

◆ Static methods and memory

In Java, **static methods** (and also static variables) belong **to the class, not to any object**.

So, they are **stored in the “Method Area”** (also called **Non-Heap memory**) — not inside the heap.

Java Memory Areas (simplified)

Let's recall the key memory sections:

Memory Area	What It Stores	Examples
Heap Memory	Objects and their instance variables	<code>new Person()</code> , arrays, etc.
Stack Memory	Local variables and method call frames	Method parameters, local vars
Method Area (Non-Heap)	Class metadata, static variables, static methods, method bytecode	Class info like <code>Person.class</code> , static fields/methods
PC Registers	Current executing instruction address	Used internally by JVM
Native Method Stack	Used by native (non-Java) methods	JNI calls, etc.

◆ Static Method Details

When the JVM loads a class:

- The **class metadata, static variables, and static methods' bytecode** are placed in the **Method Area**.
- That means **static methods are not duplicated per object**, unlike instance methods.
- You can call them **without creating an object**, e.g.:

```
MyClass.display();
```

◆ Example

```
java Copy code

class Demo {
    static void show() {
        System.out.println("Static method");
    }

    void instanceShow() {
        System.out.println("Instance method");
    }
}

public class Main {
    public static void main(String[] args) {
        Demo.show(); // Static method -> Method Area (Non-Heap)

        Demo obj = new Demo();
        obj.instanceShow(); // Instance method -> Invoked on object in Heap
    }
}
```

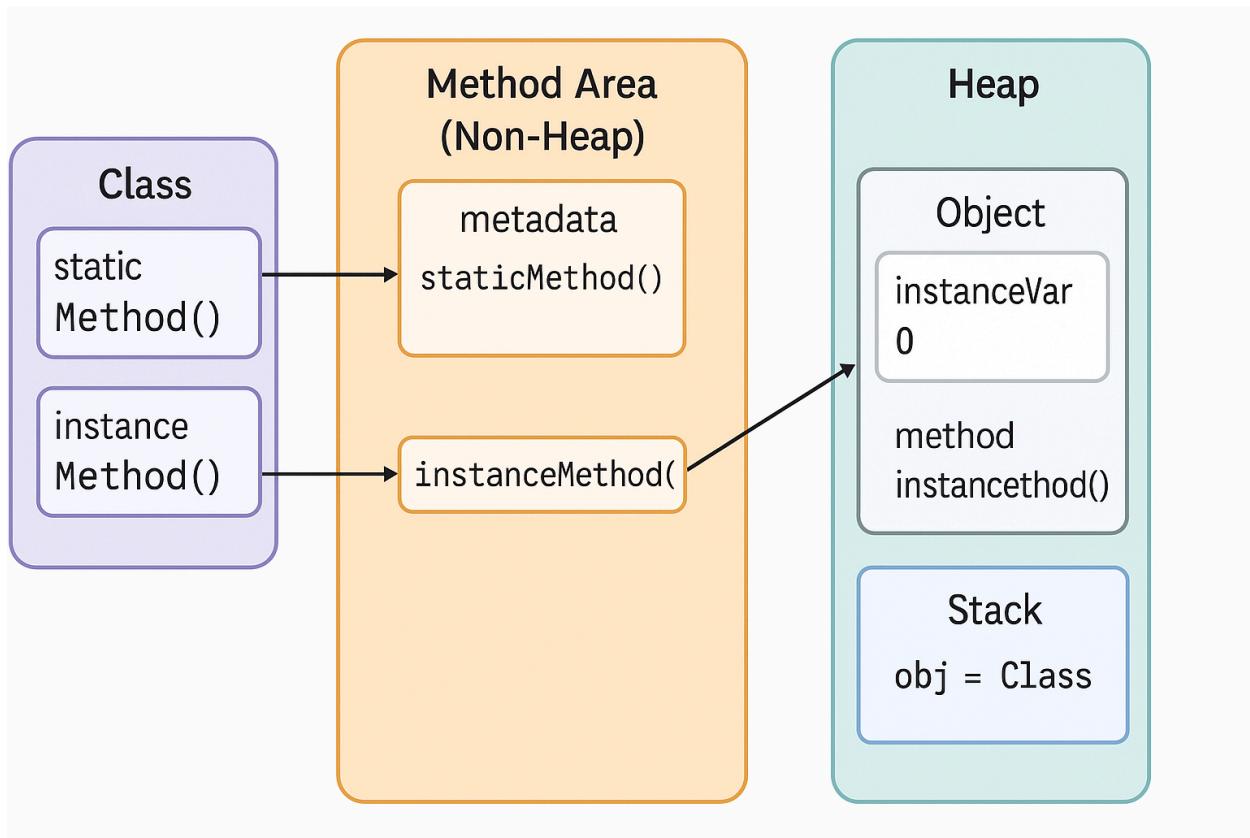
Here's what happens in memory:

- `Demo.class` → stored in **Method Area**.
- `show()` → stored in **Method Area (Non-Heap)**.
- `obj` → reference in **Stack**.
- `new Demo()` object → stored in **Heap**.



✓ Summary

Concept	Memory Location
Static method	Method Area (Non-Heap)
Static variable	Method Area (Non-Heap)
Instance method	Method Area , but invoked on objects in Heap
Object data	Heap
Local variables	Stack



🧠 1 What is “Metadata” in Java?

Metadata = “data about data.”

In Java’s memory model, **class metadata** means the information that the JVM stores about a **loaded class** — so that it knows **how to create objects, which methods exist, etc.**

📘 When a class is loaded by the JVM

When you load a class (say `Demo.class`), the JVM stores its metadata in the **Method Area (Non-Heap)**.

That metadata includes:

Metadata Component	Description
Class name	e.g. <code>"Demo"</code>
Superclass info	e.g. <code>java.lang.Object</code>

Implemented interfaces	e.g. <code>Serializable, Runnable</code>
Field info	List of all variables with data types
Method info	List of all methods, return types, access modifiers
Static variables and methods	Values and bytecode for static members
Constant pool	String literals, numeric constants, method references
Access flags	Whether the class is public, abstract, final, etc.

This **metadata** helps the JVM know:

- how to allocate memory for objects in the heap,
- how to call methods at runtime,
- and how to perform reflection (`Class.forName()`, `getDeclaredFields()`, etc.).

2 Difference between `void instanceShow()` and `public void instanceShow()`

Feature	<code>void instanceShow()</code>	<code>public void instanceShow()</code>
Access modifier	<code>default (package-private)</code>	<code>public</code>
Accessibility	Accessible only within the same package	Accessible from anywhere
Example usage	Accessible by any class in the same package only	Accessible from all packages
Bytecode difference	Same method signature but different access flag (<code>ACC_PUBLIC</code> vs none)	JVM stores access flag as <code>public</code> in metadata

Example

```
java Copy code
package pkg1;
class Demo {
    void instanceShow() { // default access
        System.out.println("Default access method");
    }
    public void publicShow() { // public access
        System.out.println("Public method");
    }
}

java Copy code
package pkg2;
import pkg1.Demo;
public class Test {
    public static void main(String[] args) {
        Demo obj = new Demo();
        obj.publicShow(); // ✓ Works (public)
        obj.instanceShow(); // ✗ Compile error (not visible outside pkg1)
    }
}
```

So the only difference between

`void instanceShow()` and `public void instanceShow()` is the **visibility (scope)** — functionally they are identical otherwise.

3 Summary Before the Diagram

Concept	Description	Stored In
Static methods & fields	Belong to the class	Method Area (Non-Heap)
Instance methods	Belong to class (only one copy of bytecode) but executed on object	Method Area (definition) + Heap (object data)
Objects & instance variables	Belong to instance	Heap
Local variables	Temporary data per method call	Stack
Metadata	Class info (fields, methods, modifiers, etc.)	Method Area

1 Access Modifiers in Java — Overview

Access modifiers define **visibility (or scope)** of **methods, variables, constructors, and classes**.

