Java
Subject code: BTIT503-18
Module 1

Java virtual machine and its architecture

Java virtual machine, or JVM, loads, verifies, and runs Java bytecode. It is known as the interpreter or the core of the Java programming language because it runs Java programming.

The role of JVM in Java

JVM is responsible for converting bytecode to machine-specific code and is necessary in both JDK and JRE. It is also platform-dependent and performs many functions, including memory management and security. In addition, JVM can run programs that are written in other programming languages that have been converted to Java bytecode.

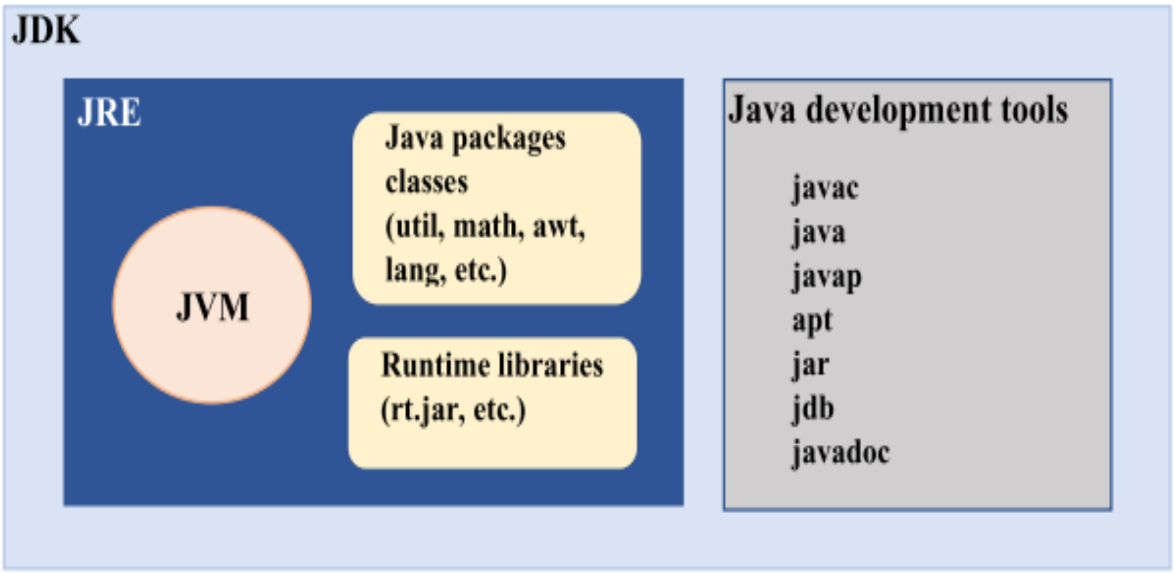
Java Native Interface (JNI) is often referred to in connection with JVM. JNI is a programming framework that enables Java code running in JVM to communicate with (that is, to call and be called by) applications that are associated with a piece of hardware and specific operating system platform. These applications are called native applications and can often be written in other languages. Native methods are used to move native code written in other languages into a Java application.

JVM components

JVM consists of three main components or subsystems:

- **Class Loader Subsystem** is responsible for loading, linking, and initializing a Java class file (that is, "Java file"), otherwise known as dynamic class loading.
- **Runtime Data Areas** contain method areas, PC registers, stack areas, and threads.
- **Execution Engine** contains an interpreter, compiler, and garbage collection area.

Architecture of JVM



JDK (Java Development Kit)
The JDK is a complete software development kit used for developing Java applications. It includes:

- Java Development Tools: Tools for developing, debugging, and monitoring Java applications.
    - javac: The Java compiler, which translates Java source code into bytecode.
    - java: The launcher for executing Java applications.
    - javap: Disassembles class files.
    - apt: The annotation processing tool.
    - jar: Used for packaging Java classes into JAR files.
    - jdb: The Java debugger.
    - Java doc: Generates HTML documentation from Java source code.

2**. JRE (Java Runtime Environment)**
The JRE provides the libraries, Java Virtual Machine (JVM), and other components necessary for running Java applications. It does not include development tools like compilers or debuggers.

- JVM: The Java Virtual Machine is the heart of the Java programming language. It runs the Java bytecode and provides the runtime environment.
- Java Packages Classes: Essential Java packages (e.g., java.lang, java.util, java.math, etc.) included in the runtime environment.
- Runtime Libraries: Libraries required for running Java applications (e.g., rt.jar).

Difference Between JDK, JRE, and JVM

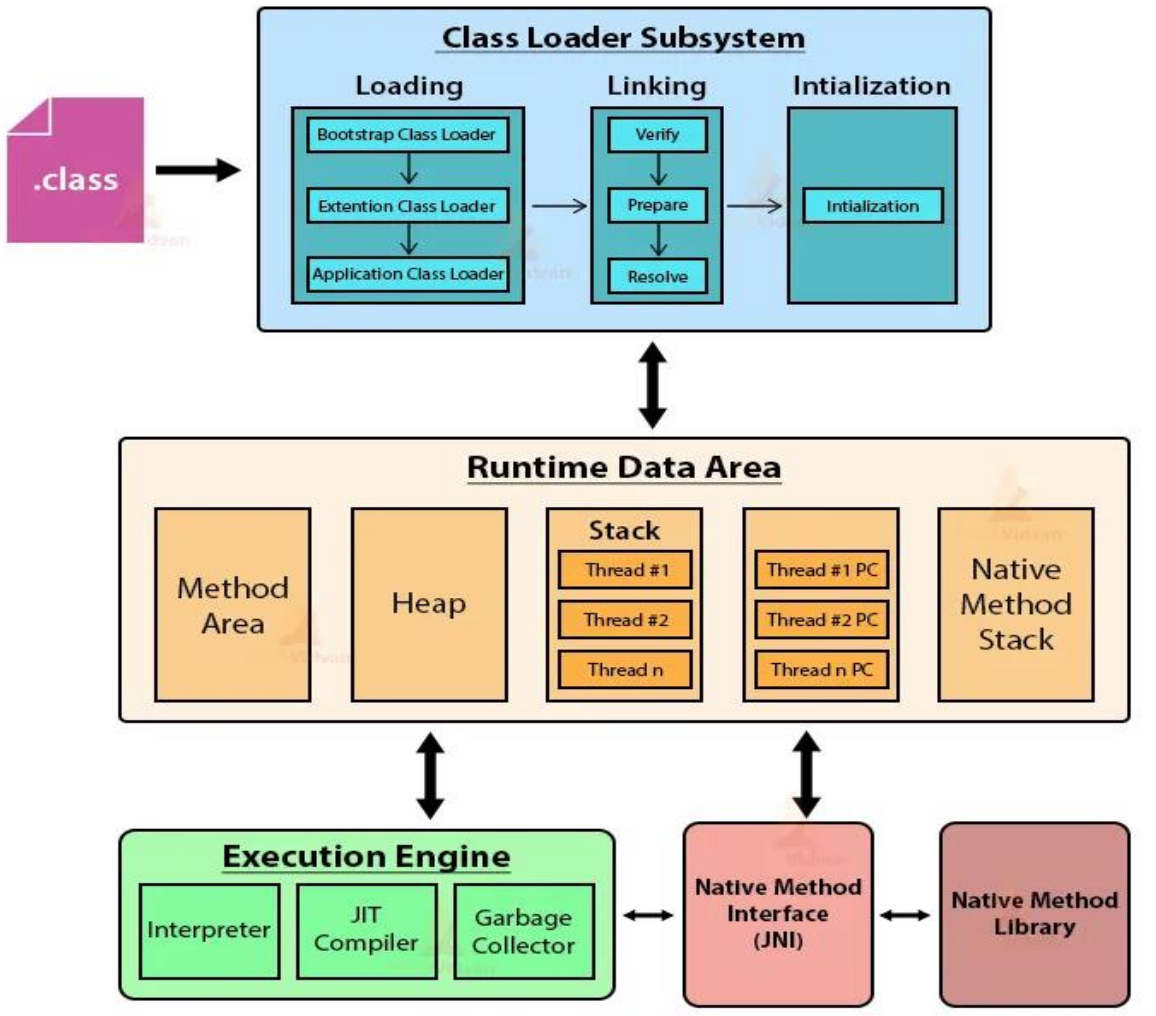
Parameter	JDK	JRE	JVM
Full-Form	The JDK is an abbreviation for Java Development Kit.	The JRE is an abbreviation for Java Runtime Environment.	The JVM is an abbreviation for Java Virtual Machine.
Definition	The JDK (Java Development Kit) is a software development kit that develops applications in Java. Along with JRE, the JDK also consists of various development tools (Java Debugger, JavaDoc, compilers, etc.)	The Java Runtime Environment (JRE) is an implementation of JVM. It is a type of software package that provides class libraries of Java, JVM, and various other components for running the applications written in Java programming.	The Java Virtual Machine (JVM) is a platform-independent abstract machine that has three notions in the form of specifications. This document describes the requirement of JVM implementation.
Functionality	The JDK primarily assists in executing codes. It	JRE has a major responsibility for creating	JVM specifies all of the implementations. It is

