

>> JVM does not look for just main method, it looks for entire method signature "public static void main(String a[])"

>> Java is called WORA -> Write Once Run Anywhere

>> **JIT:** Just In Time Compiler: The Just-In-Time (JIT) compiler is a component of the runtime environment that improves the performance of Java™ applications by compiling bytecodes to native machine code at run time

>> **AOT:** Ahead Of Time Compiler: It used to compile byte code to native code prior to the execution by the JVM.

>> **Java Features:**

Simple, Secure, Portable, Object-Oriented, Robust, Multithreaded, Architecture-neutral, Interpreted, High performance, Distributed, Dynamic.

**Secure:** In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows **ArrayIndexOutOfBoundsException** if we try to do so. That's why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the os environment which makes java programs more secure.

**Robust:** The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

>> beginning with JDK 8 it is now possible to define a default implementation for a method specified by an interface. If no implementation for a default method is created the the default defined by the interface is used.

```
public static void main(String a[])
{
    // literals

    int num1 = 0x7E;
    System.out.println(num1);
}
```

```
2
3     public static void main(String a[])
4     {
5         // literals
6
7         int num1 = 10_00_00_000;
8         System.out.println(num1);
9
10
11     }
12
13 }
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
5
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
126
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
100000000
telusko@Navins-MacBook-Pro course % █
```

```
Hello.java
2  {
3      public static void main(String a[])
4      {
5          // literals
6
7          double num1 = 12e10;
8          System.out.println(num1);
9
10
11
12      }
13 }
```

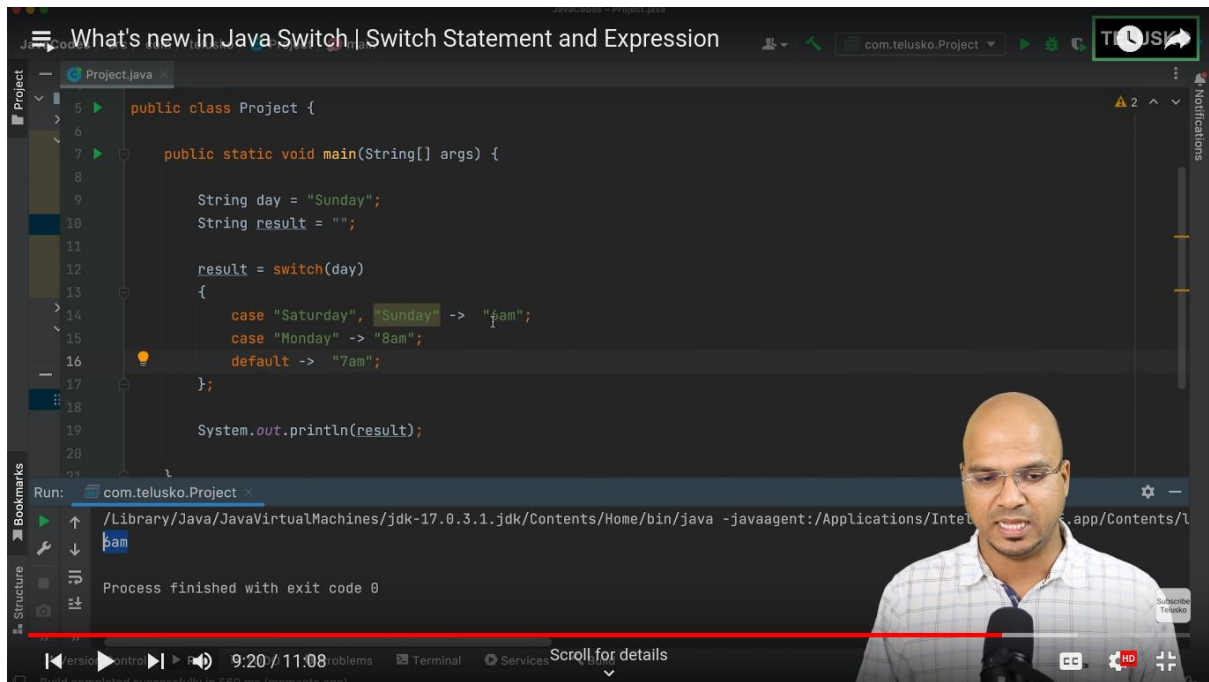
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
1000000000
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
56.0
telusko@Navins-MacBook-Pro course % java Hello
56.0
telusko@Navins-MacBook-Pro course % javac Hello.java
telusko@Navins-MacBook-Pro course % java Hello
1.2E11
telusko@Navins-MacBook-Pro course %
```

