

# **Concepts of Programming**

## **PG-DAC Sept 2022**

**Gaurav Rajput**

# Types of Computer Programming Languages

There are basically two types of computer programming languages given below:

- 1.Low level language
- 2.High level language

## Low Level Languages

The programming languages that are very close to machine code (0s and 1s) are called low-level programming languages.

The program instructions written in these languages are in binary form.

The examples of low-level languages are:

- Machine language
- Assembly language

## Machine Language

The instructions in binary form, which can be directly understood by the computer ([CPU](#)) without translating them, is called a machine language or machine code.

Machine language is also known as first generation of programming language. Machine language is the fundamental language of the computer and the program instructions in this language is in the binary form (that is 0's and 1's).

## Assembly Language

It is another low-level programming language because the program instructions written in this language are close to machine language. Assembly language is also known as second generation of programming language.

With assembly language, a programmer writes instructions using symbolic instruction code instead of binary codes.

Symbolic codes are meaningful abbreviations such as SUB is used for subtraction operation, MUL for multiply operation and so on. Therefore this language is also called the low-level symbolic language.

The set of program instructions written in assembly language are also called as mnemonic code.

Assembly language provides facilities for controlling the hardware.

## High Level Languages

The programming languages that are close to human languages (example like English languages) are called the high-level languages.

The examples of high-level languages are:

- Fortran
- COBOL
- Basic
- Pascal
- C
- C++
- Java

The high level languages are similar to English language. The program instructions are written using English words, for example print, input etc. But each high level language has its own rule and grammar for writing program instructions. These rules are called syntax of the language.

The program written in high level language must be translated to machine code before to run it. Each high level language has its own translator program.

The high level programming languages are further divided into:

- Procedural languages
- Non procedural languages
- Object oriented programming languages

## **Procedural Language**

Procedural languages are also known as third generation languages (3GLs). In a procedural language, a program is designed using procedures.

A procedure is a sequence of instructions having a unique name. The instructions of the procedure are executed with the reference of its name.

In procedural programming languages, the program instructions are written in a sequence or in a specific order in which they must be executed to solve a specific problem. It means that the order of program instructions is very important.

Some popular procedural languages are described below:

C language - Dennis Ritchie and Brian Kernighan developed it in 1972 at Bell Laboratories. It is a high level language but it can also support assembly language codes (low level codes

## Non Procedural Languages

Non procedural programming languages are also known as fourth generation languages. In non procedural programming languages, the order of program instructions is not important. The importance is given only to, what is to be done.

With a non procedural language, the user/programmer writes English like instructions to retrieve data from databases. These languages are easier to use than procedural languages. These languages provide the user-friendly program development tools to write instructions. The programmers have not to spend much time for coding the program. The most important non procedural languages and tools are discussed below:

- SQL** - it stands for structured query language. it is very popular database access language and is specially used to access and to manipulate the data of databases. The word query represents that this language is used to make queries (or enquiries) to perform various operations on data of database. However, SQL can also be used to **create tables**, **add data**, **delete data**, **update data** of database tables etc.

## Object Oriented Programming Languages

The object oriented programming concept was introduced in the late 1960s, but now it has become the most popular approach to develop software.

In object oriented programming, the software is developed by using a set of interfacing object. An object is a component of program that has a set of modules and data structure. The modules are also called methods and are used to access the data from the object. The modern technique to design the program is object oriented approach. It is a very easy approach, in which program designed by using objects. Once an object for any program designed, it can be re-used in any other program.

Now-a-days, most popular and commonly used object oriented programming (OOPs) languages are C++ and Java.

## **Computer Language Translator and its Types**

Translator is a computer program that translates program written in a given programming language into a functionally equivalent program in a different language.

Depending on the translator, this may involve changing or simplifying the program flow without losing the essence of the program, thereby producing a functionally equivalent program.

### **Types of Language Translator**

There are mainly three Types of translators which are used to translate different programming languages into machine equivalent code:

- 1.Assembler
- 2.Compiler
- 3.Interpreter



## Assembler

An assembler translates assembly language into machine code.

Assembly language consists of Mnemonics for machine Op-codes, so assemblers perform a 1:1 translation from mnemonic to a direct instruction. For example, LDA #4 converts to 0001001000100100.

Conversely, one instruction in a high level language will translate to one or more instructions at machine level

## Compiler

Compiler is a computer program that translates code written in a high level language to a low level language, object/machine code.

The most common reason for translating source code is to create an executable program (converting from high level language into machine language).

## Interpreter

An interpreter program execute other programs directly, running through program code and executing it line-by-line.

As it analyses every line, an interpreter is slower than running compiled code but it can take less time to interpret program code than to compile and then run it. This is very useful when prototyping and testing code.

Interpreters are written for multiple platforms, this means code written once can be run immediately on different systems without having to recompile for each. Examples of this include flash based web programs that will run on

# Java Overview

## Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all the Java programs are to at least some extent object oriented. OOP is thus integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.

## Two Paradigms

All computer programs consist of the two elements i.e., code and data. Moreover, a program can be organized around its code or around its data in a conceptual manner i.e., some programs are written around "what's going on ?" and other programs are written about "who's being affected ?". These are the two paradigms that govern how a program is constructed. The first style is called as the process-oriented-model. This approach characterized a program as a series of one-dimensional steps i.e., code. The process-oriented model can be supposed of as code acting on data.

To manage increasing complexity, the second approach is called as object-oriented-programming, was conceived. Object-oriented programming organizes a program around its data, objects and a set of well defined interfaces to that data. An object-oriented program can be described as data controlling access to the code.

## The Three OOP Principles

All the object-oriented programming languages provide mechanisms which helps you to implement the object-oriented model. Examples are encapsulation, inheritance, and polymorphism.