There are **four** main access modifiers in Java:

Modifier	Scope Level	Keyword
public	Most accessible	<code>public</code>
protected	Accessible in same package + subclasses	<code>protected</code>
default (package-private)	Accessible only within same package	(no keyword)
private	Least accessible	<code>private</code>

◆ 2 Access Levels (Summary Table)

Modifier	Same Class	Same Package	Subclass (diff package)	Other Packages
public	✓ Yes	✓ Yes	✓ Yes	✓ Yes
protected	✓ Yes	✓ Yes	✓ Yes (only in subclass)	✗ No
default (no modifier)	✓ Yes	✓ Yes	✗ No	✗ No
private	✓ Yes	✗ No	✗ No	✗ No

◆ 3 Examples

● Public

Accessible from **anywhere** (any package or class).

```
java
```

 Copy code

```
public class Demo {  
    public int x = 10;  
  
    public void show() {  
        System.out.println("Public method");  
    }  
}
```

```
java
```

 Copy code

```
Demo obj = new Demo();  
System.out.println(obj.x); //  Accessible anywhere  
obj.show(); //  Accessible anywhere
```

● Protected

Accessible in:

- Same package
- Subclasses (even in different packages)

```
java
```

 Copy code

```
package pkg1;  
public class Demo {  
    protected void display() {  
        System.out.println("Protected method");  
    }  
}
```

```
java
```

 Copy code

```
package pkg2;  
import pkg1.Demo;  
  
class SubDemo extends Demo {  
    void test() {  
        display(); //  Allowed (subclass)  
    }  
}
```

If you try `new Demo().display();` in `pkg2` (non-subclass),  it will not compile.

● Default (Package-Private)

If no modifier is given, it's visible only within the same package.

```
java

class Demo {      // default class
    int value = 20; // default variable
    void show() {   // default method
        System.out.println("Default access");
    }
}
```

 Copy code

Accessible only to other classes in the same package.

● Private

Accessible only within the same class — not even subclasses or other classes in same package can access.

```
java

public class Demo {
    private int data = 50;

    private void show() {
        System.out.println("Private method");
    }

    public void display() {
        show(); // ✓ accessible inside same class
    }
}
```

 Copy code

Outside the class:

```
java

Demo obj = new Demo();
System.out.println(obj.data); // ✗ Compile error
obj.show();                 // ✗ Compile error
```

 Copy code



4 Best Practices

Situation	Recommended Modifier
Class used everywhere	<code>public</code>
Method used only inside class	<code>private</code>
Method for inheritance (to be overridden)	<code>protected</code>
Helper classes in same package	<code>default</code> (no keyword)



5 Bonus Tip — Class-level modifiers

- **Top-level class** can be `public` or `default` only.
(Can't be `private` or `protected`.)
 - **Nested classes** (inner classes) can be any modifier (`private`, `protected`, etc.).
-



Quick Summary

Modifier	Visibility	Common Use
<code>public</code>	Everywhere	API methods, utility classes
<code>protected</code>	Same package + subclasses	Inheritance-related
<code>default</code>	Same package only	Package-internal helpers
<code>private</code>	Within same class	Encapsulation



1 `public void method()` vs `public static void method()`

Feature	<code>public void method()</code>	<code>public static void method()</code>
---------	-----------------------------------	--

Belongs to	The object (instance)	The class itself
Requires object to call	✓ Yes (<code>obj.method()</code>)	✗ No (can call using class name)
Can access instance variables?	✓ Yes	✗ No (only static vars)
Can access static variables?	✓ Yes	✓ Yes
Can use <code>this</code> keyword?	✓ Yes	✗ No (no object context)
Memory storage	Method Area (definition) + invoked via object in Heap	Method Area (Non-Heap)
Example call	<code>new Demo().show();</code>	<code>Demo.display();</code>

