
- Example:

```
    class Number {
          int num;
          Number(int a) {
                num = a;
          }
          Number incrementBy(int a) {   //returning obejct
                Number temp = new Number(a+num);
                return temp; }

          int getNum(){
                return num; } }

  class Demo {
    public static void main(String [] args) {
    Number n1 = new Number(10);
    Number n2 = new Number(10);
    N2 = n1.incrementBy(5); //returns object
     System.out.printl(" Value of object calling method = " + n1.getNum());
    System.out.printl(" Value of object receiving other object = " + n2.getNum());

    } }

Output:

Value of object calling method =10

Value of object receiving other object =15
```

Constructor overloading and method overloading

Constructor Overloading

- Like methods, constructors can also be overloaded. *It is* also known as **Static Polymorphism**.
- *Constructor overloading is way of having more than one constructor which does different.*
- Overloaded Constructors can be referred using this() keyword from other constructor.
- Syntax:

classClassName {

ClassName() { ***//default constructor***

// code

```
                    }
                    ClassName(Parameter-List) {//Parameterized constructor
                            // code
                    }
            }
```

Method Overloading

- *Method overloading is a way of defining two or more methods with same name but different forms. It is* also known as **Static Polymorphism**.
- Syntax

```
class ClassName {

     type method (Parameter-list-1)  {   // form1

          //code   }

     type method (Parameter-list-2)  {  //form2

          //code  }

     }
```

- Rules to achieve overloading
  Argument list must differ in one of the following
    1. Number of parameters
    2. Data type of parameters
    3. Sequence of Data type of parameters
- Return type has no effect on overloading.
- Example for method overloading and constructor overloading

```
class Shape {

      int length, breadth, height;

Shape( ) { //Default constructor

       Length = breadth = height = 0;}

Shape(int length, int breadt, int height) { // parameterized constructor

Length = breadth = height = side;
}

int volume( ) { // method with no parameters
return length * breadth * height; }

int volume(int length, int breadth, int height) { // method with parameters
return length * breadth * height;

}
public static void main(String [] args) {
System.out.println(" --Demonstrating Constructor Overloading-- ");

Shape square = new Shape(10);
Shape rectangle = new Shape();
System.out.println(" --Demonstrating Method Overloading-- ");

int volume1 = square.volume();
int volume2 = rectangle.volume(10,20, 30);
System.out.println("  -Volume of Square = " + volume1);
System.out.println("  -Volume of Rectangle = " + volume2);
} }
```

Recursion

Defintion:

- Process of calling function itself.
- A recursive procedure routine is one that has ability to call itself
- Why recursion:
    - it requires the least amount of code to perform the necessary functions.
    - Solves complicated problems in simple steps
- A recursive function has two parts:
    - **Base case:** is stopping condition to prevent an infinite loop
    - **Recursive case:** it must always get closer to base case from one invocation to another.
- Example 1: Factorial of a number

❖ Mathematical definition

$$\text{product}(a, b) = \begin{cases} 1 & \text{if } n = 0 \\ n \,.\, \text{factorial}(n-1) & \text{if } n > 1 \end{cases}$$

❖ Recursive Method

```
class RecursiveDemo  {
        int factorial(int number)
                {
            if (number == 0)
            return 1;
            else
            return (number * factorial(number - 1));
                }
            public static void main(String [] args) {

            RecursiveDemoob = new RecursiveDemo();
            int fact = ob.factorial(5);
            System.out.println("Factorial = " + fact);
            }
}
```

Example 2: Logic of Multiplication of 2 natural numbers

- Multiplication of (a*b) can be defined as
  Number 'a' added to itself b times
- Mathematical definition

$$\text{product}(a, b) = \begin{cases} 0 & \text{if } a = 0 \text{ or } b = 0 \\ 1 & \text{if } b = 1 \\ a + \text{prod}(a, b-1) & \text{if } b > 1 \end{cases}$$

```
int prod(int a, int b)
{
        if (a == 0 || b == 0)   return 0;
        if (b==1)               return 1;
        else        return a + prod(a, b-1);
}
```

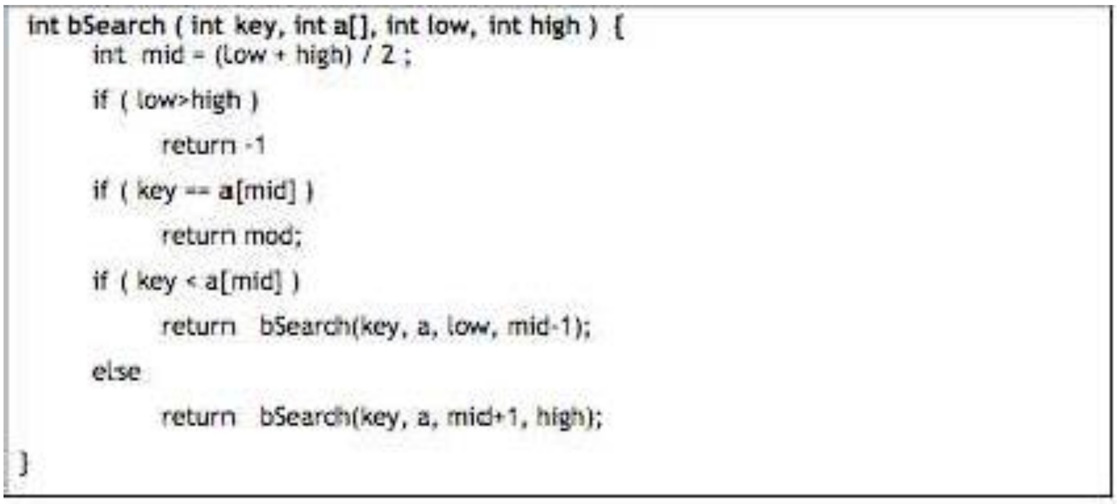
Example 3: Logic of: generating Fibonacci number for nth term

