

# INTRODUCTION TO JAVA PROG SESSION 1

## JAVA "HELLO, WORLD!" PROGRAM

```
// Our First Program
```

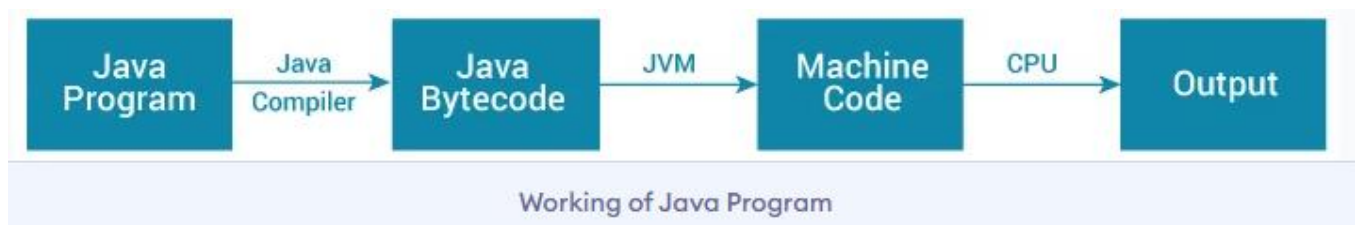
```
class HelloISBAT {  
    public static void main(String[ ] args) {  
        System.out.println("Hello, ISBAT UNIVERSITY!");  
    }  
}
```

### NOTE:

- Every valid Java Application must have a **class definition** that matches the filename (**class name** and **file name** should be same).
- The main method must be inside the class definition.
- The compiler executes the codes starting from the main function.

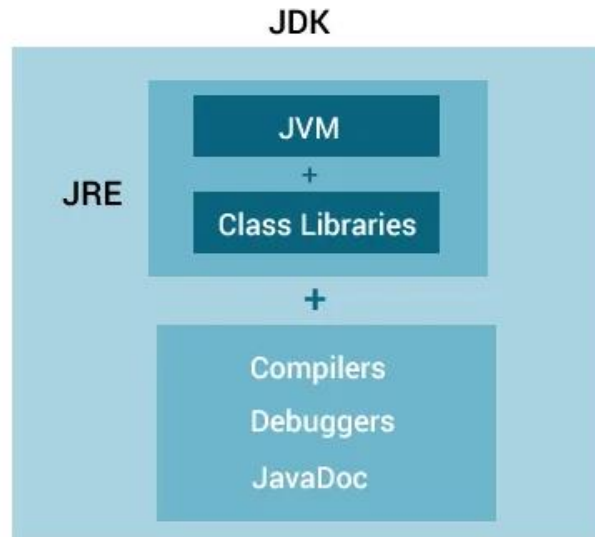
## Java JDK, JRE and JVM

**What is JVM?** JVM, **Java Virtual Machine**, is an abstract machine that enables your computer to run a Java program. When you run the **Java program**, Java compiler first compiles your **Java code** to **bytecode**. Then, the JVM translates your **bytecode** into native **machine code** (set of instructions that your computer's **microprocessor** or **CPU** executes directly). **Java** is a platform-independent language because when you write Java code, it's ultimately written for the **JVM** but not **your physical machine** (computer). JVM executes the **Java bytecode** which is platform-independent so this is what makes Java to become platform-independent.



**What is JRE?** **Java Runtime Environment** is a software package that provides **Java class libraries**, a **Java Virtual Machine (JVM)**, and **other components** that are required to run Java applications.

**What is JDK?** **Java Development Kit** is a software development kit that is needed while developing applications in Java. When you download a **JDK**, a **JRE** is also downloaded with it. In addition to JRE, JDK also contains a number of development tools e.g. (compilers, JavaDoc, Java Debugger, etc).



## JAVA VARIABLES

A **variable** is a storage location area in a computer's main memory that holds data. To indicate the storage area, each **variable** must be given a **unique name (identifier)**.

### How to Create Variables in Java

```
int scoredMark = 80;
```

Here, scoredMark is a variable of int data type and we have assigned value 80 to it. The int data type suggests that the variable can only hold integers.

You can also declare variables and assign their variables separately. For example,

```
int scoredMark;  
scoredMark = 80;
```

Note: Java is a statically-typed language which means that all variables must be declared before they can be used.

## RULES FOR NAMING VARIABLES IN JAVA

Java programming language has its own set of rules and conventions for naming variables.

- Java is **case sensitive**. Hence, **age** and **AGE** are two different variables. For example,

```
int age = 24;  
int AGE = 25;  
System.out.println(age); // prints 24  
System.out.println(AGE); // prints 25
```

- Variables must start with either a **letter** or an **underscore**, **\_** or a **dollar**, **\$** sign. E.g.,  
`int age;` // valid name and good practice

```
int _age; // valid but a bad practice
```

```
int $age; // valid but a bad practice
```

- Variable names **cannot start with numbers**. For example,  

```
int 1age; // invalid variables
```
- Variable names can't **use whitespace**. For example,  

```
int my age; // invalid variables
```

## JAVA DATA TYPES (PRIMITIVE)

**Data types** specify the type of data that can be stored inside variables in Java. Java is a statically-typed language. This means that all variables must be declared before they can be used.

```
int speed;
```

Here, **speed** is a **variable name**, and the **data type** of the variable is **int**. The **int data type** determines that the **speed variable** can only contain integers. There are **8 data types** predefined in Java, known as **primitive data types**.

## EIGHT (8) PRIMITIVE DATA TYPES IN JAVA

1. **boolean type**: A boolean data type has two possible values, either **true** or **false**.

Default value: false. They are usually used for **true/false** conditions.

### Example 1: Java boolean data type

```
class Main {  
    public static void main(String[ ] args) {  
  
        boolean flag = true;  
        System.out.println(flag); // prints true  
    }  
}
```

2. **byte type**: The byte data type can have values from -128 to 127 (8-bit signed two's complement integer). If it's a value of a variable will be within -128 to 127, then it is used instead of int to save memory. Default value: 0

### Example 2: Java byte data type

```
class Main {  
    public static void main(String[ ] args) {  
  
        byte range;  
        range = 124;  
        System.out.println(range); // prints 124  
    }  
}
```

3. **short type**: The short data type in Java can have values from -32768 to 32767 (16-bit signed two's

complement integer). If a value of a variable will be within -32768 and 32767, then it is used instead of the other integer data types (int, long). Default value: 0

### Example 3: Java short data type

```
class Main {  
    public static void main(String[] args) {  
  
        short temperature;  
        temperature = -200;  
        System.out.println(temperature); // prints -200  
    }  
}
```

4. **int type:** The int data type can have values from -2<sup>31</sup> to 2<sup>31</sup>-1 (32-bit signed two's complement integer). If you are using Java 8 or later, you can use an **unsigned 32-bit integer**. This will have a minimum value of 0 and a maximum value of 2<sup>32</sup>-1. Default value: 0

### Example 4: Java int data type

```
class Main {  
    public static void main(String[] args) {  
  
        int range = -4250000;  
        System.out.println(range); // print -4250000  
    }  
}
```

5. **long type:** The long data type can have values from -2<sup>63</sup> to 2<sup>63</sup>-1 (64-bit signed two's complement integer). If you are using Java 8 or later, you can use an **unsigned 64-bit integer** with a minimum value of 0 and a maximum value of 2<sup>64</sup>-1. Default value: 0

### Example 5: Java long data type

```
class LongExample {  
    public static void main(String[] args) {  
  
        long range = -423322000000L;  
        System.out.println(range); // prints -423322000000  
    }  
}
```

6. **double type:** The double data type is a double-precision 64-bit floating-point. It should never be used for precise values such as currency. Default value: 0.0 (0.0d)

### Example 6: Java double data type

```
class Main {  
    public static void main(String[] args) {  
  
        double number = -42.3;  
        System.out.println(number); // prints -42.3  
    }  
}
```

7. **float type:** The float data type is a single-precision 32-bit floating-point. Learn more about single-

precision and double-precision floating-point if you are interested. It should never be used for precise values such as currency. Default value: 0.0 (0.0f)

### Example 7: Java float data type

```
class Main {  
    public static void main(String[ ] args) {  
  
        float number = -42.3f;  
        System.out.println(number); // prints -42.3  
    }  
}
```

Notice that we have used -42.3f instead of -42.3 in the above program. It's because -42.3 is a double literal. To tell the compiler to treat -42.3 as float rather than double, you need to use f or F.

8. **char type:** It's a 16-bit Unicode character. The minimum value of the char data type is '\u0000' (0) and the maximum value of the is '\uffff'. Default value: '\u0000'

### Example 8: Java char data type

```
class Main {  
    public static void main(String[] args) {  
  
        char letter = '\u0051';  
        System.out.println(letter); // prints Q  
    }  
}
```

**String type:** Java also provides support for character strings via java.lang.String class. Strings in Java are not primitive types. Instead, they are objects. For example,