💡 2 Example to Understand Clearly

```
java Copy code

public class Demo {

    int instanceVar = 10;           // instance variable → belongs to object
    static int staticVar = 20;       // static variable → belongs to class

    // Instance method
    public void show() {
        System.out.println("Instance method:");
        System.out.println("instanceVar = " + instanceVar); // ✓ allowed
        System.out.println("staticVar = " + staticVar);      // ✓ allowed
    }

    // Static method
    public static void display() {
        System.out.println("Static method:");
        // System.out.println("instanceVar = " + instanceVar); ✗ ERROR
        System.out.println("staticVar = " + staticVar);      // ✓ allowed
    }

    public static void main(String[] args) {
        Demo.display(); // ✓ directly using class name
    }
}
```

↓

3 Key Rules about Static Methods

Concept	Description
Can't access instance variables/methods directly	Because they belong to objects, not class
Can access only static variables/methods directly	Both are class-level
Can be called without creating an object	Using class name
No <code>this</code> or <code>super</code> keyword	Because they refer to instance context
Useful for utility/helper methods	e.g., <code>Math.max()</code> , <code>Collections.sort()</code>
Cannot be overridden (truly)	Static methods are hidden , not overridden
Loaded once in memory	Stored in Method Area when class is loaded

4 Static Variables (Class Variables)

Property	Description
Belong to class	Shared by all instances
Memory	Stored in Method Area (Non-Heap)
Created once	When class is loaded by JVM
Common use	Constants, counters, configuration
Access	<code>ClassName.variable</code> or <code>obj.variable</code> (both work, but class name is preferred)

Example:

```
java Copy code  
  
class Counter {  
    static int count = 0; // shared among all objects  
  
    Counter() {  
        count++;  
        System.out.println("Object created, count = " + count);  
    }  
  
    public static void showCount() {  
        System.out.println("Total objects created: " + count);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        new Counter();  
        new Counter();  
        new Counter();  
  
        Counter.showCount(); // ✅ prints 3  
    }  
}
```

5 Static Blocks

Static blocks are executed **once**, when the class is loaded — useful for **initializing static data**.

```
java Copy code  
  
class Example {  
    static int value;  
  
    static {  
        System.out.println("Static block executed");  
        value = 100;  
    }  
  
    public static void show() {  
        System.out.println("Value = " + value);  
    }  
  
    public static void main(String[] args) {  
        Example.show(); // triggers class loading + static block  
    }  
}
```

Output:

```
java Copy code  
  
Static block executed  
Value = 100
```

6 Static Import (Bonus)

You can import static members of a class to use them **without class name**:

```
java                                     ⌂ Copy code

import static java.lang.Math.*; // imports all static members of Math

public class Test {
    public static void main(String[] args) {
        System.out.println(sqrt(25)); // no need for Math.sqrt()
        System.out.println(max(10, 20));
    }
}
```

7 Summary Table

Feature	Static	Non-static (Instance)
Belongs to	Class	Object
Memory	Method Area (Non-Heap)	Heap
Can access	Only static data	Both static + instance data
Requires object	 No	 Yes
this keyword	 Not allowed	 Allowed
Lifetime	Till class unloaded	Till object garbage collected
Example use	Utilities, counters	Business logic tied to object

✳️ 1 Normal `hashCode()` vs `System.identityHashCode()`

Every Java object has a `hashCode()` method (inherited from `Object`):

java

Copy code

```
public int hashCode()
```

Normally, when you don't override it →

- ➡️ it returns a hash value derived from the **object's memory reference** (like its address).

So two **different objects** will almost always have **different hash codes**.

But when you **override** `hashCode()` (like in your own classes),

- ➡️ you change its meaning to represent the **data inside the object**, not its location in memory.

🧠 Example 1: Without overriding `hashCode()`

java

Copy code

```
class A { }

public class Main {
    public static void main(String[] args) {
        A obj1 = new A();
        A obj2 = new A();

        System.out.println(obj1.hashCode()); // different
        System.out.println(obj2.hashCode()); // different
    }
}
```



Output:

Output:

366712642
1829164700

✓ These are **identity-based hash codes** — derived from the object's location in memory.

