BIRLA INSTITIUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS F213

LAB-3 [Class Design Basics - Part-2]

AGENDA **DATE: 10/09/2020** **TIME: 02 Hours**

1. static variables, methods and static blocks
2. Objects as parameter to methods
3. Wrapper classes

1. static variables, methods and static blocks -

1.1 static variables -

a) It is a variable which belongs to the class and not to an object (instance).
b) Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
c) A single copy is shared by all instances of the class.
d) A static variable can be accessed directly by the class name and doesn't need any object.
e) Syntax: <class-name>.<variable-name>; for example Math.PI;

1.2 static method

a) It is a method which belongs to the class and not to the object(instance).
b) A static method can access only static data. It cannot access non-static data (instance variables). **WHY?**
c) A static method can call only other static methods and cannot call a non-static method from it. **WHY?**
d) A static method can be accessed directly by the class name and doesn't need any object.
e) Syntax:<class-name>.<method-name>;
f) A static method cannot refer to "this" or "super" keywords in anyway. **WHY?**

Side Note: main method is static, since it must be accessible for an application to run, before any instantiation takes place.

Let's learn the use of the static keywords by doing some exercises.

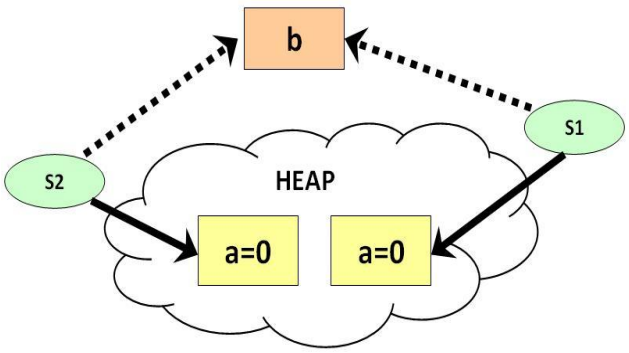
Example-1: static variables & methods

```
1. class MyStatic {

2.          int a; //initialized to zero
3.          static int b; /*initialized to zero only when class is
            loaded not for each object created.*/

4.          //Constructor incrementing static variable b
5.          MyStatic (){ b++; }
```

```
6.          public void showData(){
7.               System.out.println("Value of a = "+a);
8.               System.out.println("Value of b = "+b);
9.          }
10.
11.         //public static void increment(){
12.              //a++;
13.         //}
14.    }

15.    class StaticDemo{
16.              public static void main(String args[]){
17.                   MyStatic s1 = new MyStatic ();
18.                   s1.showData();
19.                   MyStatic s2 = new MyStatic ();
20.                   s2.showData();
21.                   // MyStatic.b++;
22.                   //s1.showData();
23.              }
24.    }
```

Run the StaticDemo class and observe the output.

Following diagram shows, how reference variables & objects are created and static variables are accessed by the different instances.



- It is possible to access a static variable from outside the class using the syntax `ClassName.VariableName`. Uncomment line# 21& 22in StaticDemo class. **Save and run the StaticDemo class and observe the output again.**

- Uncomment line 11,12 & 13of the MyStatic class. **Run the StaticDemo class and observe the output again.**What do you see as an output? ANSWER = Error. This is because it is not possible to access instance variable "a" from static method "increment".

1.3 static blocks

Static blocks are also called Static initialization blocks. A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code. And don't forget, this code will be executed when JVM loads the class. JVM combines all these blocks into one single static block and then executes. Here are a couple of points I like to mention:

- If you have executable statements in the static block, JVM will automatically execute these statements when the class is loaded into JVM.
- If you're referring some static variables/methods from the static blocks, these statements will be executed after the class is loaded into JVM same as above i.e., now the static variables/methods referred and the static block both will be executed.

Example-2: Demo of static blocks

```
public class StaticExample{

    static {
        System.out.println("This is first static block");
    }

    public StaticExample(){
        System.out.println("This is constructor");
    }

    public static String staticString = "Static Variable";

    static {
        System.out.println("This is second static block and "
                                                + staticString);
    }

    public static void main(String[] args){
        StaticExample statEx = new StaticExample();
        StaticExample.staticMethod2();
    }

    static {
        staticMethod();
        System.out.println("This is third static block");
    }
```

```
public static void staticMethod() {
        System.out.println("This is static method");
    }

    public static void staticMethod2() {
        System.out.println("This is static method2");
    }
}
```

What will happen when you execute the above code? Run the code and observe the output.