**String myString = "Java Programming";**  
**Here, myString is an object of the String class.**

### Example 1: Java boolean data type

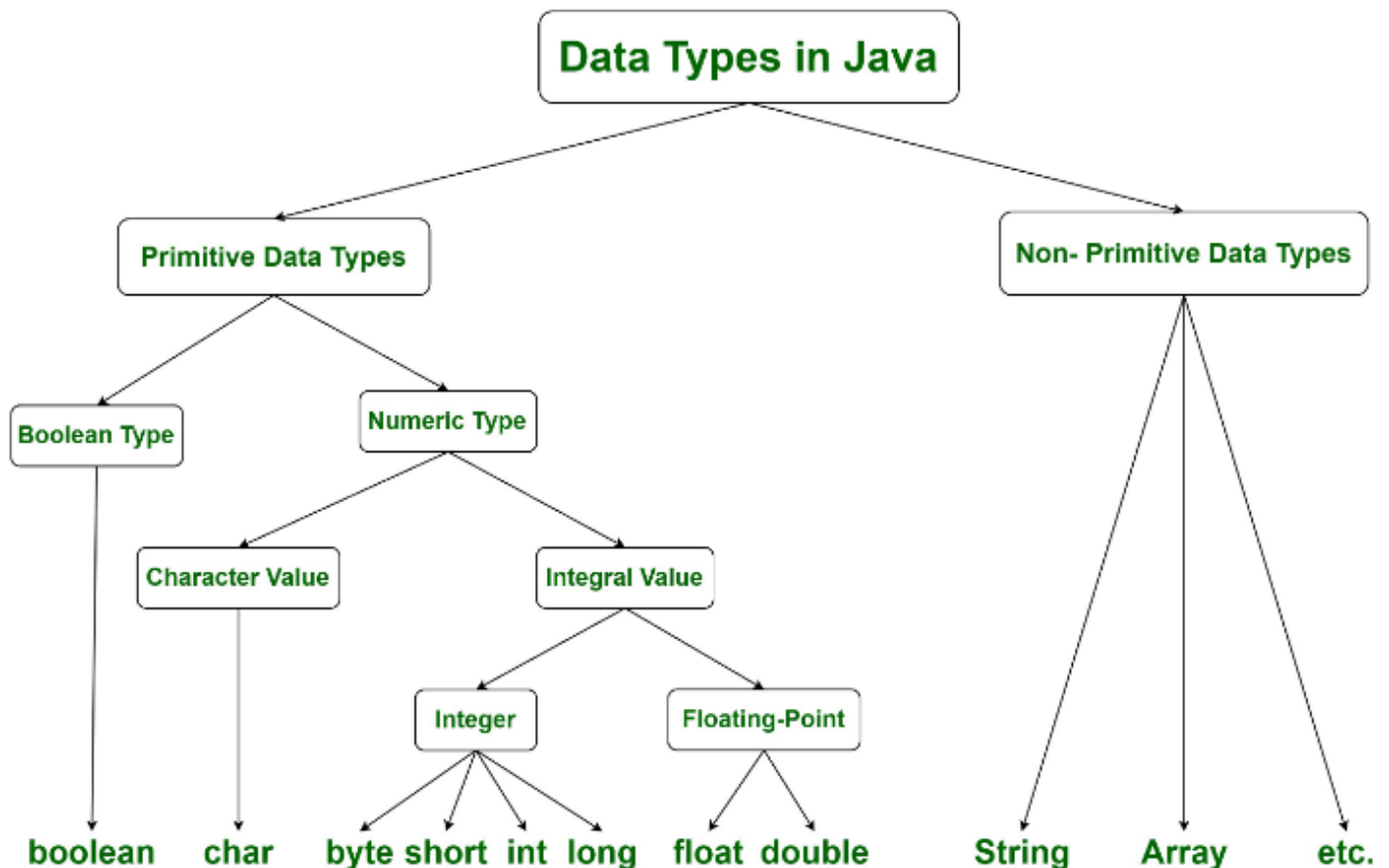
```
class Main {  
    public static void main(String[ ] args) {  
  
        boolean flag = true;  
        byte range = 124;  
        short temperature = -200;  
        double number = -42.3;  
        float number = -42.3f;  
        char letter = '\u0051';  
        System.out.println(letter); // prints Q  
        System.out.println(number); // prints -42.3  
        System.out.println(number); // prints -42.3  
        System.out.println(range); // prints -42332200000  
        System.out.println(range); // print -4250000  
    }  
}
```

```

System.out.println(temperature); // prints -200
System.out.println(flag); // prints true
System.out.println(range); // prints 124
}

```

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values



# JAVA OPERATORS

**Operators** are symbols that perform arithmetic operations on variables and values. E.g., + is an operator used for addition, while \* is also an operator used for multiplication.

Operators in Java can be classified into 5 types:

1. Arithmetic Operators
3. Relational Operators
5. Unary Operators

2. Assignment Operators
4. Logical Operators
6. Bitwise Operators

## 1. JAVA ARITHMETIC OPERATORS:

**Arithmetic operators** are used to perform arithmetic operations on variables and data.

For example, `a + b`; Here, the **+** operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

```
class Main {  
    public static void main(String[] args) {  
  
        // declare variables  
        int a = 12, b = 5;  
  
        // addition operator  
        System.out.println("a + b = " + (a + b));  
  
        // subtraction operator  
        System.out.println("a - b = " + (a - b));  
  
        // multiplication operator  
        System.out.println("a * b = " + (a * b));  
    }  
}
```

```

// division operator
System.out.println("a / b = " + (a / b));

// modulo operator
System.out.println("a % b = " + (a % b));
}
}

```

In the above example, we have used +, -, and \* operators to compute addition, subtraction, and **multiplication** operations. / Division Operator, Note the operation, **a / b** in our program. The / **operator** is the **division operator**. If we use the division operator with two integers, then the resulting **quotient** will also be an integer. And, if one of the operands is a floating-point number, we will get the result will also be in floating-point.

In Java,

(9 / 2) is 4
(9.0 / 2) is 4.5
(9 / 2.0) is 4.5
(9.0 / 2.0) is 4.5

**% Modulus Operator:** The **modulo operator %** computes the remainder. When a = 7 is divided by b = 4, the remainder is 3. Note: The % **operator is mainly used with integers**.

## 2. JAVA ASSIGNMENT OPERATORS:

**Assignment operators** are used in Java to assign values to variables. For example,

```

int age;
age = 5;

```

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, 5 is assigned to the variable age. Other examples of assignment operators available in Java include:



Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

```

class Main {
    public static void main(String[] args) {
        // create variables
        int a = 4;
        int var;

        // assign value using =
        var = a;
        System.out.println("var using =: " + var);
        // assign value using +=
        var += a;
        System.out.println("var using +=: " + var);
        // assign value using *=
        var *= a;
        System.out.println("var using *=: " + var);
        var /= a;
        System.out.println("var is using /=: " + var);
        var %= a;
        System.out.println("var is using %=: " + var);
    }
}

```

### 3.JAVA RELATIONAL OPERATORS

**Relational operators** are used to check the relationship between two **operands**. For example, // check if a is less than b

**a < b;**

Here, < operator is the relational operator. It checks if **a** is less than b or not. It returns either true or false.

Operator	Description	Example
<code>==</code>	Is Equal To	<code>3 == 5</code> returns <b>false</b>
<code>!=</code>	Not Equal To	<code>3 != 5</code> returns <b>true</b>
<code>&gt;</code>	Greater Than	<code>3 &gt; 5</code> returns <b>false</b>
<code>&lt;</code>	Less Than	<code>3 &lt; 5</code> returns <b>true</b>
<code>&gt;=</code>	Greater Than or Equal To	<code>3 &gt;= 5</code> returns <b>false</b>
<code>&lt;=</code>	Less Than or Equal To	<code>3 &lt;= 5</code> returns <b>true</b>

```
class Main {  
    public static void main(String[] args) {  
        // create variables  
        int a = 7, b = 11;  
        // value of a and b  
        System.out.println("a is " + a + " and b is " + b);  
        // == operator  
        System.out.println(a == b); // false  
        // != operator  
        System.out.println(a != b); // true  
        // > operator  
        System.out.println(a > b); // false  
        // < operator  
        System.out.println(a < b); // true  
        // >= operator  
        System.out.println(a >= b); // false  
        // <= operator  
        System.out.println(a <= b); // true  
    }  
}
```

## 4. JAVA LOGICAL OPERATORS:

**Logical operators** are used to check whether a Java expression is **true** or **false**. They are used in decision making.

Operator	Example	Meaning
<code>&amp;&amp;</code> (Logical AND)	<code>expression1 &amp;&amp; expression2</code>	<code>true</code> only if both <code>expression1</code> and <code>expression2</code> are <code>true</code>
<code>  </code> (Logical OR)	<code>expression1    expression2</code>	<code>true</code> if either <code>expression1</code> or <code>expression2</code> is <code>true</code>
<code>!</code> (Logical NOT)	<code>!expression</code>	<code>true</code> if <code>expression</code> is <code>false</code> and vice versa

```

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

        // && operator
        System.out.println((5 > 3) && (8 > 5)); // true
        System.out.println((5 > 3) && (8 < 5)); // false

        // || operator
        System.out.println((5 < 3) || (8 > 5)); // true
        System.out.println((5 > 3) || (8 < 5)); // true
        System.out.println((5 < 3) || (8 < 5)); // false

        // ! operator
        System.out.println(!(5 == 3)); // true
        System.out.println(!(5 > 3)); // false
    }
}

```

#### Working of Program

- `(5 > 3) && (8 > 5)` returns `true` because both `(5 > 3)` and `(8 > 5)` are `true`.
- `(5 > 3) && (8 < 5)` returns `false` because the expression `(8 < 5)` is `false`.
- `(5 < 3) || (8 > 5)` returns `true` because the expression `(8 > 5)` is `true`.
- `(5 > 3) || (8 < 5)` returns `true` because the expression `(5 > 3)` is `true`.
- `(5 < 3) || (8 < 5)` returns `false` because both `(5 < 3)` and `(8 < 5)` are `false`.
- `!(5 == 3)` returns `true` because `5 == 3` is `false`.
- `!(5 > 3)` returns `false` because `5 > 3` is `true`.

## 5. JAVA UNARY OPERATORS:

**Unary operators** are used with only one operand. For example, **++** is a unary operator that increases the value of a variable by 1. That is, **++5** will return **6**. Different types of unary operators are:

Operator	Meaning
<b>+</b>	<b>Unary plus:</b> not necessary to use since numbers are positive without using it
<b>-</b>	<b>Unary minus:</b> inverts the sign of an expression
<b>++</b>	<b>Increment operator:</b> increments value by 1
<b>--</b>	<b>Decrement operator:</b> decrements value by 1
<b>!</b>	<b>Logical complement operator:</b> inverts the value of a boolean

## INCREMENT AND DECREMENT OPERATORS

Java offers **increment** and **decrement** operators: **++** and **--** respectively. **++** increases the value of the operand by 1, while **--** decrease it by 1. E.g., `int num = 5;`

```
// increase num by 1
++num;
```

Here, the value of **num** gets increased to 6 from its initial value of 5.

### Example: Increment and Decrement Operators

```
class Main {
    public static void main(String[] args) {
        // declare variables
        int a = 12, b = 12;
        int result1, result2;
        // original value
        System.out.println("Value of a: " + a);
        // increment operator
        result1 = ++a;
        System.out.println("After increment: " + result1);
        System.out.println("Value of b: " + b);
        // decrement operator
        result2 = --b;
```

```
        System.out.println("After decrement: " + result2);
    }
}
```

In the above program, we have used the **++** and **--** operator as **prefixes** (**++a**, **--b**). We can also use these operators as **postfix** (**a++**, **b++**). There is a slight difference when these operators are used as prefix versus when they are used as a postfix.