	primarily functions in development.	an environment for the execution of code.	responsible for providing all of these implementations to the JRE.
Platform Dependency	The JDK is platform-dependent. It means that for every different platform, you require a different JDK.	JRE, just like JDK, is also platform-dependent. It means that for every different platform, you require a different JRE.	The JVM is platform-independent. It means that you won't require a different JVM for every different platform.
Tools	Since JDK is primarily responsible for the development, it consists of various tools for debugging, monitoring, and developing java applications.	JRE, on the other hand, does not consist of any tool- like a debugger, compiler, etc. It rather contains various supporting files for JVM, and the class libraries that help JVM in running the program.	JVM does not consist of any tools for software development.
Implementation	JDK = Development Tools + JRE (Java Runtime Environment)	JRE = Libraries for running the application + JVM (Java Virtual Machine)	JVM = Only the runtime environment that helps in executing the Java bytecode.
Why Use It?	Why use JDK? Some crucial reasons to use JDK are: • It consists of various tools required for writing Java programs. • JDK also contains JRE for executing Java programs. • It includes an Appletviewer, Java application launcher, compiler, etc. • The compiler helps in converting the code written in	Why use JRE? Some crucial reasons to use JRE are: • If a user wants to run the Java applets, then they must install JRE on their system. • The JRE consists of class libraries along with JVM and its supporting files. It has no other tools like a compiler or a debugger for Java development. • JRE uses crucial package classes like util, math, awt, lang, and	Why use JVM? Some crucial reasons to use JVM are: • It provides its users with a platform-independent way for executing the Java source code. • JVM consists of various tools, libraries, and multiple frameworks. • The JVM also comes with a Just-in-Time (JIT) compiler for converting the Java source code into a low-

	Java into bytecodes. • The Java application launcher helps in opening a JRE. It then loads all of the necessary details and then executes all of its main methods.	various runtime libraries.	level machine language. Thus, it ultimately runs faster than any regular application. • Once you run the Java program, you can run JVM on any given platform to save your time.
Features	Features of JDK • Here are a few crucial features of JDK: • It has all the features that JRE does. • JDK enables a user to handle multiple extensions in only one catch block. • It basically provides an environment for developing and executing the Java source code. • It has various development tools like the debugger, compiler, etc. • One can use the Diamond operator to specify a generic interface in place of writing the exact one. • Any user can easily install JDK on Unix, Mac, and Windows OS (Operating Systems).	Features of JRE • Here are a few crucial features of JRE: • It is a set of tools that actually helps the JVM to run. • The JRE also consists of deployment technology. It includes Java Plug-in and Java Web Start as well. • A developer can easily run a source code in JRE. But it does not allow them to write and compile the concerned Java program. • JRE also contains various integration libraries like the JDBC (Java Database Connectivity), JNDI (Java Naming and Directory Interface), RMI (Remote Method Invocation), and many more.	Features of JVM Here are a few crucial features of JVM: • The JVM enables a user to run applications on their device or in a cloud environment. • It helps in converting the bytecode into machine-specific code. • JVM also provides some basic Java functions, such as garbage collection, security, memory management, and many more. • It uses a library along with the files given by JRE (Java Runtime Environment) for running the program. • Both JRE and JDK contain JVM.

		• It consists of the JVM and virtual machine client for Java HotSpot.	• It is easily customizable. For instance, a user can feasibly allocate a maximum and minimum memory to it. • JVM can also execute a Java program line by line. It is thus also known as an interpreter. • JVM is also independent of the OS and hardware. It means that once a user writes a Java program, they can easily run it anywhere.

Internal architecture of JVM



1. Class Loader Subsystem

This subsystem is responsible for loading class files, verifying them, preparing them for execution, and initializing them.

- Loading:
  - Bootstrap Class Loader: Loads core Java classes located in the rt.jar file.
  - Extension Class Loader: Loads classes from the Java Extensions directory.
  - Application Class Loader: Loads classes from the classpath (user-defined classes).
- Linking:
  - Verify: Ensures the correctness of the bytecode. This process checks for any illegal code that violates JVM specifications.
  - Prepare: Allocates memory for class variables and initializes them to default values.
  - Resolve: Converts symbolic references into direct references. This step is often delayed until runtime (lazy resolution).
- Initialization:
  - Executes the class's static initializers and initializers for static fields.

2. Runtime Data Area
This is where the JVM allocates memory for the execution of Java programs.

- Method Area: Stores class structures such as the runtime constant pool, field and method data, and the code for methods.
- Heap: The runtime data area from which memory for all class instances and arrays is allocated.
- Stack: Each thread has a private JVM stack, created at the same time as the thread. A stack stores frames, and each frame holds local variables, operand stacks, and references to the constant pool.
- Program Counter (PC) Register: Contains the address of the Java virtual machine instruction currently being executed.
- Native Method Stack: Contains all the native method information used in the application.

3. Execution Engine

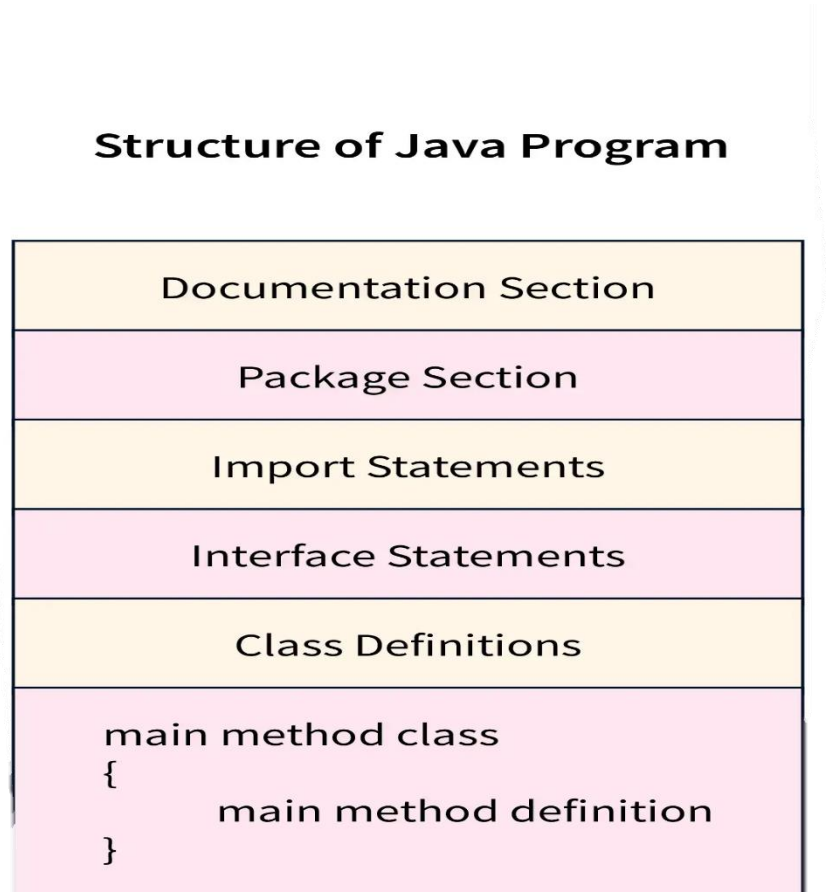
- Interpreter: Reads and executes bytecode line by line.
- JIT (Just-In-Time) Compiler: Compiles bytecode into native machine code at runtime for better performance.
- Garbage Collector: Automatically manages memory by deallocating objects that are no longer in use.

