Types of Variables in Java

In Java, **every variable must satisfy two independent classifications**:

1. **Based on the type of value stored**
2. **Based on where and how it is declared**

These classifications are **fundamental for Oracle certification** and frequently tested through tricky scenarios.

1️⃣ Classification Based on the Type of Value

A. Primitive Variables

- Store **primitive values** directly
- Hold actual data (not references)

```
int x = 10;
boolean flag = true;
```

✔ Size is fixed
✔ Stored directly in memory

B. Reference Variables

- Store **references (addresses)** to objects
- Do not store the object itself

```
Student s = new Student();
```

✔ Reference variables point to objects stored on the heap
✔ Multiple references can point to the same object

2️⃣ Classification Based on Declaration & Behavior

Based on **where they are declared and how long they live**, variables are of **three types**:

1. Instance Variables
2. Static Variables
3. Local Variables

Instance Variables (Object-Level Variables)

Definition

If a variable's value **differs from object to object**, it should be an **instance variable**.

```
class Student {
    int marks;   // instance variable
}
```

Key Characteristics

- Declared **inside a class**, but **outside methods/constructors**
- Each object gets **its own copy**
- Created when an object is created
- Destroyed when the object becomes eligible for GC
- Stored on the **heap**
- JVM provides **default values**
- Also called **object-level variables / attributes**

Access Rules

Context	Access
Instance method	Direct access
Static method	❌ Not directly (requires object reference)

```
class Test {
    int i = 10;

    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t.i); // valid
    }

    public void show() {
        System.out.println(i);  // valid
    }
}
```

❌ This is invalid:

```
System.out.println(i); // non-static variable cannot be referenced
from static context
```

Static Variables (Class-Level Variables)

Definition

If a variable's value is **common to all objects**, it should be declared as **static**.

```
class College {
    static String name;
}
```

Key Characteristics

- Declared at class level using `static`

- **Single shared copy** per class
- Created when the class is loaded
- Destroyed when the class is unloaded
- Stored in **method area / metaspace**
- JVM provides **default values**
- Also called **class-level variables / fields**

Access Rules

Context	Access
Instance method	Direct
Static method	Direct

```
System.out.println(College.name);   // recommended
```

✔ Can be accessed using:

- Class name (recommended)
- Object reference (allowed but discouraged)

Instance vs Static (Exam Favorite)

```
class Test {
    int x = 10;
    static int y = 20;

    public static void main(String[] args) {
        Test t1 = new Test();
        t1.x = 888;
        t1.y = 999;

        Test t2 = new Test();
        System.out.println(t2.x + "----" + t2.y); // 10----999
    }
}
```

✔ Instance variable → separate copy
✔ Static variable → shared copy

Local Variables (Method / Block-Level Variables)

Definition

Variables declared **inside methods, constructors, or blocks** are called **local variables**.

```
public static void main(String[] args) {
    int x = 10; // local variable
}
```

Key Characteristics

- Stored on the **stack**
- Created when the block executes
- Destroyed when the block ends
- **No default values**
- Must be **explicitly initialized**
- Thread-safe by nature (each thread has its own stack)

Most Important Rule (Very High Exam Weight)

❌ JVM **does NOT provide default values** for local variables.

```
int x;
System.out.println(x); // Compilation error
```

✔ Valid:

```
int x = 0;
System.out.println(x);
```

Initialization Inside Conditional Blocks (Common Trap)

```
int x;
if (args.length > 0) {
    x = 10;
}
System.out.println(x); // ❌ compile-time error
```

✔ Correct approach:

```
int x = 0;
if (args.length > 0) {
    x = 10;
}
```

Modifiers and Local Variables

Only **one modifier** is allowed for local variables:

✔ `final`

❌ Not allowed:

- public
- private
- protected
- static
- volatile
- transient

Variable Scope Summary

Variable Type	Stored In	Default Value	Thread Safe
Instance	Heap	Yes	❌
Static	Method Area	Yes	❌
Local	Stack	❌	✔

Arrays and Default Values (Important Clarification)

Instance / Static Array

```
int[] a;
System.out.println(a);     // null
System.out.println(a[0]);  // NullPointerException
```

After Array Creation

```
int[] a = new int[3];
System.out.println(a[0]);  // 0
```

✔ **Array elements always get default values**
✔ But the **array reference itself does not**

Variable Combinations in Java

Every variable must be:

- **Instance / Static / Local**
- **Primitive / Reference**

Valid combinations:

- instance-primitive
- instance-reference

- static-primitive
- static-reference
- local-primitive
- local-reference

Passing Variables to Methods (Conceptual Clarity)

Passing Primitive Types

- Java is **pass-by-value**
- Changes do **not affect caller**

```
void m1(int x) {
    x = x * 2;
}
```

Passing Object References

- Reference is copied, **object is shared**
- Object state changes **are reflected**

```
void update(Product p) {
    p.price = 500;
}
```

✔ This is **not pass-by-reference**, but pass-by-value of reference

---

Certification-Ready Conclusions

- Local variables **must be initialized**

- Instance variables are object-specific
- Static variables are class-specific
- Arrays get default values only after creation
- Java is **strictly pass-by-value**
- Static methods cannot access instance variables directly
- Variable scope errors are **compile-time errors**