## INCREMENT **++** AND DECREMENT **--** OPERATOR AS PREFIX AND POSTFIX

In Java, the **increment operator ++** **increases** the value of a variable by **1**. Similarly, the **decrement operator --** **decreases** the value of a variable by **1**.

```
a = 5
++a;    // a becomes 6
a++;    // a becomes 7
--a;    // a becomes 6
a--;    // a becomes 5
```

There is an important difference when these two operators are used as a **PREFIX** and a **POSTFIX**.

- If you use the **++ operator** as a **prefix** like: **++var**, the value of **var** is incremented by **1**; then it returns the value.
- If you use the **++ operator** as a **postfix** like: **var++**, the original value of **var** is returned **first**; then var is incremented by 1.
- The **--** operator works in a similar way to the **++** operator except **--** decreases the value by 1.

```
class Operator {
    public static void main(String[ ] args) {
        int var1 = 5, var2 = 5;

        // 5 is displayed
        // Then, var1 is increased to 6.
        System.out.println(var1++);

        // var2 is increased to 6
        // Then, var2 is displayed
        System.out.println(++var2);
    }
}
```

## 6. JAVA BITWISE OPERATORS

Bitwise operators in Java are used to perform operations on individual bits. For example,

```
Bitwise complement Operation of 35
```

```
35 = 00100011 (In Binary)
```

```
~ 00100011
```

```
11011100 = 220 (In decimal)
```

Here, ~ is a bitwise operator. It inverts the value of each bit (0 to 1 and 1 to 0). The various bitwise operators present in Java are:

Operator	Description
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR

## OTHER OPERATORS

Besides these operators, there are other additional operators in Java.

### Java instanceof Operator

The instanceof operator checks whether an object is an instanceof a particular class. For example,

```
class Main {  
    public static void main(String[ ] args) {  
        String str = "ISBAT UNIVERSITY";  
        boolean result;  
        // checks if str is an instance of  
        // the String class  
        result = str instanceof String;  
        System.out.println("Is str a String object? " + result);  
    }  
}
```

Here, str is an instance of the String class. Hence, the **instanceof** operator returns true. To learn more, visit Java **instanceof**.

## Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

```
variable = Expression ? expression1 : expression2
```

Here's how it works.

- If the Expression is true, expression1 is assigned to the variable.
- If the Expression is false, expression2 is assigned to the variable.

```
class Java {  
    public static void main(String[] args) {  
        int februaryDays = 29;  
        String result;  
  
        // ternary operator  
        result = (februaryDays == 28) ? "Not a leap year" : "Leap year";  
        System.out.println(result);    }  
}
```

# JAVA BASIC INPUT AND OUTPUT

## 1.JAVA OUTPUT

In Java, you can simply use the following

**System.out.println();** or

**System.out.print();** or

**System.out.printf();**

to send output to standard output (screen).

Here,

**System** is a class

**out** is a public static field: it accepts output data.

Example to output a line.

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

```
System.out.println("Java programming is interesting.");  
}}
```

### Difference between **println()**, **print()** and **printf()**

**print()** - It prints string inside the quotes.

**println()** - It prints string inside the quotes similar like **print()** method.

Then the cursor moves to the beginning of the next line.

**printf()** - It provides string formatting

### Example: **print()** and **println()**

```
class Output {  
public static void main(String[ ] args) {  
  
System.out.println("1. println ");  
System.out.println("2. println ");  
  
System.out.print("1. print ");  
System.out.print("2. print");  
  
}}
```

In the above example, we have seen the working of the **print()** and **println()** methods. Try practicing with the **printf()** method in your free time.

### Example: Printing Variables and Literals

```
class Variables {  
public static void main(String[ ] args) {  
  
Double number = -10.6;  
  
System.out.println(5);  
System.out.println(number);  
  
}  
}
```

Here, you can see that we have not used the quotation marks. It is because to display integers, variables and so on, we don't use quotation marks.

### Example: Print Concatenated Strings

```
class PrintVariables {  
public static void main(String[] args) {  
  
Double number = -10.6;  
  
System.out.println("I am " + "awesome.");  
System.out.println("Number = " + number);  
  
}
```



```
}}
```

## 2.JAVA INPUT

Java offers different ways to **get input from the user**. Let's see how to **get input** from a user using the **object of Scanner class**. To use the **object of Scanner**, we have to **import java.util.Scanner package**.

```
import java.util.Scanner;
```

Then, we need to create **an object** of the **Scanner class**. We can use the object to take input from the user e.g.

```
// create an object of Scanner
```

```
Scanner input = new Scanner(System.in);
```

```
// take input from the user
```

```
int number = input.nextInt();
```

**Example: GET INTEGER INPUT FROM A USER**

```
import java.util.Scanner;
```

```
class Input {
```

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

```
        Scanner input = new Scanner(System.in);
```

```
        System.out.print("Enter an integer: ");
```

```
        int number = input.nextInt();
```

```
        System.out.println("You have Typed " + number);
```

```
        // close the scanner object
```

```
        input.close();
```

```
    }}
```

In the above example, we have created **an object** named **input** of the **Scanner class**. We then call the **nextInt() method** of the Scanner class to get an integer input from a user.

Also, we can use **nextLong()**, **nextFloat()**, **nextDouble()**, and **next() methods** to get **long**, **float**, **double**, and **string input** respectively from a user.

Note: We have used the **close() method** to close the object. It is recommended to close the **scanner object** once the **input is taken**.

**Example: Get float, double and String Input**

```
import java.util.Scanner;
```

```

class Input {
    public static void main(String[ ] args) {
        Scanner input = new Scanner(System.in);
        // Get the float input from the user
        System.out.print("Enter float: ");
        float myFloat = input.nextFloat();
        System.out.println("Float entered = " + myFloat);

        // Get the double input from the user
        System.out.print("Enter double: ");
        double myDouble = input.nextDouble();
        System.out.println("Double entered = " + myDouble);

        // Get the String input from the user
        System.out.print("Enter text: ");
        String myString = input.next();
        System.out.println("Text entered = " + myString);
    }
}

```

## JAVA EXPRESSIONS, STATEMENTS AND BLOCKS

Java expressions, Java statements, the difference between **expression** and **statement**, and **Java blocks** with examples. In previous notes, we have at least used **expressions**, statements, and **blocks** without explaining about them. Now that we know about **variables**, **operators**, and **literals**, it will be easier to understand these concepts.

### JAVA EXPRESSIONS

A **Java expression** is made of **variables**, **operators**, **literals**, and **method calls**. We shall see or learn more about **method calls**, in the OOP section,

```

int score;
score = 90;

```

Here, **score = 90** is an **expression** that returns **an int**. Consider another example,

```

Double a = 2.2, b = 3.4, result;
result = a + b - 3.4;
Here, a + b - 3.4 is an expression.

```

```

if (number1 == number2)
    System.out.println("Number 1 is larger than number 2");

```

Here, **number1 == number2** is an expression that returns **a boolean value**. Also, **"Number 1 is larger than number 2"** is a string expression.

## JAVA STATEMENTS

In Java, **every statement** is a complete unit of execution. For example,

```
int score = 9*5;
```

In the above **statement**, **int score = 9\*5**; has a complete execution which involves multiplying integers **9** and **5** and then **assigns** the result to the variable **score**. In Java, **expressions** are part of **statements**.

## EXPRESSION STATEMENTS

We can convert **an expression** into **a statement** by terminating the expression with a **;**.

These are known as **expression statements**. For example,

```
// expression
number = 10
// statement
number = 10;
```

In the above example, we have an expression **number = 10**. By adding a semicolon (**;**), we have converted the **expression** into a **statement** (**number = 10**);.

Consider another example,

```
// expression
++number
// statement
++number;
```

Similarly, **++number** is an expression whereas **++number;** is a statement.

## DECLARATION STATEMENTS

In Java, **declaration statements** are used for declaring variables. For example,

```
Double tax = 9.5;
```

The above statement declares a **variable tax** which is also **initialized** to **9.5**.

Note: We also have Java **control flow statements** that are used **in decision making** and **looping** in Java.

## JAVA BLOCKS

A **block** is a group of statements (zero or more) that is enclosed in curly braces **{ }**. E.g.,

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

```

String band = "Afrigo";

if (band == "Afrigo") { // Start of Java Statements block
    System.out.print("Moses Matovu ");
    System.out.print("the Band Leader!");
} // end of block
}

```

Moses Matovu the Band Leader!"

In the above example, we have a block if **{....}**. Here, inside the block we have two statements:

```

System.out.print("Moses Matovu ");
System.out.print("the Band Leader!");

```

But, a block may not have any statements. Consider the following examples,

```

class Main {
    public static void main(String[ ] args) {
        if (10 > 5) { // start of block
            // end of block
        }
    }
}

```

This is a valid Java program. Here, we have a block if **{....}**. But, there no any statement inside this block.

```

class AssignmentOperator {
    public static void main(String[ ] args) { // start of block
        // end of block
    }
}

```

## JAVA COMMENTS

Let's all look at **Java comments**, why we use them, and the right way to use them. In computer programming, **comments** are a section of the source code that are completely ignored by Java compilers. They are mainly used to help programmers to understand the code. For example,

```

// declare and initialize two variables
int a =1;
int b = 3;

// print the output
System.out.println("This is our output");

```

In Java, there are two types of comments:

**single-line comment**  
**multi-line comment**

## ●SINGLE-LINE COMMENT

A single-line comment starts and ends in the same line. To write a single-line comment, we can use the `//` symbol. For example,

```
// "Hello, ISBAT UNIVERSITY!" program example
class Main {
    public static void main(String[ ] args) {
        // prints "Hello, ISBAT UNIVERSITY!"
        System.out.println("Hello, ISBAT UNIVERSITY!");
    }
}
```

The Java compiler ignores everything from `//` to the end of line. Hence, it is also known as the **End of Line comment**.

## ●MULTI-LINE COMMENT

When we want to write comments in multiple lines, we can use the multi-line comment. To write multi-line comments, we can use the `/*....*/` symbol. For example,

```
/* This is an example of multi-line comment.
 * The program prints "Hello, World!" to the standard output.
 */
```

This type of comment is also called a **Traditional Comment**. In this type of comment, the Java compiler ignores everything from `/*` to `*/`.

