## 1️⃣ Local Primitive Variable

class Demo {

void test() {

int a = 10;

System.out.println(a);

}

}

**Notes**

* a is a **local primitive variable**
* Stored in **stack**
* Value 10 is stored **directly in stack**
* No default value → must be initialized
* Lifetime: till method execution ends

## 2️⃣ Instance Primitive Variable

class Student {

int id = 10;

}

**Notes**

* id is an **instance primitive variable**
* Stored **inside the object**
* Object is in **heap**
* Value 10 is stored **inside heap object**
* Each object gets its **own copy**
* Default value exists if not initialized

## 3️⃣ Static Primitive Variable

class College {

static int count = 100;

}

**Notes**

* count is a **static primitive variable**
* Stored in **method area**
* Single shared copy for all objects
* Created when class is loaded
* Value 100 is stored **in method area**

## ✅ Reference Variables — Core Concepts

* Reference variables **do not store objects**.
* They store the **address (reference)** of an object.
* Objects are **always created in heap memory** using new.
* A reference variable can be:
  + local
  + instance
  + static
* The reference is stored in the **same memory area as the reference variable**:
  + local reference → stack
  + instance reference → heap (inside object)
  + static reference → method area
* Multiple reference variables can point to the **same object**.
* Assigning one reference to another copies the **address**, not the object.
* If all references to an object are removed, the object becomes **eligible for garbage collection**.
* Garbage Collector works **only on heap objects**, not on references themselves.

## ⭐ One-Line Golden Summary (Must Remember)

**Primitive variables store values directly.  
Reference variables store addresses.  
Objects always live in heap.  
Scope decides where the variable is stored.**

## ✅ Final Verified Rule (Correct)

* **Object** → ALWAYS stored in **Heap**
* **Reference** → Stored in the **same memory area as the reference variable**
  + local → stack
  + instance → heap (inside object)
  + static → method area

## ✅ Table View Explanation (Perfect & Exam-Ready)

| **Reference Variable Type** | **Where Reference Is Stored** | **Where Object Is Stored** | **Explanation** |
| --- | --- | --- | --- |
| **Local reference** | Stack | Heap | Reference variable is local, so it lives in stack. Object created using new is always in heap. |
| **Instance reference** | Heap (inside object) | Heap | Instance variable is part of the object, so both reference and the object it points to are in heap. |
| **Static reference** | Method Area (Class area) | Heap | Static variables belong to class, so reference is stored in method area. Object remains in heap. |

## 🧠 One-Line Golden Summary (Best to Remember)

**Objects always live in heap.  
References live wherever the reference variable is declared.**

## 4️⃣ Local Reference Variable

class Demo {

void test() {

Student s = new Student();

}

}

**Notes**

* s is a **local reference variable**
* Reference is stored in **stack**
* Object Student is stored in **heap**
* s holds **address of heap object**
* Object becomes eligible for GC when s goes out of scope

## 5️⃣ Instance Reference Variable

class College {

Student student = new Student();

}

**Notes**

* student is an **instance reference variable**
* Stored **inside the object**
* Object (College) is in **heap**
* Reference is also in **heap**
* Student object is a **separate heap object**

