Object Oriented Programming

Topic: Class & Object using Java

the basics you need to know

Dr. Shaon Bhatta Shuvo

Assistant Professor

School of Computer Science

University of Windsor

Object - Objects have states and behaviours. Example: A human has states - name, age, nationality as well as behaviours -walking, sleeping, eating.
An object is an instance of a class.

Class - A class can be defined as a template/blue print that describes the behaviours/states that object of its type support.

Instance- An instance is a unique copy of a Class that representing an Object. When a new instance of a class is created, the JVM will allocate a room of memory for that class instance.

Note: Here we'll discuss these basic features(Object & Class) of object oriented programming using java.

Objects in Java

- Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, Humans, etc. All these objects have a state and behaviour.
- If we consider a dog, then its state is - name, breed, colour, and the behaviour is - barking, wagging, running
- If you compare the software object with a real world object, they have very similar characteristics.
- Software objects also have a state and behaviour. A software object's state is stored in fields and behaviour is shown via methods.
- So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Objects & Instances are Same?

- Yeah Almost! But there is a very tiny difference (you may ignore)
- Objects are the definitions of something, instances are the physical things.
- Example:
  - Say you have 10 apples in your basket. Each of those apples is an object of type Apple, which has some characteristics (i.e. green, round, grows on trees).
  - In programming terms, you can have a class called Apple, which has variables color:green, shape:round, habitat:grows on trees. To have 10 apples in your basket, you need to *instantiate* 10 apples. Apple apple1, Apple apple2, Apple apple3 etc....

Classes in Java

- A class is a blueprint from which individual objects are created.
- A class mainly consists of Variables and Methods.
- For example-

```
public class Human{
        String name;
        int age;
       String nationality;

       void walking(){
        }

        void eating(){
        }

        void sleeping(){
        }
}
```

Variables

- A class can contain any of the following variable types-

- ✓ **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

- ✓ **Instance variables:** Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

- ✓ **Class variables:** Class variables are variables declared with in a class, outside any method, with the static keyword.

N.B: Together Instance Variables and Class Variables are also Known as **Field Variable** or **Data Member**.

Method

- A Java method is a collection of statements that are grouped together to perform an operation.
- ✓ Considering the following example to explain the syntax of a method:

```
public static int methodName(int a, int b) {
        // body
}
```

Here,

public static : modifier.

int: return type

methodName: name of the method

a, b: formal parameters

int a, int b: list of parameters

Method

Method definition consists of a method header and a method body. The same is shown below:

```
modifier returnType nameOfMethod (Parameter List) { // method body }
```

The syntax shown above includes:

- **modifier/ access modifier:** It defines the access type of the method and it is optional to use. (e.g. public, protected, private etc.)
- **returnType:** Method may return a value.
- **nameOfMethod:** This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List:** The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body:** The method body defines what the method does with statements.

Modifiers ?

Modifiers fall into **two** categories:

- Access modifiers: **public, protected, private**.
- Non-access modifiers (including **strictfp, final, and abstract**).

✓ We'll look at access modifiers first, so you'll learn how to restrict or allow access to a class you create.

❖ **Access control in Java is a little tricky because there are four access *controls* (levels of access) but only three access *modifiers*. The fourth access control level (called *default* or *package* access) is what you get when you don't use any of the three access modifiers.**

In other words, *every* class, method, and instance variable you declare has an access *control*, whether you explicitly type one or not.

Although all four access *controls* (which means all three *modifiers*) work for most method and variable declarations, a class can be declared with only public or *default* access; the other two access control levels don't make sense for a class, as you'll see.

public, protected, private, default(no modifier)

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Let's Write a Method

This method takes two parameters num1 and num2 and returns the minimum between the two:

```
/* the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
                min = n2;
        else
                min = n1;
        return min;
}
```

Method Calling

- For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value).

- The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when: return statement is executed, reaches the method ending closing brace.