## USE COMMENTS THE RIGHT WAY

NB: Comments shouldn't be a substitute for a way to [explain poorly written code in English](#). You should always write **well-structured** and **self-explaining code**. And, then use comments. In most cases, always use comments to explain **'why'** rather than **'how'** and you are good to go.

# JAVA FLOW CONTROL SESSION 2

## JAVA IF...ELSE STATEMENT

**Control flow statements** in **Java** with **if** and **if...else statements** with examples. In Java, we can ably use the **if..else statement** to run a block of code. E.g., assigning of grades (**A**, **B**, **C**) based on the percentage obtained by a student.

```
if the percentage is above 90, assign grade A
if the percentage is above 75, assign grade B
if the percentage is above 65, assign grade C
```

### 1.Java if (if-then) Statement

The syntax of an if-then statement is:

```

if (condition) {
    // statements
}

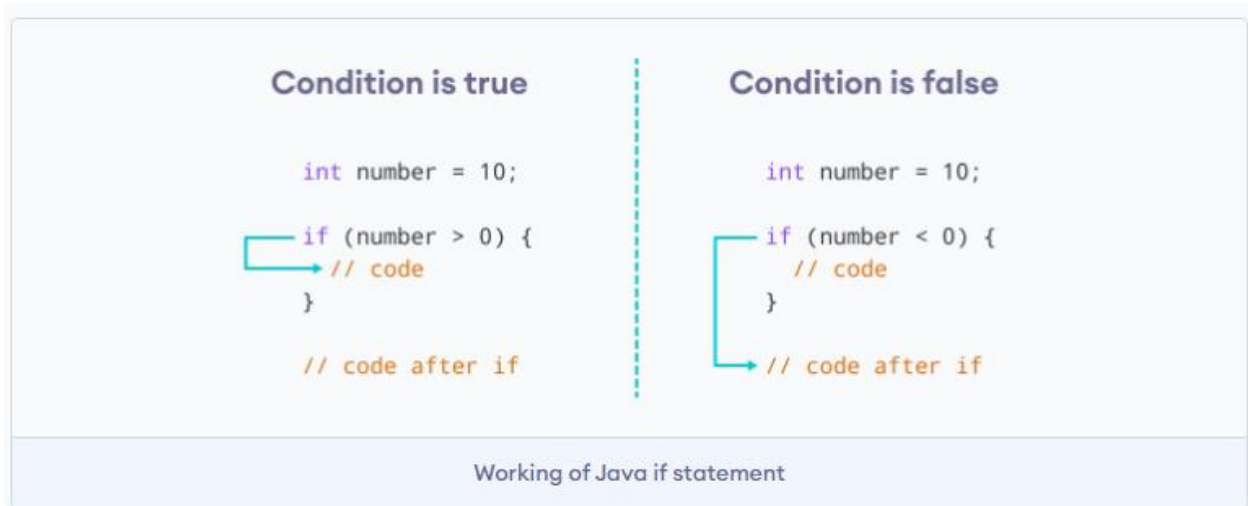
```

Here, the condition is a **boolean expression** such as `age >= 18`.

if condition evaluates to true, statements are executed

if condition evaluates to false, statements are skipped

## WORKING WITH THE IF STATEMENT IN JAVA



if the number is greater than 0, the code inside the if block is executed, otherwise code inside if block is skipped

### Example 1: Java if Statement

```

class IfStatement {
    public static void main(String[] args) {
        int number = 10;
        // checks if number is less than 0
        if (number < 0) {
            System.out.println("The number is negative.");
        }
        System.out.println("Statement outside if block");
    }
}

```

## STATEMENT OUTSIDE IF BLOCK

In the program, `number < 0` is false. Hence, the code inside the body of the **if statement** is skipped. We can also use Java Strings as the test condition.

### Example 2: Java if with String

```

class Main {

```

```

public static void main(String[ ] args) {
    // create a string variable
    String language = "Java";
    // if statement
    if (language == "Java") {
        System.out.println("The Best Programming Language");
    }
}

```

In the above example, we are comparing two strings in the **if block**.

## 2. Java if...else (if...then-else) Statement

The **if statement** executes a certain section of code if the test expression is evaluated to **true**. But, if the **test expression** is evaluated to **false**, it does nothing. In this case, we can use an **optional else block**. Statements in the body of a else block are executed if a test expression is evaluated to false. This is known as the **if-...else statement** in Java.

The syntax of the if...else statement is:

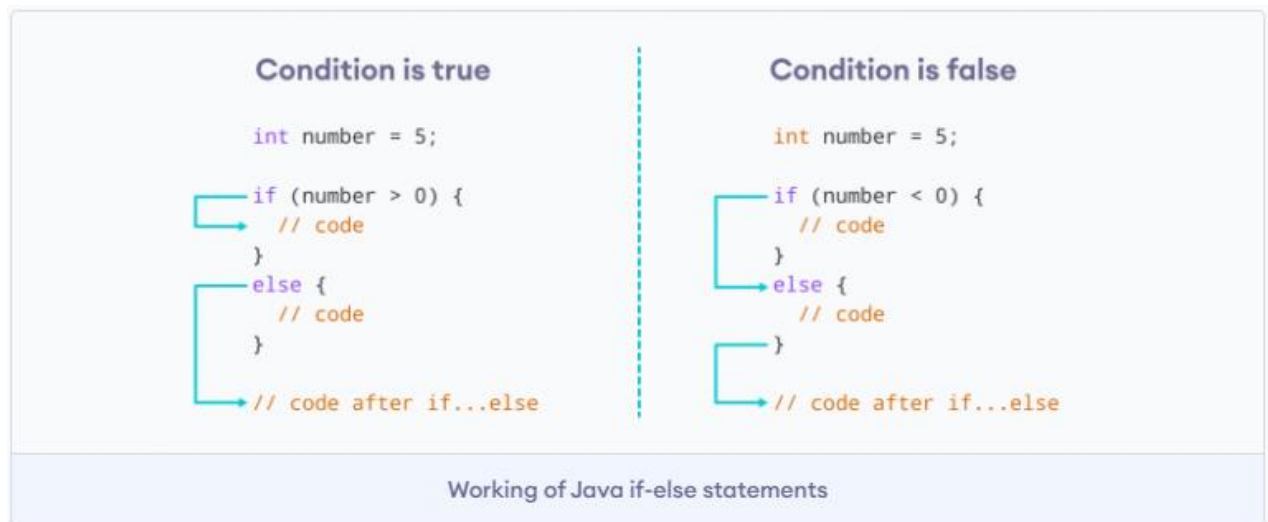
```

if (condition) {
    // codes in if block
}
else {
    // codes in else block
}

```

Here, the program will do one task (codes inside the if block) if the condition **is true** and another task (codes inside **else block**) if the condition **is false**.

## How the if...else statement works?



If the condition **is true**, the code inside the **if block** is executed, otherwise, code inside the **else block** is executed

### Example 3: Java if...else Statement

```
class Main {  
    public static void main(String[] args) {  
        int number = 10;  
  
        // checks if number is greater than 0  
        if (number > 0) {  
            System.out.println("The number is positive.");  
        }  
  
        // execute this block if the number is not greater than 0  
        else {  
            System.out.println("The number is not positive.");  
        }  
  
        System.out.println("Statement outside if...else block");  
    }  
}
```

### 3. Java if...else...if Statement

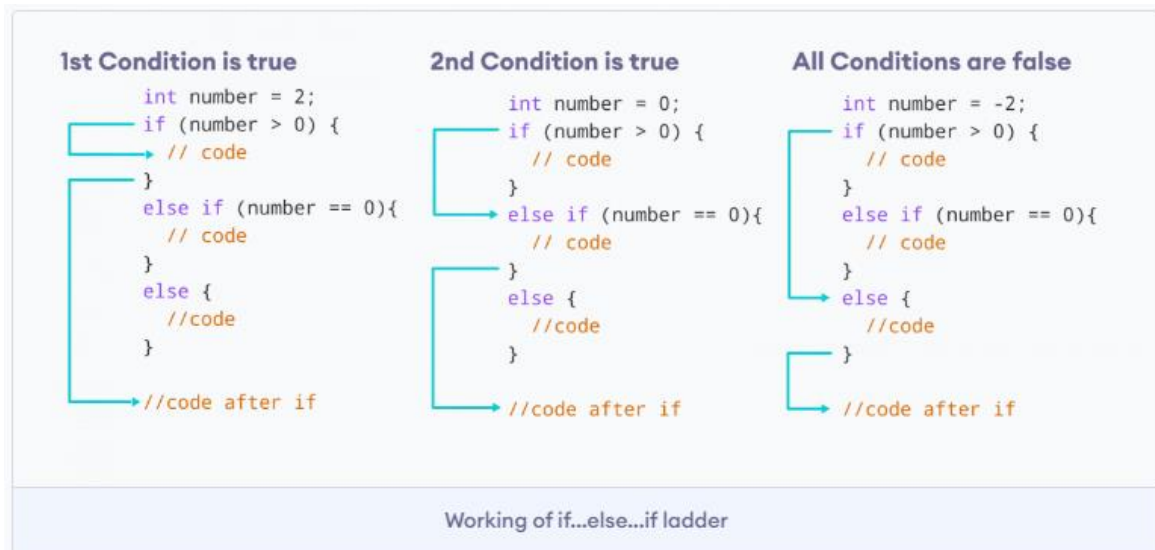
We also have an **if...else...if ladder**, that can be used to execute one block of code among multiple or many other blocks.

```
if (condition1) {  
    // codes  
}  
else if(condition2) {  
    // codes  
}  
else if (condition3) {  
    // codes  
}  
.  
.  
else {  
    // codes  
}
```



Here, the **if statements** are executed from the top towards the bottom. When the test condition **is true**, codes in the body of that **if block is executed**. And, program control **jumps outside** the **if...else...if ladder**. If all test expressions are false, codes inside the body of else are executed.

## How the if...else...if ladder works?



If the first test condition is true, code inside first if block is executed, if the second condition is true, block inside second if is executed, and if all conditions are false, the else block is executed.