🧠 Example 2: With overridden `hashCode()`

```
java Copy code

class Student {
    int roll;
    Student(int roll) { this.roll = roll; }

    @Override
    public int hashCode() {
        return roll; // based on data, not memory
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(101);
        Student s2 = new Student(101);

        System.out.println(s1.hashCode()); // 101
        System.out.println(s2.hashCode()); // 101
    }
}
```

✿ Here both `s1` and `s2` have different memory locations,
but same data (`roll = 101`) → so their `hashCode()` is same.

👉 That's useful for logical equality in `HashSet`, `HashMap`, etc.
But sometimes, we want to see the real physical identity (not overridden logic).



🧠 2 Enter `System.identityHashCode(obj)`

This method gives you the original, JVM-assigned identity-based hash,
even if you override `hashCode()`.

`System.identityHashCode(Object obj)`

- Returns a hash code based on the object's identity, not the overridden `hashCode()` method.
- This means even if a class overrides `hashCode()` (e.g., for equality or hashing in collections), this method will still return a number derived from the actual memory

reference of the object (like its "address fingerprint").

So:

- It doesn't give the **real physical memory address** (Java hides that for safety).
- But it gives a **unique integer identifier** that reflects the object's **identity in the JVM**.

```
java

class Student {
    int roll;
    Student(int roll) { this.roll = roll; }

    @Override
    public int hashCode() {
        return roll;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(101);
        Student s2 = new Student(101);

        System.out.println("s1.hashCode(): " + s1.hashCode()); // 101
        System.out.println("s2.hashCode(): " + s2.hashCode()); // 101

        System.out.println("s1.identityHashCode(): " + System.identityHashCode(s1));
        System.out.println("s2.identityHashCode(): " + System.identityHashCode(s2));
    }
}
```

Output:

```
makefile
↓

s1.hashCode(): 101
s2.hashCode(): 101
s1.identityHashCode(): 366712642
s2.identityHashCode(): 1829164700
```

what is the purpose of using `toString` method?

ChatGPT said:

The **purpose of the `toString()` method** in Java is to provide a **human-readable string representation** of an object.

In simple words — it's used to **convert an object into a string form** that describes its contents, rather than just showing a memory address.

◆ Default Behavior

Every class in Java **inherits** the `toString()` method from `Object` class.

```
public String toString()
```

If you **don't override** it, the default version from `Object` looks like this:

```
ClassName@HexadecimalHashCode
```

For example:

For example:

```
java Copy code
class Student {
    int id;
    String name;
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.toString());
    }
}
```

Output:

```
css Copy code
Student@5acf9800
```

⚠ This output isn't helpful because it just shows the class name and memory hash.

♦ Why Override `toString()`?

To make the output **meaningful** and **readable**.

You override `toString()` to describe the **object's data** instead of memory info.

Example:



Example:

```
java Copy code
class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{id=" + id + ", name='" + name + "'}";
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student(101, "Raj");
        System.out.println(s);
    }
}
```

Output:

```
bash Copy code
Student{id=101, name='Raj'}
```



◆ Common Use Cases

1. Debugging and Logging

- Makes logs readable when printing objects

```
System.out.println(user); // prints useful info instead of hashCode
```

2. Printing Collections

- Collections like `ArrayList` or `HashMap` internally call `toString()` on elements.

3. String Concatenation

Automatically called when an object is added to a string:

```
"User info: " + user
```

- → compiler calls `user.toString()` implicitly.

