BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS F213

LAB 3

AGENDA **TIME: 02 Hrs**

- The static keyword
- Method overloading and overriding
- Use of this and super keywords

3.1 The Static Keyword

The **static** keyword is used in Java for three main purposes:

1) Static variables
2) Static methods
3) Static blocks

3.1.1 Static Variables

Static variables are variables that belong to a class, and not to an object (instance of a class). They are initialized only once, when the class is loaded into the memory, before the execution of the program.

Static variables are defined by adding the **"static"** keyword before variable declaration.

```
static int x;
static int y = 10;
```

Since static variables are associated with a class, and not an object, they can be accessed directly by using the class-name, instead of requiring an object.

```
class Math {
    static double PI = 3.14;
}

class Example {
    public static void main() {
        double pi = Math.PI;
    }
}
```

Static variables are used to keep track of information that relates logically to an entire class, as opposed to information that varies from instance to instance. For example, a class representing a dialog box might use a static variable to specify the default size for new dialog box instances, or a class representing a car in a racing game might use a static variable to specify the maximum speed of all car instances.

3.1.2 Static Methods

Just like static variables, static methods are methods that belong to a class, and not to an object (instance of a class). They are defined just like static variables, by adding the **“static”** keyword before the method signature. We can again use the class name, with the method name, to call static methods.

```
class MyStatic {
    public static void staticMethod() {
        // Body of method
    }
}
class Example {
    public static void main() {
        MyStatic.staticMethod();
    }
}
```

Static methods can only access static data. This is because static methods are not connected with an instance, due to which they cannot access instance variables. Because of similar reasons, static methods can only call other static methods, or use the **“super”** and **“this”** keywords.

Program 3.1.a

```
class MyStatic {
    int a;        // initialized to 0 for every object
    static int b; // initialized to 0 only once when class loads

    // Constructor to increment b
    MyStatic() { b++; }

    public void showData() {
        System.out.println("Value of a = " + a);
        System.out.println("Value of b = " + b);
    }

    /** Comment 1
```

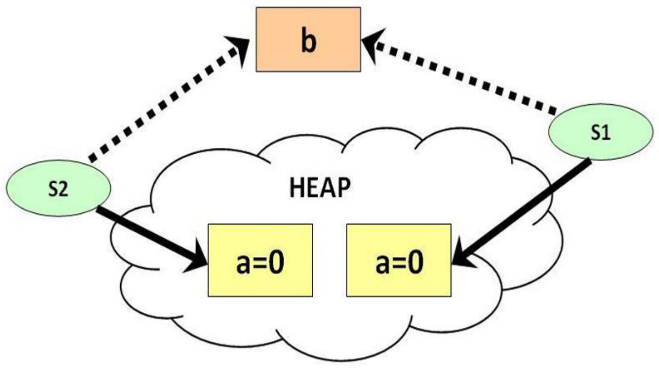
```
        public static void increment() {
            a++;
        }
    */
}


class StaticDemo {
    public static void main(String args[]) {
        MyStatic s1 = new MyStatic();
        s1.showData();
        MyStatic s2 = new MyStatic();
        s2.showData();
        /** Comment 2
            MyStatic.b++;
            s1.showData();
            s2.showData();
        */
    }
}
```

1. Run the above program, and observe the output. Can you explain why this particular output is seen?

   ```
   Expected Output:
   Value of a = 0
   Value of b = 1
   Value of a = 0
   Value of b = 2
   ```

   Since a is an instance variable, it is initialized to 0 every time a new object of the class is instantiated. On the other hand, since b is a static variable, all objects of the class share the same copy of the variable. Hence, the value of b keeps increasing with every new object created.

The following figure would help you to have a clearer understanding of the same



2. Uncomment the comment block labeled Comment 2, and then try to explain the output again.

Expected Output:

```
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
Value of a = 0
Value of b = 3
Value of a = 0
Value of b = 3
```

The output demonstrates that the static variables of the class can easily be accessed by using the syntax <className>.<varaibleName>. This also further demonstrates the point that the value of b is shared across objects, and no object reference is necessary to access static variables.

