

Object Lifecycle Explanation



give me the deep and detailed explanation of Object Lifecycle so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explanation

Sure! ✓

I will explain **Object Lifecycle in Java** in a very **deep, detailed, and easy way** so that you will understand everything clearly without any doubt.

Let's begin step-by-step.

🚀 Object Lifecycle in Java (Complete Detailed Explanation)

In Java, an **object** goes through different phases from the moment it is **created**, **used**, and finally **destroyed**.

This complete journey is called **Object Lifecycle**.

♦ 1 Class Declaration → Blueprint Stage

Before creating an object, we need a **class**.

- ✓ A **class** is just a blueprint/template
- ✓ It does **not occupy memory** for instance variables
- ✓ It only defines **properties (variables)** and **behaviors (methods)**

Example:

```
java

class Student {
    String name;
    int age;

    void study() {
        System.out.println(name + " is studying");
    }
}
```

At this stage → **No object exists yet**

♦ 2 Object Creation → Instantiation Stage

Objects are created using the **new keyword**

👉 Memory allocation happens inside the **Heap Memory**

Example:

```
java
Student s1 = new Student();
```

What happens internally?

Action	Description
new keyword	Allocates memory in heap
Constructor call	Initializes object values
Reference variable	Stores address of object in stack

📌 After creation

→ Object lives in **Heap Memory**

→ Reference variable (s1) lives in **Stack Memory**

◆ 3 Object Initialization

After creation, Java initializes the object:

✓ Default values assigned to variables if not initialized manually:

Data Type	Default Value
int	0
float	0.0
boolean	false
Object (String etc.)	null

Constructor can also initialize values:

```
java
Student s2 = new Student("Piyush", 21);
```

◆ 4 Object Usage Stage

The object now performs actions:

- ✓ Methods are called
- ✓ Values are updated
- ✓ Object interacts with other objects

Example:

```
java

s1.name = "John";
s1.age = 20;
s1.study();
```

🚧 During usage, object keeps living in heap memory until it is no longer reachable.

♦ **5 Object Becomes Unreachable (Eligible for Garbage Collection)**

An object becomes **unreachable** when:

Case **1** Reference assigned to another object

```
java

Student s1 = new Student();
s1 = new Student(); // old object lost
```

Case **2** Reference variable goes out of scope

```
java

void test() {
    Student s = new Student(); // Local object
} // After method ends → 's' gets destroyed → object unreachable
```

Case **3** Reference set to null explicitly

```
java

s1 = null;
```

🚧 When Java detects no references pointing to an object →
It becomes **eligible** for **Garbage Collection**.

♦ **6 Garbage Collection (GC) → Automatic Cleanup**

Java has **Automatic Memory Management**.

The **Garbage Collector** removes unused objects from heap memory.

- ✓ GC uses mark and sweep algorithm
- ✓ Frees memory space
- ✓ Prevents memory leaks
- ✓ Improves performance

You can request GC (Not guaranteed immediately)

```
java
System.gc();
```

7 Finalization (Before Destruction)

Before the object is destroyed, GC calls:

```
java
protected void finalize()
```

Example:

```
java
@Override
protected void finalize() {
    System.out.println("Object is being destroyed");
}
```

⚠ Finalize is **deprecated** after Java 9 → Not recommended (unpredictable timing)

8 Object Destruction

Finally, unused objects are removed from heap memory permanently.

- ✓ Memory is released
- ✓ Object no longer exists

GC handles destruction → Programmer does NOT manually delete objects (unlike C++)

📌 Summary Table

Stage	Description	Who Controls?
Class Definition	Blueprint	Programmer
Object Creation	Memory allocated	Programmer (new)
Initialization	Variables set	Constructor
Usage	Methods executed	Programmer
Unreachable State	Object no longer referenced	Runtime
Garbage Collection	Automatic cleanup	JVM GC
Destruction	Memory released	JVM

🧠 Important Notes

Concept	Meaning
Stack Memory	Stores reference variables
Heap Memory	Stores actual objects
JVM	Manages lifecycle automatically
GC	Frees unused objects
finalize()	Called (sometimes) before destruction

Real World Example (Full Lifecycle)

```
java

class Car {
    String name;

    Car(String name) {
        this.name = name;
    }

    void drive() {
        System.out.println(name + " is driving...");
    }

    @Override
    protected void finalize() {
        System.out.println(name + " destroyed.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c1 = new Car("BMW"); // created
        c1.drive(); // used

        c1 = null; // unreferenced

        System.gc(); // request GC to destroy
    }
}
```

Output (may vary):

```
csharp

BMW is driving...
BMW destroyed.
```

★ Why Object Lifecycle is Important?

- ✓ Efficient memory usage
- ✓ No memory leaks
- ✓ Better performance
- ✓ Clean and managed object handling
- ✓ Makes Java safer than languages like C++

Final Understanding

"Objects are born, they live, and then they die...
But in Java, JVM takes care of their life and death."

If you want, I can also give you:

- ✓ A flowchart of Object Lifecycle
- ✓ A diagram showing stack vs heap memory
- ✓ Interview questions based on this topic
- ✓ MCQs for practice
- ✓ Notes in PDF format

Would you like that? 😊