4. Native Method Interface (JNI)
Allows Java code running in the JVM to call and be called by native applications and libraries written in other languages like C or C++.

5. Native Method Library
Holds the native libraries required for execution of the native methods.

Program structure in java

```
import java.io.*;

public class Main{

public static void main(String[] args) throws Exception {

   System.out.println("Hello, Java!");
 }

}
```



Documentation Section

The documentation section in the structure of a Java program serves as a vital but optional part of a Java program, providing essential details about the program. This includes the author's name, creation date, version, program name, company name, and a brief description. While these details enhance the program's readability, the Java compiler ignores them during program execution. To include these details, programmers typically use comments.

Comments are non-executable parts of a program. A compiler does not execute a comment. It is used to improve the readability of the code.

Writing comments is a good practice as it improves the documentation of the code. There are three different types of comments- single-line, multi-line, and documentation comments.

- **Single line comment**- It is a comment that starts with // and is only used for a single line.

```
//Single-line comment
```

- **Multi line comment**- It is a comment that starts with /* and ends with */ and is used when more than one line has to be enclosed as comments.

```
/* Multiline comment
in Java */
```

- **Documentation comment**- It is a comment that starts with /** and ends with */

```
/** Documentation comment */
```

Package Declaration

Declaring the package in the structure of Java is optional. It comes right after the documentation section. You mention the package name where the class belongs. Only one package statement is allowed in a Java program and must come before any class or interface declaration. This declaration helps organize classes into different directories based on the modules they're used in. You use the keyword package followed by the package name. For instance:

```
package scaler; //scaler is the package name
package com.scaler; //com is the root directory, and scaler is the subdirectory
```

In Java, we have to save the program file name with the same name as the name of public class in that file.

The above is a good practice as it tells JVM which class to load and where to look for the entry point (main method).

The extension should be java. Java programs are run using the following two commands:

```
javac fileName.java // To compile the Java program into byte-code
java fileName // To run the program
```

Import Statements

Import statements are used to import classes, interfaces, or enums that are stored in packages or the entire package. A package contains many predefined classes and interfaces. We need to mention which package we are using at the beginning of the program. We do it by using the import keyword. We either import the entire package or a specific class from that package.

The following is the description of how we can write the import statement.

```
import java.util.*; //imports all the classes in util package
```

```
import java.util.StringTokenizer; // imports only the StringTokenizer class from util package
```

Explanation-
We have imported java.util package in the first and second lines we have imported only the StringTokenizer class from java.util package.

Interface Section

This is an optional section. The keyword interface is used to create an interface. An interface comprises a set of cohesive methods that lack implementation details., i.e. method declaration and constants.

```
interface Code {
   void write();
   void debug();
}
```

Explanation-
In the above code, we have defined an interface named Code, which contains two method declarations, namely write() and debug(), with no method body. The body of abstract methods is implemented in those classes that implement the interface Code.

Class Definition

This is a mandatory section in the structure of Java program. Each Java program has to be written inside a class as it is one of the main principles of Object-oriented programming that Java strictly follows, i.e., its Encapsulation for data security.
There can be multiple classes in a program. Some conventions need to be followed to name a class. They should begin with an uppercase letter.

```
class Program{
   // class definition
}
```

Class Variables and Variables

Identifiers are used to name classes, methods, and variables. It can be a sequence of uppercase and lowercase characters. It can also contain '_' (underscore) and '$' (dollar) signs. It should not start with a digit(0-9) and not contain any special characters.
Variables are also known as identifiers in Java. It is a named memory location which contains a value. In a single statement, we're able to declare multiple variables of the same type.

Syntax:
<Data-Type> <Variable Name> or <Data-Type> <Variable Name> = value;

Example:

```
int var=100;
int g;
char c,d; // declaring more than one variable in a statement
```

Rules for Naming a Variable-

- A variable may contain characters, digits and underscores.
- The underscore can be used in between the characters.
- Variable names should be meaningful and depict the program's logic.
- Variable names should not start with a digit or a special character.

Main Method Class

This is a compulsory part of the structure of Java program. This is the **entry point** of the compiler where the execution starts. It is called/invoked by the Java Virtual Machine or JVM. The main() method should be defined inside a class. We can call other functions and create objects using this method. The following is the syntax that is used to define.

Syntax

```
public static void main(String[] args) {
   // Method logic
}
```

Methods and Behaviors

A method is a collection of statements that perform a specific task.Method names typically begin with a lowercase letter.
It provides the reusability of code as we write a method once and use it many times by invoking the method. The most important method is Java's main() method.

The following are the important components of the method declaration.

Modifier- Defines access type, i.e., the method's scope.

The return data type- It is the data type returned by the method or void if does not return any value.

Method Name- The method names should start with a lowercase letter (convention) and not be a keyword.

Parameter list- It is the list of the input parameters preceded by their data type within the enclosed parenthesis. If there are no parameters, you must put empty parentheses ().

Exception list- The exceptions that a method might throw are specified.
Method body- It is enclosed between braces {}. The code to be executed is encapsulated within them.

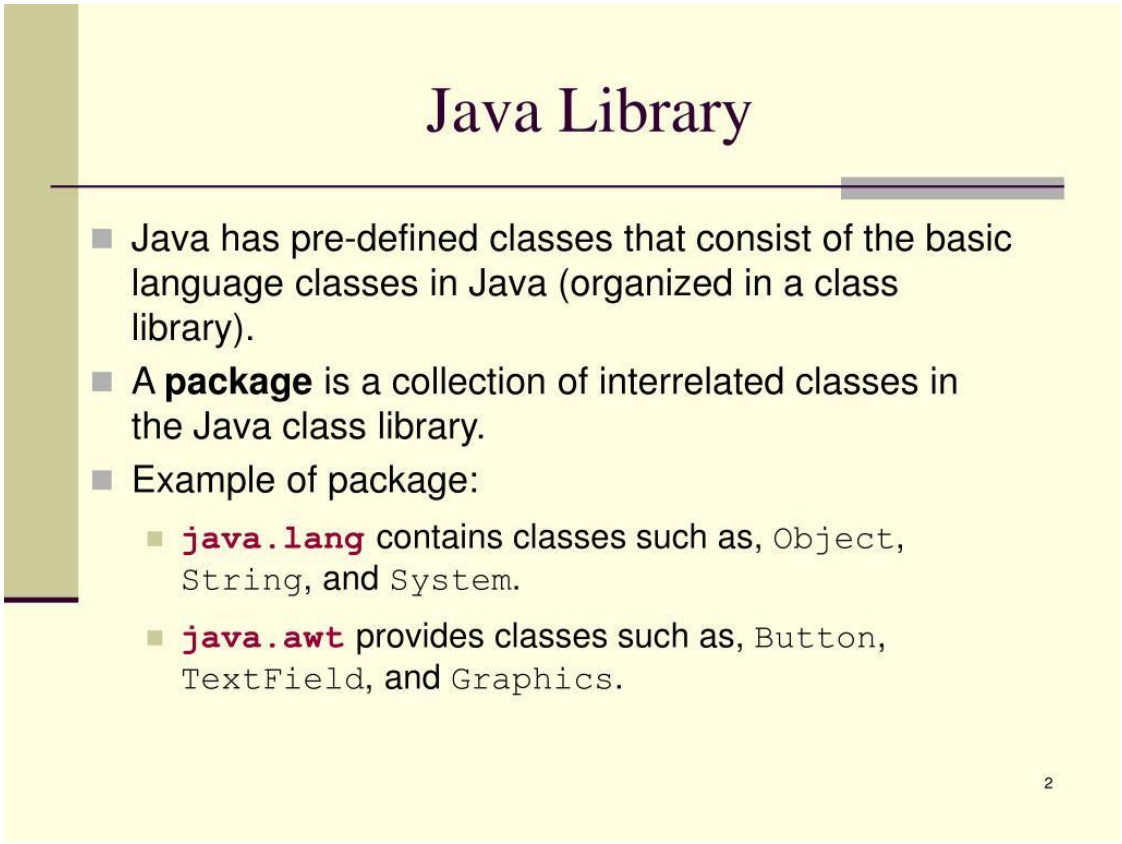
Method signature- It contains the method name and a parameter list (number, type, and order of the parameters). The return data type and exceptions are not a part of it.

