MultiThreading Notes HariPrasanth H A

What is a Thread?

A **thread** in programming is the smallest unit of execution within a process. In Java (and most modern programming languages), a process (like your Java application) can have multiple threads running concurrently. Each thread represents a separate path of execution, meaning it can run code independently of other threads within the same application.

- In Java, every application starts with a main thread (the thread that runs your `main()` method).
- Threads can be created to perform tasks such as calculations, I/O operations, or background services simultaneously.

Why Do We Use Threads?

Threads are used to achieve **multitasking** and to make programs more efficient and responsive. Here are the main reasons:

1. **Concurrency & Parallelism**
   Threads allow multiple tasks to run at the same time. For example, downloading a file while updating the user interface.

2. **Better CPU Utilization**
   On multi-core systems, threads can run on different cores, utilizing the CPU more efficiently.

3. **Improved Responsiveness**
   Threads keep applications responsive. For example, in a GUI application, one thread can handle user input while another performs heavy computations, preventing the UI from freezing.

4. **Simplified Program Structure**
   Threads can simplify the design of certain programs. For example, a server can use one thread per client, making the code easier to manage.

5. **Asynchronous Operations**
   Long-running operations (like network or file I/O) can be performed in separate threads, so the main program doesn't have to wait for them to finish.

Example

Suppose you are building a chat application:

- One thread receives incoming messages.
- Another thread sends outgoing messages.
- A third thread updates the UI.

Without threads, all these tasks would have to be handled sequentially, leading to slow and unresponsive software.

Built-in Threads in Java

Java provides several built-in threads that are created and managed by the Java Virtual Machine (JVM) automatically. These threads are essential for the internal functioning of the JVM, and most of them are daemon threads. Here are some common built-in threads in Java:

1. Main Thread

- The entry point of every Java application is the main thread.
- It starts when the `main()` method is invoked.
- It is a user thread (i.e., not a daemon).

2. Garbage Collector Thread

- This thread is responsible for automatic memory management—reclaiming memory used by objects that are no longer referenced.
- It is a daemon thread.

3. Finalizer Thread

- Invokes the `finalize()` method on objects before garbage collection.
- Used to perform cleanup operations on objects.

4. Reference Handler Thread

- Handles reference objects like `SoftReference`, `WeakReference`, and `PhantomReference`.
- Ensures proper management and cleanup of memory referenced by these special reference types.

5. Signal Dispatcher Thread

- Handles native OS signals sent to the JVM (such as interrupts or termination requests).

6. Attach Listener Thread

- Used internally for dynamic attach operations, such as when monitoring or profiling tools connect to the JVM.

7. Other Background Threads

- Threads for Just-In-Time (JIT) compilation.
- Threads for class loading.
- Threads for thread pool management (like `ForkJoinPool` in parallel streams).

Note:

- Most JVM internal threads are daemon threads.
- These threads improve performance, memory management, and system responsiveness.

Application developers typically interact with the main thread and explicitly created threads, but understanding built-in threads helps with debugging and JVM tuning.

How to Create thread in java

1. By Extending the Thread Class

You create a subclass of `Thread` and override its `run()` method. Then, you create an instance of your subclass and call `start()` to begin execution in a new thread.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running!");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // Starts the new thread, which calls run()
    }
}
```

2. By Implementing the Runnable Interface

You create a class that implements the `Runnable` interface and define the `run()` method. Pass an instance of your class to a `Thread` object, then call `start()`.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running!");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start(); // Starts the new thread, which calls run()
    }
}
```

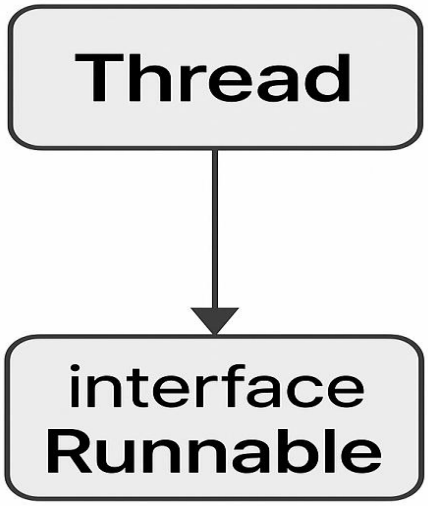
3. Using Lambda Expression (Java 8+)

You can also use a lambda expression for a more concise syntax

```
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(() -> System.out.println("Thread running!"));
        t.start();
    }
}
```

Note:

- Always use `start()` to launch a thread. Directly calling `run()` will not start a new thread; it just runs the code in the current thread.
- Implementing `Runnable` is preferred when your class needs to extend another class, as Java does not support multiple inheritance.