3. Now, uncomment the block labeled Comment 1, and try to explain the output this time.

Expected Output: Error

The error occurs because increment is a static method, but it is trying to modify a, which is an instance variable. Since static methods are not called via an object, the value of variable a cannot be determined, as each object would have a different value of the variable associated with it.

3.1.3 Static Blocks

Since static variables are initialized only once, when the classes are loaded into memory, what if we need to have static variables which are not constants known at the time of writing the program, but require some extra computations before they are initialized? Where should one write these calculations? For exactly this reason, Java provides a feature known as static blocks. Static blocks are simple blocks of code, written within a pair of curly braces, and preceded by the **“static”** keyword. Static blocks are executed when the class is loaded into the memory by the JVM. Static blocks are handled in the following manner:

- 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 to 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.
- 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.

Program 3.1.b

```
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");
     }
}
```

1. Run the above program, and carefully observe the output. The order of statements printed gives a useful insight about the manner in which the statements are executed. Based on the above information, can you logically explain the order of execution?

   Expected Output:

   ```
   This is first static block
   This is second static block and Static Variable
   This is static method
   This is third static block
   This is constructor
   This is static method2
   ```

   As already stated earlier, the static blocks will first get executed, while the class is being loaded. The order of execution is the same as they appear in the program, which explains the first 2 lines of the output. Since the last static block invokes staticMethod, the method gets executed first, and then the control resumes within the static block to print the 4th line of the output. At this point, all the static blocks have now been executed, and all static variables have been initialized. Now, the control moves to the main method, where the constructor is invoked when a new object is being created, and then staticMethod2 is invoked.

Limitations of static blocks:

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

3.2 Overloading and Overriding

3.2.1 Method Overloading in Java

If a class has multiple methods having the same name but different in parameters, it is known as Method Overloading. If we have to perform only one operation, having the same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

3.2.1.1 Why method overloading?

It increases the readability of the program.
An example of overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
      for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

In this overloading example, the two bark methods can be invoked by using different parameters. Compiler know that they are different because they have different method signatures (method name and method parameter list).

3.2.1.2 Different ways to overload

There are two ways to overload the method in java

1. By changing number of arguments

Program 3.2.a

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Here in the above program we have two add methods which are differentiated by the number of arguments which are 2 and 3 respectively.

2. By changing the data type

Program 3.2.b

```
static int plusMethod(int x, int y) {
  return x + y;
}

static double plusMethod(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethod(8, 5);
  double myNum2 = plusMethod(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

Here the number of arguments are the same but the type of the methods and arguments are different between the two methods.

3.2.1.3 Some more examples of overloading.

Run the programs given below and see what output you get? Try to play around with arguments and types and see what outputs and errors you get.

Program 3.2.c

```
public class Main {

    public static int foo(int a) { return 10; }
    public static char foo(int a, int b) { return 'a'; }

    public static void main(String args[])
    {
        System.out.println(foo(1));
        System.out.println(foo(1, 2));
    }

/**

Expected Output: 10
                 a
Explanation:
There are two methods in the above program which differ both in their
return type and number of arguments. So if we pass a single int then 10 is
returned and if we pass two integers  character 'a' is returned.
*/
}
```

Now try the following programs on your own and infer the output.

Program 3.2.d

```
class HelperService {
    private String formatNumber(int value) {
        return String.format("%d", value);
    }

    private String formatNumber(double value) {
        return String.format("%.3f", value);
    }

    private String formatNumber(String value) {
        return String.format("%.2f", Double.parseDouble(value));
    }
    public static void main(String[] args) {
        HelperService hs = new HelperService();
        System.out.println(hs.formatNumber(500));
        System.out.println(hs.formatNumber(89.9934));
        System.out.println(hs.formatNumber("550"));
    }
}
// Can you also try it with long and double, see what happens?
```

Program 3.2.e

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
//What would happen if we pass  an int and a double as the parameters?Can
you try that and see?
```

Program 3.2.f

```
class MethodOverloading {

    // this method accepts int
    private static void display(int a){
        System.out.println("Got Integer data.");
    }

    // this method  accepts String object
    private static void display(String a){
        System.out.println("Got String object.");
    }

    public static void main(String[] args) {
        display(1);
        display("Hello");
    }
}
//Why not try and print the input parameter with the output string?Can you
do that?
```

3.2.2 Method Overriding

If a subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent classes, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.[1]
3. There must be an IS-A relationship (inheritance).