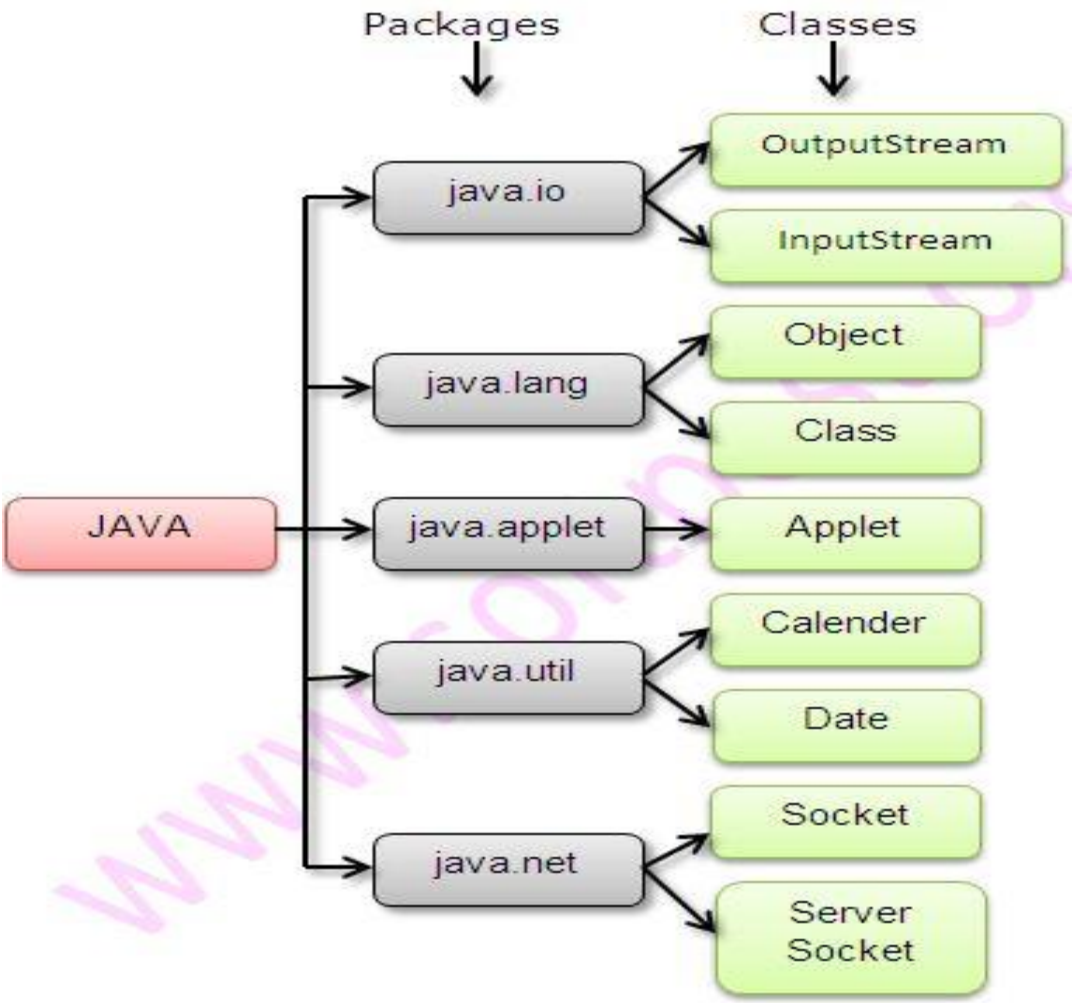
Example:

```
public void add(int a, int b){
   int c = a + b;
   System.out.println(c);
}
```

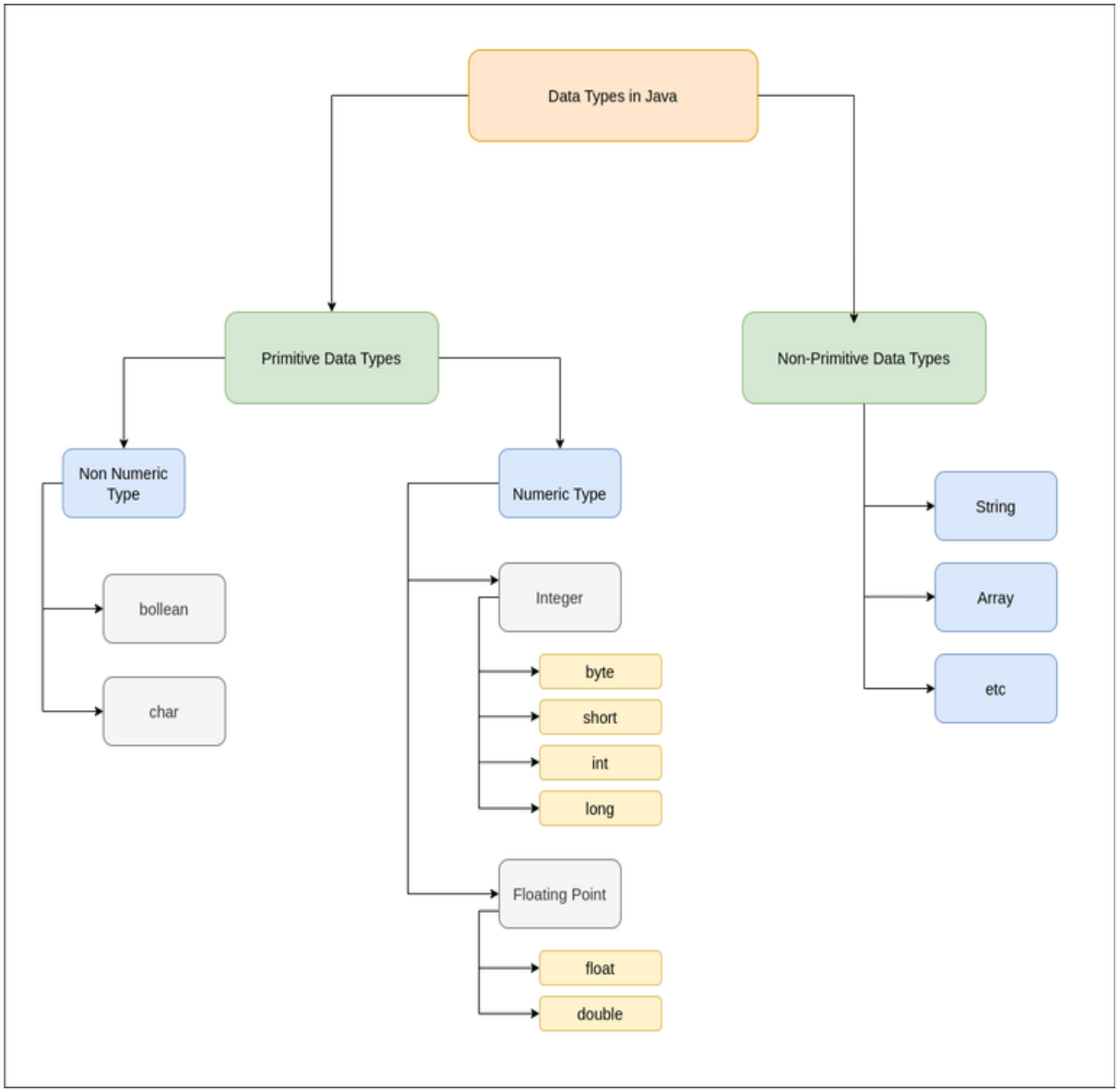
Explanation-
In the above method, we added two numbers passed as parameters and printed the result after adding them. This is only executed when the method is called.

