JIT(just in time compiler) - just in time compiler present inside

JVM and it tells the JVM to do not convert same code into machine code again and again instead use it from JIT. For example some function has been already converted into machine code so no need to convert it into it again and again. Instead use the already converted machine code from JIT.

**Primitive variable** -

int a = 10;

Here, a is a primitive variable.

It directly stores the value 10 in memory (no reference, no object).

There is no object and no "reference value" here.

so a itself contains the number 10.

**Reference variable: -**

A **reference variable** stores the "address" (or reference) of an object in memory, not the actual object itself.

It’s like a remote control pointing to the actual TV.

Example:

String name = "Amit";

Here:

name is a reference variable.

It doesn’t directly hold the string "Amit", but rather the reference (memory address) of the String object stored in the heap.

String s = "Hello";

Here,

**Reference variable** = **s**

**Object** = Hello

**Primitive types** (int, char, double, boolean, etc.) → they are **variables**, they store actual values.

**Objects (classes, arrays, strings, etc.)** → are stored in heap and accessed using **reference variables**.

Variable Arguments (Varargs)

Normally, a method has a fixed number of parameters.  
But with **varargs**, you can pass **0 or more arguments** to a method.

👉 Syntax:

returnType methodName(type... varName) { }

* type... means variable arguments.
* Inside the method, varargs behave like an **array**.
* Variable argument always come at end.  
  for example : - String...v come at end of the arguments

static void multiple(int a, int b, String...v) {  
   
  
}

**Method With Varargs**

import java.util.Arrays;  
  
public class VariableArgument {  
 public static void main(String[] args) {  
 *fun*(2,3,4,5,6,7,8,9); // return an array  
  
 }  
  
 // another method with multiple parameter.two integer one variable argument  
 static void multiple(int a, int b, String...v) {  
 // String...v works as an Array  
  
 }  
  
  
  
 // method with one variable argument  
 static void fun(int... v) {  
 System.*out*.println(Arrays.*toString*(v));  
 }  
}

👉 int... numbers can take **any number of int values**.  
Inside the method, numbers acts like an array.

**🔹 Key Rules**

1. Only **one varargs parameter** per method.  
   ✅ void test(int... x)  
   ❌ void test(int... x, int... y)
2. Varargs must be the **last parameter**.  
   ✅ void test(String name, int... x)  
   ❌ void test(int... x, String name)

**Method Overloading**

In Java, **Method Overloading** means **defining multiple methods with the same name** in the same class, but with **different parameter lists** (number, type, or order of parameters).

It is also called **Compile-Time Polymorphism** because the decision of which method to call is made by the compiler.

## ✅ Rules for Function Overloading

1. Method **name must be the same**.
2. Parameter list must be **different** (in number, type, or order).
3. Return type **alone cannot differentiate methods**.

## 🔹 Example 1 – Different Number of Parameters

class OverloadExample {

static int add(int a, int b) {

return a + b;

}

static int add(int a, int b, int c) {

return a + b + c;

}

public static void main(String[] args) {

System.out.println(add(5, 10)); // calls add(int, int)

System.out.println(add(5, 10, 15)); // calls add(int, int, int)

}

}

## 🔹 Example 2 – Different Parameter Types

class OverloadExample {

static int multiply(int a, int b) {

return a \* b;

}

static double multiply(double a, double b) {

return a \* b;

}

public static void main(String[] args) {

System.out.println(multiply(4, 5)); // calls multiply(int, int)

System.out.println(multiply(4.5, 5.5)); // calls multiply(double, double)

}

}

## 🔹 Example 3 – Different Order of Parameters

class OverloadExample {

static void show(String name, int age) {

System.out.println("Name: " + name + ", Age: " + age);

}

static void show(int age, String name) {

System.out.println("Age: " + age + ", Name: " + name);

}

public static void main(String[] args) {

show("Amit", 26); // calls show(String, int)

show(26, "Amit"); // calls show(int, String)

}

}

## ❌ Not Allowed – Only Return Type Different

class OverloadExample {

static int test(int a, int b) { return a + b; }

// static double test(int a, int b) { return a + b; } // ❌ ERROR

}

👉 Because compiler won’t know which one to call.

## 🔑 Key Point

* Overloading → **same method name, different parameter list**.
* Happens at **compile time** → also called **Compile-Time Polymorphism**.

**Formal paramter and actual parameter**

**Formal parameter :**

Defined during function definition.  
example :   
int add(int a, int b) {

}

Int a and int b are formal parameter.

**Actual Parameter / Argument :**

Parameter that we passed when we are calling the function

Example :

add(5,6)  
add(x,y)  
  
5 and 6 are actual parameter.

x and y are actual parameter.

**Pass by value and pass by reference :**

Pass by value and pass by reference are the two different Ways to pass actual parameter.

Pass by value : here copies the value of actual parameters. Called function creates its own copy.

// create a method named as sum

Static int sum(int a, int b) {

}

// calling sum

Int x = 5, y = 10;

sum(x, y)

* when sum is called then it creates a copy of a and b from

Static int sum(int a, int b) {

}

Pass by reference: here pass the the parameter as reference(address). Called function does not creates its own copy.