### Example 4: Java if...else...if Statement

```
class Main {  
    public static void main(String[] args) {  
        int number = 0;  
        // checks if number is greater than 0  
        if (number > 0) {  
            System.out.println("The number is positive.");  
        }  
        // checks if number is less than 0  
        else if (number < 0) {  
            System.out.println("The number is negative.");  
        }  
        // if both condition is false  
        else {  
            System.out.println("The number is 0.");  
        } } }
```

In the above example, we are checking whether number is positive, negative, or zero. Here, we have two condition expressions:

**number > 0** - checks if number is greater than 0

**number < 0** - checks if number is less than 0

Here, the value of number is 0. So both conditions evaluate to **false**. Hence the statement inside the body of the else is executed. NB: Java offers a **special operator** called **ternary operator**, which is a kind of shorthand notation of the if...else...if statement.

## 4. Java Nested if..else Statement

In Java, it is similarly possible to use the **if..else statements inside** an **if...else statement**. It's called the **nested if...else statement**.

Here's a program to find the largest of 3 numbers using the nested if...else statement.

**Example 5: Nested if...else Statement**

```
class Main {
    public static void main(String[ ] args) {
        // declaring double type variables
        Double n1 = -1.0, n2 = 4.5, n3 = -5.3, largest;
        // checks if n1 is greater than or equal to n2
        if (n1 >= n2) {
            // if...else statement inside the if block
            // checks if n1 is greater than or equal to n3
            if (n1 >= n3) {
                largest = n1;
            }
            else {
                largest = n3;
            }
        } else {
            // if..else statement inside else block
            // checks if n2 is greater than or equal to n3
            if (n2 >= n3) {
                largest = n2;
            }
            else {
                largest = n3;
            }
        }
    }
}
```

```

    }
    System.out.println("Largest Number: " + largest);
}
}

```

## ●JAVA SWITCH STATEMENT

Let's use the **switch statement** in Java to control the flow of our program's execution with examples. The switch statement allows us to execute a block of code among many alternatives. The syntax of the switch statement in Java is:

```

switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    ...
    ...
    default:
        // default statements
}

```

How does the **switch-case statement** work? The expression is evaluated once and compared with the values of each case. If expression matches with **value1**, the code of case **value1** are executed. Also, the code of case **value2** is executed if expression matches with **value2**. If there is no match, the code of the **default case** is executed. Note: The working of the **switch-case** statement is similar to the Java **if...else...if ladder**. However, **the syntax of the switch statement is cleaner and much easier to read and write.**

**Example:** Java switch Statement

// Java Program to check the size using the switch...case statement

```

class Main {
    public static void main(String[ ] args) {
        int number = 44;
        String size;
        // switch statement to check size
        switch (number) {
            case 29:

```

```

        size = "Small";
        break;

case 42:
    size = "Medium";
    break;

// match the value of week
case 44:
    size = "Large";
    break;

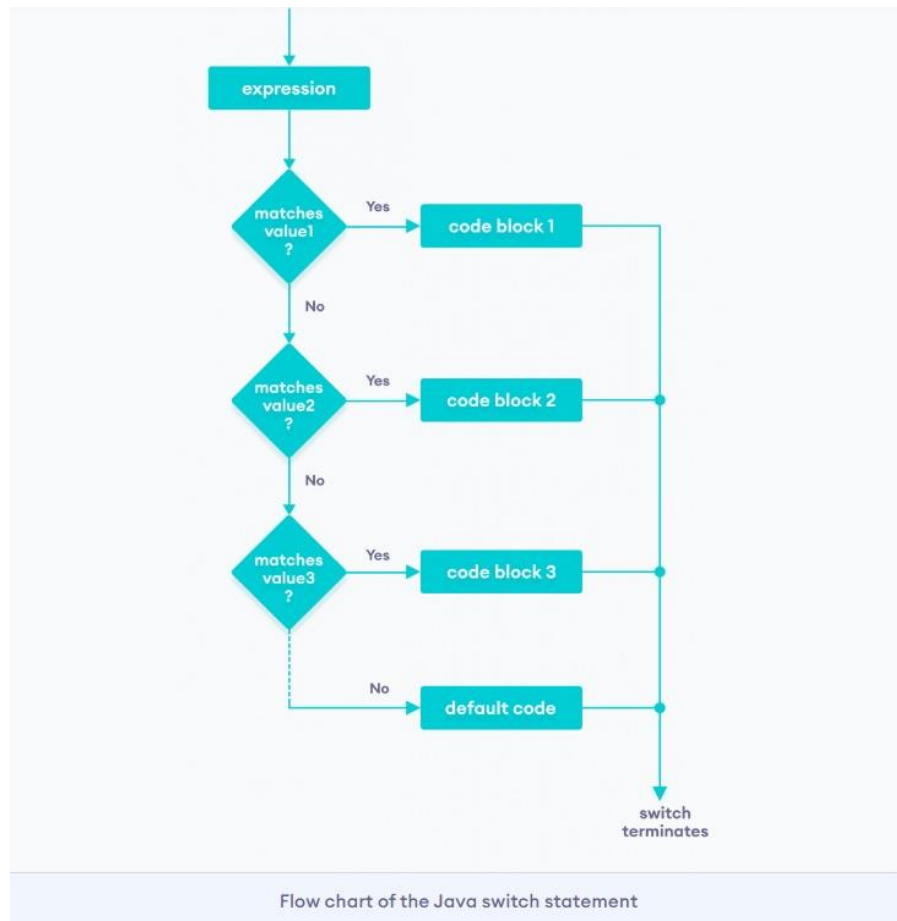
case 48:
    size = "Extra Large";
    break;

default:
    size = "Unknown";
    break;
    }
    System.out.println("Size: " + size);
}
}

```

Recommended: **Create a Simple Calculator Using the Java switch Statement**

## **Flowchart of switch Statement**



The break statement is used to terminate the switch-case statement. If break is not used, all the cases after the matching case are also executed. For example,

```
class Main {  
    public static void main(String[ ] args) {  
        int expression = 2;  
        // switch statement to check size  
        switch (expression) {  
            case 1:  
                System.out.println("Case 1");  
                // matching case  
            case 2:  
                System.out.println("Case 2");  
            case 3:  
                System.out.println("Case 3");  
            default:  
                System.out.println("Default case");  
        }  
    }  
}
```

**Default Case:** In the above example, the expression matches with **case 2**. Here, we have not used the break statement after each case. Hence, all the cases after **case 2** are also executed. This is why the **break statement is needed to terminate the switch-case statement after the matching case**. To learn more, visit [Java break Statement](#).

## The default case in Java switch-case

The switch statement also has an **optional default case**. It is executed when an expression doesn't match any of the cases. For example,

```
class Main {  
    public static void main(String[] args) {  
        int expression = 9;  
        switch(expression) {  
            case 2:  
                System.out.println("Small Size");  
                break;  
            case 3:  
                System.out.println("Large Size");  
                break;  
            // default case  
            default:  
                System.out.println("Unknown Size");  
        }  
    }  
}
```

In the above example, we have created a switch-case statement. Here, the value of expression doesn't match with any of the cases. Hence, the **code inside the default case is executed**.

**Note: The Java switch statement only works with:**

- Primitive data types: byte, short, char, and int
- Enumerated types
- String Class
- Wrapper Classes: Character, Byte, Short, and Integer.

## ●JAVA FOR LOOP

Let's use the **for loop** in Java with examples. In computer programming, **loops** are used to repeat a block of code. E.g., if you want to show a **message 100 times**, then rather than typing the same code 100 times, you can **use a loop**.

In Java, there are three types of loops.

**for loop**

**while loop**

**do...while loop**

## JAVA FOR LOOP

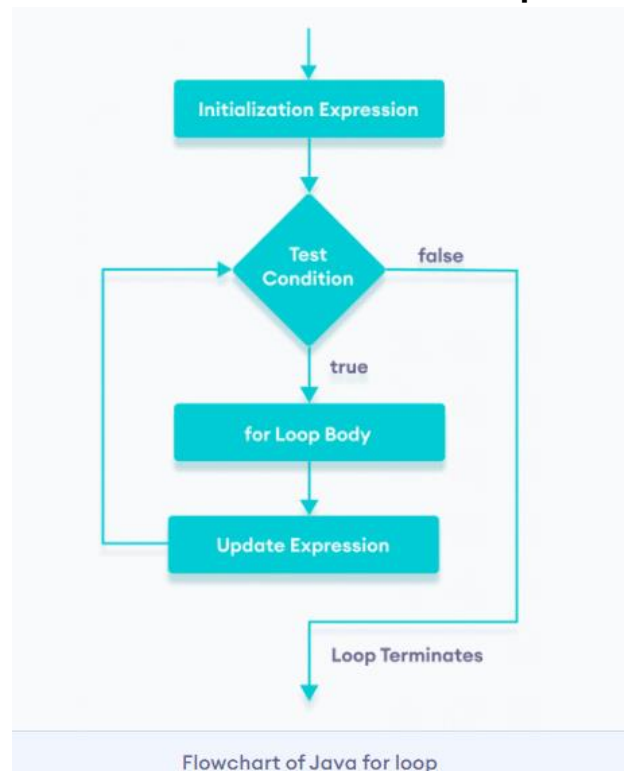
**Java** for loop is used to run a block of code for a certain number of times. The syntax of a for loop is:

```
for (initialExpression; testExpression; updateExpression) {  
    // body of the loop  
}
```

Here,

- The **initialExpression** initializes and/or declares variables and executes only once.
- The **condition** is evaluated. If the condition is true, the body of the for loop is executed.
- The **updateExpression** updates the value of initialExpression.
- The condition is evaluated again. The process continues until the condition is false.

### Flowchart of Java for loop



### Example 1: Display Text Five Times

// Program to print a text 5 times

```
class Main {
```

```

public static void main(String[ ] args) {
    int n = 5;
    // for loop
    for (int i = 1; i <= n; ++i) {
        System.out.println("ISBAT UNIVERSITY");
    } }

```

Here is how this program works.

Iteration	Variable	Condition: i <= n	Action
1st	i = 1 n = 5	true	Java is fun is printed. i is increased to 2.
2nd	i = 2 n = 5	true	Java is fun is printed. i is increased to 3.
3rd	i = 3 n = 5	true	Java is fun is printed. i is increased to 4.
4th	i = 4 n = 5	true	Java is fun is printed. i is increased to 5.
5th	i = 5 n = 5	true	Java is fun is printed. i is increased to 6.
6th	i = 6 n = 5	false	The loop is terminated.

