

## **Session 2- Facilitation Guide**

### **Index**

- I. Recap of previous session**
- II. Using break and continue**
- III. Unlabeled Statements**
- IV. Labeled Statements**
- V. Passing Variables into Methods**
- VI. Passing Object Reference Variables**
- VII. Does Java Use Pass-By-Value Semantics?**
- VIII. Initialization Blocks**

### **For (2 hrs) ILT**

#### **I. Recap from the previous session**

In our last session we discussed various types of flow controls in Java. We learnt about if-else, switch Statements and loops. We understood the difference between while loops and for loops. When we know the number of times a code block will be executed we use a for loop. But when we have no idea about the exact number of times execution will be repeated it is better to use while or do loop. However, a programmer should decide what flow control statements should be best fitted in the application so that code is optimized, reusable as well as readable.

In addition to flow controls we have also learnt how to provide inputs to a Java program using Console class, Scanner class and BufferedReader class. Java offers several methods for user input. It is better to leave the choice of which method is suitable for a particular application to the programmer.

#### **II. Using break and continue**

In Java, the break and continue statements are control flow statements used to modify the normal flow of execution within loops (such as for, while, and do-while loops) and

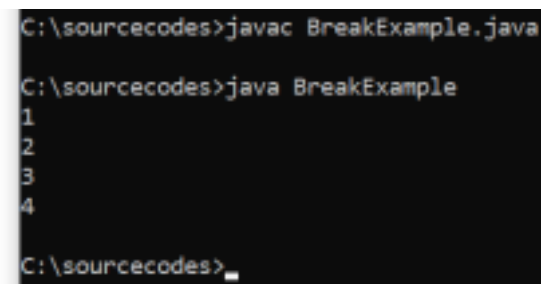
switch statements. They allow us to control when the loop terminates or skips certain iterations based on specific conditions.

Actually, break and continue are jump statements that ignore specific statements within the loop or abruptly terminate the loop without executing the test expression.

Following is the Java program that demonstrates the use of break statements inside a While loop.

```
Java
public class BreakExample {
public static void main(String[] args) {
    //initiating while loop
    int a=1;
    while(a<=10){
        if(a==5){
            //using break statement
            a++;
            break;//it will break the while loop
        }
        System.out.println(a);
        a++;
    }
}
}
```

### Output:



```
C:\sourcecodes>javac BreakExample.java
C:\sourcecodes>java BreakExample
1
2
3
4
C:\sourcecodes>_
```

Here initially the while loop was designed to be executed for 10 times. But the loop was terminated before the 5th time using a break statement. The break statement is useful if there is a need to terminate the loop when a certain item is found or an error occurs or the user wants to terminate the program.

For example, let us consider a simple example. Suppose, in a menu-driven program, the

user is provided with multiple menu items. The users might choose options from a menu until they decide to exit the menu. In this case, the break statement can be used to exit the loop when the user selects the exit option.

See this example for better understanding of the above scenario:

Java

```
import java.util.Scanner;
public class MenuExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int choice;

        while (true) {
            System.out.println("1. Option A");
            System.out.println("2. Option B");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    // Code for Option A
                    break;
                case 2:
                    // Code for Option B
                    break;
                case 3:
                    System.out.println("Exiting the menu.");
                    scanner.close();
                    return; // Exit the loop and the program
                default:
                    System.out.println("Invalid choice. Try again.");
            }
        }
    }
}
```

**Output:**

```
C:\sourcecodes>java MenuExample
1. Option A
2. Option B
3. Exit
Enter your choice: 1
1. Option A
2. Option B
3. Exit
Enter your choice: 2
1. Option A
2. Option B
3. Exit
Enter your choice: 3
Exiting the menu.
C:\sourcecodes>
```

In this example, the break statement is used to exit the while loop when the user selects "Exit" from the menu.

In Java, statements can be categorized as unlabeled statements and labeled statements, depending on whether they have a label associated with them or not.

### III. Unlabeled Statements

Unlabeled statements are regular statements that do not have any special label associated with them. They are the most common type of statements used in Java and are used for normal program flow control.

