Understanding Methods and Variables Scope

- Variables
- Member Modifiers
- Encapsulation
- Constructors
- Methods
    - Pass Objects by Reference and Value
    - Var-Arg Methods
    - Method Overloading
    - Main Method

Variables

- Depending on type of contents

  a). Reference Variables : to hold object references

  Ex : String s = "Java";

  b). Primitive Variables: to hold primitive values

  Ex : int i=10;

- Depending on the position at which variable is declared

  a). Local Variables b). Instance Variables c). Static Variables

Local Variables (stack/temporary/automatic variables)

- Declared inside a method or block or constructor or in the method arguments
- Gets created as part of method execution and gets destroyed as soon as the method execution completes
- Must be initialized before using a local variable

```
public void print() {
    int i=10;
    System.out.println(i);
}  //Error : variable i might not have been initialized
```

Instance Variables (properties/attributes/member variables)

- If the values of the variables are varied from instance to instance such type of variables are called as instance variables
- Can be declared within a class, outside of any method or block
- Gets created as soon as new object is created and destroyed whenever garbage collector destroys that object
- Gets default values, no need to perform explicit initialization

```
public class Student
{
  int no;
  String name;

  public static void main(String args[])
 {
  Student s1 = new Student();
  System.out.println(s1.no + "...." + s1.name); //0....null

  s1.no = 101;
  s1.name = "Sam";
  System.out.println(s1.no + "...." + s1.name); //101....Sam

  Student s2 = new Student();
  s2.no = 102;
  s2.name = "Tom";
  System.out.println(s2.no + "...." + s2.name); //102....Tom
  }
}
```

Static Variables (fields/class variables)

- static keyword can be applied for methods, variables and inner classes , but not for calsses
- If a single copy of the variable needs to be maintained & shared by all instances, such variables are called as static
- Value of static variable is same for all instances
- Gets created when the class is loaded into the memory and destroyed when the class is unloaded from memory
- Gets default values, no need to perform explicit initialization
- Can be accessed by using class name (highly recommended) or object reference from static and non-static contexts

```
public class Employee {
    String empName;
    static String orgName;
    public static void main(String[] args) {
        Employee e1 = new Employee();
     System.out.println(e1.empName+" works for "+e1.orgName);
        e1.empName = "Sam";
        e1.orgName = "Sun";
     System.out.println(e1.empName+" works for "+e1.orgName);
        Employee e2 = new Employee();
        e2.empName="Tom";
     System.out.println(e2.empName+" works for "+e2.orgName);
        orgName = "Oracle";
     System.out.println(e1.empName+" works for "+e1.orgName);
     System.out.println(e2.empName+" works for "+orgName);
    }
}
```

- Instance variables can not be accessed from static context directly.

```
public class Employee1 {
     String empName ="Sam";
    public static void main(String[] args) {
        System.out.println(empName);
//Error: non-static variable empName cannot be referenced
// from a static context
    } }
```

- static variables are used to define class level CONSTANTS(final)

```
public class Employee2 {
    public static final String ORGANIZATION_NAME ="Oracle";
    public static void main(String[] args) {
        System.out.println(ORGANIZATION_NAME);
    }
}
```

Member Modifiers

- default : (package level modifier)
  default member is visible with in the current package only
- public : can be acessed from anywhere within the package or outside the package
- private : can be accessed within the class only and can not be accessed it from outside the class
- protected : can be accessed in every class within the package, and accessible only in child classes from outside the package
  protected = default (current package) + child classes

```java
package var;
public class Variables {
    String sDef ="default";
    public String sPub = "public";
    private String sPri = "private";
    protected String sPro = "protected";
}

package var;
public class VarSample1 {
    public static void main(String[] args) {
        Variables v = new Variables();
        System.out.println(v.sDef);
        System.out.println(v.sPub);
        System.out.println(v.sPri);
      // Error : sPri has private access in var.Variables
        System.out.println(v.sPro);
   } }
```

```
package met;
import var.Variables;
public class VarSample2 {
    public static void main(String[] args) {
        Variables v = new Variables();
        System.out.println(v.sDef);
//Error : sDef is not public in Variables1; cannot be accessed
        System.out.println(v.sPub);
        System.out.println(v.sPri);
//Error : sPri has private access in Variables1
        System.out.println(v.sPro);
//Error : sPro has protected access in Variables1
    }
}
```

- protected members in child class must be accessed with child class refererence only

```
package met;
import var.Variables;
public class VarSample3 extends Variables{
    public static void main(String[] args) {
        Variables v = new Variables();
        System.out.println(v.sPro);
  // Error : sPro has protected access in var.Variables

        VarSample3 v1= new VarSample3();
        System.out.println(v1.sPro);

        Variables v2= new VarSample3();
        System.out.println(v2.sPro);
  // Error : sPro has protected access in var.Variables
    }
}
```