Java class libraries




Packages
Classes
JAVA
java.io
java.lang
java.applet
java.util
java.net
OutputStream
InputStream
Object
Class
Applet
Calender
Date
Socket
Server Socket

Data types


Data Types in Java
Primitive Data Types
Non-Primitive Data Types
Non Numeric Type
Numeric Type
String
bollean
Integer
Array
byte
etc
char
short
int
long
Floating Point
float
double

1. Primitive Data Types
Primitive data types are the most basic types and are predefined by the Java language. There are eight primitive data types, divided into non-numeric and numeric types.

Non-Numeric Types

- **boolean**:
  - Represents one bit of information, which can be either true or false.
  - **Syntax**:
    java
    Copy code
    boolean flag = true;
    boolean isValid = false;

- **char**:
  - Represents a single 16-bit Unicode character. Values range from '\u0000' to '\uffff' (0 to 65,535 inclusive).
  - **Syntax**:
    java
    Copy code
    char letter = 'A';
    char digit = '1';
    char unicodeChar = '\u0041'; // Unicode for 'A'

Numeric Types

Integer Types
- **byte**:
  - An 8-bit signed integer. Range: -128 to 127.
  - **Syntax**:
    java
    Copy code
    byte smallNumber = 100;

- **short**:
  - A 16-bit signed integer. Range: -32,768 to 32,767.
  - **Syntax**:
    java
    Copy code
    short mediumNumber = 32000;
- **int**:
  - A 32-bit signed integer. Range: -2^31 to 2^31-1.
  - **Syntax**:
    java
    Copy code
    int number = 100000;

- **long**:
  - A 64-bit signed integer. Range: -2^63 to 2^63-1.
  - **Syntax**:

    java
    Copy code
    long largeNumber = 100000L;

Floating-Point Types

- **float**:
  - A single-precision 32-bit IEEE 754 floating point.
  - **Syntax**:

    java
    Copy code
    float decimalNumber = 5.75f;
- **double**:
  - A double-precision 64-bit IEEE 754 floating point.
  - **Syntax**:

    java
    Copy code
    double bigDecimalNumber = 19.99;

2. Non-Primitive Data Types

Non-primitive data types are created by the programmer and are more complex than primitive data types. They can be used to call methods to perform certain operations.

- **String**:
  - Represents a sequence of characters. It is not a primitive type but is heavily used.
  - **Syntax**:

    java
    Copy code
    String message = "Hello, World!";
- **Array**:
  - A collection of elements of the same type. Arrays can store primitive data types as well as objects.
  - **Syntax**:

    java
    Copy code
    // Array of integers
    int[] numbers = {1, 2, 3, 4, 5};

    // Array of strings
    String[] names = {"John", "Jane", "Doe"};

Non-Primitive Types

Includes classes, interfaces, and other data structures.

- **Classes**:

```
java
Copy code
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

Person person = new Person("Alice", 30);
person.display();
```

- **Interfaces**:

```
java
Copy code
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}

Animal dog = new Dog();
dog.makeSound();
```

Variables

Variables are containers for storing data values.
Syntax
type variableName = value;

In Java, there are different **types** of variables, for example:

- String - stores text, such as "Hello". String values are surrounded by double quotes
examples

```
public class Main {
  public static void main(String[] args) {
    String name = "hello CGCians";
    System.out.println(name);
  }
}
```

- int - stores integers (whole numbers), without decimals, such as 123 or -123

```
public class Main {
  public static void main(String[] args) {
    int myNum = 15;
    System.out.println(myNum);
  }
}
```

- float - stores floating point numbers, with decimals, such as 19.99 or -19.99

```
public class Main {
  public static void main(String[] args) {
   float myNum = 5.457f;
    System.out.println(myNum);
  }
}
```

- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes

```
public class Main {
  public static void main(String[] args) {
  char x = 'j';
    System.out.println(x);
  }
}
```

- boolean - stores values with two states: true or false

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

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Examples

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt;

    System.out.println(myInt);
    System.out.println(myDouble);
  }
}
```

Narrowing Casting

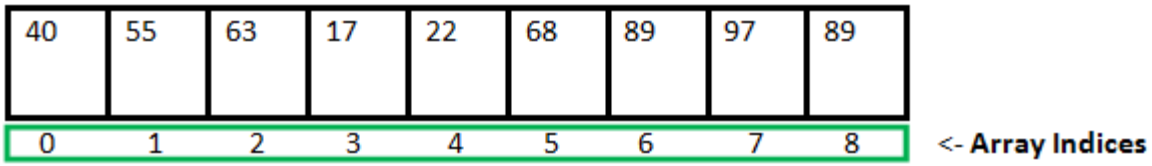
Narrowing casting must be done manually by placing the type in parentheses () in front of the value:

Example

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble;

    System.out.println(myDouble);
    System.out.println(myInt);
  }
}
```

Arrays

- In Java, all arrays are dynamically allocated. (discussed below)
- Arrays may be stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using size of.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning with 0.
- Java array can also be used as a static field, a local variable, or a method parameter.



Array Length = 9
First Index = 0
Last Index = 8

Examples

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    System.out.println(cars[0]);
  }
}
```

Automatic types promotion in expression

Automatic type promotion, also known as widening conversion, happens in Java when a smaller data type is automatically converted to a larger data type during an expression evaluation. This is done to prevent data loss and ensure compatibility between different types. Here are the key points about automatic type promotion in Java:

Rules:

- byte, short, and char promotion:

All byte, short, and char operands are promoted to int before any arithmetic operation.

- Higher type promotion:

If one operand is of type long, float, or double, the other operand is promoted to the same type.

- Order of promotion:

The promotion happens in a specific order: byte/short/char -> int -> long -> float -> double.

Examples:

Java

```
byte b = 10;

int i = 20;

long l = 30;


// b is promoted to int before addition

int result1 = b + i;

// i is promoted to long before addition

long result2 = i + l;


// b is promoted to int, then to long before addition

long result3 = b + l;
```

Benefits:

- Prevents data loss: By promoting smaller types to larger types, automatic type promotion ensures that no data is lost during arithmetic operations.
- Simplifies coding: Developers don't need to explicitly cast variables in many cases, leading to cleaner and more concise code.
- Improves performance: Implicit conversions are often faster than explicit casting operations.

Considerations:

- Loss of precision:

While promoting from int to float or double prevents data loss, it can lead to a loss of precision due to the way floating-point numbers are represented.

- Overflow:

If the result of an expression exceeds the range of the promoted type, overflow can occur, leading to unexpected results.
In summary, automatic type promotion is a valuable feature in Java that helps prevent data loss and simplifies coding. However, it's important to be aware of its rules and potential pitfalls to write correct and efficient code.

Operators –

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Example 2: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21

}}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-
11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{
public static void main(String args[]){
   //For positive number, >> and >>> works same
   System.out.println(20>>2);
   System.out.println(20>>>2);
   //For negative number, >>> changes parity bit (MSB) to 0
   System.out.println(-20>>2);
   System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.
The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.
The bitwise | operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
true
true
true
10
true
11
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.
Java Ternary Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Another Example:

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

5

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.
Java Assignment Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

14
16

Java Assignment Operator Example