Example of an unlabeled statement:

```
Java
...
int x = 10;
int y = 20;
int sum = x + y; // This is an unlabeled statement
...
```

### IV. Labeled Statements

Labeled statements, on the other hand, are statements that have a label identifier

followed by a colon (:) before the actual statement. Labels are user-defined identifiers, and they provide a way to name a specific statement within a Java program. Labeled statements are often used in conjunction with break and continue statements to control the flow of nested loops or to break out of multiple nested loops.

Example of a labeled statement:

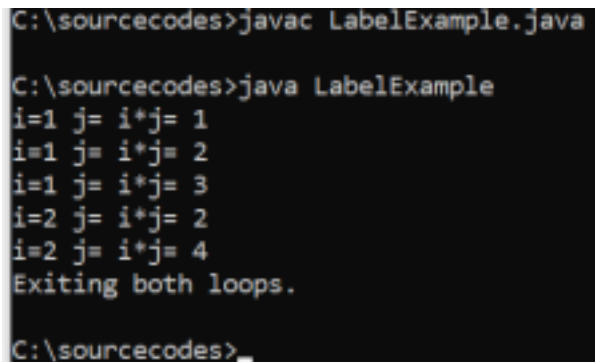
Java

```
public class LabelExample {

    public static void main(String[] args) {

        outerLoop: // This is a labeled statement
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i * j > 5) {
                    System.out.println("Exiting both loops.");
                    break outerLoop; // Using the labeled statement with break
to exit both loops
                }
                System.out.println("i=" + i + " j=" + " i*j= " + i * j);
            }
        }
    }
}
```

**Output:**



```
C:\sourcecodes>javac LabelExample.java

C:\sourcecodes>java LabelExample
i=1 j= i*j= 1
i=1 j= i*j= 2
i=1 j= i*j= 3
i=2 j= i*j= 2
i=2 j= i*j= 4
Exiting both loops.

C:\sourcecodes>
```

In the above example, we have a labeled statement outerLoop before the outer for loop. When the condition  $i * j > 5$  is met, the break outerLoop; statement is executed,

causing the program to exit both the inner and outer loops simultaneously.

Labeled statements provide a way to reference specific points within nested loops and improve control over flow control in complex looping scenarios. However, it's essential to use labeled statements judiciously, as excessive use can make code harder to read and maintain.

## V. Passing Variables into Methods

In previous sessions we have discussed how to call a method in Java. Each method has its name. You can pass data into a method via parameters. A method also has a return type defining the type of data it returns. Each programming language has its own method for passing the variables into the method. There are two ways of passing the variables:

- **Pass by value:** The method parameter values are copied to another variable and then the copied object is passed to the method. The method uses the copy.
- **Pass by reference:** An alias or reference to the actual parameter is passed to the method. The method accesses the actual parameter.

Pass by reference refers to the passing of a reference or pointer to a variable and pass by value refers to the passing a copy of that variable into the method. C language supports both the ways of passing. The Pass By reference is supported by use of pointers in C language. Every Java beginner wonders whether Java is a Pass By Value or a Pass By reference or “both?” Before answering that question, we would like to explain the difference between Passing Object Reference variable and Passing primitive variable. We will also provide some examples to explain how Java treats the reference variable when passed as an object to a method parameter.

### Passing Primitive Variables

The fundamental concepts in any programming language are “values” and “references”. In Java, **Primitive variables store the actual values, whereas Non-Primitives store the reference variables which point to the addresses of the objects they're referring to.** Both values and references are stored in the stack memory.

Arguments in Java are always passed-by-value. During method invocation, a copy of each argument, whether its a value or reference, is created in stack memory which is then passed to the method.

In case of primitives, the value is simply copied inside stack memory which is then passed to the callee method; in case of non-primitives, a reference in stack memory points to the actual data which resides in the heap. When we pass an object, the reference in stack memory is copied and the new reference is passed to the method. We will explain this concept with more examples in this session.

## **Passing Primitive Types**

The Java Programming Language features eight primitive data types. Primitive variables are directly stored in stack memory. Whenever any variable of primitive data type is passed as an argument, the actual parameters are copied to formal arguments and these formal arguments accumulate their own space in stack memory.