Encapsulation

- Data Hiding : Restricting direct data access from outside of the class by declaring all the data memebers as private
- Abstraction : Hiding internal implementation
- Encapsulation = Data Hiding + Abstraction
- Hiding data behind methods is the key concept of Encapsulation
- Benefits : security, easy to inhance, maintainability
- A class is said to be tightly encapsulated if and only if all the data memebers declared as private
- If a parent class is not tightly encapsulated, then no child class will be tighly encapsulated class

```
public class EncapSample1
{
  private String name;

  public void setName(String name) { this.name=name; }
  public String getName() { return name; }

  public static void main(String[] args){
      EncapSample1 s= new EncapSample1();
      s.setName("java");
      System.out.println(s.getName());
    }
}
```

Constructors

- Purpose: perform initialization of all data members of an object
- Gets called automatically to initialize data members when a object is created
- Applicable for every class including abstract class, but not interfaces
- Name of the constructor must be same as class name
- Allowed modifiers are public, default, protected and private
- Applying any other modifier to constructor will give compilation error : " Modifier xxx is not allowed here."
- Though it is legal, not a good practice to give return type, even void also. If we give it compiler / jvm treats it as a normal method, not as constructor

Default Constructor

- If programmer written constructor does not exist, then compiler will generate a default constructor.
- Either programmer written constructor or compiler generated constructor must present in a class. But not both at the same time.
- The default constructor is always no-arg constructor
- The access modifier of the default constructor is same as class modifier (public & default)
- The default constructor contains only one statement which is no-arg call to super class constructor.

super() & this()

- Allowed to keep more than one constructor inside a class which is called Constructor overloading
- Allowed only in constructors, not anywhere else
- Constructor can have either super() or this(), but not both.
- Must be first statement in constructor

```
public  class ConstStudent {
  int rollNo;
  String studentName;

  ConstStudent () { this(10,"Sam"); }

  ConstStudent ( int rollNo, String studentName) {
        super();
        this.rollNo = rollNo;
        this.studentName  = studentName ;
  }
  public static void main(String args[]) {
    ConstStudent s1 = new ConstStudent ();
    ConstStudent s2 = new ConstStudent (101, "Tom");

    System.out.println(s1.rollNo+"......"+s1.studentName);
    System.out.println(s2.rollNo+"......"+s2.studentName);
   }
  }
// Output:
10......Sam
101......Tom
```

- Constructors are not inherited and hence overriding a constructor is not possible
- Recursive constructor calls:

```
public  class ConstStudent1 {
  int rollNo;
  String studentName;

  ConstStudent () { this(10,"Sam"); }

  ConstStudent ( int rollNo, String studentName) {
        this(); // Error : recursive constructor invocation
        this.rollNo = rollNo;
        this.studentName  = studentName ;
  }
  public static void main(String args[]) {
    ConstStudent s1 = new ConstStudent ();
    System.out.println(s1.rollNo+"......"+s1.studentName);
    }
  }
```

Pass Objects by Reference and Value

- Passing primitives by value to methods

```
public class PassValSample {
    int no1 ;
    int no2 ;

    public void swap(int a, int b) {
        int c;
        System.out.println("Before swap a: "+a+"  b:"+b);
        c=a;         a=b;         b=c;
        System.out.println("After swap a: "+a+"  b:"+b);
    }

 public static void main(String[] args) {
        PassValSample s = new PassValSample();
        s.no1=10;
        s.no2=20;
 System.out.println("Before swap no1: "+s.no1+"  no2:"+s.no2);
        s.swap(s.no1,s.no2);
 System.out.println("After swap no1: "+s.no1+"  no2:"+s.no2);
 }
}
```

- Passing objects by reference to methods

```
public class PassRefSample {
   int no1 ;
   int no2 ;

   public void swap(PassRefSample s1) {
       int no;
 System.out.println("In swap no1: "+s1.no1+"  no2:"+s1.no2);
       no=s1.no1;
       s1.no1=s1.no2;
       s1.no2=no;
System.out.println("After swap no1: "+s1.no1+"  no2:"+s1.no2);
   }

   public static void main(String[] args) {
   PassRefSample s = new PassRefSample();
       s.no1=10;
       s.no2=20;
 System.out.println("Before call no1: "+s.no1+"  no2:"+s.no2);
       s.swap(s);
 System.out.println("After call no1: "+s.no1+"  no2:"+s.no2);
   }
}
```