## 6️⃣ Static Reference Variable

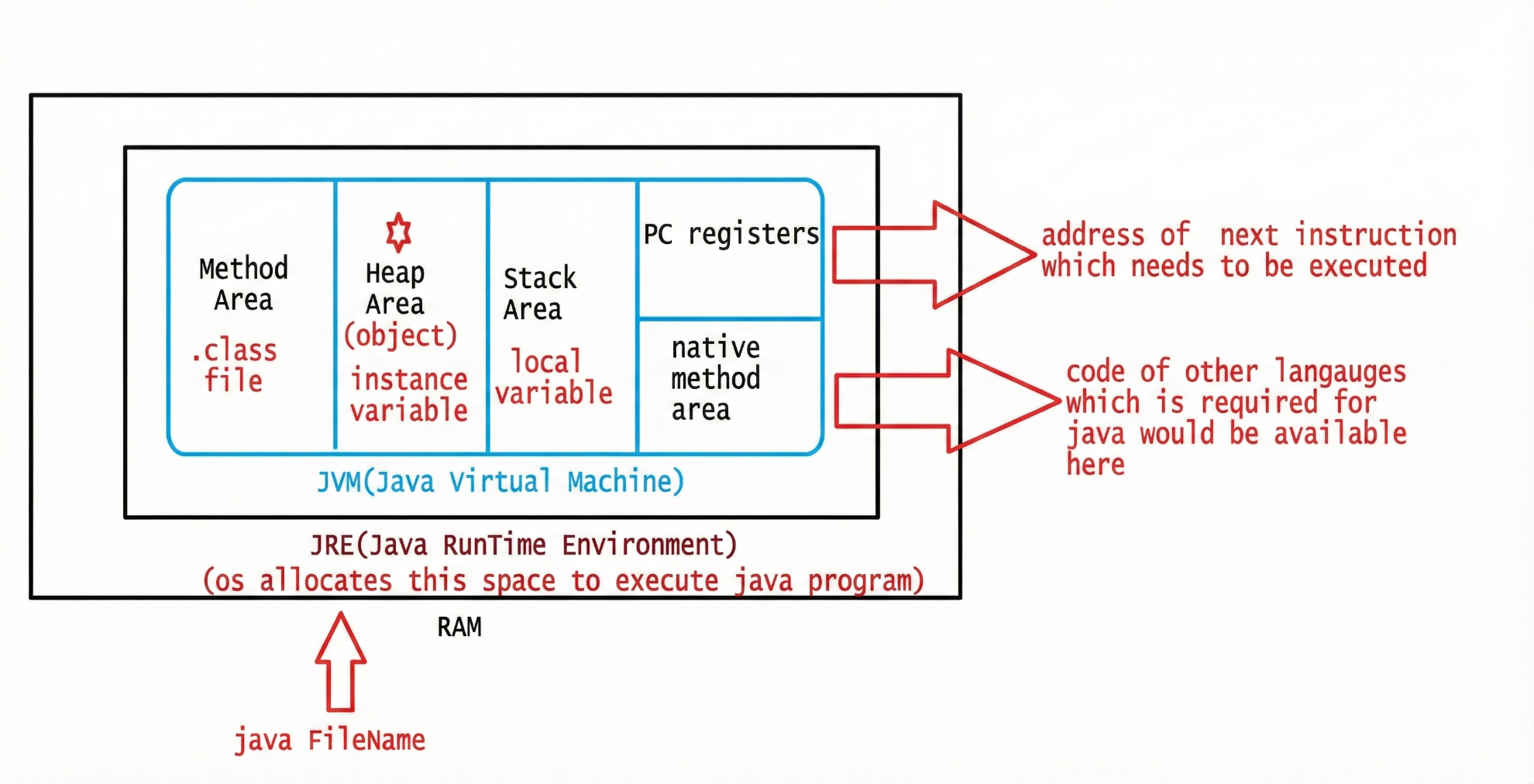
class College {

static Student topper = new Student();

}

**Notes**

* topper is a **static reference variable**
* Reference is stored in **method area**
* Object Student is stored in **heap**
* Single shared reference for all objects



## 1️⃣ JRE (Java Runtime Environment)

**Meaning**

* JRE is the **environment required to run Java programs**
* OS allocates **RAM** to JRE
* Inside JRE, **JVM is created**

👉 **JRE = JVM + supporting libraries**

## 2️⃣ JVM (Java Virtual Machine)

**Role**

* JVM executes Java bytecode
* JVM manages **memory, execution, and garbage collection**

The blue box in the diagram is **JVM memory structure**.

## 3️⃣ Method Area (Class Area) — Detailed Explanation

### What is the Method Area?

The Method Area is a part of JVM memory where class-level information is stored.