**Example 2:** Display numbers from 1 to 5

// Program to print numbers from 1 to 5

```

class Main {
    public static void main(String[ ] args) {
        int n = 5;
        // for loop
        for (int i = 1; i <= n; ++i) {
            System.out.println(i);
        } }

```

Here is how the program works.



Iteration	Variable	Condition: $i \leq n$	Action
1st	$i = 1$ $n = 5$	true	1 is printed. i is increased to 2.
2nd	$i = 2$ $n = 5$	true	2 is printed. i is increased to 3.
3rd	$i = 3$ $n = 5$	true	3 is printed. i is increased to 4.
4th	$i = 4$ $n = 5$	true	4 is printed. i is increased to 5.
5th	$i = 5$ $n = 5$	true	5 is printed. i is increased to 6.
6th	$i = 6$ $n = 5$	false	The loop is terminated.

### Example 3: Display Sum of n Natural Numbers

// Program to find the sum of natural numbers from 1 to 1000.

```
class Main {
    public static void main(String[] args) {
        int sum = 0;
        int n = 1000;
        // for loop
        for (int i = 1; i <= n; ++i) {
            // body inside for loop
            sum += i;    // sum = sum + i
        }
        System.out.println("Sum = " + sum);
    }
}
```

Here, the value of sum is 0 at first. The **for loop** is iterated from  $i = 1$  to **1000**. In each iteration,  $i$  is added to sum and its value is increased by 1. When  $i$  becomes 1001, the test condition is false and sum will be equal to  $0 + 1 + 2 + \dots + 1000$ .

The above program to add the sum of natural numbers can also be written as

// Program to find the sum of natural numbers from 1 to 1000.

```
class Main {
    public static void main(String[] args) {
        int sum = 0;
        int n = 1000;
```

```

// for loop
for (int i = n; i >= 1; --i) {
    // body inside for loop
    sum += i;    // sum = sum + i
}
System.out.println("Sum = " + sum);
}
}

```

## ●JAVA FOR-EACH LOOP

**Java for loop** has an another syntax that makes it easy to iterate through arrays and collections. For example,

```

// print array elements
class Main {
    public static void main(String[ ] args) {
        // create an array
        int[ ] numbers = {3, 7, 5, -5};
        // iterating through the array
        for (int number: numbers) {
            System.out.println(number);
        }
    }
}

```

Here, we have used the **for-each loop** to print each element of the numbers array one by one. In the first iteration of the loop, number will be 3, number will be 7 in second iteration and so on.

## ●JAVA INFINITE FOR LOOP

If we set the **test expression** in such a way that it never **evaluates to false**, the **for loop** will run **forever**. This is called **infinite for loop**. For example,

```

// Infinite for Loop
class Infinite {
    public static void main(String[ ] args) {
        int sum = 0;
        for (int i = 1; i <= 10; --i) {
            System.out.println("Hello");
        }
    }
}

```

## ●JAVA FOR-EACH LOOP

The **Java for-each loop** the for-each loop is used to iterate through elements of arrays and collections (like an ArrayList). It is also known as the **enhanced for loop**.

### for-each Loop Syntax

```
for(dataType item : array) {  
    ...  
}
```

Here,

- **array** – is an array or a collection.
- **item** - each item of the array or collection is assigned to this variable.
- **dataType** – is the data type of the array or collection.

#### Example 1: Print Array Elements

```
// print array elements  
class Main {  
    public static void main(String[] args) {  
        // create an array  
        int[ ] numbers = {3, 9, 5, -5};  
  
        // for each loop  
        for (int number: numbers) {  
            System.out.println(number);  
        }  
    }  
}
```

Here, we have used the for-each loop to print each element of the numbers array one by one.

- In the first iteration, the item will be 3.
- In the second iteration, the item will be 9.
- In the third iteration, the item will be 5.
- In the fourth iteration, the item will be -5.

#### Example 2: Sum of Array Elements

```
// Calculate the sum of all elements of an array  
class Main {  
    public static void main(String[] args) {  
        // an array of numbers  
        int [ ] numbers = {3, 4, 5, -5, 0, 12};  
        int sum = 0;  
  
        // iterating through each element of the array  
        for (int number: numbers) {
```

```

        sum += number;
    }
    System.out.println("Sum = " + sum);
}

```

In the above program, the execution of the for each loop looks as:

Iteration	Variables
1	<div>number = 3</div> <div>sum = 0 + 3 = 3</div>
2	<div>number = 4</div> <div>sum = 3 + 4 = 7</div>
3	<div>number = 5</div> <div>sum = 7 + 5 = 12</div>
4	<div>number = -5</div> <div>sum = 12 + (-5) = 7</div>
5	<div>number = 0</div> <div>sum = 7 + 0 = 7</div>
6	<div>number = 12</div> <div>sum = 7 + 12 = 19</div>

As we can see, we have added each element of the numbers array to the sum variable in each iteration of the loop.

**Let's see how a for-each loop is different from a regular Java for loop.**

#### 1. Using for loop

```

class Main {
    public static void main(String[] args) {
        char[] vowels = {'a', 'e', 'i', 'o', 'u'};
        // iterating through an array using a for loop
        for (int i = 0; i < vowels.length; ++i) {
            System.out.println(vowels[i]);
        }
    }
}

```

#### 2. Using for-each Loop

```

class Main {
    public static void main(String[] args) {
        char[] vowels = {'a', 'e', 'i', 'o', 'u'};
        // iterating thru an array using a for-each loop
        for (char item: vowels) {
            System.out.println(item);
        }
    }
}

```

Here, the output of both programs is the same. However, the **for-each loop** is easier to write and understand. This is why the **for-each loop** is ideal over the **for loop** when working with arrays and collections.

## ●Java while and do...while Loop

In computer programming, **loops** are used to repeat a block of code. E.g., if you want to show a message 100 times, then you can use a loop. It's just a simple example; you can achieve much more with loops. Here, we are going to learn about the **while** and **do...while loops**.

### 1.Java while loop

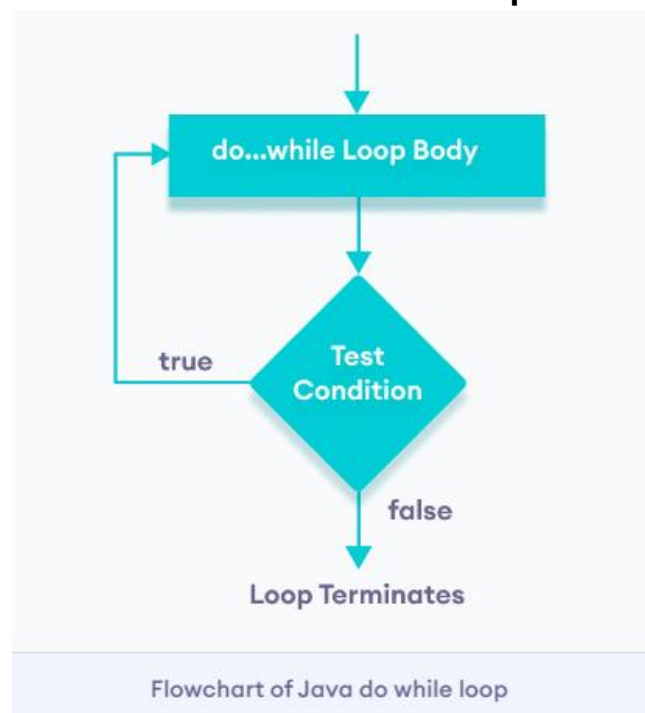
**Java while loop** is used to run a specific code until a certain condition is met. The syntax of the while loop is:

```
while (testExpression) {  
    // body of loop  
}
```

Here,

- A while loop evaluates the textExpression inside the parenthesis ().
- If the textExpression evaluates to true, the code inside the while loop is executed.
- The textExpression is evaluated again.
- This process continues until the textExpression is false.
- When the textExpression evaluates to false, the loop stops.

### Flowchart of while loop



### Example 1: Display Numbers from 1 to 5

```
// Program to display numbers from 1 to 5
class Main {
    public static void main(String[] args) {
        // declare variables
        int i = 1, n = 5;

        // while loop from 1 to 5
        while(i <= n) {
            System.out.println(i);
            i++;
        }
    }
}
```

Here is how this program works.

Iteration	Variable	Condition: i <= n	Action
	i = 1 n = 5	not checked	1 is printed. i is increased to 2.
1st	i = 2 n = 5	true	2 is printed. i is increased to 3.
2nd	i = 3 n = 5	true	3 is printed. i is increased to 4.
3rd	i = 4 n = 5	true	4 is printed. i is increased to 5.
4th	i = 5 n = 5	true	5 is printed. i is increased to 6.
5th	i = 6 n = 5	false	The loop is terminated

### Example 2: Sum of Positive Numbers Only

// Java program to find the sum of positive numbers

```
import java.util.Scanner;
```

```
class Main {
    public static void main(String[ ] args) {
        int sum = 0;

        // create an object of Scanner class
        Scanner input = new Scanner(System.in);

        // take integer input from the user
        System.out.println("Enter a number");
        int number = input.nextInt();
    }
}
```

```

// while loop continues
// until entered number is positive
while (number >= 0) {
    // add only positive numbers
    sum += number;
    System.out.println("Enter a number");
    number = input.nextInt();
}
System.out.println("Sum = " + sum);
input.close();
}}

```

In the above Java program, we used the **Scanner class** to take input from the user. Here, **nextInt()** takes integer input from the user. The **while loop** continues until the user enters a negative number. During each iteration, the number entered by the user is added to the sum variable. When the user enters a negative number, the loop terminates.