The lifespan of these formal parameters lasts only as long as that method is running, and upon returning, these formal arguments are cleared away from the stack and are discarded.

```
Java
public class PassingParamTest{

    public static void main(String[] args) {

        int x = 1;
        int y = 2;

        // Before Modification
        System.out.println(x+" "+y); //1 2

        modify(x, y);

        // After Modification
        System.out.println(x+" "+y); //1 2
    }

    public static void modify(int x1, int y1) {
        x1 = 5;
        y1 = 10;
    }
}
```

## Output:

```
C:\sourcecodes>javac PassingParamTest.java

C:\sourcecodes>java PassingParamTest
1 2
1 2

C:\sourcecodes>_
```

Let's try to understand the assertions in the above program by analyzing how these values are stored in memory:

The variables “x” and “y” in the main method are primitive types and their values are directly stored in the stack memory.

When we call method `modify()`, an exact copy for each of these variables is created and stored at a different location in the stack memory.

Any modification to these copies affects only them and leaves the original variables unaltered.

Initial Stack space

x = 1
y = 2

Stack space when  
*modify()* method called

x = 1
y = 2
x1 = 1
y1 = 2

Stack space after  
*modify()* method call

x = 1
y = 2
x1 = 5
y1 = 10

## VI. Passing Object Reference Variables

In Java, object references are always passed by value. Even though a variable may contain a reference to the object, that reference is a pointer to the memory location of the object. Therefore, object references are also passed by value in Java. There is



confusion about how Java Objects are passed to methods. The reason for the confusion between these terms is often due to the concept of object references in Java. Both reference data types and primitive data types are passed by value but for the object the value of the reference of pointer is passed and that reference can not be modified. It sounds complicated but concepts will be clear in the following examples.

Let us consider a simple Balloon class which has one attribute color of type String.

Java

```
public class Balloon {
    private String color;
    public Balloon() {}
    public Balloon(String c) {
        this.color = c;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}
```

Now we will pass the instance of Balloon class to methods and will inspect the behavior.

Java

```
public class BalloonTest {
    public static void main(String[] args) {
        Balloon red = new Balloon("Red"); // Let us assume this object has
        // a memory reference = 50.
        Balloon blue = new Balloon("Blue"); // memory reference = 100

        System.out.println("Initial Values of colors" );
        System.out.println(" Balloon with `red` color value = " +
red.getColor());
        System.out.println(" Balloon with `blue` color value = " +
blue.getColor());

        swap(red, blue); // swapping the balloon objects
        System.out.println("After the swap method executes:");
    }
}
```

```

        System.out.println(" Balloon with `red` color value = " +
red.getColor());
        System.out.println(" Balloon with `blue` color value = " +
blue.getColor());

        changeValue(blue);
        System.out.println("After the changeValue method executes:");
        System.out.println(" Balloon with `blue` color value = " +
blue.getColor());

    }
    /*
    * Generic swap method.
    * The references of the two objects are swapped.
    */
    public static void swap(Object o1, Object o2){
        Object temp = o1;
        o1 = o2;
        o2 = temp; // Reference are changed but it has no effect on
        // the original reference passed.
    }
    /*
    *
    */
    private static void changeValue(Balloon balloon) { // balloon = 100
        balloon.setColor("Red"); // This the balloon with reference 100
        balloon = new Balloon("Orange"); // A new balloon with reference
200

        balloon.setColor("Pink"); // Working on balloon with reference 200
    }
}

```

**Output:**

```
C:\sourcecodes>javac Balloon.java
C:\sourcecodes>javac BalloonTest.java
C:\sourcecodes>java BalloonTest
Initial Values of colors
  Balloon with `red` color value = Red
  Balloon with `blue` color value = Blue
After the swap method executes:
  Balloon with `red` color value = Red
  Balloon with `blue` color value = Blue
After the changeValue method executes:
  Balloon with `blue` color value = Red
```

So when we use the swap method we can see that there is no change in the actual object after the method is called. However, we will notice that there is a change in the value of color after the changeValue method is called. The reason behind this is that Java passes the reference of the object as a value. That means this reference value can not be changed by calling the method. But we can modify the object's property using setter methods.

## VII. Does Java use Pass-By-Value semantics?

In the previous sections we have discussed how to call a method in Java and pass parameters. We can pass data into a method via parameters. Each programming language has its own method for passing the variables into the method. We already discussed that there are two ways of passing the variables - pass by value and pass by reference. Pass by reference refers to the passing of a reference or pointer to a variable and pass by value refers to the passing a copy of that variable into the method.