Program 3.2.g

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
/**
Output Expected: I am a dog.

Explanation: The displayInfo method is called from the dog class and dog
class contains the overriding method of the displayInfo hence the method
from dog class is executed.
/*
```

Program 3.2.h

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
//How do you think overriding will work for a different animal and a case
in which it wont?
```

Program 3.2.i

```java
class Employee {
    public static int base = 10000;
    int salary()
    {
        return base;
    }
}

// Inherited class
class Manager extends Employee {
    // This method overrides salary() of Parent
    int salary()
    {
        return base + 20000;
    }
}

// Inherited class
class Clerk extends Employee {
    // This method overrides salary() of Parent
    int salary()
    {
        return base + 10000;
    }
}

// Driver class
class Main {
    // This method can be used to print the salary of
    // any type of employee using base class reference
    static void printSalary(Employee e)
    {
        System.out.println(e.salary());
    }
    public static void main(String[] args)
    {
        Employee obj1 = new Manager();

        // We could also get type of employee using
        // one more overridden method.loke getType()
        System.out.print("Manager's salary : ");
        printSalary(obj1);

        Employee obj2 = new Clerk();
        System.out.print("Clerk's salary : ");
        printSalary(obj2);
    }
}
//Can you do this without a static keyword and could overloading come into
picture with this?
```

Program 3.2.j

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }  }
// What more can we add to the vehicle class that will require overriding,
let say how a car and a bike work differently?
```

3.2.3 **Overriding vs. Overloading**

Here are some important facts about Overriding and Overloading:

1. The real object type in the run-time, not the reference variable's type, determines which overridden method is used at runtime. In contrast, reference type determines which overloaded method will be used at compile time.
2. Polymorphism applies to overriding, not to overloading.
3. Overriding is a run-time concept while overloading is a compile-time concept.

3.3 This and Super Keywords:

“this” keyword - refers to the currently active instance of the class you are working with. It is mainly used for the following purposes

1. To refer to current class instance variables.
2. To invoke current class method within the class (implicitly - done by the compiler)
3. "this( )" is used to involve current class constructors. We can also pass arguments in it, if we have parameterized constructors as well.
4. Can be passed as an argument to a method and constructor calls within the class in order to work with the current active instance.
5. Can be used to return the current class instance from the method. To ensure the state of the object is updated explicitly.

We are going to see some of the above-mentioned uses of "this" keyword with their implementations in the following programs.

Point to remember: "this" refers to the current executing/ active instance(object form) of the class you are dealing with.

Quick question for you!
Can we have a static block inside the class, where we could use "this" keyword? State your reasons.

Program 3.3.a

As the first example, we are trying to map a student's details to their respective campus in BITS. For the above purpose, we have taken two classes: "Campus" class and "Student" class, to illustrate the uses of "this" - to refer to current class active instance variables and also passing the active instance of the current class to other class constructors.

Before trying to copy-paste and run the program. Debug the program yourself, what output we would get, and state your observations. What is happening in line 4? After successfully executing the program, answer the questions in the comment section of the program.

Note: Name the file with the class name which has the “main( )” method and it should be public. Here, it’s “TestClass1”.