### Encapsulation

**Encapsulation** is the process which binds together the code and the data it manipulates, and keeps both safe from the outside interference and misuse. One way to guess about encapsulation is as a protective wrapper that prevents the code and the data from being arbitrarily accessed by the other code that defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled by a well-defined interface.

### Inheritance

**Inheritance** is the process by which one object adopts the properties of another objects. This is important because it supports the concept of hierarchical classification.

### Polymorphism

**Polymorphism** (from Greek, meaning "many forms") is a characteristic which allows one **interface** to be used for a general class of actions. The particular action is decided by the exact nature of the situation.

## A First Simple Program

Now that the basic of object-oriented support of Java has been discussed. Now let's look at the simple [Java program](#):

```
/* Java Overview - Example Program */

public class JavaProgram
{
    /* A Java Program Begins with a call to main() */

    public static void main(String args[])
    {

        System.out.println("This is a Simple Java Program.");

    }
}
```

## Java Program Structure

Before proceed to learn Java, it is important to understand the basic structure of a Java program. To know about the structure of a Java Program, just look at the following simple example:

```
/* Java Program Example - Java Program Structure
```

```
* This is a simple Java program, prints
```

```
* "Hello World" on the Screen
```

```
*/
```

```
public class JavaProgram
```

```
{
```

```
    /* This
```

```
    * is
```

```
    * a
```

```
    * multi-line
```

```
    * comments
```

```
    */
```

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

```
    {
```

```
        // single line comments
```

```
        System.out.println("Hello World");
```

```
    }
```

```
}
```

Let's explain the above Java Program:

•**public class JavaProgram:** This line of code has three parts:

- **public** : This is an access modifier keyword, tells the compiler to access the class. Various values of access modifiers are public, protected, private or default (no value).
- **class** : This keyword is used to declare the class. Name of the class (here JavaProgram) followed by this keyword.
- **JavaProgram** : This is the name of the class. You can give this name according to your demand/program.

•**Comment Section** : You can write comments in two ways:

- **Single Line Comments** : It start with two forward slashes i.e. `//` and continue to the end of the current line. Line comments do not require an ending symbol.
- **Multi Line Comments** : Also called as block comments start with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`).Block comments can also extend across as many lines as needed.

• **public static void main (String args[])** : This line of code has three keywords, main, and string arguments:

- **public** : Access Modifier
- **static** : static is reserved keyword which means that a method is accessible and usable even though no objects of the class exist.
- **void** : This keyword declares nothing would be returned from method. Method can return any primitive or object
- **main** : This is the main where program starts running or from here our compiler starts checking and running our java program/code.
- **Arguments** : This is for the arguments purposes which will be included inside the curly bracket.

**System.out.println("Hello World")**: This line of code has also four parts:

**System** : It is the name of Java utility class.

**out** : It is an object which belongs to System class used in sending the data which is inside the bracket to the console.

**println** : It is print line which is utility method name which is used to send any String to console.

Hello World : It is String literal set as argument to println method

## Java Basic Syntax

For Java programmer, it is very important to keep in mind about the following points.

- Case Sensitivity** - Java is a case sensitive language, which means that the identifier Hello, hello, HeLLo, hElLo, helLo, HELLO. All are different in Java.

- Method Names** - All the method names should start with a Lower Case letter.

If several words are used to form the name of the method, then each first letter of inner word should be in Upper Case. [Lower camelCase] **Example:** public void employeeRecords(), public void myMethodName(), public void employeeNumber() etc.

- Class Names** - For all class names, the first letter should be in Uppercase

If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

[Upper Camel Case ] **Examples:** class MyJavaProgram, class MyThirdJavaProgram, class StudentRecords etc.

- public static void main(String args[]):** Java program processing starts from the method main() which is a mandatory part of every Java program.