```
class Hello
{
    public static void main(String args[])
    {
        //byte b = 125;
        int a = 257;
        byte k = (byte) a;
        System.out.println(k);
    }
}
```

```
127
telusko@navin-mac course % javac Hello.java
Hello.java:5: error: incompatible types: possible lossy conversion from int to byte
        byte b = 257;
        ^
1 error
telusko@navin-mac course % java Hello.java
127
telusko@navin-mac course % java Hello.java
125
telusko@navin-mac course % java Hello.java
Hello.java:7: error: incompatible types: possible lossy conversion from int to byte
        byte k = a;
        ^
1 error
error: compilation failed
telusko@navin-mac course % java
12
telusko@navin-mac course % java
1
telusko@navin-mac course %
```

Here 257 is out of range so printing 257%256



&gt;&gt;

## 1. **\*\*Class Loading\*\***:

- **Class Verification**: The JVM verifies the bytecode of the `Car` class for correctness and security.
- **Class Preparation**: The JVM allocates memory for static variables of the `Car` class and initializes them to default values.
- **Class Resolution**: The JVM resolves symbolic references in the `Car` class to actual references.

## 2. **Reference Variable Creation**:

- **Reference Variable Creation**: A reference variable `c` of type `Car` is declared. If `c` is a local variable, it is created on the stack. If it is an instance variable, it is part of the memory layout of the containing object, which resides on the heap.

## 3. **Object Creation**:

- **Memory Allocation**: The JVM allocates memory for a new `Car` object on the heap. The size of the allocated memory depends on the fields defined in the `Car` class.
- **Default Initialization**: The memory allocated for the `Car` object is initialized to default values (e.g., `0` for numeric types, `null` for reference types, `false` for boolean).