👉 It is created when a class is loaded into JVM  
👉 It exists until the JVM shuts down

## What Is Stored in the Method Area (Elaborated)

### 1️⃣ .class File Data

* Bytecode generated by the Java compiler
* JVM reads this bytecode to execute methods

### 2️⃣ Class Structure

Includes:

* Class name
* Package name
* Access modifiers
* Fields (variables) information
* Methods information
* Inheritance details
* Interfaces implemented

📌 JVM needs this metadata to understand how the class is designed

### 3️⃣ Method Bytecode

* Actual instructions of methods
* Used when methods are executed
* Stored once per class, not per object

### 4️⃣ Constant Pool

* String literals
* Numeric constants
* Method references
* Field references
* Symbolic references used by bytecode

📌 Helps JVM in efficient execution and linking

### 5️⃣ ⭐ Static Variables (Important — Elaborated)

Static variables are class-level variables, so JVM stores them in the Method Area.

class College {

static int count = 100;

}

**What happens internally**

When College class is loaded:

* JVM allocates memory for count in Method Area
* Only one copy of count exists
* All objects of College share the same variable

📌 Static variables:

* Are not part of any object
* Do not get separate copies per object
* Belong to the class itself

**Why Static Variables Are NOT Stored in Heap**

* Heap is meant for objects
* Static variables exist even without objects
* They must live as long as the class is loaded
* Method Area matches this lifetime perfectly

### 6️⃣ Static Blocks (missed earlier)

* Static block bytecode is stored in the Method Area
* Executed once when the class is loaded into JVM
* Used to initialize static variables before main() runs

## Key Notes (Expanded)

* One copy per class
* Created during class loading
* Shared across all threads and objects
* Static variables and static methods live here
* Removed only when JVM terminates (or class unloading happens)

## 4️⃣ Heap Area

**What is stored here**

* Objects
* Instance variables (inside objects)

**Key Notes**

* Objects are created using new
* Heap is shared by all threads
* Garbage Collector works here
* Objects stay until GC removes them

📌 **Objects always live in heap**

## 5️⃣ Stack Area — Detailed Explanation

### What is the Stack Area?

The **Stack Area** is the part of JVM memory used for **method execution and local data**.

👉 Each thread has its **own stack**  
👉 Stack works in **LIFO (Last In, First Out)** order

### What Is Stored in the Stack Area

#### 1️⃣ Method Call Frames

* Each method call creates a **stack frame**
* Stack frame stores method execution details
* Removed when method execution completes

#### 2️⃣ Local Variables

* Variables declared inside methods or blocks
* Stored in stack frames

void test() {

int a = 10;

Student s = new Student();

}

* a → local primitive (stored in stack)
* s → local reference (stored in stack)
* Object created by new → stored in heap

#### 3️⃣ Local Primitive Variables

* Store **actual values**
* Fast access
* Destroyed when method ends

#### 4️⃣ Local Reference Variables

* Store **addresses of heap objects**
* Reference removed when method ends
* Object becomes eligible for GC if no other references exist

6️⃣ PC Register (Program Counter Register)

The Program Counter (PC) Register stores the address of the next bytecode instruction to be executed by the JVM. Each thread has its own PC Register, which keeps track of the current execution position. After an instruction is executed, the PC Register is automatically updated to point to the next instruction, allowing the JVM to maintain proper execution flow and instruction sequencing during program execution.

7️⃣ Native Method Area

Native methods are declared in Java using the native keyword, but their actual implementation is written in C/C++. These implementations are stored inside native libraries (.dll, .so). When such a method is called, JVM loads the required native library and executes the code through the Native Method Area, allowing Java to interact directly with the operating system and perform low-level, platform-dependent tasks.

# 🔑 CORE CONCEPT: HASHCODE (First Principles)

## 1️⃣ What is an Object in Java?

In Java:

* An **object is a real entity created in memory**
* All objects are created **only in Heap memory**
* Each object must have:
  + Its **own identity**
  + Its **own memory space**
  + Its **own data**

👉 JVM must be able to **uniquely identify every object** it creates.

