Java Multithreading Basics

Here's a **beginner-friendly guide to Multithreading in Java** — from **scratch**, with **simple examples and clear explanations**.

🧠 What is Multithreading in Java?

Multithreading is a Java feature that allows the **concurrent execution** of two or more parts of a program for maximum utilization of CPU.

- A **thread** is the smallest unit of execution in a program.
- Java supports multithreading via the **`java.lang.Thread`** class and the **`Runnable`** interface.

✅ Why use Multithreading?

- To perform **multiple tasks simultaneously**.
- To improve **performance**.
- To make programs **responsive** (like GUI apps or servers).

🔧 Ways to Create Threads in Java

There are **two main ways** to create a thread in Java:

1. By Extending the `Thread` class

```
java

class MyThread extends Thread {
    public void run() {
        // Code that runs in this thread
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread: " + i);
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create thread object
        t1.start(); // Start thread
    }
}
```

🔍 Explanation:

- **`MyThread`** is a subclass of **`Thread`**.
- **`run()`** contains the code that runs in the new thread.
- **`start()`** begins execution of the thread (calls **`run()`** internally).

---

2. By Implementing the `Runnable` Interface

```
java

class MyRunnable implements Runnable {
    public void run() {
        // Code to execute in thread
        for (int i = 1; i <= 5; i++) {
            System.out.println("Runnable: " + i);
        }
    }

    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t1 = new Thread(r); // Wrap Runnable in Thread
        t1.start();
    }
}
```

🔍 Explanation:

- **`Runnable`** is a functional interface with **`run()`** method.
- You pass the **`Runnable`** object to the **`Thread`** constructor.
- Then call **`start()`** to run it.

🔄 Running Multiple Threads

```
java

class MyTask extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(getName() + " - Count: " + i);
        }
    }

    public static void main(String[] args) {
        MyTask t1 = new MyTask();
        MyTask t2 = new MyTask();

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

⏱️ Output (varies each run):

```
mathematica

Thread-0 - Count: 1
Thread-1 - Count: 1
Thread-0 - Count: 2
Thread-1 - Count: 2
...
```

> Threads run independently and might interleave differently each time.

🛌 Sleep Example (Pausing a Thread)

```
java

class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            try {
                Thread.sleep(1000); // 1 second pause
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
            System.out.println(getName() + ": " + i);
        }
    }

    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

🔒 Synchronization (Prevent Data Inconsistency)

If two threads try to access a shared resource, we use **`synchronized`** to avoid **race conditions**.

```
java

class Counter {
    int count = 0;

    public synchronized void increment() {
        count++;
    }
}

class MyThread extends Thread {
    Counter c;

    MyThread(Counter c) {
        this.c = c;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            c.increment();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        MyThread t1 = new MyThread(c);
        MyThread t2 = new MyThread(c);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + c.count);
    }
}
```

🔚 Summary Table

Topic	Explanation
`Thread` class	Extend and override **`run()`**
`Runnable`	Implement **`run()`** and pass to **`Thread`**
`start()`	Starts a new thread
`sleep(ms)`	Pauses a thread for given ms
`join()`	Waits for thread to finish
`synchronized`	Used to lock methods/blocks

Would you like **hands-on assignments**, or should I continue with **Thread Pools**, **Executors**, or **real-world examples** like downloaders or ATM simulations?

Here's a **beginner-friendly explanation of the Java Thread Lifecycle**, including **each state**, **transitions**, and a **code example**.

🔄 Java Thread Lifecycle

A thread in Java can be in one of the following **6 states** defined in the `Thread.State` enum:

📊 Thread Lifecycle Diagram

```sql
NEW → RUNNABLE → RUNNING → TERMINATED
         ↓           ↑
     BLOCKED / WAITING / TIMED_WAITING
```

🧩 Thread States Explained

State	Description
NEW	Thread is created but not started yet.
RUNNABLE	Thread is ready to run and waiting for CPU.
RUNNING	Thread is actively executing.
BLOCKED	Thread is waiting to acquire a lock.
WAITING	Thread is waiting indefinitely for another thread's action.
TIMED_WAITING	Thread is waiting for a specified period (e.g., sleep).
TERMINATED	Thread has completed or exited due to error.

✅ State Transitions with Code

```java
class ThreadLifecycle extends Thread {
    public void run() {
        System.out.println("State in RUNNING: " + Thread.currentThread().getState());

        try {
            Thread.sleep(2000); // TIMED_WAITING
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }

        System.out.println("Exiting Thread...");
    }

    public static void main(String[] args) {
        ThreadLifecycle t = new ThreadLifecycle();

        System.out.println("State after creation: " + t.getState()); // NEW

        t.start();
        System.out.println("State after start(): " + t.getState()); // RUNNABLE

        try {
            Thread.sleep(100); // Let thread go into RUNNING and then sleep
```

```
            System.out.println("State during sleep: " + t.getState()); // TIMED_WAITING
            t.join(); // wait for thread to finish
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("State after finish: " + t.getState()); // TERMINATED
    }
}
```

🧠 Thread State Mapping Summary

Method or Action	Transitions Thread To
`new Thread()`	NEW
`start()`	RUNNABLE
CPU assigns time slice	RUNNING
`sleep(ms)`	TIMED_WAITING
`wait()`	WAITING
`join()` (on a thread)	WAITING or TIMED_WAITING
`synchronized` lock wait	BLOCKED
`run()` ends or error	TERMINATED

🚦 Real-Life Analogy

Thread State	Real-World Analogy
NEW	A person is born but not working yet.
RUNNABLE	The person is waiting in job queue.
RUNNING	The person is working.
BLOCKED	The person wants a machine that's in use.
WAITING	The person is waiting for someone else.
TIMED_WAITING	The person is taking a short nap.
TERMINATED	The person's task is done.

Would you like a **visual simulation of thread states**, or a **mini-project like thread-based file downloader or ATM queue simulation** to apply these concepts practically?