- The methods returning void is considered as call to a statement. Lets consider an example:

```
System.out.println("This is going to be displayed !");
```

- The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

Method Calling(Example)

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber{
        public static void main(String[] args) {
                int a = 11;
                int b = 6;
                int c = minFunction(a, b);
                System.out.println("Minimum Value = " + c);
        }
/** returns the minimum of two numbers */

        public static int minFunction(int n1, int n2) {
                int min;
                if (n1 > n2)
                        min = n2;
                else
                        min = n1;
                return min;
        }
}
```

This would produce the following result:

```
Minimum value = 6
```

The void Keyword

- The void keyword allows us to create methods which do not return a value.
- Call to a void method must be a statement i.e. *methodRankPoints(255.7);*
- It is a Java statement which ends with a semicolon as shown below.

Let's look at the example-

```
public class ExampleVoid {
        public static void main(String[] args) {
                methodRankPoints(255.7);
        }
        public static void methodRankPoints(double points) {
                if (points >= 202.5) {
                        System.out.println("Rank:A1");
                }
                else if (points >= 122.4) {
                        System.out.println("Rank:A2");
                } else {
                        System.out.println("Rank:A3");
                }
        }
}
```

This would produce the following result:

```
        Rank:A1
```

Constructor

- A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.
- Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.
- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

```
public class Puppy{
        public Puppy(){// This constructor has no parameter
        }
        public Puppy(String name){ // This constructor has one parameter, name.
        }
}
```

Creating an Object

- As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java, the new key word is used to create new objects.
- There are three steps when creating an object from a class:
  - ✓ **Declaration:** A variable declaration with a variable name with an object type.
  - ✓ **Instantiation:** The 'new' key word is used to create the object.
  - ✓ **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example

Example of creating an object is given below:

```
public class Puppy{
        public Puppy(String name){ // This constructor has one parameter, name.
            System.out.println("Passed Name is :" + name );
        }
        public static void main(String[] args){
           // Following statement would create an object myPuppy
           Puppy myPuppy = new Puppy( "tommy" );
        }
}
```

If we compile and run the above program, then it would generate the following output:

```
Passed Name is :tommy
```

Accessing Instance Variables and Methods

- ✓ Instance variables and methods are accessed via created objects. To access an instance variable the fully qualified path should be as follows:

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example

```
1  public class Puppy{
2
3     int puppyAge;
4
5     public Puppy(String name){
6        // This constructor has one parameter, name.
7        System.out.println("Name chosen is :" + name );
8     }
9
10    public void setAge( int age ){
11       puppyAge = age;
12    }
13
14    public int getAge( ){
15       System.out.println("Puppy's age is :" + puppyAge );
16       return puppyAge;
17    }
18
19    public static void main(String []args){
20       /* Object creation */
21       Puppy myPuppy = new Puppy( "tommy" );
22
23       /* Call class method to set puppy's age */
24       myPuppy.setAge( 2 );
25
26       /* Call another class method to get puppy's age */
27       myPuppy.getAge( );
28
29       /* You can access instance variable as follows as well */
30       System.out.println("Variable Value :" + myPuppy.puppyAge );
31    }
32 }
33
```

Output:

Name chosen is :tommy
Puppy's age is :2
Variable Value :2

Let’s see something that you need to know …

MORE ON METHODS & CONSTRUCTOR

Call by Value

```
class CallByValue {
  public static void main ( String[] args ) {
    int x =3;
    System.out.println ( "Value of x before calling increment() is "+x);
    increment(x);
    System.out.println ( "Value of x after calling increment() is "+x);
  }

  public static void increment ( int a ) {
    System.out.println ( "Value of a before incrementing is "+a);
    a= a+1;
    System.out.println ( "Value of a after incrementing is "+a);
  }
}
```

Output:

Value of x before calling increment() is 3
Value of a before incrementing is 3
Value of a after incrementing is 4
Value of x after calling increment() is 3

Note: As is evident from the output, the value of x has remain unchanged, even though it was passed as a parameter to the increment() method. (This program contains two static methods. The method increment() was also specified to be static since only static members can be accessed by a static method)

Call by Reference