**Note:** C and C++ support both the ways of passing. The Pass By reference is supported with the use of pointers.

When you pass a primitive data type (like int, float, char, etc.) or an object reference as

an argument to a method, Java makes a copy of the value of the argument and passes that copy to the method. This is known as pass-by-value.

In the case of primitive data types, the value itself is passed to the method. Any changes made to the parameter inside the method do not affect the original value outside the method.

We have explained in the previous section that in the case of objects, the object reference (memory address) is passed by value. This means a copy of the reference is passed to the method, not a copy of the actual object. Both the original reference and the method parameter will point to the same object in memory. If the method modifies the object's state (i.e., the content of the object), those changes will be visible outside the method since both the original reference and the method parameter are pointing to the same object.

However, if the method reassigns the reference to a new object, that change will not affect the original reference outside the method because only the copy of the reference was modified within the method's scope. We clearly explained this concept in our previous section. We have seen in the previous examples that using the swap method we are not able to actually swap the objects because actual reference can not be altered in Java while passing objects as parameters.

In summary, Java uses pass-by-value semantics, whether the parameter is a primitive type or an object reference. For primitives, it passes the actual value, and for objects, it passes a copy of the reference to the object.

## **VIII. Initialization Blocks**

Initialization blocks in Java are used to execute code that initializes instance variables or performs common initialization tasks before the constructors are called. There are two types of initialization blocks in Java: instance initialization blocks and static initialization blocks.

## Instance Initialization Blocks:

Instance initialization blocks are used to initialize instance variables of an object. They are executed whenever a new object of the class is created, before the constructor is called.

Example:

Java

```
public class MyClass {
    private int x;
    private int y;

    // Instance initialization block
    {
        x = 10;
        y = 20;
        System.out.println("Instance initialization block executed.");
    }

    // Constructor
    public MyClass() {
        System.out.println("Constructor executed.");
    }

    public void display() {
        System.out.println("x: " + x + ", y: " + y);
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

**Output:**

```
C:\sourcecodes>javac MyClass.java

C:\sourcecodes>java MyClass
Instance initialization block executed.
Constructor executed.
x: 10, y: 20

C:\sourcecodes>
```

In this example, the instance initialization block sets the values of the instance variables `x` and `y`. It gets executed when an object of the class is created, before the constructor is called.

### Static Initialization Blocks:

Static initialization blocks are used to initialize static variables of a class. They are executed only once when the class is loaded into the memory, before any static method or constructor is called.

Example:

Java

```
public class StaticInitializationBlockTest {
    private static int count;

    // Static initialization block
    static {
        count = 0;
        System.out.println("Static initialization block executed.");
    }

    // Constructor
    public StaticInitializationBlockTest () {
        count++;
        System.out.println("Constructor executed.");
    }

    public static void displayCount() {
        System.out.println("Count: " + count);
    }
}
```

```

    }

    public static void main(String[] args) {
        StaticInitializationBlockTest obj1 = new StaticInitializationBlockTest
    };
        StaticInitializationBlockTest obj2 = new StaticInitializationBlockTest
    };
        StaticInitializationBlockTest.displayCount();
    }
}

```

## Output:

```

C:\sourcecodes>javac StaticInitializationBlockTest.java

C:\sourcecodes>java StaticInitializationBlockTest
Static initialization block executed.
Constructor executed.
Constructor executed.
Count: 2

C:\sourcecodes>_

```

In this example, the static initialization block initializes the static variable count. It gets executed only once when the class is loaded, before any object is created.

Initialization blocks provide a convenient way to perform common initialization tasks for objects or classes. They are executed automatically at appropriate times, making the code more organized and maintainable.

## Differences Between Static and Instance Initializer Block

Static Block	Instance Initializer Block
It executes during class loading	It executes during class instantiation
It can only use static variables	It can use static or non-static (instance variables).
It can not use this keyword	It can use this keyword

It executes only once during the entire  
execution of the program when the class  
loads into the memory

It can run many times whenever there is a  
call to the constructor

***After the 2 hr ILT, the student has to do a 1 hour live lab. Please refer to the Session 2 Lab doc in the LMS.***