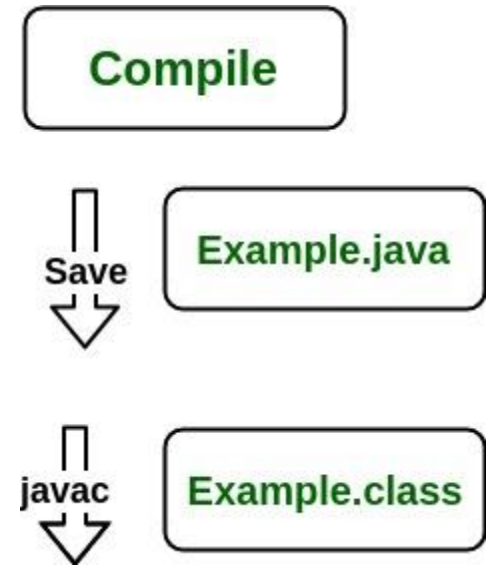
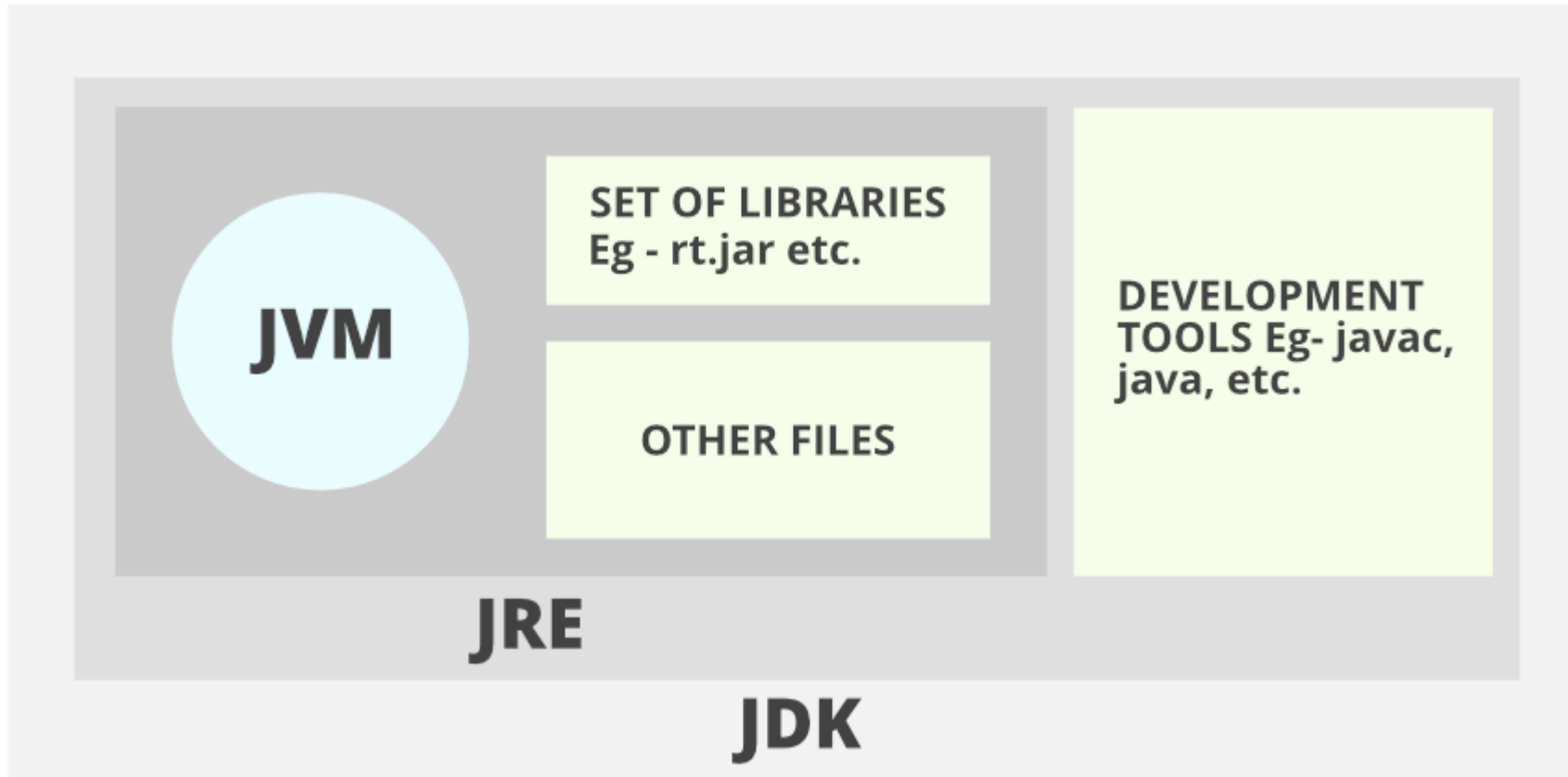
## What is JVM?

- 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 bytecode into native machine code (set of instructions that a computer's CPU executes directly).
- Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.

## Java JDK, JRE and JVM



Working of Java Program



1. JDK (Java Development Kit) is a Kit that provides the environment to develop and execute(run) the Java program.

JDK is a kit(or package) that includes two things

- Development Tools(to provide an environment to develop your java programs)
- JRE (to execute your java program).

2. JRE (Java Runtime Environment) is an installation package that provides an environment to only run(not develop) the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

3. JVM (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.

# Java Primitive Data Types

- 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.

Java defines the following eight primitive data types:

- 1.boolean
- 2.byte
- 3.short
- 4.int
- 5.long
- 6.Double
- 7.float
- 8.Char

The primitive data types are also commonly referred to as simple types. These can be put in the following four groups :

- 1.Integers - This group included byte, short, int, and long, which are for whole-valued signed numbers.
- 2.Floating-Point numbers - This group includes float and double, these represent the numbers with fractional precision.
- 3.Characters - This group includes char, which represents the symbols in a character set, like letters and numbers.
- 4.Booleans - This group includes boolean, which is a special type for representing the true/false values.

## 1. boolean type

- The `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  
    }  
}
```



Run Code >>

## 2. byte type

- The `byte` data type can have values from **-128** to **127** (8-bit signed two's complement integer).
- If it's certain that the 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  
    }  
}
```



Run Code >>

### 3. short type

- The `short` data type in Java can have values from **-32768** to **32767** (16-bit signed two's complement integer).
- If it's certain that the value of a variable will be within -32768 and 32767, then it is used instead of 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  
    }  
}
```



Run Code >>

## 4. int type

- The `int` data type can have values from  $-2^{31}$  to  $2^{31}-1$  (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  $2^{32}-1$ . To learn more, visit [How to use the unsigned integer in java 8?](#)
- 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  
    }  
}
```



Run Code >>



## 5. long type

- The `long` data type can have values from  $-2^{63}$  to  $2^{63}-1$  (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  $2^{64}-1$ .
- Default value: 0

### Example 5: Java long data type

```
class LongExample {  
    public static void main(String[] args) {  
  
        long range = -42332200000L;  
        System.out.println(range);    // prints -42332200000  
    }  
}
```



Run Code >>

Notice, the use of `L` at the end of `-42332200000`. This represents that it's an integer of the `long` type.