```
public class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

13
9
18
9

Java Assignment Operator Example: Adding short

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
//a+=b;//a=a+b internally so fine
a=a+b;//Compile time error because 10+10=20 now int
System.out.println(a);
}}
```

Output:

Compile time error

After type cast:

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

20

operators precedence

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
	prefix	++expr --expr +expr -expr ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

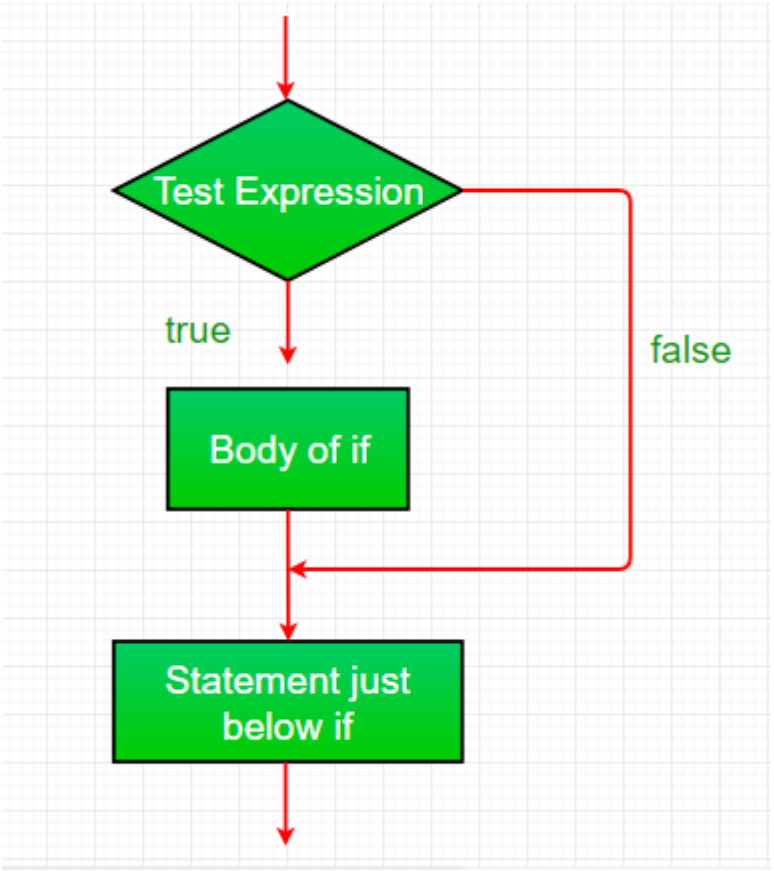
Control statements

Selection statements

1. **if:** if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

   Syntax:

   ```
   if(condition)
   {
     // Statements to execute if
     // condition is true
   }
   ```



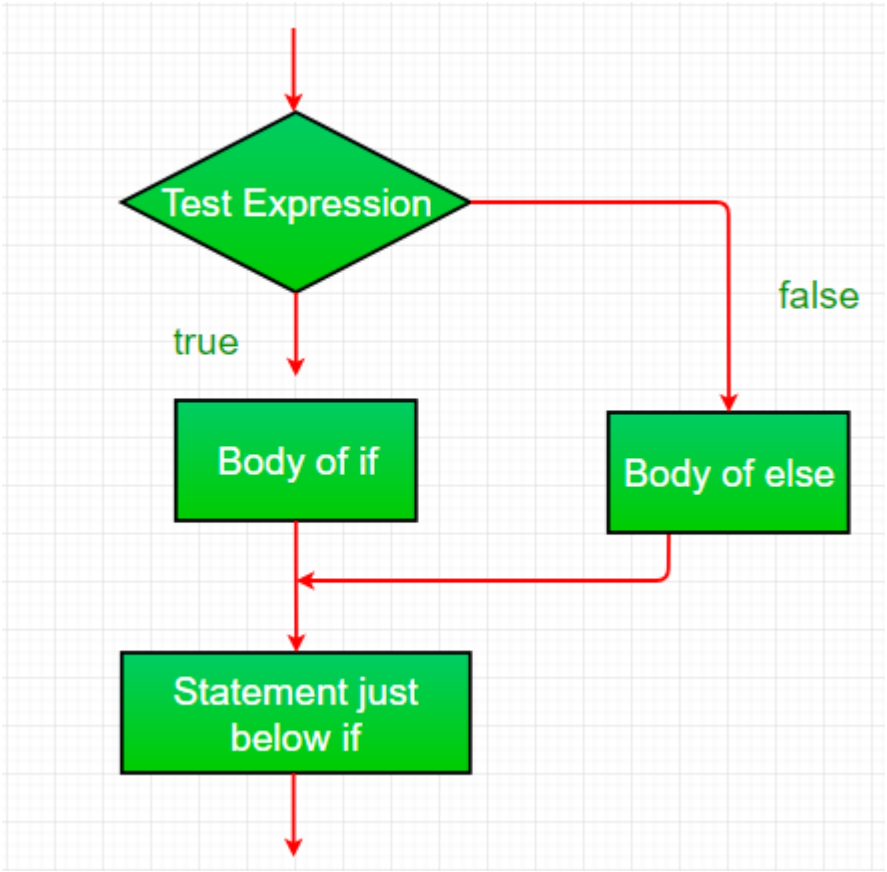
```
import java.util.*;
 class IfDemo {
   public static void main(String args[])
   {
      int i = 10;
      if (i < 15)
         System.out.println("Inside If block");
         System.out.println("10 is less than 15");

      System.out.println("I am Not in if");
   }
}
```

Time Complexity: O(1)

Auxiliary Space : O(1)

2. **if-else**: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false? Here comes the else statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

```
if(condition)
{
  // Statements to execute if
  // condition is true
}
else
{
  // Statements to execute if
  // condition is false
}
```



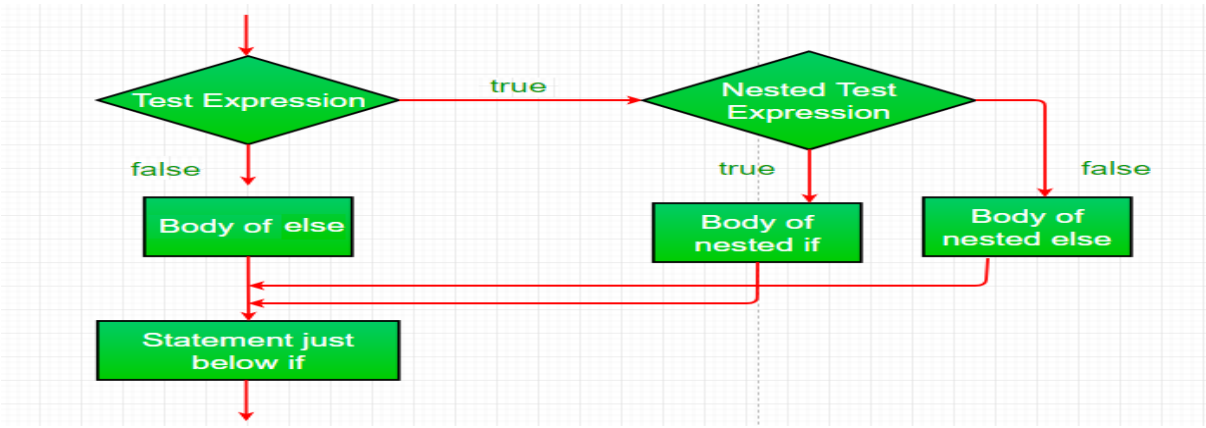

```
import java.util.*;

class IfElseDemo {
   public static void main(String args[])
   {
      int i = 10;

      if (i < 15)
         System.out.println("i is smaller than 15");
      else
         System.out.println("i is greater than 15");
   }
}
```

Time Complexity: O(1)

Auxiliary Space : O(1)

3. **nested-if:** A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition2)
  {
    // Executes when condition2 is true
  }
}
```