Variable Argument (var-arg) Methods

- Allow us to specify that a method can take multiple arguments of the same types and allows no. of variables to be variable

```
public class VarArgSample1
{   static int add(int... x) {
    int sum = 0;
    System.out.println("Number of arguments: " + x.length);

    // using for loop to show array representation
    // for (int i = 0; i < x.length; i++) sum = sum + x[i];

    for (int a : x)  sum = sum + a; // using for each loop
        return sum;
    }

public static void main(String[] args) {
 System.out.println("Result="+add()); // no parameter
 System.out.println("Result="+add(1,2)); // 2 params
 System.out.println("Result="+add(1, 2, 3, 4)); // 4 params
   }  }
```

- var-arg parameter must be last in the parameter declarations, when we have other regular parameters.

```
public class VarArgSample2
{
//public void show( int... a, String name) { //Error : ')' expe
   public void show(String name, int... a){
        System.out.println(name+" scores :");
        for (int i:a)     System.out.println(i);
    }

  public static void main(String[] args) {
        VarArgSample2 x = new VarArgSample2();
        x.show("Sam", 90,80);
        x.show("Tom", 56, 90, 80, 50 );
    }
}
```

- In var-arg method, we are restricted to have only one var-arg param and it must be last parameter.

```
public class VarArgSample3 {
   public static void main(String[] args) {
       VarArgSample3 x = new VarArgSample3();
       x.print(1,2,3,'a','b','c','d');
   }

//public void print(int... a, char... c)//Error : ')' expected
  public void print(int... a)
   {
       for(int i:a)  System.out.println(i);
     /*   for (char ch : c) System.out.println(ch); */
   }
}
```

- Var-arg method will get last priority when there is method exists with exact no.of parameteres match.

```
public class VarArgSample4 {

  public void print(int... a) {
      System.out.println("Var-Arg Method");  }

  public void print(int a) {
      System.out.println("Regular Method");  }

  public static void main(String[] args) {
      VarArgSample4 x = new VarArgSample4();
      x.print(2);
      x.print(1,2);
  }
}
```

Method Signature

- Composed of method name and argument list (the order of arguments is also important)
- Return type is not part of signature
- Used by Compiler to resolve method calls within a class
- A class cannot have 2 methods with same signature

```
class SignatureSample {
public int m1(int i) {}
public void m1(int i) {}
}
//Error : method m1(int) is already defined in class SignatureS
```

Method Overloading

- 2 methods are said be overloaded if and only if they have same method name but different parameter list (atleast in order)
- return type, access modifier and throws clause are not considered in overloading

```
public class OLSample1 {
public void m1(){System.out.println("No-Arg");}
public int m1(int i){System.out.println("Int-Arg"); return i;}
public void m1(double d){System.out.println("Double-Arg");}
  public static void main(String[] args) {
       OLSample1 s = new OLSample1();
       s.m1(); // No-Arg
       s.m1(5); // Int-Arg
       s.m1(5.5); // Double-Arg
   } }
```

Method resolution - primitives

- Automatic promotion of parameteres in overloading

  byte -> short/char -> int -> long -> float -> double

```
public class OLSample2 {
   public void m1(int i, float f)
          { System.out.println("Int-Float"); }
   public void m1(float f, int i)
          { System.out.println("Float-Int"); }

   public static void main(String[] args){
       OLSample2 s= new OLSample2();
       s.m1(10,10.5f); //Int-Float
       s.m1(10.5f,10); //Float-Int
       s.m1(10,10);  //Error: reference to m1 is ambiguous;
   }
}
```

Method resolution - object references

- Compiler chooses the method based on the object reference type, not on run time object.

```
public class OLsample3 {
public void m1(Object o){System.out.println("ObjectVersion");}
public void m1(String s){System.out.println("StringVersion");}
public void m1(OLsample3 ols)
            { System.out.println("OLsample3 version"); }
public static void main(String[] args) {
        Object o= new Object();
        String s = "Java";
        OLsample3  ol= new OLsample3();

        ol.m1(s); // String Version
        ol.m1(ol); // OLsample3 version
        ol.m1(o); // Object Version
        o=ol;  ol.m1(o); // Object Version
       // ol.m1(null); //Error : reference to m1 is ambiguous;
    }
}
```

Main Method

- Running such class without main method , will give the error : " NoSuchMethodError : main "
- Any of below changes to main method
  static -> non-static
  void -> some other return type
  main -> any other name
  will give the error : " NoSuchMethodError : main " while running the class
- Order of public & static can be interchanged
- Main method can be declared as final & synchronized => No Compilation Error and Run time Exception

References :

- OCA Java SE 8 Programmer I Study Guide
- https://docs.oracle.com/javase/tutorial/java/javaOO/index.html
- You can text me on slack : @raju / raju.sandepogu@gmail.com