First all static blocks are positioned in the code and they are executed when the class is loaded into JVM. Since the static method staticMethod() is called inside third static block, its executed before calling the main method. But the staticMethod2() static method is executed after the class is instantiated because it is being called after the instantiation of the class.

Again if you miss to precede the block with "static" keyword, the block is called "constructor block" and will be executed when the class is instantiated. The constructor block will be copied into each constructor of the class. Say for example you have four parameterized constructors, and then four copies of constructor blocks will be placed inside the constructor, one for each. Let's execute the below example and see the output.

Example-3: Demo of constructor blocks

```
public class ConstructorBlockExample{

    {System.out.println("This is first constructor block");}

    public ConstructorBlockExample(){
        System.out.println("This is no parameter constructor");
    }

    public ConstructorBlockExample(String param1){
        System.out.println("This is single parameter
                                                constructor");
    }

    {System.out.println("This is second constructor block");}

    public static void main(String[] args){
        ConstructorBlockExample constrBlockEx =
                            new ConstructorBlockExample();
        ConstructorBlockExample constrBlockEx1 =
                            new ConstructorBlockExample("param1");
    }
}
```

Run the code and observe the output.

Now let's go back to static blocks. There is an alternative to static blocks —you can write a private static method.

```
class PrivateStaticMethodExample {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {
        //initialization code goes here
    }
}
```

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable. So get more flexibility with a private static method in comparison to the corresponding static initialization block. This should not mislead that a 'public' static method can't do the same. But, we are talking about a way of initializing a class variable and there is hardly any reason to make such a method 'public'.

Advantages of static blocks

- If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.

Disadvantages for static blocks

- There is a limitation of JVM that a static initialization block should not exceed 64K.
- You cannot throw Checked Exceptions. (will be discussed later)
- You cannot use this keyword since there is no instance.
- You shouldn't try to access super since there is no such a thing for static blocks.
- You should not return anything from this block.
- Static blocks make testing a nightmare.

Example-4: alternate use of static blocks

```
public class StaticBlock {

        static int[] values = initializeArray(10);

      private static int[] initializeArray(int N) {
      int[] arr = new int[N];
      for (int i = 0; i <arr.length; i++) {
      arr [i] = i;
          }
        }
```

```
void listValues() {
          for (int value : values) {
            System.out.println(value);
          }
        }

        public static void main(String[] args) {
      StaticBlock example = new StaticBlock();
          System.out.println("\nFirst object:");
          example.listValues();

          example = new StaticBlock();
          System.out.println("\nSecond object:");
          example.listValues();
        }

}
```

Example-5 Consider the class Circle. It has methods to compute the radius and circumference of a circle.

```
public class Circle{

    static double PI; /* variables PI is a class variable, it
                      is not instance specific */
private double radius;

    //overloaded constructor
Circle(double radius) {
        this.radius = radius;Circle.PI = 3.141;
    }

    //accessor method
public double getRadius() {return radius;}

//mutator method
public void setRadius(double radius) {this.radius = radius;}

    //method to find the area
    public double area() {return(PI * radius * radius);}

    //static method's are not instance specific
    public static void getCircumference(double radius) {
     // here radius variable is a local variable
        System.out.println("Circumference = " + 2 * PI * radius);
    }
}// End of circle
```

```
class TestCircle{
    public static void main(String args[]) {

        Circle c1 = new Circle(2.3);
        c1.area();

          // accessing static method with class name
          Circle.getCircumference(2.3);

        Circle c2 = new Circle(3.45);
          c2.area();
        // accessing static method with references is discouraged
        c2.getCircumference (3.45);

/* 1. Make the area function as static and observe the
          output
2. Remove the formal argument from getCircumference()
               method and observe the output
3. Rename static to final and observe the error(s) and
               correctthem */
    } // end of main
} // end of class
```

2. Objects as parameters to methods

When you are passing any Object as a parameter to a method then two values are passed: state of the object as well as a reference variable of some type which is pointing to that object. An object is always passed by reference with respect to its instance field values i.e. state of the object (it can be changed byusing the same reference variable).If you are changing the values of instance fields by using same reference variable that has been passed from caller method then changes will be reflected in the caller method also. This is what we mean by call by reference. Object reference itself is passed by value not by reference. The called method if tries to change the reference of passed parameter to another object ofsame type then that change remains local only and will not be reflected to caller method.
Consider the following program to calculate the distance between two points in a 2-dimensional coordinate plane. Observe how we can pass objects as parameters to the methods.