- Mathematical definition

$$\text{fib}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } b > 1 \end{cases}$$

```
int fib ( int n )
{
        if (n == 0 || n == 1)   return n;
        else        return  fib(n-1)  +  fib(n-2);
}
```

Example 4: Logic of: Binary Search



```
int bSearch ( int key, int a[], int low, int high ) {
    int mid = (Low + high) / 2 ;
    if ( low>high )
        return -1
    if ( key == a[mid] )
        return mod;
    if ( key < a[mid] )
        return  bSearch(key, a, low, mid-1);
    else
        return  bSearch(key, a, mid+1, high);
}
```

Understanding static

- Generally to accesses normal members of other class ***"an object is must".***
- But to Access Static member of other class, object is not needed.
- Static members *can be accessed before any objects of its class are created.*
- Static members *can be accessed using a dot operator with the class name.*
- Syntax:

```
className.StaticVariable;

className.StaticMethod();
```

- Most common example of a static member is main( ).
- main() is declared as static because it must be called before any object.
- Different uses of static are:
    - *static variable*
    - *static method*
    - *static block*
- **static variable**
    - 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.
    - A static variable can be accessed Using dot operator with class name.

Syntax

```
className.staticVariable
```

- **static method**
    - **Restrictions on static method**

- They can only call other static methods.
- They must only access static data.
- They cannot refer to 'this' or 'super' in any way.
- They cannot refer instance variables.

- A static variable can be accessed using dot operator with class name.

Syntax:

```
className. staticmethod( )
```

```
class StaticDemo {
static float basic = 5000;
static float DA;
     static {
          System.out.println("..Static
          block ..");
          DA = basic * 0.10;
          }
static void static ShowSalary() {
     System.out.println("..Static Method ..");
     float salary = basic + DA ;
     System.out.println("Net salary = " + salary);
     }

public static void main(String[] args) {
StaticDemo.staticShowSalary(); // No object needed }
}
```

Static variable

Static block

Static method

Execution steps:

1. StaticDemo class is loaded which Run all static statements

2. basic is set to 5000,

3. static block executes and DA is set,

4. main( ) is called, o *calls staticShowSalary( )*

Introducing Nested and Inner classes

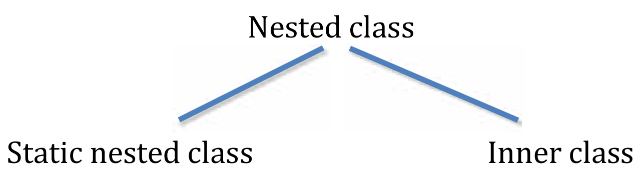
Definition

- Nested class is a class defined within another class.
- Syntax

```
classEnclosingClass {
.....
classNestedClass {
…….
  }
}
```

Thus, nested class does not exist independently of Enclosing class

- **Types of Nested class**



- **Static Nested class**

```
Class EnclosingClass {
        ……
        static class StaticNested {
        ……
        }
}
```

- Creating instance of static Nested class

  EnclosingClass.staticNestedob = new EnclosingClass.staticNested();
- A static nested class has the static modifier applied.
- Because it is,
  - it must access the members of its enclosing class through an object.
  - it cannot refer to members of its enclosing class directly.
  - Because of this restriction, static nested classes are seldom used.
- **Inner class**

```
Class OuterClass {
        ……
        classInnerClass{
        ……
        }
}
```

- Creating instance of inner class
  OuterClassoutob =new OuterClass();
  OuterClass.InnerClassinob = outob.newInnerClass();
- Inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class
- may refer to them directly in the same way that other non-static members of the outer class do.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

2. 8 Varargs: Variable –length arguments

- **Varargs:** stands for variable-length argument
- A method that takes a variable number of arguments is called a "***variable-arity***" method, or simply a ***varargs***method.
- **Example**: printf( ) method that is part of Java"s I/O library.
- **Different ways to handle variable-length argument**
  - if the number of arguments was ***small and known***, then use method overloading, one for each.
  - if the number of arguments was ***larger, or unknowable***, then use Array and add arguments into an array.
  - Use three periods (...) to specify variable-length argument.
- Syntax

```
<returnType>MethodName (int… arrayName) {
--
}
```

```
static void vaTest (int ... v) { ------------------

}
```

- This syntax tells the compiler that vaTest( ) can be called with zero or more arguments.
- As a result, v is implicitly declared as an array of type int[ ].

- Thus, inside vaTest( ), v is accessed using the normal array syntax.

```
public class VarArgsDemo {

    public static void main(String[] args) {

        VarArgsDemoob = new VarArgsDeno();
        int a = ob.getSum(10, 20);
        System.out.println("Sum="+a);

        a = ob.getSum(10, 20, 30, 40);
        System.out.println("Sum="+a);
     }
    intgetSum (int ... v) {
        int sum = 0;
        for ( intnum: v )
            sum = sum + num;
        return sum;
    }
}
```