## 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  
    }  
}
```



Run Code >>

## 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  
    }  
}
```



Run Code »

## 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  
    }  
}
```

Run Code »

Here, the Unicode value of `Q` is `\u0051`. Hence, we get `Q` as the output.

Here is another example:

```
class Main {  
    public static void main(String[] args) {  
  
        char letter1 = '9';  
        System.out.println(letter1); // prints 9  
  
        char letter2 = 65;  
        System.out.println(letter2); // prints A  
  
    }  
}
```



Run Code »

Here, we have assigned `9` as a character (specified by single quotes) to the `letter1` variable. However, the `letter2` variable is assigned `65` as an integer number (no single quotes).

Hence, `A` is printed to the output. It is because Java treats characters as an integer and the ASCII value of `A` is 65. To learn more about ASCII, visit [What is ASCII Code?](#).

## 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. To learn more, visit [Java Strings](#).

## Java Variables and Literals:

Java Variables: A variable is a location in memory (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

### Identifiers in Java

In Java, an identifiers are used to name things, such as classes, variables, and methods.

An identifier may be any sequence of uppercase and lowercase letters, number or the underscore and dollar-sign characters. As you know, Java is case-sensitive, therefore, **VALUE** is a different identifier than **Value**.

Following are the rules to declare Identifiers in Java:

- All identifiers can begin with a letter (A to Z or a to z) or dollar currency character (\$) or an underscore (\_).
- After the first character identifiers can have any combination of characters.
- Most importantly identifiers are case-sensitive.
- A keyword cannot be used as an identifier since it has reserved words and have some special meaning.
- Examples of illegal identifiers: 123abc, -salary etc.
- Examples of legal identifiers: AvgTemp, count, a4, this\_is\_ok, age, \$salary, \_value, \_\_1\_value, customers etc

## Java Literals

In Java, a constant value is created by using a *literal* representation of it. For example, below are some literals :

**100** - specifies an integer literal

•**98.6** - specifies a floating-point literal

•**'X'** - specifies a character constant

•**"This is a test"** - this literal specifies a string literal

A literal can be used anywhere a value of its type is allowed.

## Types of Literals

In Java, there are the following types of literals:

•Boolean Literals

•Integer Literals

•Floating-point Literals

•Character Literals

•String Literals



## 1. Boolean Literals

In Java, boolean literals are used to initialize boolean data types. They can store two values: true and false. For example,

```
boolean flag1 = false;  
boolean flag2 = true;
```

Here, `false` and `true` are two boolean literals.

## 2. Integer Literals

An integer literal is a numeric value(associated with numbers) without any fractional or exponential part. There are 4 types of integer literals in Java:

1. binary (base 2)
2. decimal (base 10)
3. octal (base 8)
4. hexadecimal (base 16)

For example:

```
// binary
int binaryNumber = 0b10010;
// octal
int octalNumber = 027;

// decimal
int decNumber = 34;

// hexadecimal
int hexNumber = 0x2F; // 0x represents hexadecimal
// binary
int binNumber = 0b10010; // 0b represents binary
```

### 3. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponential form. For example,

```
class Main {  
    public static void main(String[] args) {  
  
        double myDouble = 3.4;  
        float myFloat = 3.4F;  
  
        // 3.445*10^2  
        double myDoubleScientific = 3.445e2;  
  
        System.out.println(myDouble); // prints 3.4  
        System.out.println(myFloat); // prints 3.4  
        System.out.println(myDoubleScientific); // prints 344.5  
    }  
}
```



Run Code >>

## 4. Character Literals

Character literals are unicode character enclosed inside single quotes. For example,

```
char letter = 'a';
```

Here, `a` is the character literal.

We can also use escape sequences as character literals. For example, `\b` (backspace), `\t` (tab), `\n` (new line), etc.

---

## 5. String literals

A string literal is a sequence of characters enclosed inside double-quotes. For example,

```
String str1 = "Java Programming";  
String str2 = "Programiz";
```

Here, `Java Programming` and `Programiz` are two string literals.

# Java Type Casting

The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting.

We will only focus on the major 2 types of casting .

## 1. Widening Type Casting (Implicit Type Casting)

In the case of Widening Type Casting, the lower data type (having smaller size) is converted into the higher data type (having larger size). Hence there is no loss in data. This is why this type of conversion happens automatically.

## 2. Narrowing Type Casting (Explicit Type Casting)

In the case of Narrowing Type Casting, the higher data types (having larger size) are converted into lower data types (having smaller size). Hence there is the loss of data. This is why this type of conversion does not happen automatically.

# Widening Type Casting

In **Widening Type Casting**, Java automatically converts one data type to another data type.



## Example: Converting int to double

```
class Main {  
    public static void main(String[] args) {  
        // create int type variable  
        int num = 10;  
        System.out.println("The integer value: " + num);  
  
        // convert into double type  
        double data = num;  
        System.out.println("The double value: " + data);  
    }  
}
```



Run Code >>

## Output

```
The integer value: 10  
The double value: 10.0
```

In the above example, we are assigning the `int` type variable named `num` to a `double` type variable named `data`.

# Narrowing Type Casting

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

## Example: Converting double into an int

```
class Main {  
    public static void main(String[] args) {  
        // create double type variable  
        double num = 10.99;  
        System.out.println("The double value: " + num);  
  
        // convert into int type  
        int data = (int)num;  
        System.out.println("The integer value: " + data);  
    }  
}
```



Run Code >>

## Output

```
The double value: 10.99  
The integer value: 10
```

## Example 1: Type conversion from int to String

```
class Main {  
    public static void main(String[] args) {  
        // create int type variable  
        int num = 10;  
        System.out.println("The integer value is: " + num);  
  
        // converts int to string type  
        String data = String.valueOf(num);  
        System.out.println("The string value is: " + data);  
    }  
}
```



Run Code >>

### Output

```
The integer value is: 10  
The string value is: 10
```



## Example 2: Type conversion from String to int

```
class Main {  
    public static void main(String[] args) {  
        // create string type variable  
        String data = "10";  
        System.out.println("The string value is: " + data);  
  
        // convert string variable to int  
        int num = Integer.parseInt(data);  
        System.out.println("The integer value is: " + num);  
    }  
}
```



Run Code >>

### Output

```
The string value is: 10  
The integer value is: 10
```

# Java Operators

Operators are symbols that perform operations on variables and values. For example, + 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
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Unary 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
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Modulo Operation (Remainder after division)

## Example 1: Arithmetic Operators

```
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));  
    }  
}
```



Run Code >>

## 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`.

Let's see some more assignment operators available in Java.

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

## Example 2: Assignment Operators