```
class  Number {
  int  x;
}
class CallByReference {
  public static void main ( String[] args ) {
    Number a = new Number();
    a.x=3;
    System.out.println("Value of a.x before calling increment() is "+a.x);
    increment(a);
    System.out.println("Value of a.x after calling increment() is "+a.x);
  }
  public static void increment(Number n) {
    System.out.println("Value of n.x before incrementing x is "+n.x);
    n.x=n.x+1;
    System.out.println("Value of n.x after incrementing x is "+n.x);
  }
}
```

Output:

Value of a.x before calling increment() is 3
Value of n.x before incrementing x is 3
Value of n.x after incrementing x is 4
Value of a.x after calling increment() is 4

Call by Value Vs. Call by Reference

- ✓ Now, there is a remarkable difference between the outputs obtained in the above two programs:
- ➢ In the first program, the change made to the variable a inside the increment() method had no effect on the original variable x that was passed as an argument.
- ➢ On the other hand, in the second program, changes made to the variable x that was a part of the object in the increment() method had an effect on the original variable (the object which contained that integer variable) that was passed as an argument.
- ➢ The difference lies in the type of the variable that was passed as an argument. int is a primitive data type while Number is a reference data type.
- ➢ Primitive data types in Java are passed by value while reference data types are passed by reference.

- ✓ What we mean by passing a variable by value is that the value held in the variable that is passed as an argument is copied into the parameters that are defined in the method header. That is why changes made to the variable within the method had no effect on the variable that was passed.
- ✓ On the other hand, when objects are passed, the object itself is passed. No copy is made.

Call by Value Vs. Call by Reference

The concept of call by reference can be better understood if one tries to look into what a reference actually is and how a variable of a class type is represented.

- When we declare a reference type variable, the compiler allocates only space where the memory address of the object can be stored. The space for the object itself isn't allocated.
- The space for the object is allocated at the time of object creation using the new keyword.
- A variable of reference type differs from a variable of a primitive type in the way that a primitive type variable holds the actual data while a reference type variable holds the address of the object which it refers to and not the actual object.

Consider the following program which illustrates theses concepts.

```
class  Number {
  int  x;
}

public class  Reference {
  public static void main ( String[] args ) {
    Number  a = new Number();
    a.x = 4;
    System.out.println(a.x);
    Number  b = a;
    b.x = 5;
    System.out.println(b.x);
  }
}
```

The output would be:

4

5

This program creates just one object and not two. The statement Number b doesn't create a new object. Instead it only allocates some space where the address of the object to which b refers would be stored. The statement b=a; simply copies the value stored in a to b. This value isn't the object itself but is simply the address at which the object is stored. Therefore, both a and b refer to the same object. Any change made to the object through either of the variables gets reflected on the other variable also. That is why when we have changed the value of x through the variable name b to 5, the change was reflected on the value of x accessed through a. this is because, a and b are simply different names for the same object in the computers 'memory.

Method Overloading Vs. Overriding

- When a class has two or more methods by same name but different parameters, it is known as method overloading.
  In overriding a method has same method name, type, number of parameters etc.
- Method overloading is performed *within class*.
  Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.

- Method Overloading is a perfect example of ***static binding*** and Method Overriding is ***dynamic binding.***

- Method overloading is the example of *compile time polymorphism*.
  Method overriding is the example of *run time polymorphism*.
- We'll briefly discuss about this topic in the section of Polymorphism.

Types of Constructor

- Constructors can be declared in two ways-
  - ✓Constructor without Parameter
  - ✓Constructor with Parameter/ Parameterized Constructor

Constructor without Parameter(Default Constructor)

Here is a simple example that uses a constructor without parameters:

```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass() {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce the following result:

```
10 10
```

Parameterized Constructor

Here is a simple example that uses a constructor with parameter:

```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass(int i ) {
      x = i;
   }
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce the following result:

```
10 20
```

***Note:**

- ✓ constructor **without** parameter **always** inherited automatically to the child class.
- ✗ constructor **with** parameter **never** inherited automatically, it can be inherited through explicit call using ***super*** keyword.

Inheritance of default constructor

