Java Memory Management

Here’s a **complete beginner-friendly guide to Java Memory Management**, starting from scratch — with **diagrams**, **code examples**, and clear **step-by-step explanations**.

🧠 What Is Java Memory Management?

Java memory management is the process of **allocating**, **using**, and **releasing** memory in a Java program. Java uses an automatic system called the **Garbage Collector (GC)** to manage memory.

🗂️ Java Memory Areas

When you run a Java program, memory is divided mainly into:

Area	Purpose
Heap	Stores objects and class instances
Stack	Stores method calls and local variables
Method Area (MetaSpace)	Stores class-level data (e.g. method code, static variables)
PC Register	Stores current executing instruction for each thread
Native Method Stack	Stores native (non-Java) methods

🔄 Java Memory Lifecycle

1. **Code is compiled**
2. **JVM loads classes**
3. **Objects created on heap**
4. **Method calls use stack frames**
5. **Garbage collector frees unused memory**

🔥 Java Heap vs Stack – Key Differences

Feature	Heap	Stack
Stores	Objects	Method calls, local vars
Access	Global	Thread-limited
Memory size	Larger	Smaller
Lifetime	Until GC collects it	Until method returns
Speed	Slower	Faster

✅ Code Example: Stack and Heap in Action

```java
public class MemoryExample {
    public static void main(String[] args) {
        int localVar = 10; // stored in stack

        Person person = new Person("John"); // object in heap
        person.sayHello();
    }
}

class Person {
    String name; // stored in heap

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

    void sayHello() {
        String greeting = "Hello " + name; // greeting is on stack
        System.out.println(greeting);
    }
}
```

🔍 Explanation:

- **`localVar`** and **`greeting`** → go on the **stack**.
- **`person`** (reference) → on stack, but the actual **`Person`** object → on **heap**.
- **`name`** field → on **heap** because it's part of the object.

🗑️ Java Garbage Collection (GC)

- Java automatically frees memory using the **Garbage Collector**.
- It removes objects from heap that are **no longer reachable**.

Example:

```java
public class GCExample {
    public static void main(String[] args) {
        Person p1 = new Person("Alice");
        p1 = null; // no reference now

        System.gc(); // Hint to JVM to run garbage collection
    }
}
```

Output:

You *may* see:

```csharp
Person object is garbage collected
```

To print a message on GC:

```java
@Override
protected void finalize() throws Throwable {
    System.out.println("Person object is garbage collected");
}
```

> ⚠️ `finalize()` is deprecated in Java 9+ and not recommended for production use.

🧪 Memory Leak Example in Java

Although Java manages memory, **you can still create memory leaks** if you hold references you no longer need:

```java
import java.util.*;

public class MemoryLeakExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        while (true) {
            list.add("Leak" + new Random().nextInt()); // keeps growing
        }
    }
}
```

This causes **OutOfMemoryError** eventually because the `list` grows infinitely and objects are never freed.

🧱 JVM Memory Diagram

```pgsql
+------------------------+
|        Method Area     |
|  - Class info          |
|  - Static variables    |
+------------------------+
|          Heap          |
|  - Objects             |
|  - Instance variables  |
+------------------------+
|         Stack          |
|  - Method calls        |
|  - Local variables     |
+------------------------+
```

🛠️ Tips for Better Memory Management in Java

- Avoid unnecessary object creation.
- Use local variables as much as possible (stack memory).
- Nullify object references when no longer needed.
- Use `WeakReference` when appropriate to avoid memory leaks.
- Monitor heap usage using tools like:
    - **VisualVM**
    - **Eclipse Memory Analyzer (MAT)**
    - **JConsole**

🚨 Common Errors

Error	Cause
`OutOfMemoryError`	Heap memory full
`StackOverflowError`	Too many recursive method calls
`GC Overhead Limit`	Too much time spent in GC

📘 Summary

- Stack stores method calls and local variables.
- Heap stores objects and instance variables.
- Garbage Collector frees unused heap memory.
- You can leak memory in Java if references are not properly cleared.

Would you like a **visual animation or diagram** of memory flow or a **hands-on mini-project** to solidify your understanding?