```
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);  
    }  
}
```



Run Code >>

## Output

```
var using =: 4  
var using +=: 8  
var using *=: 32
```

### 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>

### Example 3: Relational Operators

```
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 an 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

## Example 4: Logical Operators

```
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  
    }  
}
```

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

## Example 5: 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);  
    }  
}
```

## Output

```
Value of a: 12  
After increment: 13  
Value of b: 12  
After decrement: 11
```

## 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
<code>~</code>	Bitwise Complement
<code>&lt;&lt;</code>	Left Shift
<code>&gt;&gt;</code>	Right Shift
<code>&gt;&gt;&gt;</code>	Unsigned Right Shift
<code>&amp;</code>	Bitwise AND
<code>^</code>	Bitwise exclusive OR

# Conditional and Looping Statements

## Java if...else Statement

In programming, we use the if..else statement to run a block of code among more than one alternatives.

For example, assigning 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, `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 of if Statement


### Condition is true

```
int number = 10;  
  
if (number > 0) {  
    // code  
}  
  
// code after if
```



### Condition is false

```
int number = 10;  
  
if (number < 0) {  
    // code  
}  
  
// code after if
```



## 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");  
    }  
}
```

Run Code >>

## Output

```
Statement outside if block
```

In the program, `number < 0` is `false`. Hence, the code inside the body of the `if` statement is **skipped**.



## 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`. However, if the test expression is evaluated to `false`, it does nothing.

In this case, we can use an optional `else` block. Statements inside the body of `else` block are executed if the 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 `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?

### Condition is true

```
int number = 5;
```

```
if (number > 0) {  
    // code  
}  
else {  
    // code  
}  
// code after if...else
```



### Condition is false

```
int number = 5;
```

```
if (number < 0) {  
    // code  
}  
else {  
    // code  
}  
// code after if...else
```



Working of Java if-else statements

### 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 number is not greater than 0  
        else {  
            System.out.println("The number is not positive.");  
        }  
  
        System.out.println("Statement outside if...else block");  
    }  
}
```

Run Code >>

### Output

```
The number is positive.  
Statement outside if...else block
```

In the above example, we have a variable named `number`. Here, the test expression

`number > 0` checks if `number` is greater than 0.

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

In Java, we have an **if...else...if** ladder, that can be used to execute one block of code among multiple other blocks.

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


Here, `if` statements are executed from the top towards the bottom. When the test condition is `true`, codes inside 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?


## 1st Condition is true

```
int number = 2;  
if (number > 0) {  
    // code  
}  
else if (number == 0){  
    // code  
}  
else {  
    //code  
}  
//code after if
```



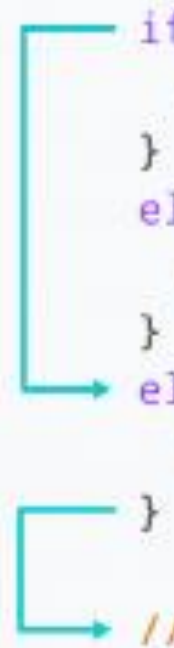
## 2nd Condition is true

```
int number = 0;  
if (number > 0) {  
    // code  
}  
else if (number == 0){  
    // code  
}  
else {  
    //code  
}  
//code after if
```



## All Conditions are false

```
int number = -2;  
if (number > 0) {  
    // code  
}  
else if (number == 0){  
    // code  
}  
else {  
    //code  
}  
//code after if
```



## 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.");  
        }  
    }  
}
```

Run Code >>

## Output

The number is 0.

# Java switch Statement

The switch statement allows us to execute a block of code among many alternatives.

The syntax of the `switch` statement in Java is:

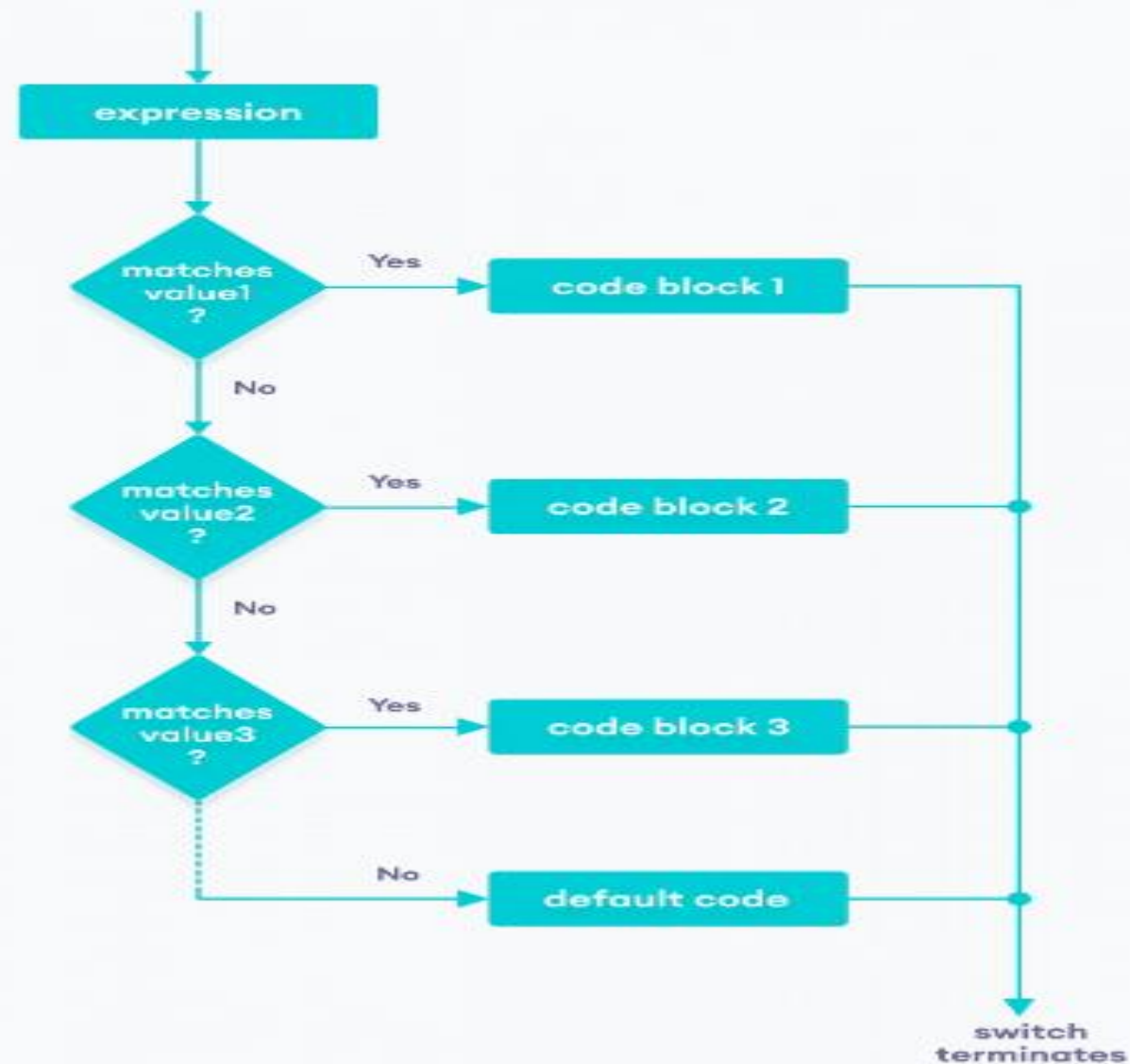
```
switch (expression) {  
  
    case value1:  
        // code  
        break;  
  
    case value2:  
        // code  
        break;  
  
    ...  
    ...  
  
    default:  
        // default statements  
}
```