- **Constructor Call**: The constructor of the `Car` class is called, which may further initialize the fields of the `Car` object with specific values.

## 4. **Reference Assignment**:

- **Assign Memory Address**: The memory address of the newly created `Car` object is assigned to the reference variable `c`.

## 5. **Garbage Collection**:

- **Reachability Check**: The `Car` object remains in memory as long as it is reachable through the reference variable `c` or any other references.

- **Object Eligibility**: When the reference variable `c` goes out of scope or is assigned to another object or `null`, the `Car` object becomes eligible for garbage collection.

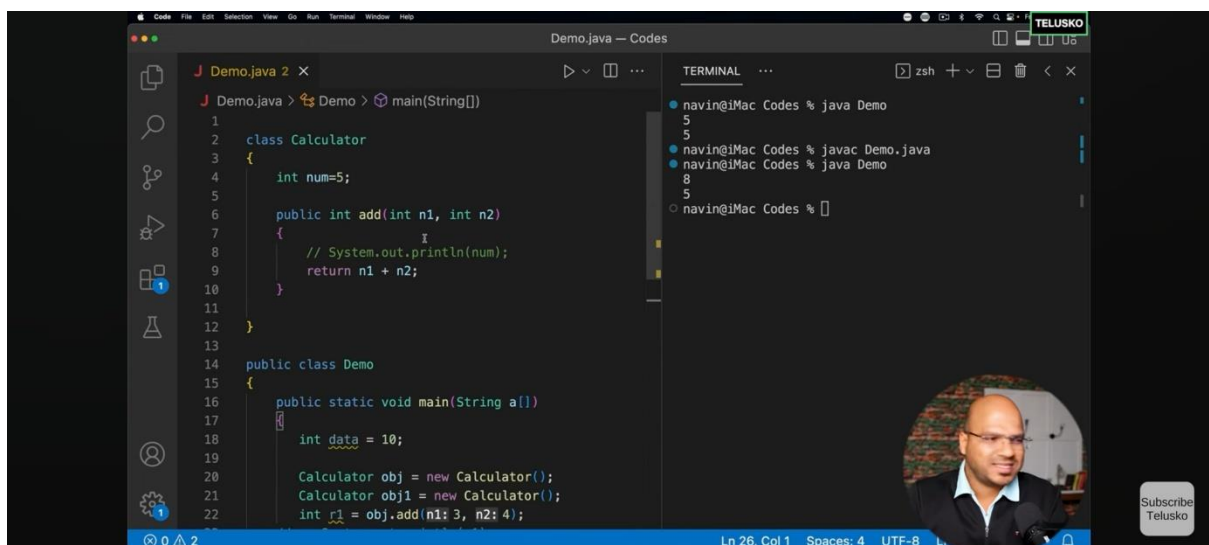
- **Garbage Collection Process**: The garbage collector eventually reclaims the memory used by the `Car` object, making it available for future allocations.

So the correct order is:

1. Class loading (if needed)
2. Reference variable creation (`c` on the stack)
3. Object creation (memory allocation on the heap)
4. Reference assignment (heap address assigned to `c`)
5. Garbage collection (if applicable)

This revised sequence correctly reflects how the JVM manages memory when the statement `Car c = new Car();` is executed.

-----

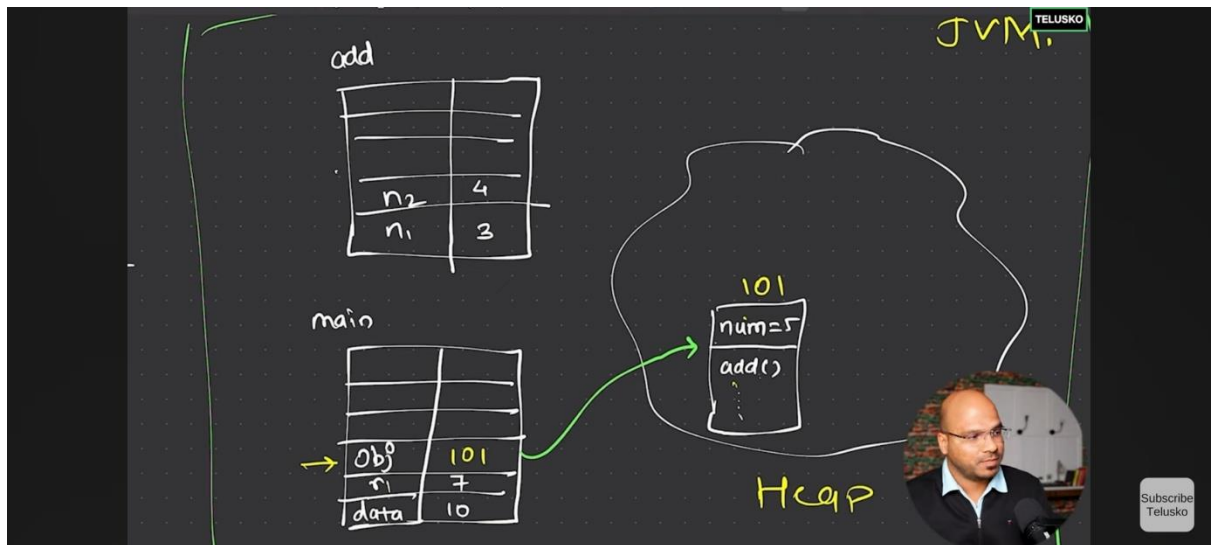


The screenshot shows an IDE with a Java file named 'Demo.java'. The code defines a 'Calculator' class with an 'add' method and a 'Demo' class with a 'main' method. The 'main' method creates a 'Calculator' object and calls its 'add' method with arguments 3 and 4. The terminal output shows the execution of 'java Demo', which prints '5'. A 'Subscribe Telusko' button is visible in the bottom right corner.