Methods of Thread class

Method	Description
`start()`	Starts the thread, which calls `run()` method internally.
`run()`	Entry point of the thread when started.
`sleep(long millis)`	Causes the thread to sleep for specified milliseconds.
`join()`	Waits for the thread to die.
`join(long millis)`	Waits for the thread to die for the specified time.
`setName(String name)`	Changes the name of the thread.
`getName()`	Returns the name of the thread.
`getId()`	Returns the thread's ID.
`setPriority(int priority)`	Sets thread priority (1 to 10).
`getPriority()`	Returns thread priority.
`isAlive()`	Returns true if thread is alive.
`isDaemon()`	Checks if thread is daemon thread.
`setDaemon(boolean on)`	Marks thread as daemon thread. Must be called before start().
`interrupt()`	Interrupts the thread (if it’s sleeping/waiting).
`isInterrupted()`	Checks if thread has been interrupted.
`currentThread()`	Static method: returns reference to currently executing thread.
`yield()`	Pauses the current thread to allow others to execute.
`activeCount()`	Returns the number of active threads.

Method	Description
`enumerate(Thread[] tarray)`	Copies every active thread into the array.

Thread class Constructor

Constructor Signature	Description
`Thread()`	Creates a new thread with a unique name and no target.
`Thread(Runnable target)`	Creates a thread that executes the specified `Runnable` task.
`Thread(Runnable target, String name)`	Creates a thread with a `Runnable` task and a custom name.
`Thread(String name)`	Creates a thread with a custom name, no `Runnable` task.

`start()` Method Example

```
class StartExample extends Thread {
    public void run() {
        System.out.println("Thread is running using start()");
    }

    public static void main(String[] args) {
        StartExample t = new StartExample();
        t.start(); // creates a new thread and calls run()
    }
}
```

`run()` Method Example (without start)

```
class RunExample extends Thread {
    public void run() {
```

```
        System.out.println("Thread is running directly using run()");
    }

    public static void main(String[] args) {
        RunExample t = new RunExample();
        t.run(); // behaves like a normal method call (no multithreading)
    }
}
```

`sleep()` Method Example

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

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

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

`join()` Method Example

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

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

        t1.setName("Thread-1");
        t2.setName("Thread-2");

        t1.start();
        try {
            t1.join(); // main thread waits for t1 to complete
        } catch (InterruptedException e) {
            System.out.println("Join interrupted");
        }
```

```
        t2.start(); // starts after t1 completes
    }
}
```

What is Multithreading in Java?

Multithreading **is a programming technique where** multiple threads run concurrently **within a single program (process). It allows you to** perform multiple tasks at the same time**, which can improve the performance and responsiveness of applications.**

- One thread handles video decoding.
- Another handles audio.
- Another one processes your touch input.

Example

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }

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

        t1.start();  // starts a new thread
        t2.start();
```

```
    }
}
```

What is a Race Condition?

A **race condition** happens when **two or more threads access shared data at the same time**, and the **final outcome depends on the order** in which the threads execute.