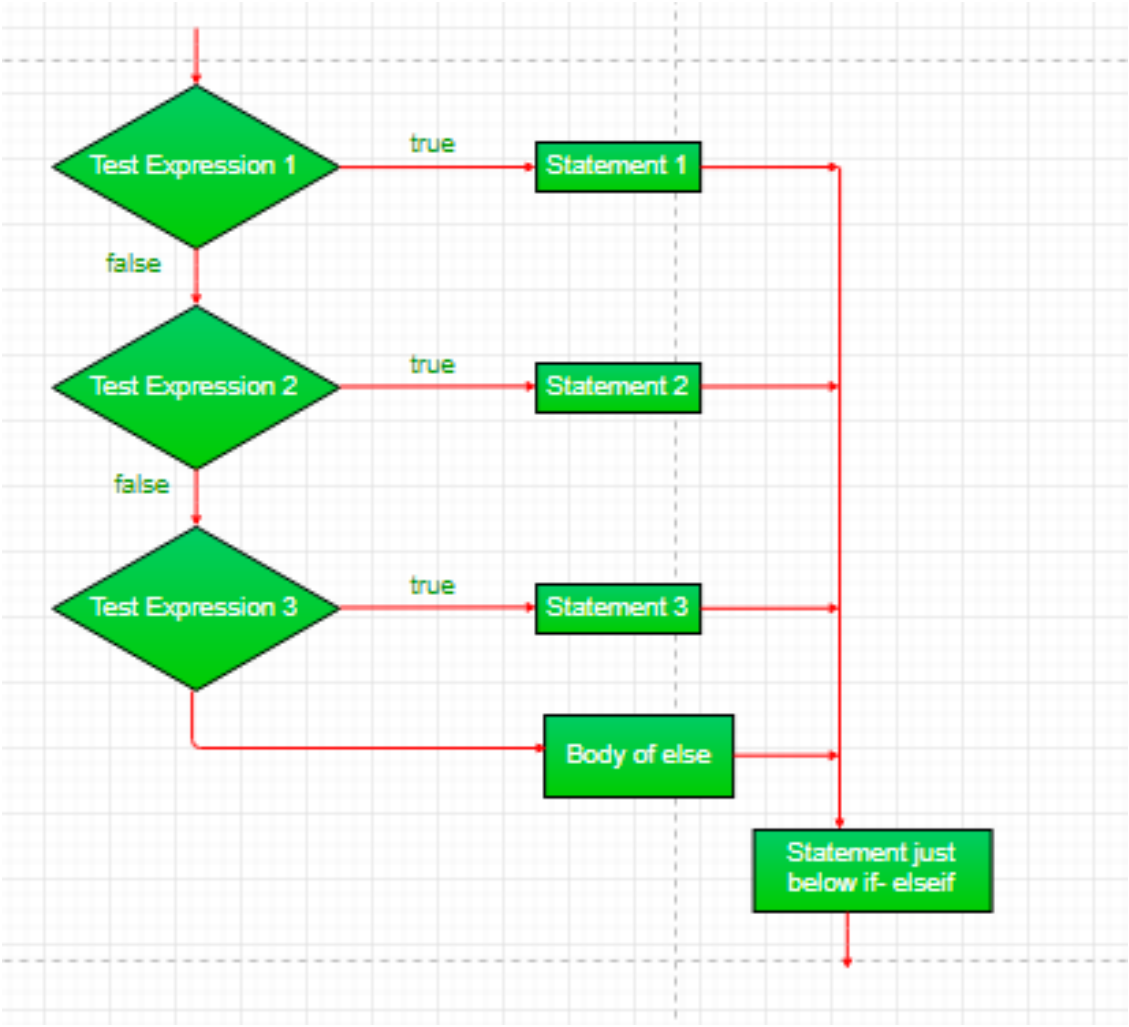

```
import java.util.*;

class NestedIfDemo {
   public static void main(String args[])
   {
      int i = 10;

      if (i == 10 || i<15) {
         // First if statement
         if (i < 15)
            System.out.println("i is smaller than 15");

         // Nested - if statement
         // Will only be executed if statement above
         // it is true
         if (i < 12)
            System.out.println(
               "i is smaller than 12 too");
      } else{
            System.out.println("i is greater than 15");
      }
   }
}
```

Time Complexity: O(1)

Auxiliary Space: O(1)

4. **if-else-if ladder:** Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that 'if' is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. There can be as many as 'else if' blocks associated with one 'if' block but only one 'else' block is allowed with one 'if' block.

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```



```
import java.util.*;

class ifelseifDemo {
   public static void main(String args[])
   {
      int i = 20;

      if (i == 10)
         System.out.println("i is 10");
      else if (i == 15)
         System.out.println("i is 15");
      else if (i == 20)
         System.out.println("i is 20");
      else
         System.out.println("i is not present");
   }
}
```

Time Complexity: O(1)

Auxiliary Space: O(1)

5. **switch-case:** The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

```
switch (expression)
{
 case value1:
   statement1;
   break;
 case value2:
   statement2;
   break;
 .
 .
 case valueN:
   statementN;
   break;
 default:
   statementDefault;
}
```

Example-

```
import java.io.*;

class GFG {
   public static void main (String[] args) {
      int num=20;
        switch(num){
        case 5 :  System.out.println("It is 5");
                  break;
        case 10 : System.out.println("It is 10");
                  break;
        case 15 : System.out.println("It is 15");
                  break;
        case 20 : System.out.println("It is 20");
                  break;
        default:  System.out.println("Not present");

      }
   }
}
```

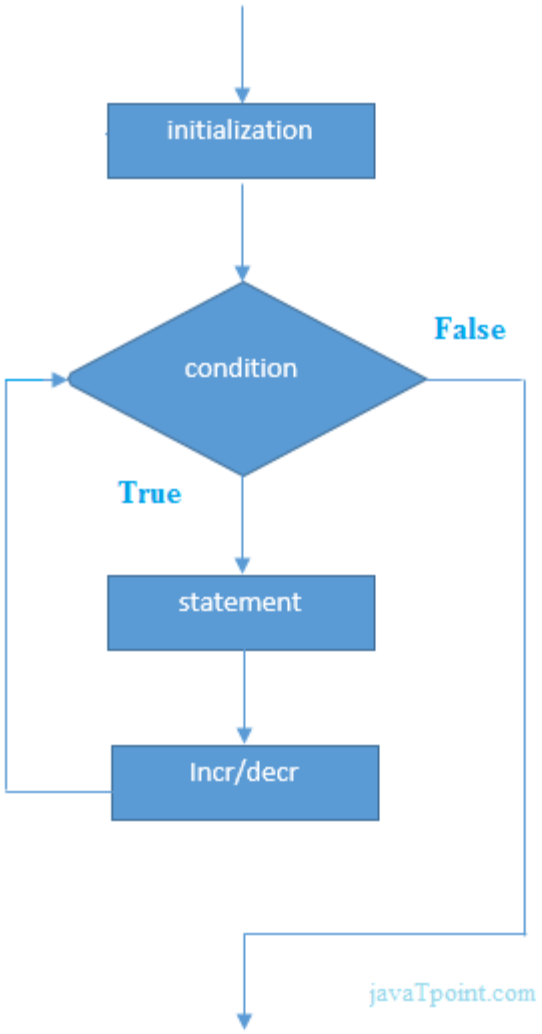
Time Complexity: O(1)
Space Complexity: O(1)

Iteration statement

For Loop:
For loops iterate over a range of values or elements in a collection. These are mainly used where the number of iterations is known beforehand like iterating through elements in an array or predefined range.