```
1 class Calculator
2 {
3     int num=5;
4     public int add(int n1, int n2)
5     {
6         // System.out.println(num);
7         return n1 + n2;
8     }
9 }
10
11
12
13
14 public class Demo
15 {
16     public static void main(String a[])
17     {
18         int data = 10;
19
20         Calculator obj = new Calculator();
21         Calculator obj1 = new Calculator();
22         int r1 = obj.add(n1: 3, n2: 4);
```

```
navin@iMac Codes % java Demo
5
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
8
5
navin@iMac Codes %
```





>>**How Main method works:** In Java, the main class (the class containing the `main` method) serves as the entry point for the application. When you run a Java application, the Java Virtual Machine (JVM) starts executing the `main` method without explicitly creating an instance of the main class. Here's how memory management works in this context:

1. **Static Context**: The `main` method is `static`, meaning it belongs to the class itself rather than an instance of the class. Static methods and variables are stored in a special area of memory known as the **method area** (part of the heap in many JVM implementations).
2. **Method Area**: The method area stores class-level data, including static variables, method definitions (including the `main` method), and other class-level structures. When the JVM loads the main class, it allocates memory for these static members in the method area.
3. **Stack Memory**: When the JVM starts executing the `main` method, it creates a stack frame for it in the **stack memory**. The stack is used



for method execution, including local variables and method call management. Each thread has its own stack.

4. **\*\*Heap Memory\*\***: If the ``main`` method or any other part of the program creates objects using the ``new`` keyword, these objects are allocated memory in the **\*\*heap\*\***. The heap is shared among all threads and is managed by the JVM's garbage collector, which reclaims memory from objects that are no longer reachable.

To summarize, the ``main`` method and any other static members of the main class reside in the method area, while objects created during the execution of the program are stored in the heap. Local variables within the ``main`` method (or any other method) are stored in the stack. This division of memory areas helps in efficient memory management and execution of Java programs.

---

>>

### **Instances Variables vs Other Variables:**

Instance variables in Java differ from other types of variables in several key ways:

#### 1. **\*\*Scope and Lifetime\*\***:

- **\*\*Instance Variables\*\***: These are defined within a class but outside any method, constructor, or block. They are created when an object of the class is instantiated and destroyed when the object is destroyed. Each object has its own copy of instance variables.

- **\*\*Local Variables\*\***: These are defined within a method, constructor, or block. They are created when the method, constructor, or block is entered and destroyed once it exits. They are not accessible outside their defining method, constructor, or block.

- **\*\*Static Variables\*\***: Also known as class variables, these are defined within a class using the ``static`` keyword but outside any method, constructor, or block. There is only one copy of a static variable, regardless of how many

objects of the class are created. They are associated with the class itself rather than any specific instance.

## 2. **Memory Location**:

- **Instance Variables**: Stored in the heap, as part of the object.
- **Local Variables**: Stored in the stack, as part of the stack frame for the method, constructor, or block in which they are defined.
- **Static Variables**: Stored in the method area (part of the heap in many JVM implementations), shared among all instances of the class.

## 3. **Default Values**:

- **Instance Variables**: Automatically initialized to default values (e.g., `0` for integers, `null` for object references) if not explicitly initialized.
- **Local Variables**: Must be explicitly initialized before use; they do not get default values.
- **Static Variables**: Automatically initialized to default values if not explicitly initialized.

## 4. **Access Modifiers**:

- **Instance Variables**: Can be marked with access modifiers (`private`, `protected`, `public`) to control their visibility.
- **Local Variables**: Cannot have access modifiers; their scope is limited to the method, constructor, or block in which they are declared.
- **Static Variables**: Can also be marked with access modifiers and are accessed using the class name.

## 5. **Use Case**:

- **Instance Variables**: Hold the state of an object. Different objects can have different values for their instance variables.

- **Local Variables**: Used for temporary storage of data within a method, constructor, or block.

- **Static Variables**: Used for values shared among all instances of a class, such as constants or counters. Here is a quick example to illustrate the differences:

Java  Copy code

```
public class Example {
    // Instance variable
    private int instanceVar;

    // Static variable
    private static int staticVar;

    public Example(int instanceVar)
    {
        this.instanceVar =
instanceVar;
    }

    public void method() {
        // Local variable
        int localVar = 10;
        System.out.println("Instance
Var: " + instanceVar);
        System.out.println("Static
Var: " + staticVar);
        System.out.println("Local
Var: " + localVar);
    }

    public static void main(String[]
args) {
        Example obj1 = new
Example(1);
        Example obj2 = new
Example(2);

        obj1.method();
        obj2.method();
    }
}
```

In this example:

- ``instanceVar`` is an instance variable, so each ``Example`` object has its own copy.
- ``staticVar`` is a static variable, shared among all ``Example`` objects.
- ``localVar`` is a local variable, accessible only within the ``method``.