If **synchronization is not used**, you can get **wrong or unpredictable results**.

```
class Counter {
    int count = 0;

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

public class RaceConditionExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
```

```
            for (int i = 1; i <= 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();
        System.out.println("Final Count: " + counter.count);
    }
}
```

Fixing the Race Condition (Using `synchronized`)

```
class Counter {
    int count = 0;

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

What is `synchronized` in Java?

- The `synchronized` keyword ensures **only one thread can access a block or method at a time**.
- It works by acquiring a **lock on an object** (also called a monitor).
- **While one thread holds the lock**, all other threads trying to enter a `synchronized` block **must wait**.

How Object Lock Works

When a method is declared `synchronized`, the **lock is acquired on the current object** (`this`)

Thread-1 ──────┐

        │ acquires lock on `counter` object → enters synchronized method

Thread-2 ──────┘ waits until Thread-1 releases the lock

Fixed With `synchronized` – Using Object Lock

```
class Counter {
    int count = 0;

    // synchronized method acquires lock on the object (this)
    public synchronized void increment() {
        count++;
    }
}

public class SynchronizedExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 1000; i++) {
                counter.increment();
            }
        });
```

```
        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();
        System.out.println("Final Count: " + counter.count);
    }
}
```

How It Works

- When `t1` calls `counter.increment()`, it **acquires the lock** on the `counter` object.
- While `t1` holds the lock, `t2` **cannot enter** the `synchronized` method.
- Once `t1` finishes and **releases the lock**, `t2` acquires it and runs.
- This **avoids interference**, ensuring `count++` executes safely.

Object Lock

- Every Java object has an **intrinsic lock**.
- `synchronized` ensures **one thread at a time** accesses critical code.
- Helps prevent **race conditions** by **serializing access** to shared resources.

What is a Class Lock in Java?

A **class lock** is used when you want to synchronize **static methods or blocks** of code so that only **one thread** at a time can execute them, **across all instances of a class**.

```
class SharedCounter {
    static int count = 0;
```

```
    // class lock - applies to all threads across all instances
    public static synchronized void increment() {
        count++;
    }
}

public class ClassLockExample {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                SharedCounter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                SharedCounter.increment();
            }
        });

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

        System.out.println("Final Count: " + SharedCounter.count);
```

```
    }

}
```

When to Use Class Lock (`synchronized static`):

- When you are accessing or modifying **static variables** (shared across all objects)
- When the operation must be **mutually exclusive for all threads**, regardless of instance
- For **utility classes** or **singleton classes** with shared resources

Synchronized Block with Object Lock

A **synchronized block** with **object lock** allows only one thread to execute the **critical section** of code for a **specific object**.

```
synchronized (this) {



}
```

How it works:

- Lock is acquired on the **current object (`this`)**.
- If two threads operate on the **same object**, only one can execute the block at a

Synchronized Block with Class Lock

A **class lock** is used to synchronize code that affects **static variables or methods**, shared across all instances.

```
synchronized (ClassName.class) {

}
```

How it works:

- Lock is acquired on the **`Class` object** of the class (`ClassName.class`).
- Affects **all instances** of that class

```
class Logger {

   static void log(String message) {

      synchronized (Logger.class) {

         System.out.println(Thread.currentThread().getName() + " logging: " + message);
```

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}


public class ClassLockExample {
    public static void main(String[] args) {
        Runnable task = () -> Logger.log("System Event");

        Thread t1 = new Thread(task, "Thread-A");
        Thread t2 = new Thread(task, "Thread-B");

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

While **synchronization** in Java is essential to avoid problems like race conditions, **improper or excessive use** of synchronization can lead to **new problems** that affect performance and system stability.

Deadlock

Deadlock is a situation where **two or more threads are waiting for each other** to release resources, but **none of them ever release their lock**. So they all remain blocked forever.

```
class Resource {
    String name;
```

```
    Resource(String name) {
        this.name = name;
    }
}

public class DeadlockWithSynchronizedBlock {
    public static void main(String[] args) {
        Resource r1 = new Resource("Resource-1");
        Resource r2 = new Resource("Resource-2");

        // Thread 1: locks r1 then tries to lock r2
        Thread t1 = new Thread(() -> {
            synchronized (r1) {
                System.out.println("Thread-1 locked " + r1.name);

                try { Thread.sleep(100); } catch (InterruptedException e) {}

                synchronized (r2) {
                    System.out.println("Thread-1 locked " + r2.name);
                }
            }
        });