## 2️⃣ Why does JVM need Hashcode?

Imagine JVM creates **millions of objects**.

JVM must:

* Distinguish one object from another
* Track objects in memory
* Manage garbage collection
* Support fast lookup (HashMap, HashSet)

📌 For this purpose, JVM assigns **one unique number to every object**.

That number is called **hashcode**.

## 3️⃣ What exactly is Hashcode?

### ✅ Definition (core concept)

**Hashcode is a unique integer value generated by JVM to identify an object in memory.**

Simple words:

* Hashcode = **object identity number**
* Just like:
  + Aadhaar number for a person
  + Roll number for a student

## 4️⃣ Very Important Truths about Hashcode

### ✅ Hashcode represents:

* Object identity
* Object existence in heap
* JVM-level identification

### ❌ Hashcode does NOT represent:

* Memory address (not exact)
* Object data (id, name, etc.)
* Reference variable

## 5️⃣ When is Hashcode created?

📌 **At the moment the object is created**

new Student();

The instant this line executes:

* Heap memory is allocated
* Object is created
* Hashcode is generated
* Identity is fixed for that object

## 6️⃣ Relationship between Object and Reference Variable

### VERY IMPORTANT CONCEPT ⭐

Student obj = new Student();

* obj is **NOT the object**
* obj is a **reference variable**
* Reference variable stores:  
  👉 **object’s identity (hashcode reference)**

📌 That’s why diagrams show:

obj ───▶ Student Object

## 7️⃣ Default Values – Required Concept

When an object is created:

* JVM assigns **default values** automatically

| **Data type** | **Default value** |
| --- | --- |
| int | 0 |
| boolean | false |
| double | 0.0 |
| Object / String | null |

📌 This happens **before** you assign values manually.

## 8️⃣ One Golden Core Line (remember forever)

**Every object in Java has a unique hashcode that represents its identity in heap memory, and reference variables store that identity, not the object itself.**

# 🧠 NOW APPLYING THE CONCEPT TO YOUR EXAMPLE

Now that the core is clear, let’s use it.