```
class Campus{
    Student s;
    String campusName;
    Campus(Student t){    //line1
        s = t;   //line2
        campusName = "Pilani";
    }
    void display(){
     System.out.println("Student Details are: \n" + s.id+ " " + s.name +
     " " +  campusName);
       //HashCode Statement
       System.out.println("From Campus class method - "+ s.hashCode());
      }
    }
 class Student{
    int id;
    String name;
    Student(int id, String name){
        this.id = id;    //line3
        this.name = name;
        Campus c = new Campus(this);  //line4
        c.display();
    }
 }
```

```java
public class TestClass1{
    public static void main(String[] args) {
     Student s2 = new Student(842325,"Surya");
    //HashCode Statement
     System.out.println("From Main Class - " +s2.name+" "+ s2.hashCode());

     Student s3 = new Student(423258,"Sundar");
    //HashCode Statement
     System.out.println("From Main Class - " +s3.name+" "+ s3.hashCode());
   }
}

/*
Hey Java Beans!
Some Quick Questions for you!
1. What happens when we change the variable name 't' to 's' in line1?
2. How can we have the same name for the Student instance variable as well
as the constructor local variable?
3. What are the outputs for HashCode Statements? Are they Different? Why
is it the way it's printed?
*/
```

What is the possible reason behind the output of “HashCode Statements”? State your reasons. For an instance variable of the class (Say s2). What are the outputs for its HashCode Statements and why is it so?

Program 3.3.b

In the next program, We will be looking at the reusing of the constructor definition using “this” keyword. As we could guess, reusing the existing definition of the constructor involves the concept of “constructor overloading”. We need to have different signatures with the same method name within a class to achieve “Overloading” w.r.t to a method.

For the following purpose,

We consider the below class, which has overloaded constructors which differ by no. of arguments. As we can see in line1, It is essentially using the existing constructor definition for the purpose of populating the data for an object.

The statement `this(id,name,course);` refers to the constructor `Student(int id ,String name,String course)` line2 leads to the invocation of the constructor in line1.

```
class Student{
    int id;
    String name, course;
    float fee;

    Student(int id ,String name,String course){  //line1
        this.id  = id;
        this.name=name;
        this.course = course;
    }

    Student(int id,String name,String course,float fee){
        //comment1
        this(id,name,course); //line2
        this.fee = fee; //line3
    }

    void display(){
        System.out.println(id+" "+name+" "+course+" "+fee);
     }
}

class TestClass2{
  public static void main(String args[]){
        Student s1=new Student(79,"ankit","OOP");
        Student s2=new Student(80,"surya","Java",6000f);
        s1.display();
        s2.display();
  }
}

/*
Hello Again!
Gist out your learnings about reusing the constructor. Meanwhile, I have a
small doubt!
Could you see lines number 2 and 3? What if we swap those statement
```

```
positions?
*/
```

What happens, when we swap or interchange line2 and line3 statements? Would the compiler allow it to execute? Try to comprehend it.

Hint:- Imagine any operation on the variables like id, name at the “comment1” statement. Having a "**this(..)**"call inside the constructor, our compiler will not allow any other statement to be there before it or as the first line of the constructor preserving constructor chaining.

“super” keyword - The super keyword in Java is a reference variable which is used to refer to a direct parent class instance for a given class. Whenever you create the instance of a subclass, an instance of the parent class is created implicitly which is referred by “**super**” reference variable.

Usage of Java super Keyword:

1. can be used to refer to immediate parent class instance variables.
2. can be used to invoke the immediate parent class method.
3. “super( )” can be used to invoke immediate parent class constructor. We can also pass arguments in it if we have parameterized constructors in the parent class as well.

Program 3.3.c

The Following Program explains the use of super( ), which can be explicitly invoked from the child class constructor and also to refer to the immediate parent class instance variable. We have Vehicle and Bike class, having the same instance variables in them (int wheels;). Vehicle class instance variable “wheels” can be accessed in Bike class using “**super**”.