## 2. Java do...while loop

The **do...while loop** is similar to while loop. But, the body of do...while loop is executed once before the test expression is checked. For example,

```

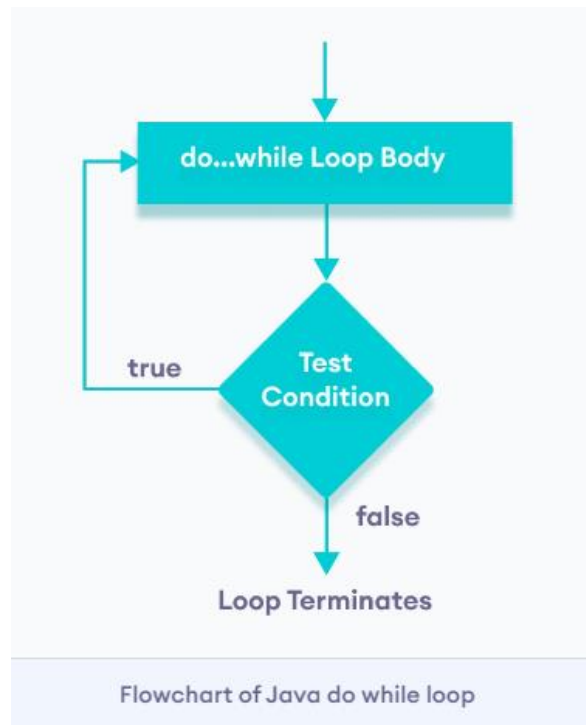
do {
    // body of loop
} while(textExpression);

```

Here,

- The body of the loop is executed at first. Then the textExpression is evaluated.
- If the textExpression evaluates to true, the body of the loop inside the do statement is executed again.
- The textExpression is evaluated once again.
- If the textExpression evaluates to true, the body of the loop inside the do statement is executed again.
- This process continues until the textExpression evaluates to false. Then the loop stops.

### Flowchart of do...while loop in Java



### do...while loop examples.

**Example 3:** Display Numbers from 1 to 5

// Java Program to display numbers from 1 to 5

**import java.util.Scanner;**

// Program to find the sum of natural numbers from 1 to 100.

**class Main {**

**public static void main(String[ ] args) {**

**int i = 1, n = 5;**

**// do...while loop from 1 to 5**

**do {**

**System.out.println(i);**

**i++;**

**} while(i <= n);**

**}}**



Iteration	Variable	Condition: $i \leq n$	Action
	$i = 1$ $n = 5$	not checked	1 is printed. i is increased to 2.
1st	$i = 2$ $n = 5$	true	2 is printed. i is increased to 3.
2nd	$i = 3$ $n = 5$	true	3 is printed. i is increased to 4.
3rd	$i = 4$ $n = 5$	true	4 is printed. i is increased to 5.
4th	$i = 5$ $n = 5$	true	5 is printed. i is increased to 6.
5th	$i = 6$ $n = 5$	false	The loop is terminated

#### Example 4: Sum of Positive Numbers

// Java program to find the sum of positive numbers

```
import java.util.Scanner;
```

```
class Main {
```

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

```
        int sum = 0;
```

```
        int number = 0;
```

```
        // create an object of Scanner class
```

```
        Scanner input = new Scanner(System.in);
```

```
        // do...while loop continues
```

```
        // until entered number is positive
```

```
        do {
```

```
            // add only positive numbers
```

```
            sum += number;
```

```
            System.out.println("Enter a number");
```

```
            number = input.nextInt();
```

```
        } while(number >= 0);
```

```
        System.out.println("Sum = " + sum);
```

```
        input.close();
```

```
    }}
```

Here, the user enters a **positive number**, that number is added to the **sum variable**. And this process continues until a number is negative. When the number is negative, the loop terminates and displays the sum without adding the negative number.

## Infinite while loop

If the condition of a **loop** is always true, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loop
while(true){
    // body of loop
}
```

Here is an example of an infinite do...while loop.

```
// infinite do...while loop
int count = 1;
do {
    // body of loop
} while(count == 1)
```

In the above Java programs, the **textExpression** is always true. Hence, the loop body will run for infinite times.

## for and while loops

The **for loop** is used when the number of iterations is known. For example,

```
for (let i = 1; i <=5; ++i) {
    // body of loop
}
```

And while and **do...while loops** are generally used when the number of iterations is unknown. For example,

```
while (condition) {
    // body of loop
}
```

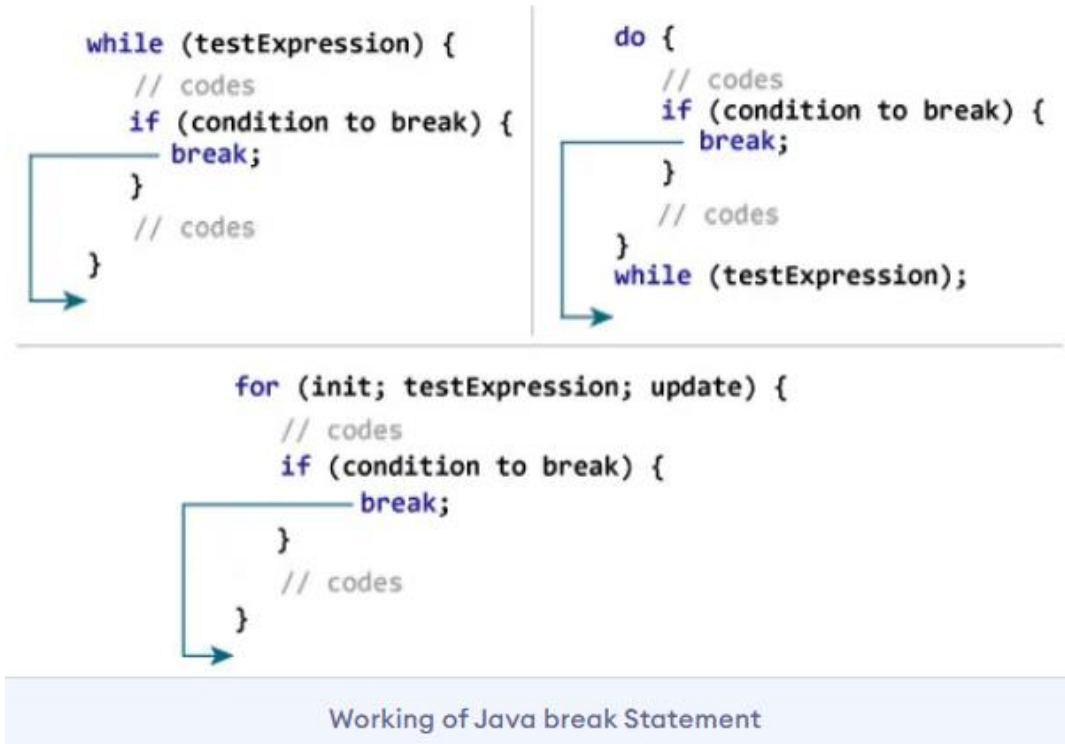
## Java break statement

While working with **loops**, it is sometimes desirable to skip some statements inside a **loop** or **terminate** the loop immediately without checking the test expression. In such cases, the **break** and **continue** statements are used. The **break statement** in Java terminates a loop instantly, and the program control moves to the next statement following the loop. It is almost always used with decision-making statements (Java if...else Statement).

Here is the syntax of the break statement in Java:

**break;**

How break statement works?



**Example 1:** Java break statement

```
class Test {  
    public static void main(String[] args) {  
        // for loop  
        for (int i = 1; i <= 10; ++i) {  
            // if the value of i is 5 the loop terminates  
            if (i == 5) {  
                break;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

In the above program, we are using the **for loop** to print the value of **i** in each iteration.

```
if (i == 5) {  
    break;  
}
```

This means when the value of **i** is equal to **5**, the loop terminates. Hence we get the output with values less than **5** only.

## Example 2: Java break statement

The Java program below calculates the **sum of numbers** entered by the user until user enters a negative number. To take input from the user, we have used the **Scanner object**.

```
import java.util.Scanner;
class UserInputSum {
    public static void main(String[] args) {
        Double number, sum = 0.0;
        // create an object of Scanner
        Scanner input = new Scanner(System.in);
        while (true) {
            System.out.print("Enter a number: ");
            // takes double input from user
            number = input.nextDouble();
            // if number is negative the loop terminates
            if (number < 0.0) {
                break;
            }
            sum += number;
        }
        System.out.println("Sum = " + sum);
    }
}
```

In the above program, the test expression of the **while loop** is always true. Here, notice the line,

```
if (number < 0.0) {
    break;
}
```

This means when the user input **negative numbers**, the while loop is terminated.

## Java break and Nested Loop

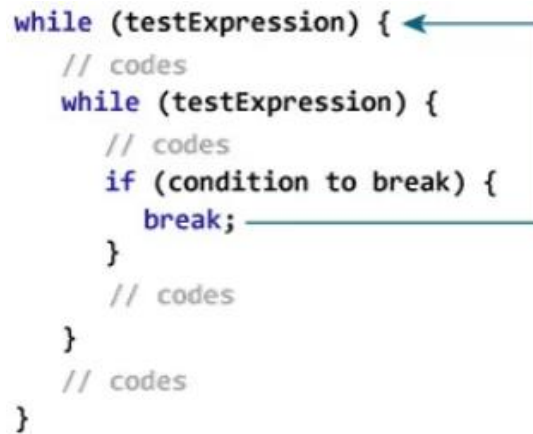
The break statement terminates the innermost while loop in case of nested loops.

### Working of a break Statement with Nested Loops

```

while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (condition to break) {
            break;
        }
        // codes
    }
    // codes
}

```



Here, the break statement terminates the innermost while loop, and control jumps to the outer loop.

## Labeled break Statement

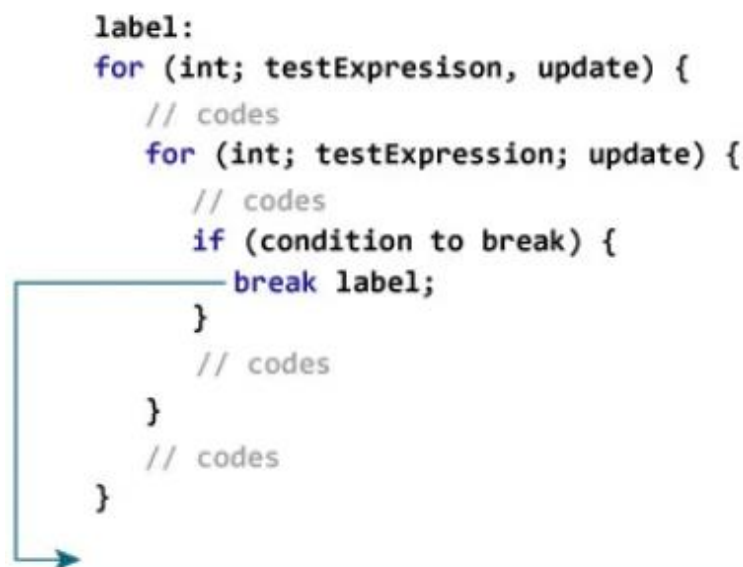
Till now, we have used the **unlabeled break statement**. It ends the **innermost loop** and switch statement. But, there is another form of break statement in Java known as the labeled break.