## Example Code

Student obj = new Student(); // id = 10, name = pavan

Student obj1 = new Student(); // id = 7, name = kalyan

## Step 1️⃣ First Object Creation

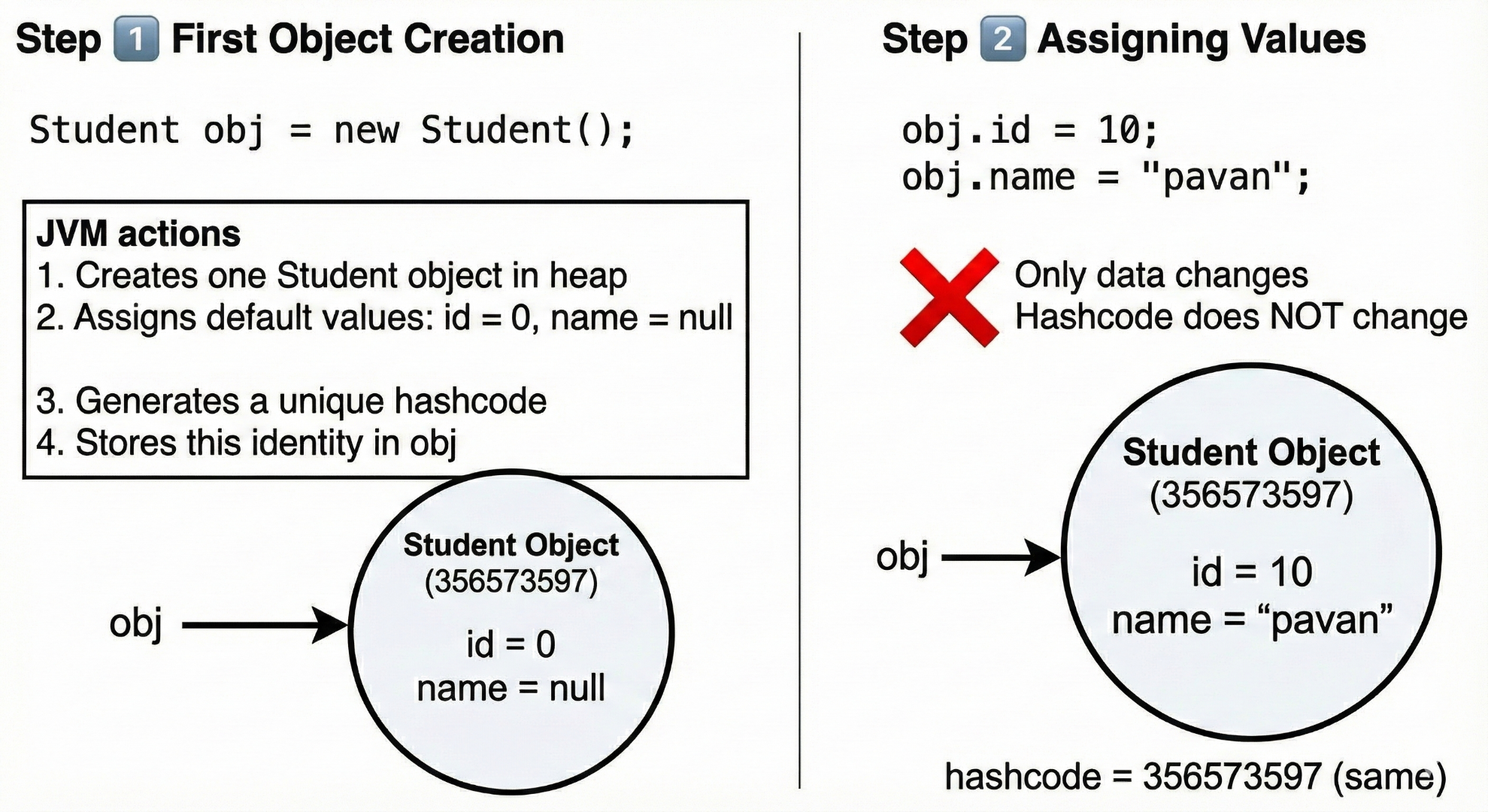
Student obj = new Student();

JVM actions:

1. Creates **one Student object** in heap
2. Assigns default values:
3. id = 0
4. name = null
5. Generates a **unique hashcode**
6. hashcode = H1 (example)
7. Stores this identity in obj

Representation:

obj ───▶ Student Object (hashcode H1)



## Step 2️⃣ Assigning Values

obj.id = 10;

obj.name = "pavan";

Only **data changes**  
❌ Hashcode does NOT change

hashcode = H1 (same)

## Step 3️⃣ Second Object Creation

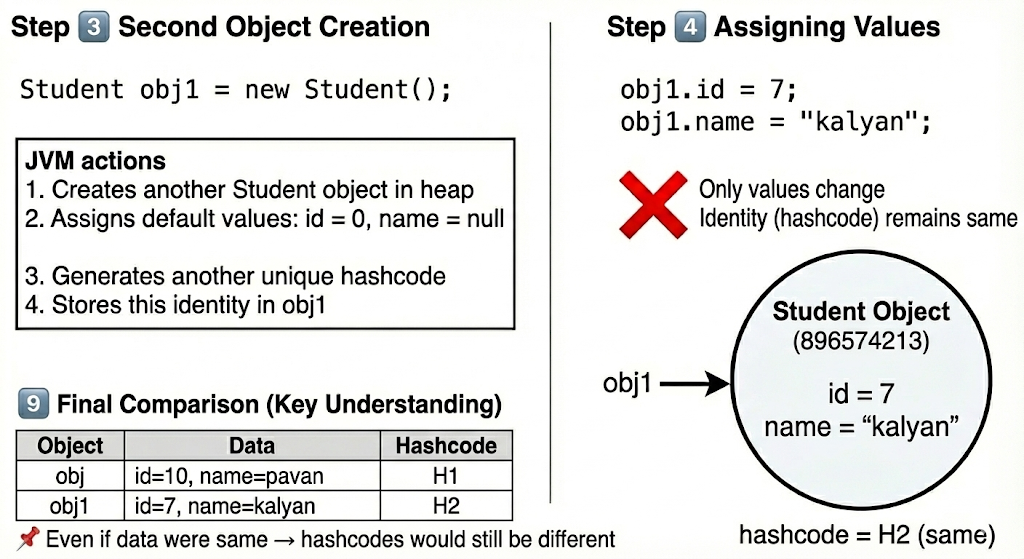
Student obj1 = new Student();