```
 * @author SHUVO
 */
public class A {
    A(){
        System.out.println("This is A");
    }
}
public class B extends A {
    B(){
        System.out.println("This is B");
    }
}
public class C extends B{
   /* C(){System.out.println("This is C");}*/
    C(int x){
        System.out.println(x);
    }
    public static void main(String[] s){
        System.out.println("Default constructor will always be Inheritated ");
        C ob = new C(5);
    }
}
```

Output:

Default constructor will always be inherited
This is A
This is B
5

***Note:** *even the first three lines of output will be same if we use no constructor or constructor without parameter for child class C !*

- ✓ Don't worry ! You'll understand this program as soon as we start the discussion on ***inheritance*** ☺

Can a Constructor be private?

✓ YES! The **use of private constructor** is to serve singleton classes. A singleton class is one which limits the number of objects creation to one. Using private constructor we can ensure that no more than one object can be created at a time.

```
public class SingleTonClass {
   //Static Class Reference
   private static SingleTonClass obj=null;
   private SingleTonClass(){
      /*Private Constructor will prevent
       * the instantiation of this class directly*/
   }
   public static SingleTonClass objectCreationMethod(){
        /*This logic will ensure that no more than
         * one object can be created at a time */
        if(obj==null){
            obj= new SingleTonClass();
        }
        return obj;
   }
   public void display(){
        System.out.println("Singleton class Example");
   }
   public static void main(String args[]){
        //Object cannot be created directly due to private constructor
        //This way it is forced to create object via our method where
        //we have logic for only one object creation
        SingleTonClass myobject= SingleTonClass.objectCreationMethod();
        myobject.display();
   }
}
```

***Note:**

✗ A class with private constructor can't be inherited.

➢ Because child class must call the constructor implicitly or explicitly using ***super*** keyword.

Constructor Properties

❑ **What can not?**

1. "**A constructor cannot be abstract, static, final, native, strictfp, or synchronized**".
2. Interface cannot have constructor.
3. Constructor cannot return a value.

❑ **What can?**

1. A constructor can be private.
2. Abstract class can have constructor.
3. A constructor can be overloaded.

Note:

Instance variables and methods of a class are known as **members** of a class. **Constructors are not members**. For this reason, Constructors cannot be called with objects like **d1.display()**. To access a constructor create an object or access with subclass constructor (explicitly with **super()**).

this Keyword

- **this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.
  - ✓ In general the keyword *this* is used to :Differentiate the instance variables from local variables if they have same names, within a constructor or a method.
  - ✓ this keyword can be used to refer current class instance variable.

```
class Student{

    int age;
    Student(int age){
    this.age=age;
    }

}
```

Call one type of constructor(parametrized constructor or default) from other in a class. It is known as explicit constructor invocation .

```
class Student{

    int age
    Student(){
    this(20);
    }

    Student(int age){
    this.age=age;
    }

}
```

- ***this*** keyword can be very useful in the handling of Variable Hiding. We can not create two instance/local variables with the same name. However it is legal to create one instance variable & one local variable or Method parameter with the same name. In this scenario the local variable will hide the instance variable this is called Variable Hiding.
- Example:

```
1  class JBT {
2
3   int variable = 5;
4
5   public static void main(String args[]) {
6   JBT obj = new JBT();
7
8   obj.method(20);
9   obj.method();
10  }
11
12  void method(int variable) {
13  variable = 10;
14  System.out.println("Value of Instance variable :" + this.variable);
15  System.out.println("Value of Local variable :" + variable);
16  }
17
18  void method() {
19  int variable = 40;
20  System.out.println("Value of Instance variable :" + this.variable);
21  System.out.println("Value of Local variable :" + variable);
22  }
23 }
```

Output:
Value of Instance variable :5
Value of Local variable :10
Value of Instance variable :5
Value of Local variable :40

- ***"this"*** keyword can be used inside the constructor to call another overloaded constructor in the same Class. This is called the Explicit Constructor Invocation. This occurs if a Class has two overloaded constructors, one without argument and another with argument. Then the ***"this"*** keyword can be used to call constructor with argument from the constructor without argument. This is required as the constructor can not be called explicitly.
- **Example:**