```
import java.util.Scanner;
class Vehicle{
    int wheels;
    Vehicle(){
        System.out.println("Vehicle is created" );
    }
}

class Bike extends Vehicle{
    int wheels;
    double price;
    Bike(double price){
        super(); //line1
        this.price = price; //line2
        System.out.println("Bike is created");
        details(this);
    }
    void details(Bike b1){
        b1.wheels = 2;
        super.wheels = b1.wheels; //line3
        System.out.print("No of Wheels for this vehicle is "+
super.wheels);
        System.out.println(" and Price is "+ b1.price);
    }
}

class TestClass3{
    public static void main(String args[]){
            Scanner sc = new Scanner(System.in);
            double price = sc.nextDouble();
            Bike b = new Bike(price);
      }
}

/*

1. We can have a doubt about the output right? If not, please explain why
is it that way?
2. What if we remove the "line1" statement, still the output would be the
same?
3. Can we swap "line1" and "line2"? Why not?
4. Play with line3 statement. Remove the "super" keyword in front of it.
See what happens? Change to b1.wheels to wheels, do you see any change? Why
not?

*/
```

Note: State your reasons for the above questions in the comment question.

Program 3.3.d

Hey, I want to generate our BITS Student ID on the basis of the student data like year, branchCode, and rollno. Would there be any problem with the below code? For simplicity purposes, consider the year, rollno as strings. Any problem at line1?

```
class IDGen{
    static int num;
    String id;
    String campus;
    IDGen(String s){
        num++;
        id = s;
        campus="P";
    }
    void display(){
        System.out.println("Student"+ num+" ID is "+ id+campus);
    }
}

class Details extends IDGen{
    public Details(String year, String branchCode, String rollno){
        //Procedure to generate student ID
        String id = year + branchCode + rollno; //Line1
        super(id);
        display();
    }
}
public class TestClass4 {
    public static void main(String[] args) {
       Details d = new Details("2020","H103","0142");
       new Details("2018","H103","0198");
    }
}

/*
```

```
Expected output:
Student1 ID is 2020H1030142P
Student2 ID is 2018H1030198P

Try to run the above code yourself, If it is printing out errors, state
your reasons.
*/
```

The reason why we are getting errors at line1 is that, if we have any **super( )** call in our constructor, the compiler won't allow us to write any other statement as the first line of the constructor definition. This ensures that the state of the parent is created before the child. Observe program 3.3.e, which is a way to deal with this issue.

Program 3.3.e

```
class IDGen{
    String id;
    String campus;
    IDGen(String s){
        campus="P";
        id = s + campus;
    }
    void display(){
        System.out.println("Student ID is "+ id);
    }
}
class Details extends IDGen {
    public Details(String year, String branchCode, String rollno){ //line1
        this(year + branchCode + rollno);
    }

    private Details(String id) { //line2
        super(id); //line3
        display();
    }
}
public class TestClass5 {
    public static void main(String[] args) {
       Details d = new Details("2020","H103","0142");
    }
}
/*
1. Observe the line numbers from 1 to 3, What are you thinking about them?
Note down the reasons behind those changes from program 4?
*/
```

Program 3.3.f

Extension of Program 3.3.e - we introduced the concepts of overriding with the help of method “display( )” as well as using the “static” keyword. Observe how the output is generated?

```
class IDGen{
    static int num; //line1
    String id;
    String campus;
    IDGen(String s){
        num++; //line2
        campus="P";
        id = s + campus;
    }
    void display(){  //line3
        System.out.println("Student"+ num+" ID is "+ id);
    }
}
class Details extends IDGen {
    String year,branchCode,rollno;
    public Details(String year, String branchCode, String rollno){
        this(year + branchCode + rollno);
        this.year = year;
        this.branchCode = branchCode;
        this.rollno= rollno;
        display();
    }
```

```java
    private Details(String id) {
        super(id);
        super.display(); //line4
    }
   void display(){
    System.out.println("Year -  " +year+ " BranchCode - "+branchCode+ "
Rollno - " +rollno);
   }

}
public class TestClass5 {
    public static void main(String[] args) {
       Details d = new Details("2020","H103","0142");
       new Details("2018","H103","0198"); //line5
    }
}

/*
Hey JavaBeans!
1. Observe the line numbers from 1 to 5, What are you thinking about them?
What are your thoughts about line 4?
2. Explaining the working of line1 and line2 with the help of a diagram.
3. What are your thoughts about line 5?
*/
```