        // Thread 2: locks r2 then tries to lock r1 (opposite order)
        Thread t2 = new Thread(() -> {
```

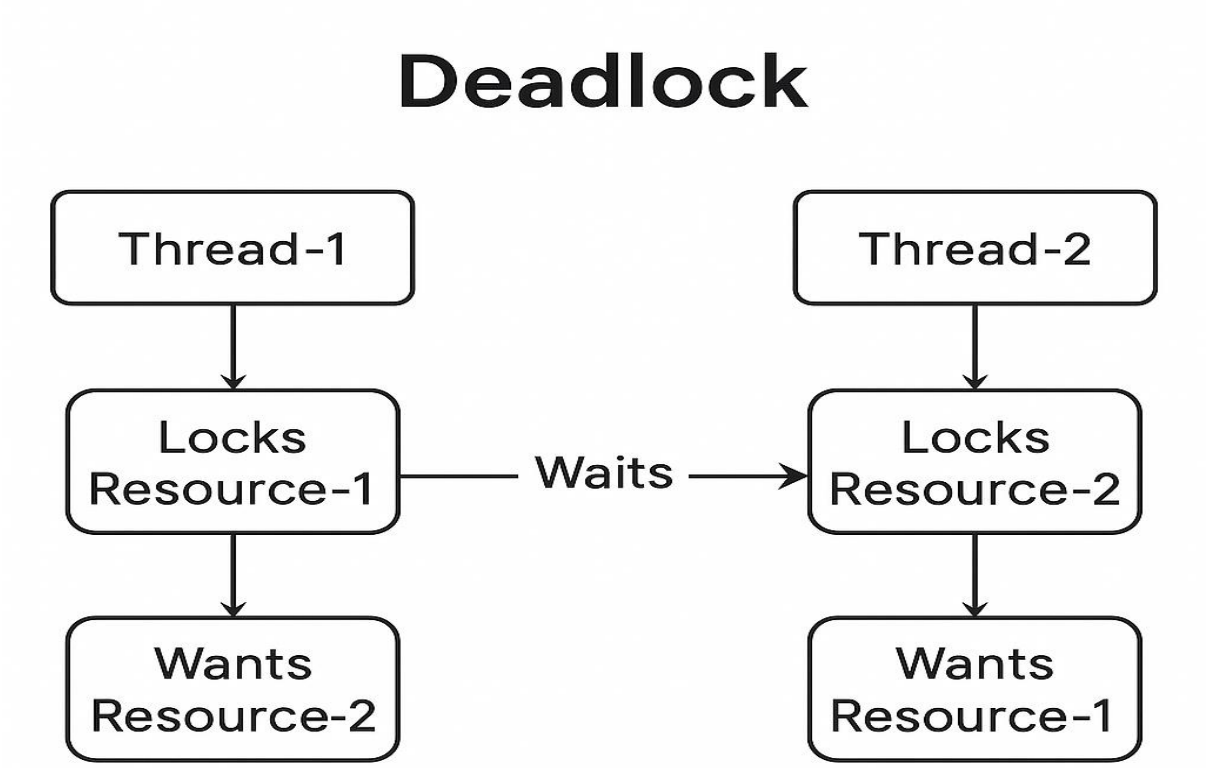
```
            synchronized (r2) {
                System.out.println("Thread-2 locked " + r2.name);

                try { Thread.sleep(100); } catch (InterruptedException e) {}

                synchronized (r1) {
                    System.out.println("Thread-2 locked " + r1.name);
                }
            }
        });

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

- **Resources:**
  Two objects, `r1` and `r2`, represent shared resources.
- **Two Threads:**
  - `Thread-1` locks `r1` first, then tries to lock `r2`.
  - `Thread-2` locks `r2` first, then tries to lock `r1`.
  - Both threads sleep for 100ms after acquiring their first lock to increase the chance of deadlock.



What is `volatile`?

- The `volatile` keyword in Java is used to mark a variable as being stored in main memory.
- Every read of a volatile variable will be read from the computer's main memory, and every write will be written to main memory.
- This ensures **visibility** of changes to variables across threads.

Why Use `volatile`?

- In a multithreaded environment, threads may cache variables locally, leading to stale or inconsistent values.
- Declaring a variable as `volatile` guarantees that any thread reading the variable will see the most recently written value by any other thread.