## 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. Similarly, 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.

# Flowchart of switch Statement



Flow chart of the 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);  
    }  
}
```

Output:

Size: Large

## break statement in Java switch...case

Notice that we have been using `break` in each case block.

```
...
case 29:
    size = "Small";
    break;
...
```

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");
        }
    }
}
```

### Output

```
Case 2
Case 3
Default case
```

## Java for Loop

In computer programming, loops are used to repeat a block of code. For example, 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

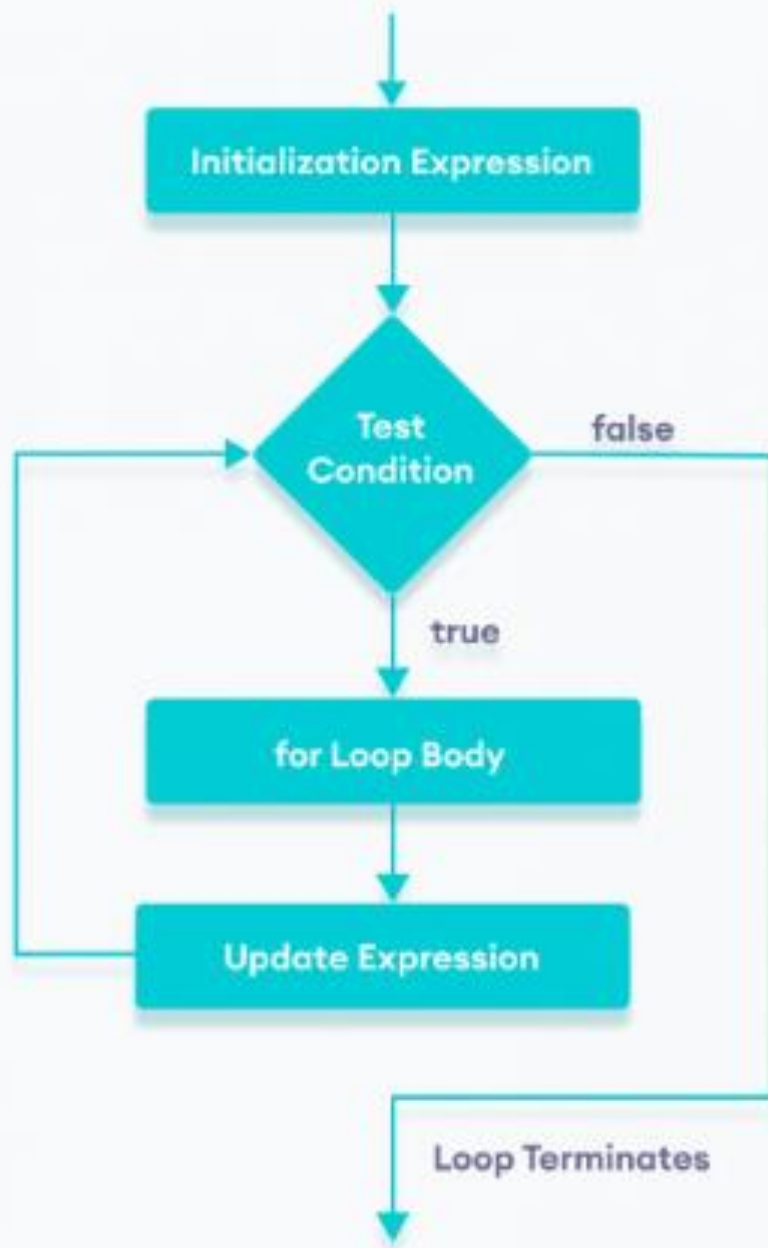
# Java for Loop

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

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

Here,

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



Flowchart of Java for loop

## Example 1: Display a 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("Java is fun");
        }
    }
}
```

## Output

```
Java is fun
Java is fun
Java is fun
Java is fun
Java is fun
```

## Example 2: Display numbers from 1 to 5

Here is how the program works.

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

Output

1  
2  
3  
4  
5

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

## 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");
        }
    }
}
```

Run Code >>

Here, the test expression, `i <= 10`, is never `false` and `Hello` is printed repeatedly until the memory runs out.

