Java as a Compiled Language vs Interpreted Language

Java is considered **both compiled and interpreted** because it uses a **two-step execution model**:

Compiled Language (Java Part)

- Java source code (`.java`) is first **compiled by `javac` (Java compiler)**.
- Compiler converts the code into **bytecode (`.class` files)**.
- Bytecode is **platform-independent** (can run on any OS with JVM).

Example:

```
javac MyProgram.java   // compilation step → MyProgram.class (bytecode)
```

Interpreted Language (Java Part)

- The **Java Virtual Machine (JVM)** reads and **interprets bytecode** line by line.
- In modern JVMs, the **JIT (Just-In-Time) compiler** converts bytecode into **native machine code** for faster execution.

Example:

```
java MyProgram   // JVM interprets and runs the bytecode
```

Summary:

- **Compiled** → `.java` → `.class` (bytecode)
- **Interpreted** → JVM executes bytecode line by line (or compiles it just-in-time).
- That's why Java is often called a **"compiled + interpreted language"**.

Packages in Java

What is a Package?

- A **package** in Java is like a **folder** that groups related classes, interfaces, and sub-packages together.
- Used to **organize code**, **avoid name conflicts**, and **provide access control**.

Think of it like folders on your computer:

```
com.techvision.utils
```

Here,

- `com` = top-level package
- `techvision` = sub-package
- `utils` = sub-package containing classes

Types of Packages

1. **Built-in Packages** (provided by Java)
   - Example:
     - `java.util` → Scanner, ArrayList
     - `java.io` → File, BufferedReader
2. **User-defined Packages** (created by programmers)
   - You can create your own packages for organizing project files.

Creating and Using a Package

Step 1 – Create a Package

```
// File: MyClass.java
package mypackage;   // declare package

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass in mypackage");
    }
}
```

Step 2 – Compile with package

```
javac -d . MyClass.java
// javac -d . *.java                // for all file
```

(`-d .` creates folder structure for package)

Step 3 – Use the Package

```
// File: Main.java
import mypackage.MyClass;   // import the package

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

Benefits of Packages

- **Code Reusability** – Organized libraries.
- **Avoids Naming Conflicts** – Two classes with the same name can exist in different packages.
- **Access Protection** – `public`, `protected`, `default` access levels depend on package use.
- **Modularity** – Large projects are easier to manage.

Access Modifiers in Java

1. What are Access Modifiers?

- **Access Modifiers** in Java are **keywords** that define the **scope (visibility)** of classes, methods, variables, and constructors.
- They control **who can access what** in a program.

2. Types of Access Modifiers

Java provides **four main access modifiers**:

Modifier	Class	Package	Subclass	Other Packages
private	Yes	No	No	No
default (no keyword)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

3. Explanation with Examples

(i) private

- Members are **accessible only inside the same class**.
- Cannot be accessed outside the class, not even by subclasses.

Example:

```
class A {
    private int data = 40;
    private void show() {
        System.out.println("Private method");
    }
}

public class Main {
    public static void main(String[] args) {
        A obj = new A();
        // System.out.println(obj.data); // Error
        // obj.show(); // Error
    }
}
```

(ii) default (no modifier)

- When no modifier is specified, it is **package-private**.
- Accessible only **within the same package**.

Example:

```
class A {
    int data = 50; // default
    void show() {
        System.out.println("Default method");
    }
}

class B {
    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj.data); // Allowed (same package)
        obj.show();
    }
}
```

(iii) protected

- Accessible within the **same package** and also in **subclasses of other packages**.

Example:

```
package pack1;
public class A {
    protected void display() {
        System.out.println("Protected method in A");
    }
}

package pack2;
import pack1.A;

class B extends A {
    public static void main(String[] args) {
        B obj = new B();
        obj.display(); // Allowed (subclass in different package)
    }
}
```

(iv) public

- Accessible **from anywhere** in the program.

Example:

```
class A {
    public void show() {
        System.out.println("Public method");
    }
}

public class Main {
    public static void main(String[] args) {
        A obj = new A();
        obj.show(); // Accessible everywhere
    }
}
```

4. Quick Summary Table

Modifier	Scope
private	Only within the same class
default	Within the same package only
protected	Within same package + subclasses in other packages
public	Accessible everywhere

Multithreading in Java

- **Multithreading** in Java is the ability to run **multiple parts of a program (threads)** at the same time.
- A **thread** is the smallest independent unit of execution in a program.
- Java supports multithreading by providing the **`Thread` class** and the **`Runnable` interface**.

Advantages of Multithreading

1. **Efficient CPU usage** – prevents CPU from staying idle.
2. **Faster execution** – multiple tasks run in parallel.
3. **Better responsiveness** – useful in GUI or server applications.
4. **Easy asynchronous execution** – tasks can run in background.

Creating Threads in Java

1. By Extending `Thread` class

- We can create a new class that **extends `Thread`** and overrides its `run()` method.
- Then create an object and call `start()` method.

```
class MyThread1 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("MyThread1 running: " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread1 t1 = new MyThread1(); // create object of thread
        t1.start();  // starts the thread, internally calls run()

    }
}
```

Here `run()` contains the task of the thread. `start()` is always used to begin execution, not `run()` directly.

2. Multiple Threads Example

- We can create multiple threads that run **simultaneously**.
- Output order is **not guaranteed** because threads run in parallel.

```
class Task1 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task1 running: " + i);
        }
    }
}

class Task2 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task2 running: " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Task1 t1 = new Task1();
        Task2 t2 = new Task2();

        t1.start(); // start first thread
        t2.start(); // start second thread
    }
}
```

Both threads run together, so the output may look **mixed** like:

```
Task1 running: 1
Task2 running: 1
Task1 running: 2
Task2 running: 2
...
```

3. By Implementing `Runnable` interface

- Another way is to create a class that **implements `Runnable`** and defines the `run()` method.
- Then create a `Thread` object and pass the `Runnable` object to it.

```
class Task1 implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task1 running: " + i);
        }
    }
}

class Task2 implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task2 running: " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Create runnable objects
        Task1 task1 = new Task1();
        Task2 task2 = new Task2();

        // Wrap runnable objects inside Thread objects
        Thread t1 = new Thread(task1);
        Thread t2 = new Thread(task2);

        // Start both threads
        t1.start();
        t2.start();
    }
}
```

Output (order may vary)

```
Task1 running: 1
Task2 running: 1
Task1 running: 2
Task2 running: 2
...
```

This method is often preferred because Java supports **multiple inheritance through interfaces**, not classes.

Commonly Used Thread Methods

- `start()` → starts the execution of a thread.
- `run()` → contains the actual code for the thread.
- `sleep(ms)` → pauses the thread for given milliseconds.
- `join()` → waits for another thread to complete.
- `getName()` / `setName()` → gets/sets the name of a thread.
- `getPriority()` / `setPriority()` → manages priority (1–10).
- `isAlive()` → checks if the thread is still running.

Java Thread Life Cycle

In Java, a **thread** goes through different states from its creation to termination. These states are managed by the **JVM** and **Thread Scheduler**.

Thread States

1. **New (Created State)**

When a thread object is created using the `Thread` class or `Runnable` interface but **not yet started** using `start()`.

Example:

```
Thread t = new Thread();
```

2. Runnable (Ready State)

When `start()` is called, the thread enters the **Runnable pool**.
It is ready to run but waiting for the **CPU scheduler** to give it time.

Example:

```
t.start();
```

3. Running

When the CPU scheduler picks the thread, it goes into the Running state.
Only one thread per core can be in the running state at a time.

4. Waiting

A thread can be put into waiting using methods like `wait()`.
It waits until another thread signals it to continue.

5. Timed Waiting (Sleep/Join State)

A thread is in **timed waiting** when it is made to wait for a specific amount of time.

Example:

```
    sleep(milliseconds)
    join(milliseconds)
```

6. Terminated (Dead)

When a thread finishes execution, it enters the **Terminated state**.
Once dead, a thread cannot be restarted.

Diagram (Thread Life Cycle):

```
New → Runnable → Running → Waiting / Timed Waiting → Runnable → Running →
Terminated
```

Java Thread Class Constructors

The `Thread` class provides several constructors to create thread objects in different ways.

Constructors of Thread Class

1. `Thread()`

Creates a thread with no target and a default name.