◆ Summary

Purpose	Description
Human-readable output	Gives meaningful information about object contents
Debugging aid	Makes printed/logged info understandable
Implicitly called	Used when printing or concatenating objects
Recommended practice	Always override in custom classes\

✳️ 1. Printing Objects Directly

Whenever you print an object using `System.out.println(obj)`, Java automatically calls `obj.toString()` behind the scenes.

```
java Copy code

class Car {
    String model;
    int year;

    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    @Override
    public String toString() {
        return "Car{model='" + model + "', year=" + year + "}";
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Tesla Model Y", 2024);

        System.out.println(car); // ✅ Java calls car.toString() automatically
    }
}
```

Output: `Car{model='Tesla Model Y', year=2024}`

👉 Without the override, you'd see something like `Car@5acf9800`.

✳️ 2. String Concatenation

When you combine an object with a string using `+`, the compiler **automatically invokes `toString()`**.

```
System.out.println("Car details: " + car);
```

This internally becomes:

```
System.out.println("Car details: " + car.toString());
```

Output:

```
Car details: Car{model='Tesla Model Y', year=2024}
```

3. Logging

When you log objects (e.g., with frameworks like Log4j, SLF4J, or `System.out`), `toString()` helps produce meaningful logs.

```
System.out.println("Saving to database: " + car);
```

Output:

```
Saving to database: Car{model='Tesla Model Y', year=2024}
```

Without a proper `toString()`, the log would be useless like:

```
Saving to database: Car@5acf9800
```

4. Collections (Lists, Sets, Maps)

Collections automatically call `toString()` for each element when printed.

```
List<Car> cars = new ArrayList<>();  
  
cars.add(new Car("Tesla", 2023));  
  
cars.add(new Car("BMW", 2022));  
  
System.out.println(cars);
```

Output: [Car{model='Tesla', year=2023}, Car{model='BMW', year=2022}]

Each element's `toString()` is called internally by `ArrayList.toString()`.

5. Debugging in IDEs

When you hover over an object during debugging (e.g., in IntelliJ or Eclipse), the IDE often calls `toString()` to show a quick summary of the object.

In summary

Case	<code>toString()</code> Called Automatically?	Example
<code>System.out.println(obj)</code>	 Yes	<code>System.out.println(car)</code>
String concatenation	 Yes	<code>"Car: " + car</code>
Logging	 Yes	<code>logger.info("Obj: " + obj)</code>
Printing Collections	 Yes	<code>System.out.println(list)</code>
Debugger display	 Yes	IDE shows readable info

⌚ What is an Iterator?

An **Iterator** is an object provided by Java collections to **traverse (loop through) elements** of a collection **sequentially, without exposing the internal representation** (like array, linked list, etc.).

You usually get it by:

```
Iterator<Integer> it = list.iterator();
```

Then use:

```
while (it.hasNext()) {  
    Integer val = it.next();  
    System.out.println(val);  
}
```

✓ Benefits of Using Iterator

1. Uniform Way to Traverse Any Collection

- Whether it's an `ArrayList`, `HashSet`, `LinkedList`, or even custom collections — all provide `iterator()`.
- You don't need to know how elements are stored internally.

Example:

```
java Copy code

Collection<String> col = new HashSet<>();
col.add("A"); col.add("B"); col.add("C");

Iterator<String> it = col.iterator(); // works for any collection
while (it.hasNext()) {
    System.out.println(it.next());
}
```

2. Safe Removal of Elements During Traversal

- Removing elements directly from a collection while looping (like using `for-each`) causes `ConcurrentModificationException`.
- But `Iterator` provides a **safe way** via `iterator.remove()`.

Example:

```
java Copy code

List<Integer> list = new ArrayList<>(List.of(1, 2, 3, 4));
Iterator<Integer> it = list.iterator();

while (it.hasNext()) {
    if (it.next() % 2 == 0) {
        it.remove(); // Safe removal
    }
}
System.out.println(list); // [1, 3]
```

If you did this with a `for-each` loop, it would crash.

3. Hides Implementation Details (Encapsulation)

- You don't need to know *how* the collection stores elements.
- The collection exposes a consistent way to iterate without leaking its structure.

Example:

- `ArrayList` uses an internal array.
- `LinkedList` uses nodes.
- With an iterator, you don't care — you just get one element at a time.