# 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:

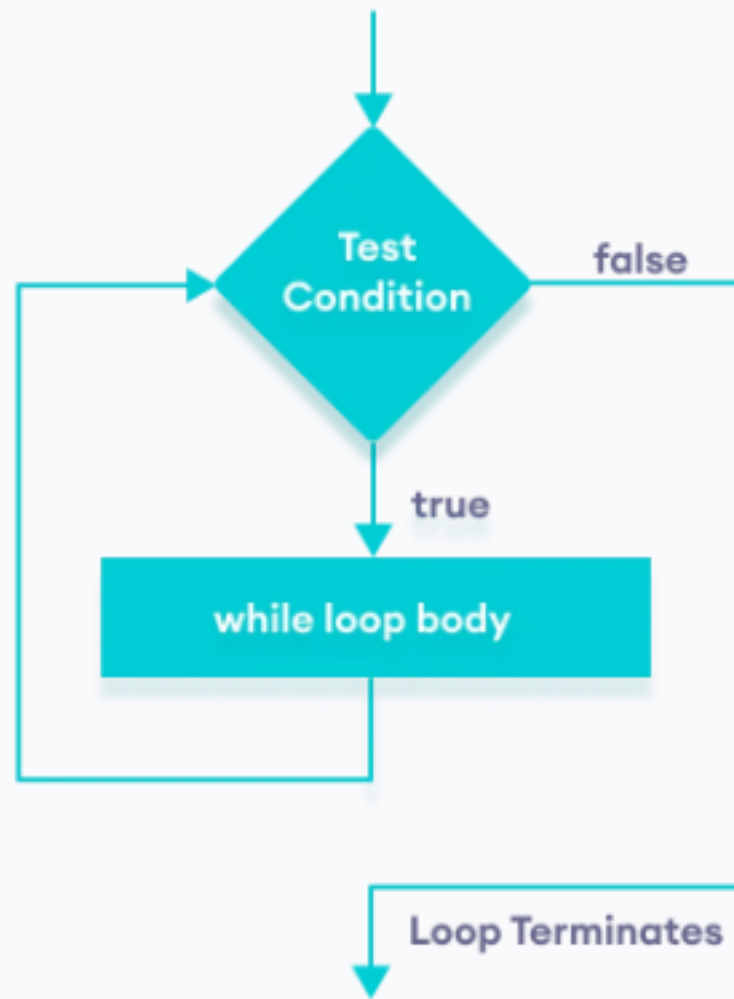
```
while (testExpression) {  
    // body of loop  
}
```

Here,

1. A `while` loop evaluates the **testExpression** inside the parenthesis `()`.
2. If the **testExpression** evaluates to `true`, the code inside the `while` loop is executed.
3. The **testExpression** is evaluated again.
4. This process continues until the **testExpression** is `false`.
5. When the **testExpression** evaluates to `false`, the loop stops.



## Flowchart of while loop



Flowchart of Java while loop

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

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

Here is how this program works.

```
// 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++;
        }
    }
}
```

Iteration	Variable	Condition: i <= 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

### Output

```
1
2
3
4
5
```

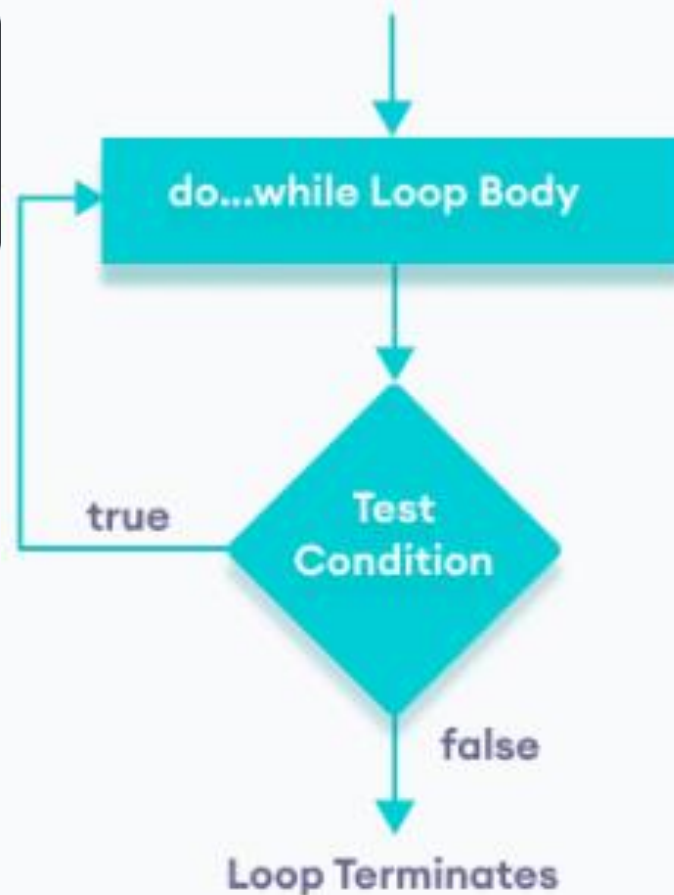
# Java do...while loop

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

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

Here,

1. The body of the loop is executed at first. Then the **textExpression** is evaluated.
2. If the **textExpression** evaluates to `true`, the body of the loop inside the `do` statement is executed again.
3. The **textExpression** is evaluated once again.
4. If the **textExpression** evaluates to `true`, the body of the loop inside the `do` statement is executed again.
5. This process continues until the **textExpression** evaluates to `false`. Then the loop stops.