Syntax

```
for i in range(5):
   print(i)
```

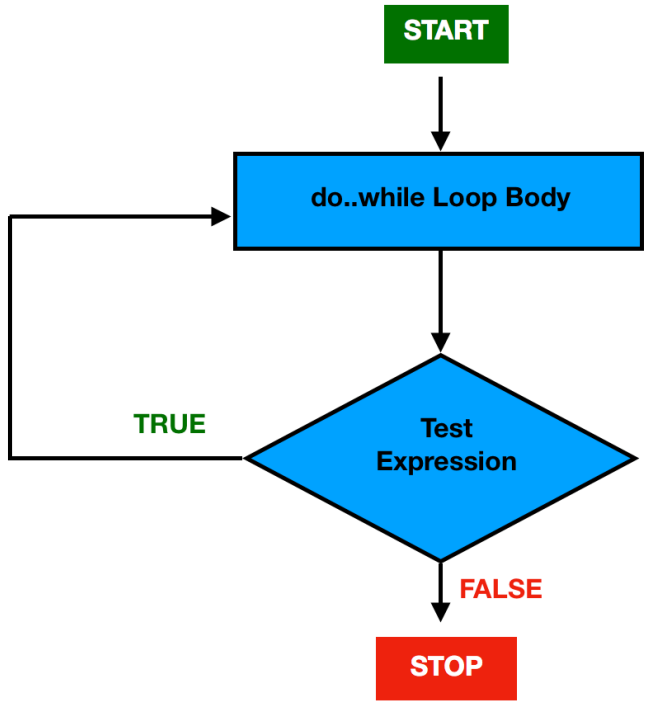


2. While Loops

While loops execute as long as specifies condition evaluates to true. These are suitable for situations where the number of iterations is not known and depends on dynamic conditions.

Syntax

```
count = 0
while count < 5:
    print(count)
    count += 1
```

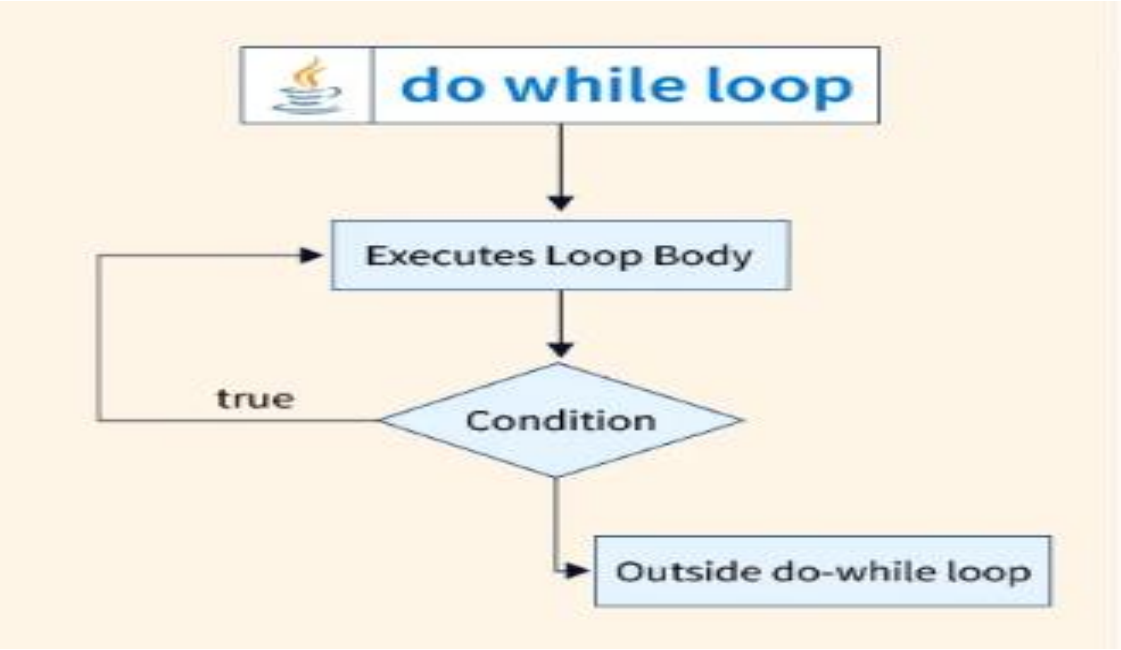


3. Do-While Loops

Do while loops are similar to while loop but guarantee at least one execution of the loop body before checking the condition. These are useful when loop must execute at least once regardless of the condition.

Syntax

```
int count = 0;
do {
    cout << count << endl;
    count++;
} while (count < 5);
```



Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	for(init;condition;incr/decr) { // code to be executed }	while(condition){ //code to be executed }	do{ //code to be executed }while(condition);

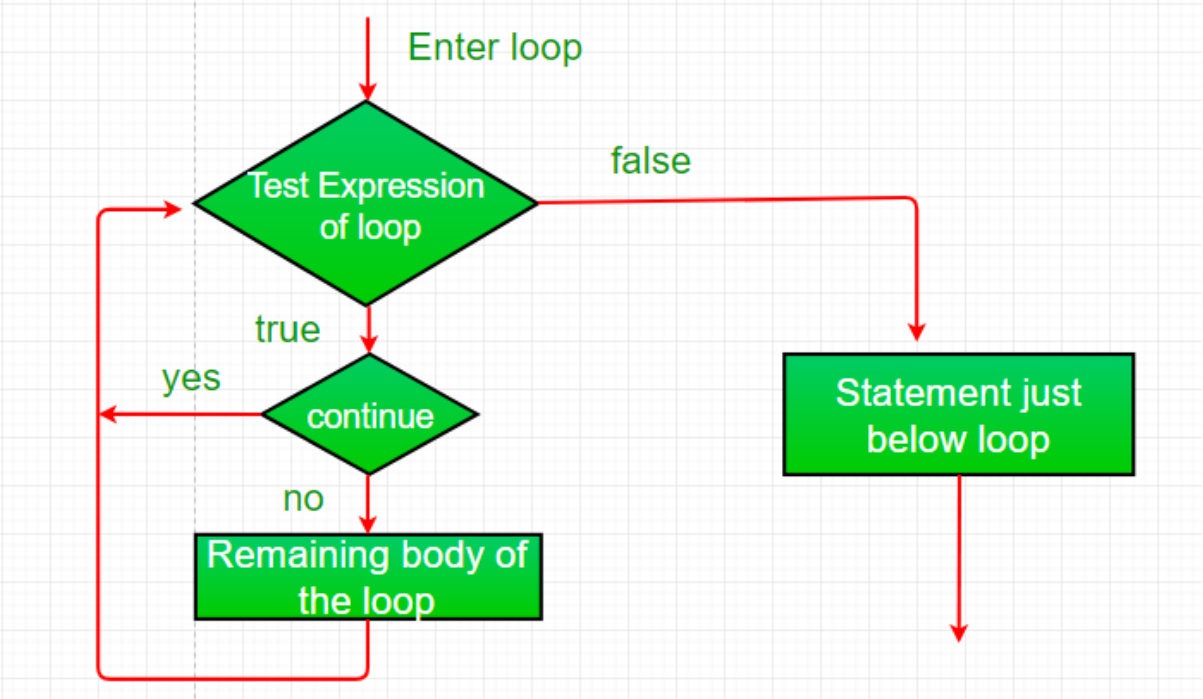
Example	//for loop
	for(int i=1;i<=10;i++){
	System.out.println(i);
	}
		//while loop
		int i=1;
		while(i<=10){
		System.out.println(i);
		i++;
		}
			//do-while loop
			int i=1;
			do{
			System.out.println(i);
			i++;
			}while(i<=10);

Syntax for infinitive loop	for(;;){
	//code to be executed
	}
		while(true){
		//code to be executed
		}
			do{
			//code to be executed
			}while(true);

Jump statement

jump: Java supports three jump statements: break, continue and return. These three statements transfer control to another part of the program.

- Break: In Java, a break is majorly used for:
  - Terminate a sequence in a switch statement (discussed above).
  - To exit a loop.
  - Used as a “civilized” form of goto.
  -
- Continue: Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action.



```
import java.util.*;

class ContinueDemo {
   public static void main(String args[])
   {
      for (int i = 0; i < 10; i++) {
         // If the number is even
         // skip and continue
         if (i % 2 == 0)
            continue;

         // If number is odd, print it
         System.out.print(i + " ");
      }
   }
}
```

Time Complexity: O(1)

Auxiliary Space: O(1)

- **Return:** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

```
import java.util.*;

public class Return {
   public static void main(String args[])
   {
      boolean t = true;
      System.out.println("Before the return.");

      if (t)
         return;

      // Compiler will bypass every statement
      // after return
      System.out.println("This won't execute.");
   }
}
```

Time Complexity: O(1)

Auxiliary Space: O(1)