```
1  class JBT {
2
3   JBT() {
4   this("JBT");
5   System.out.println("Inside Constructor without parameter");
6   }
7
8   JBT(String str) {
9   System.out
10  .println("Inside Constructor with String parameter as " + str);
11  }
12
13  public static void main(String[] args) {
14  JBT obj = new JBT();
15  }
16 }
```

OUTPUT:
Inside Constructor with **String** parameter **as** JBT
Inside Constructor without parameter

***Note:**

- ✓ ***this*** keyword can only be the first statement in Constructor.
- ✓ A constructor can have either ***this*** or ***super*** keyword but not both.
  - ***super*** keyword will be discussed briefly in the section of ***inheritance.***

this keyword with method

this keyword can also be used inside Methods to call another Method from same Class.

- Example:

```
1  class JBT {
2
3   public static void main(String[] args) {
4   JBT obj = new JBT();
5   obj.methodTwo();
6   }
7   void methodOne(){
8   System.out.println("Inside Method ONE");
9   }
10
11  void methodTwo(){
12  System.out.println("Inside Method TWO");
13  this.methodOne();// same as calling methodOne()
14  }
15 }
```

OUTPUT:

Inside Method TWO

Inside Method ONE

Example of this keyword as Method parameter

```
1
2  public class JBTThisAsParameter {
3
4   public static void main(String[] args) {
5   JBT1 obj = new JBT1();
6   obj.i = 10;
7   obj.method();
8   }
9
10 }
11
12 class JBT1 extends JBTThisAsParameter {
13  int i;
14
15  void method() {
16  method1(this);
17  }
18
19  void method1(JBT1 t) {
20  System.out.println(t.i);
21  }
22 }
23
```

*Note: ***this*** keyword can be used only inside non-static method !

✓ Don't worry ! You'll understand this program as soon as we start the discussion on ***inheritance*** ☺

Using Objects as Parameters

- ✓ It's correct and common to pass objects to methods.

❖ Example:

```
// 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));
  }
}
```

Output:

Ob1 == Ob2: true

Ob1 == Ob3: false

- One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter.

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

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

  // Notice this constructor. It takes an object of type Box.
  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); // create copy of 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);
  }
}
```

Variable Arguments(var-args)

- JDK 1.5 and above enable you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:
  - ✓ typeName... parameterName in the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Variable Arguments(var-args) Example

```
public class VarargsDemo {

   public static void main(String args[]) {
      // Call method with variable args
          printMax(34, 3, 3, 2, 56.5);
      printMax(new double[]{1, 2, 3});
   }

   public static void printMax( double... numbers) {
   if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
   }

   double result = numbers[0];

   for (int i = 1; i <  numbers.length; i++)
      if (numbers[i] >  result)
      result = numbers[i];
      System.out.println("The max value is " + result);
   }
}
```

This would produce the following result:

```
The max value is 56.5
The max value is 3.0
```

static Keyword

- If you apply static keyword with any method, it is known as static method.
  - A static method belongs to the class rather than object of a class.
  - A static method invoked without the need for creating an instance of a class.
  - **Static methods** are **inherited in Java** but they **don't take part in polymorphism.**
    If we attempt to override the **static methods** they **will** just hide the superclass **static methods** instead of overriding them.
  - static method can access static data member and can change the value of it.
  - Even a non-static method can also do the same!
- If you apply static keyword with any variable all instance share the same copy of the variable.
  - Can be accessed directly with the class without creating instance/object.
  - static variable == class variable.

static Keyword (Example)

Program of changing the common property of all objects(static field) using static method.

```
class Student9{
 int rollno;
 String name;
 static String college = "ITS";

 static void change(){
 college = "BBDIT";
 }

 Student9(int r, String n){
 rollno = r;
 name = n;
 }

 void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){
Student9.change();

Student9 s1 = new Student9 (111,"Indian");
Student9 s2 = new Student9 (222,"American");
Student9 s3 = new Student9 (333,"China");

s1.display();
s2.display();
s3.display();
}  }
```

Output:

111 Indian BBDIT

222 American BBDIT

333 China BBDIT

static Keyword (Example)

changing static member value using non-static method.

```
package teststatic;
/*
 @author SHUVO
 */
public class C{
    static String s ="C++";
    String ns= "non static!";
    void change(){
        s = "JAVA";
        ns = "lets change it for current object!";
    }
    public static void main(String[] args){
        //System.out.println("This is: Inheritated ");
        C c1 = new C();
        C c2 = new C();
        C c3 = new C();
        System.out.println("Beforecalling change method s = "+s+" "+ "and ns = "+c1.ns);
        c1.change();
        System.out.println("For object c1: s= "+c1.s+" and "+"ns= "+c1.ns);
        System.out.println("For object c2: s= "+c2.s+" and "+"ns= "+c2.ns);
        System.out.println("For object c1: s= "+c3.s+" and "+"ns= "+c3.ns);
    }
}
```

Output:

Before calling change method s = C++ and ns = non static!
For object c1: s= JAVA and ns= lets change it for current object!
For object c2: s= JAVA and ns= non static!
For object c1: s= JAVA and ns= non static!

*Note:

- ✓ A static method can access only static members and its local members!
- ✓ A non-static method doesn't have such restrictions.

static Block

- ✓ Static blocks are nothing but a normal block of code, enclosed in braces { },preceded with static keyword.
- ✓ These static blocks will be called when JVM loads the class into memory.
- ✓ In case a class has multiple static blocks across the class, then JVM combines all these blocks as a single block of code and executes it.
- ✓ Static blocks will be called only once, when it is loaded into memory. These are also called initialization blocks.

static Block (Example)

```
package mystatictest;
/* @author SHUVO */
public class MyStaticTest {
    static{
        System.out.println("This is static block!");
    }
    public void display(){
        System.out.println("This is display methods!");
    }
    static{
        System.out.println("JVM combines all the static blocks!");
    }
    public static void main(String[] args) {
       MyStaticTest mst = new MyStaticTest();
    }
}
```

Output:
This is static block!
JVM combines all the static blocks!

static Block vs. Constructor

```java
package mystatictest;

/* @author SHUVO*/
public class MyStaticTest {
    static{
        System.out.println("This is static block!");
        System.out.println("Static block will be called first!");
        System.out.println("Static block will be called only once!");
    }
    public MyStaticTest(){
        System.out.println("This is constructor!");
    }
    public static void main(String[] args) {
       MyStaticTest mst = new MyStaticTest();
       MyStaticTest mst2 = new MyStaticTest();
       MyStaticTest mst3 = new MyStaticTest();
    }
}
```

***Note:**

- ✓ Java static blocks will be called when JVM loads the class into memory, means it will be called only once.
- ❑ But constructor will be called every time when you create an object.

Output:

This is static block!
Static block will be called first!
Static block will be called only once!
This is constructor!
This is constructor!
This is constructor!

final Keyword

- A variable can be declared as **final**. Doing so prevents its contents from being modified.
  - This means that you must initialize a **final** variable when it is declared. For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

  - Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.
- It is a common coding convention to choose all uppercase identifiers for **final** variables.
- Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant
- Once you declare a Method as final it **can be inherited but can't be overridden.**
- If you declare a Class as final it **can't be extended.**
  - **N.B: We'll learn more about final keyword after done with the inheritance.**

Resources I've Used to Prepare The Lecture

- ✓ https://docs.oracle.com/javase/8/
- ✓ http://www.javatpoint.com/
- ✓ http://www.tutorialspoint.com/
- ✓ http://java2novice.com/
- ✓ http://www.javawithus.com/
- ✓ https://software.intel.com/
- ✓ http://www.instanceofjava.com/
- ✓ http://javabeginnerstutorial.com/
- ✓ http://stackoverflow.com/
- ✓ http://way2java.com/
- ✓ http://beginnersbook.com/
- ✓ JAVA: the complete reference By Herbert Scheldt.
- ✓ Sun Certified Programmer for Java- Study Guide by Kathy Sierra and Bert Bates.
- ✓ For Problem Solving Practice:
  Introduction to Java Programming by Y. Daniel Liang.

Happy Coding ☺