Flowchart of Java do while loop

### 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);
    }
}
```

### Output

```
1
2
3
4
5
```

Here is how this program works.

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

## 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 programs, the **textExpression** is always `true`. Hence, the loop body will run for infinite times.


## Java break Statement

While working with loops, it is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

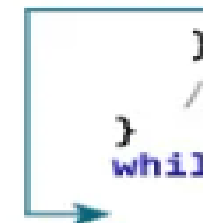
In such cases, break and continue statements are used

### How break statement works?


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



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



```
for (init; testExpression; update) {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}
```



## 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);  
        }  
    }  
}
```

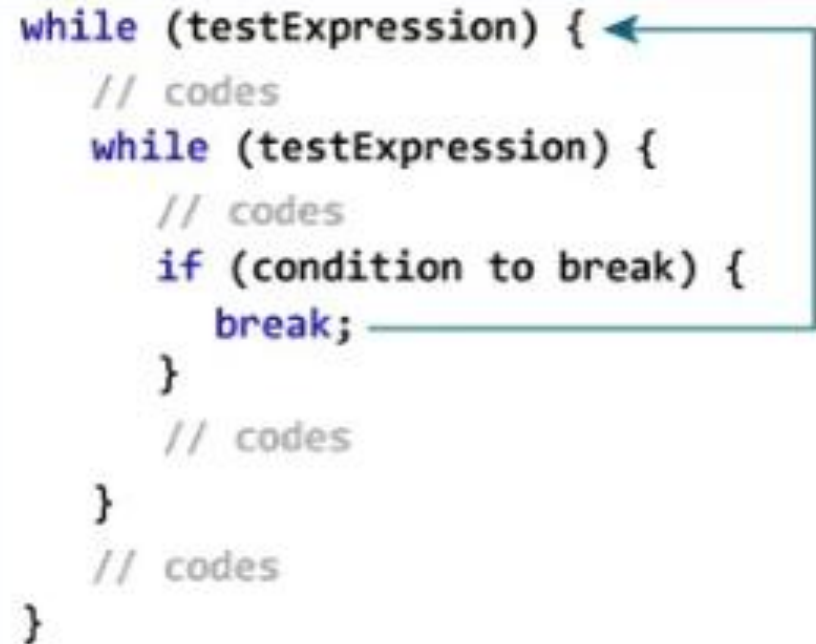
Output:

1  
2  
3  
4

## Java break and Nested Loop

In the case of **nested loops**, the `break` statement terminates the innermost loop.

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



Working of break Statement with Nested Loops

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

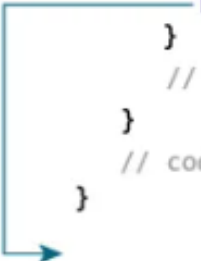


## Labeled break Statement

Till now, we have used the unlabeled break statement. It terminates the innermost loop and switch statement. However, 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  
}
```

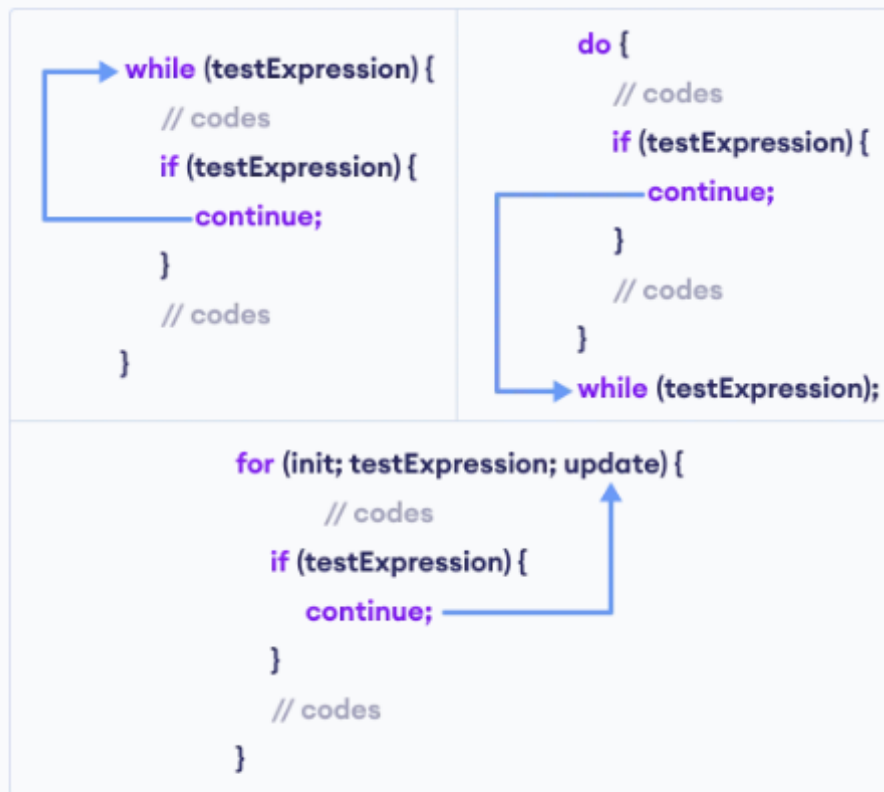
In the above example, when the statement `break second;` is executed, the `while` loop labeled as `second` is terminated. And, the control of the program moves to the statement after the second `while` loop.

# 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, test expression is evaluated (update statement is evaluated in case of the for loop).

## Working of Java continue statement



Working of Java continue Statement

## 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);  
        }  
    }  
}
```

Output:

1  
2  
3  
4  
9  
10

# Java continue with Nested Loop

In the case of [nested loops in Java](#), the `continue` statement skips the current iteration of the innermost loop.

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



## Labeled continue Statement

Till now, we have used the unlabeled `continue` statement. However, 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

## 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);
            }
        }
    }
}
```

### Output:

```
i = 1; j = 1
i = 2; j = 1
i = 4; j = 1
i = 5; j = 1
```