JVM again:

1. Creates **another Student object**
2. Assigns default values
3. Generates **another hashcode**
4. hashcode = H2 (different from H1)
5. Stores identity in obj1

Representation:

obj1 ───▶ Student Object (hashcode H2)



## Step 4️⃣ Assigning Values

obj1.id = 7;

obj1.name = "kalyan";

Again:

* Only values change
* Identity (hashcode) remains same

## 9️⃣ Final Comparison (Key Understanding)

| **Object** | **Data** | **Hashcode** |
| --- | --- | --- |
| obj | id=10, name=pavan | H1 |
| obj1 | id=7, name=kalyan | H2 |

📌 Even if data were same → **hashcodes would still be different**

## 🔥 Final One-Line Summary (Perfect Notes)

Hashcode is a JVM-generated unique identity for every object created in heap memory; reference variables store this identity, while object data can change without affecting the hashcode.

# 🔹 Accessing Instance Variables

Instance variables belong to an **object**.  
So they are accessed **only through an object context**.

There are **two valid ways** to access instance variables in Java.

## 1️⃣ Using the this keyword

### 🔑 Core Idea

this refers to the **current object**, so it can be used to access instance variables of that object.

### 📌 Where it is used

* Inside **constructors**
* Inside **instance methods**

### 📌 When it is REQUIRED

* When instance variable and local variable have the **same name**

### 📌 When it is OPTIONAL

* Inside instance methods when **no naming conflict** exists

### Example

this.name = name;

Here:

* this.name → instance variable
* name → local variable

### Important Note

❌ this cannot be used inside static methods  
Reason: static methods are not associated with any object

Eg: Instance\_Variable\_This\_Keyword\_3

## 2️⃣ Using an object of the class

### 🔑 Core Idea

Instance variables can be accessed using a **reference variable** that points to the object.

### 📌 Syntax

objectReference.variableName;

### Example

Instance\_Variable\_1 obj = new Instance\_Variable\_1();

System.out.println(obj.i);

Here:

* obj → reference variable
* i → instance variable stored in heap

### Important Note

* Reference variable stores **object identity**
* Using that reference, JVM locates the instance variable in heap
* Eg: Instance\_Variable\_1
* Instance\_Variable\_2

## ⚠️ Important Rule (Very Important)

❌ Instance variables **cannot** be accessed directly from static context  
✔ They must be accessed using:

* An object reference **OR**
* Through this (inside instance area)

## 🔥 Missing but Important Point (Added)

### Inside Instance Methods

Instance variables can be accessed **directly by name**:

System.out.println(i);

Why?

* Because instance method is already executed using an object
* JVM internally treats it as:

this.i

## 📊 Summary Table (Perfect for Notes)

| **Way** | **Where Used** | **Object Needed** | **Example** |
| --- | --- | --- | --- |
| this keyword | Constructor / Instance method | Yes (current object) | this.i |
| Object reference | Anywhere (including static) | Yes | obj.i |
| Direct access | Instance methods only | Implicit | i |

## ⭐ Final One-line Summary

Instance variables can be accessed using the this keyword inside instance areas or by using an object reference, while direct access is allowed only inside instance methods.