We can use the labeled break statement to terminate the outermost loop as well.

```

label:
for (int; testExpresison, update) {
    // codes
    for (int; testExpression; update) {
        // codes
        if (condition to break) {
            break label;
        }
        // codes
    }
    // codes
}

```



Working of the labeled break statement in Java

As you can see in the above image, we have used the **label identifier** to specify the outer loop. Now, notice how the **break statement** is used (**break label;**). Here, the **break statement** is terminating the labeled statement (i.e. outer loop). Then, the control of the program jumps to the statement after the labeled statement.

Here's another example:

```

while (testExpression) {
    // codes
    second:

```

```

while (testExpression) {
    // codes
while(testExpression) {
    // codes
    break second;
}
}
// control jumps here
}

```

In the above example, when the statement `break second;` is executed, the while loop labeled as `second` is terminated. And, the control of the program moves to the statement after the second while loop.

**Example 3:** labeled break Statement

```

class LabeledBreak {
    public static void main(String[ ] args) {
        // the for loop is labeled as first
        first:
        for( int i = 1; i < 5; i++) {
            // the for loop is labeled as second
            second:
            for(int j = 1; j < 3; j ++ ) {
                System.out.println("i = " + i + "; j = " +j);
                // the break statement breaks the first for loop
                if ( i == 2)
                    break first;
            } } } }
    }
}

```

In the above example, the **labeled break statement** is used to terminate the loop labeled as `first`. That is,

```

first:
for(int i = 1; i < 5; i++) {...}

```

if we change the statement `break first;` to `break second;` the program will behave differently. In this case, for loop labeled as `second` will be terminated. For example,

```

class LabeledBreak {
    public static void main(String[ ] args) {
        // the for loop is labeled as first
        first:

```

```

for( int i = 1; i < 5; i++) {
    // the for loop is labeled as second
    second:
    for(int j = 1; j < 3; j ++ ) {
        System.out.println("i = " + i + "; j = " +j);

        // the break statement terminates the loop labeled as second
        if ( i == 2)
            break second;
    } } }

```

Note: The break statement is also used to terminate cases inside the switch statement. To learn more, visit the Java switch statement.

## ●Java continue Statement

While working with loops, sometimes you might want to skip some statements or terminate the loop. In such cases, break and continue statements are used.

### Java continue

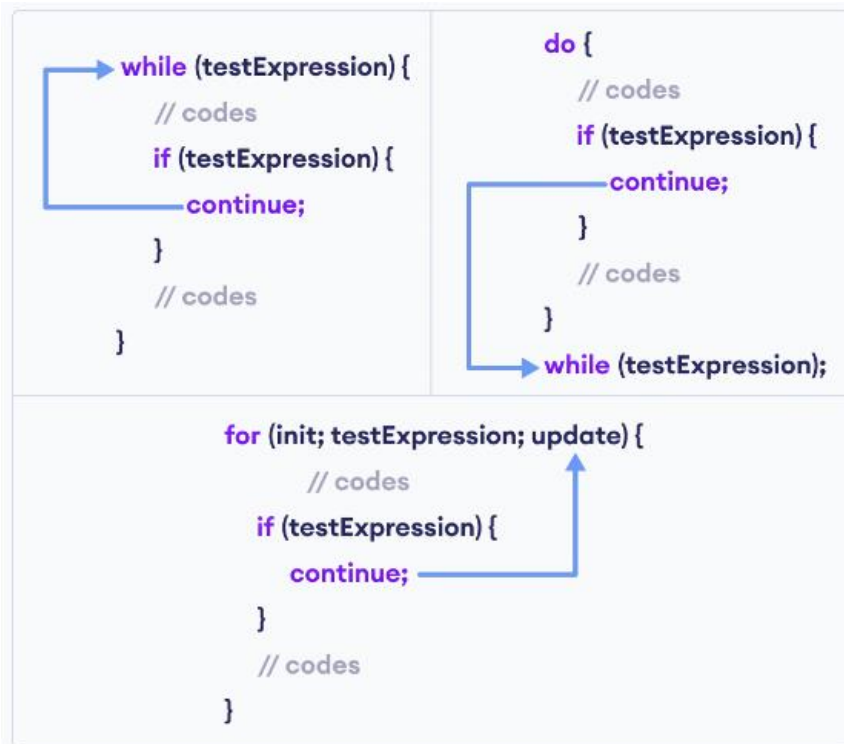
The continue statement skips the current iteration of **a loop (for, while, do...while, etc)**. After the **continue statement**, the program moves to the end of the loop. And, the test expression is then evaluated (update statement is evaluated in case of the for loop).

**the syntax of the continue statement.**

**continue;**

NB: The continue statement is always used in decision-making statements (if...else Statement).

### Working of Java continue statement



Working of continue statement with the Java **while**, **do...while**, and **for** loop.

### Example 1: Java continue statement

```

class Main {
    public static void main(String[] args) {
        // for loop
        for (int i = 1; i <= 10; ++i) {
            // if value of i is between 4 and 9
            // continue is executed
            if (i > 4 && i < 9) {
                continue;
            }
            System.out.println(i);
        }
    }
}

```

```

if (i > 4 && i < 9) {
    continue;
}

```

In the above program, we are using the **for loop** to print the value of *i* in each iteration. Notice the next statement **if (i > 4 && i < 9) {.....**

Here, the continue statement is executed when the value of *i* becomes more than 4 and less than 9. It then skips the print statement for those values. Hence, the output skips the values 5, 6, 7, and 8.

**Example 2: Compute the sum of 5 positive numbers**  
**import java.util.Scanner;**



```

class Main {
    public static void main(String[] args) {
        Double number, sum = 0.0;
        // create an object of Scanner
        Scanner input = new Scanner(System.in);
        for (int i = 1; i < 6; ++i) {
            System.out.print("Enter number " + i + " : ");
            // takes input from the user
            number = input.nextDouble();
            // if number is negative
            // continue statement is executed
            if (number <= 0.0) {
                continue;
            }
            sum += number;
        }
        System.out.println("Sum = " + sum);
        input.close();
    }
}

```

In the above example, we have used the for loop to print the sum of 5 positive numbers. Notice the line,

```

    if (number < 0.0) {
        continue;
    }

```

Here, when the user enters a negative number, the continue statement is executed. This skips the current iteration of the loop and takes the program control to the update expression of the loop. Note: To take input from the user, we have used the Scanner object.

## Java continue with Nested Loop

In the case of **nested loops** in Java, the **continue statement** skips the current iteration of the innermost loop. The continue statement skips the innermost loop while working with the nested loop in Java.

### Working of Java continue statement with Nested Loops

```

while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (testExpression) {
            continue;
        }
        // codes
    }
    // codes
}

```



Working of Java continue statement with Nested Loops

### Example 3: continue with Nested Loop

```

class Main {
    public static void main(String[] args) {
        int i = 1, j = 1;
        // outer loop
        while (i <= 3) {
            System.out.println("Outer Loop: " + i);

            // inner loop
            while(j <= 3) {
                if(j == 2) {
                    j++;
                    continue;
                }
                System.out.println("Inner Loop: " + j);
                j++;
            }
            i++;
        }
    }
}

```

In the above example, we used the **nested while loop**. Note that we have used the continue statement inside the inner loop.

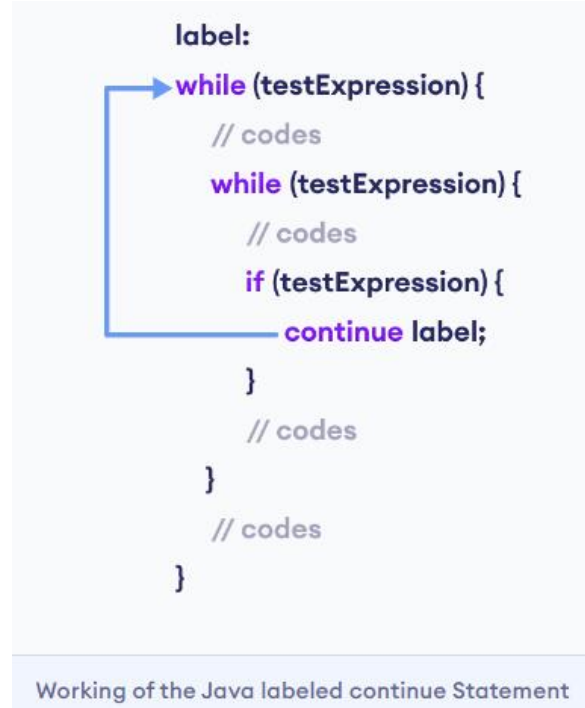
## Labeled continue Statement

There is another form of continue statement in Java known as labeled continue. It includes the **label** of the loop along with the continue keyword. For example,

**continue label;**

Here, the continue statement skips the current iteration of the loop specified by label.

### Working of the Java labeled continue Statement



We can see that the **label identifier** specifies the outer loop. Notice the use of the continue inside the inner loop. Here, the continue statement is skipping the current iteration of the labeled statement (i.e. outer loop). Then, the program control goes to the next iteration of the labeled statement.

### Example 4: labeled continue Statement

```
class Main {
    public static void main(String[] args) {
        // outer loop is labeled as first
        first:
        for (int i = 1; i < 6; ++i) {
            // inner loop
            for (int j = 1; j < 5; ++j) {
                if (i == 3 || j == 2)
                    // skips the current iteration of outer loop
                    continue first;
            }
        }
    }
}
```

```
        System.out.println("i = " + i + "; j = " + j);  
    }  
} } }
```

In the above example, the labeled continue statement is used to skip the current iteration of the loop labeled as first.

Note: The use of labeled continue is often discouraged as it makes your code hard to understand. If you are in a situation where you have to use labeled continue, refactor your code and try to solve it in a different way to make it more readable.