```
Thread t1 = new Thread();
```

2. `Thread(Runnable target)`

Creates a thread with a **Runnable target**.

```
Runnable r = () -> System.out.println("Task running...");
Thread t2 = new Thread(r);
```

3. `Thread(String name)`

Creates a thread with a given **name**.

```
Thread t3 = new Thread("MyThread");
```

4. `Thread(Runnable target, String name)`

Creates a thread with both **Runnable target** and a **name**.

```
Runnable r2 = () -> System.out.println("Named thread running...");
Thread t4 = new Thread(r2, "WorkerThread");
```

5. `Thread(ThreadGroup group, Runnable target)`

Assigns the thread to a **ThreadGroup** with a Runnable task.

6. `Thread(ThreadGroup group, String name)`

Creates a named thread inside a specific **ThreadGroup**.

7. `Thread(ThreadGroup group, Runnable target, String name)`

Creates a thread with group, target, and name.

1. Thread(String name) Constructor

- This constructor is used to create a **thread with a specific name**.
- It does not take any task (Runnable) as argument, so you must **override `run()`** method separately.

Example

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);   // call parent constructor to set thread name
    }

    public void run() {
        System.out.println("Thread is running: " + getName());
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Worker-1");
        MyThread t2 = new MyThread("Worker-2");

        t1.start();
        t2.start();
    }
}
```

Output (order may vary):

```
Thread is running: Worker-1
Thread is running: Worker-2
```

2. Thread(Runnable target, String name) Constructor

This constructor is used when we already have a **Runnable task** and we also want to **give a name** to the thread.

Example

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Task executed by: " +
Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        Runnable task = new MyTask();

        Thread t1 = new Thread(task, "TaskThread-1");
        Thread t2 = new Thread(task, "TaskThread-2");

        t1.start();
        t2.start();
    }
}
```

Output (order may vary):

```
Task executed by: TaskThread-1
Task executed by: TaskThread-2
```

Key Difference

- `Thread(String name)` → Only sets a name. You must extend `Thread` and override `run()`.
- `Thread(Runnable target, String name)` → Directly assigns a **task + name**, no need to extend `Thread`.

Thread Priorities in Java

- Each thread in Java has a **priority number** (an integer).
- Priorities help the **Thread Scheduler** decide which thread to execute first.
- By default, every thread gets **priority = 5 (NORM_PRIORITY)**.
- Thread priorities range from **1 (MIN_PRIORITY)** to **10 (MAX_PRIORITY)**.

Constants in Thread class

- `Thread.MIN_PRIORITY = 1`
- `Thread.NORM_PRIORITY = 5` (default)
- `Thread.MAX_PRIORITY = 10`

Methods for priority

- `setPriority(int p)` → sets the priority of a thread.
- `getPriority()` → returns the priority of a thread.

Example

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running thread: " + getName() +
                           " | Priority: " + getPriority());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.NORM_PRIORITY); // 5
        t3.setPriority(Thread.MAX_PRIORITY); // 10

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output (order may vary):

```
Running thread: Thread-0 | Priority: 1
Running thread: Thread-1 | Priority: 5
Running thread: Thread-2 | Priority: 10
```

Note: Priority is just a *hint* to the scheduler. It does not guarantee execution order.

Important Thread Methods

The `Thread` class provides many useful methods for managing threads:

Example with sleep, join, and currentThread

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + " is 
running...");
            try {
                Thread.sleep(2000); // pause for 2 sec
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Work done by " + 
Thread.currentThread().getName());
        });

        t.setName("Worker");
        t.start();

        t.join(); // wait until 't' finishes
        System.out.println("Main thread finished after Worker.");
    }
}
```

Output:

```
Worker is running...
Work done by Worker
Main thread finished after Worker.
```