Example-6: Object as parameter to a method

```
public class Point{

private double x; // Instance field x-coordinate
private double y; // Instance field y-coordinate

Point(double x, double y){ this.x=x;this.y=y; }
```

```
public double getX() { return this.x; }
public double getY() { return this.y; }
public void setX(double x) { this.x = x; }
public void setY(double y) { this.y = y; }

public String toString() { return "X="+x+" Y ="+y; }

public static void changeState(Point other){
other.setX(-20); // Here you can write other.z = 20 also
                 // because this code is inside the class
other.setY(-20); // Here you can write other.z = 20 also
                 // because this code is inside the class
// Note that changeState Method changing the state of the Passed
    // object by using the same reference
// So this change in state will be reflected in caller method
    // also
}

public static void changeReference(Point other){
Point other = new Point(-20,-20); // Will this change be
                                  // reflected in caller method
// Note that this Method has changed the reference from
     // incoming Point object to some other Point object
// So this change will not be reflected in caller method.
}

} // End of class Point
```

```
public class PointTest{
     public static void main(String args[]){

          // Pass By Reference
          Point p1 = new Point(10,20);
          System.out,println(p1); // See the o/p for this line
          Point.changeState(p1);
          System.out,println(p1); // See the o/p for this line

          // Pass By value
          Point p2 = new Point(100,200);
          System.out,println(p2); // See the o/p for this line
          Point.changeReference(p2);
          System.out,println(p2); // See the o/p for this line

     } // End of Method main()
} // End of class PointTest
```

Example-7: Object as parameter to a method

```
public class BOX{
     private double length, width, height; // Instance Fields

     BOX(double l,double b,double h){ // Constructor Method
          length = l; width = b; height =h;
     }

     // Mutator methods for Length, Width and Height
     public void setLength(double l) { length = l;}
     public void setWidth(double b) { width = b;}
     public void setHeight(double h) { height = h;}

     // Accessor Methods for Length, Width and Height
     public double getLength() { return length;}
     public double getWidth() { return width; }
     public double getHeight() { return height;}

     public String toString() {
     return "Length:"+length+" Width:"+width+" Height:"+height;  }

     public double area() {
          return 2 * (length * width + width*height +
                                                  height*length);
     }

     public double volume() { return length*width*height;}

     public static void swapBoxes(BOX b1, BOX b2) {
          BOX temp = b1; b1 = b2; b2 = temp;
     }

}// End of BOX
```

There are two driver classes being given for the Box class. Observe the output in each case.

First case	Second Case

`public class BOXTest{`
`public static void main(String args[]){`
`BOX b1 = new BOX(10,40,60);`
`BOX b2 = new BOX(20,30,80);`
`System.out.println(b1);`
`System.out.println(b2);`
`BOX.swapBoxes(b1,b2);`
`System.out.println(b1);`
`System.out.println(b2);`
`}`
`}//End of Main`
	`class BOXtest{`
	`public static void main(String args[]){`
	`BOX b1 = new BOX(10,40,60);`
	`BOX b2 = new BOX(20,30,80);`
	`System.out.println(b1);`
	`System.out.println(b2);`
	`BOX temp = b1;`
	`b1 = b2;b2 = temp;`
	`System.out.println(b1);`
	`System.out.println(b2);`
	`}`
	`}//End of Main`

3. Wrapper Classes and Primitive Types

Java supports 8 primitive types (byte, short, int, long, boolean, float, double, char). The primitive type values are not implemented as objects. But there are certain situations when there is a need to convert a primitive type value into an object type. e.g. we know that System.out.println() statement displays the output by converting the arguments to string form. Now just think what happens for the following statement:

```
System.out.println(10);
```

Can toString() from Object class can be applied to int type. No, toString() cannot be called thru a primitive value. Rather compiler converts a primitive type value first to its corresponding wrapper type object (Integer in this case) and then invokes toString() over Integer type object. So at compile time the above statement is converted to

```
System.out.println(new Integer(10).toString());
```

Similarly there are many other situations where it is must to convert a primitive type value to a wrapper type object. The table given below shows primitive data types as well as their corresponding wrapper classes.

Primitive Types	boolean	float	double	char
Wrapper Types	Boolean	Float	Double	Character

Primitive Types	byte	short	int	long
Wrapper Types	Byte	Short	Integer	Long

Auto boxing is the process of converting a primitive type value to its corresponding wrapper type object automatically and Similarly the process of getting a primitive type value back from a wrapper type object is known as auto Unboxing. Latest java compiler supports both auto boxing / and Unboxing automatically. But earlier java versions (1.3 and earlier) they do not support these features automatically and in that version of java programmer has to perform these operations explicitly.

To know more about these operations try to compile and execute the following programs.

```
                              /* CASE-1 */
public class Test1{
     public static void main(String args[]){
          Integer I = new Integer(10);
          Integer J = new Integer(20);
          // Observe the output for following two
          // statements carefully
          System.out.println(I.intValue());
          System.out.println(I);
          // Observe the output for following two
          // statements carefully
          System.out.println(J.intValue());
          System.out.println(J);
          Integer K1 = new Integer(I.intValue()+J.intValue());
          // What happens for the following statement
          // (Auto Unboxing)
          Integer K2 = I + J + K1; System.out.println(K2);
     }
}
                              /* CASE-2 */
class Test2 {
     public static void main(String args[]){
          Integer I = 10; // what happens for this statement
          Integer J = 20; // what happens for this statement
          // observe the output for following two
          // statements carefully
          System.out.println(I.intValue());
          System.out.println(I);
          // Observe the output for following two
          // statements carefully
          System.out.println(J.intValue());
          System.out.println(J);
          Integer K1 = new Integer(I.intValue()+J.intValue());
          Integer K2 = I + J + K1;
          System.out.println(K2);
     }
}
```

Exercises:

1.

a. Correct the following code and predict the output.

```
class test {
    int x = 10;
    public static void main(String[] args)
    {
        System.out.println(x);
    }
    static
    {
    System.out.print(x + " ");
    }
}
```

static int

b. Can we declare static local variables?

```
class test {
    static int y;
    int z;
    public static void main(String args[]) {
        System.out.println(method1());
    }
    public static int method1(){
        /* Q1. Declare a static variable x and try to modify the values of y and
        z. Study the behavior of the variables x, y, and z */
        return x++;
    }
}
```

z -> static ref to non-static field

static local variable cannot be declared, use final instead

c. Correct the following code and check whether it yields the output as predicted.

Order of execution:
0) Static variables instantiated
1) Static blocks
2) Calls from static blocks
3) main
4) Everything else

Static blocks are called the first time main is run (that is class is loaded)

```
class stest {
      private static String[] Str;
      static {
            System.out.println("1.%");
      }
      public static void main(String args) {
          System.out.println("2.@");
      }
      static {
          System.out.println("3.&");
      }
      static {
          /* Q1. Invoke main method of the test class and stest class using
          appropriate syntax. For both the methods pass the variable Str as
          argument. Check whether the code compiles */
          System.out.println("4.~");
      }
      static {
          System.out.println("5.$");
      }
}
class test{
      public static void main(String[] args) {
          /* Q2. Create an object for the stest class. Check what happens
          when this statement is omitted and remaining part of the code is
          executed */
          stest.main(args);
          stest.main(new String());
          System.out.println("6.#");
      }
}
```

2. Write a Java class Complex for dealing with complex number. Your class must have the following features:

   a. Instance variables:
   realPart for the real part of type double
   imaginaryPart for imaginary part of type double.

   b. Constructor:
   public Complex (): A default constructor, it should initialize the number to 0, 0)

public Complex (double realPart, double imaginaryPart): A constructor with parameters, it creates the complex object by setting the two fields to the passed values.
public Complex(Double …c): A constructor with variable length arguments. If one argument is passed the imaginary part is 0 else its value is assigned.

c. Instance methods:
public Complex add (Complex otherNumber): This method will find the sum of the current complex number and the passed complex number. The methods returns a new Complex number which is the sum of the two.
public Complex subtract (Complex otherNumber): This method will find the difference of the current complex number and the passed complex number. The method returns a new Complex number which is the difference of the two.
public Complex multiply (Complex otherNumber): This method will find the product of the current complex number and the passed complex number. The method returns a new Complex number which is the product of the two.
public Complex divide (Complex otherNumber): This method will find the division of the current complex number and the passed complex number. The method returns a new Complex number.
public void setRealPart (double realPart): Used to set the real part of this complex number.
public void setImaginaryPart (double realPart): Used to set the imaginary part of this complex number.
public double getRealPart(): This method returns the real part of the complex number
public double getImaginaryPart(): This method returns the imaginary part of the complex number

d. Write a separate class **ComplexDemo** with a main() method and test the Complex class methods, wrapper class and variable length arguments.