Example

```
class VolatileExample extends Thread {

    volatile boolean running = true; // shared variable
```

```
    public void run() {
        System.out.println("Thread started...");
        while (running) {
            // Loop until 'running' becomes false
        }
        System.out.println("Thread stopped...");
    }

    public static void main(String[] args) throws InterruptedException {
        VolatileExample obj = new VolatileExample();
        obj.start();

        Thread.sleep(1000); // Let the thread run for a bit
        obj.running = false; // Main thread changes the flag
        System.out.println("Main thread updated running to false");
    }
}
```

Explanation:

- `running` is a **shared variable** between the main thread and the child thread.
- `volatile` makes sure the **value is always read from main memory**, not a cached version in the thread's CPU.
- Without `volatile`, the child thread **might keep reading the old value `true`** even if the main thread sets it to `false`.

Example

```
class VolatileExample extends Thread {
```

```
    volatile boolean running = true; // shared variable

    public void run() {
        System.out.println("Thread started...");
        while (running) {
            // Keeps running until 'running' becomes false
        }
        System.out.println("Thread stopped...");
    }

    public static void main(String[] args) throws InterruptedException {
        VolatileExample obj = new VolatileExample();
        obj.start(); // Thread-1

        Thread.sleep(1000); // Let Thread-1 run for a second

        obj.running = false; // Main thread (Thread-2) updates the variable
        System.out.println("Main thread updated running to false");
    }
}
```

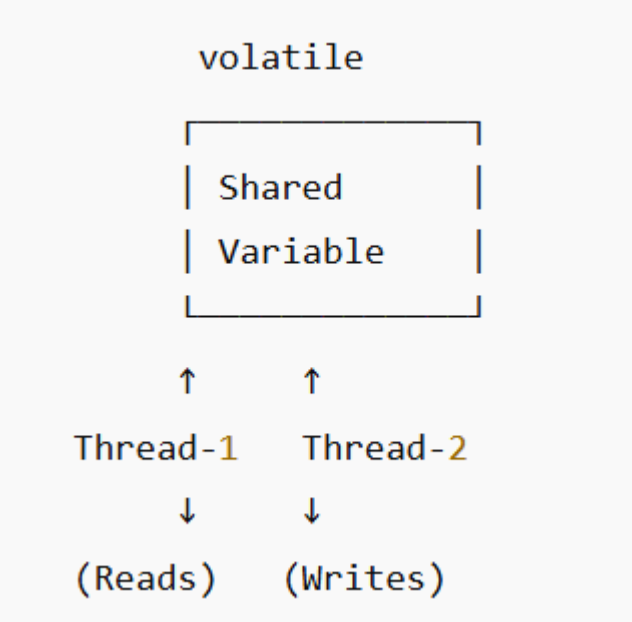


Image Element	Program Equivalent
Shared Variable	`volatile boolean running`
Thread-1	`obj.start()` → the thread that loops while `running` is true
Thread-2	`main()` method → the thread that updates `running = false`
Arrows	Both threads accessing the same variable. Volatile ensures updates are visible

Why `volatile` Is Needed

Without `volatile`, **Thread-1** might cache the `running` value as `true`, and never see the update made by **Thread-2**.
Using `volatile` ensures that:

- `Thread-1` always **reads** the current value from **main memory**
- `Thread-2` **writes** changes directly to **main memory**

Java Thread Life Cycle Stages

1. New (Created)

- Thread is **created** using `Thread` class but not yet started.

Thread t = new Thread();

2. Runnable

- Thread is **ready to run** and waiting for CPU to schedule it.

3. Running

- The thread is **executing** its `run()` method.
- The thread is picked by the **thread scheduler**

4. Waiting/Timed Waiting

- Thread is **waiting indefinitely** for another thread to perform an action.

Example: `obj.wait();` or `join();` without timeout.

- Thread is **waiting for a specified period**.

Example: `Thread.sleep(1000);` or `join(1000);`

5. Terminated (Dead)

- Thread has **finished execution** or has been **abruptly stopped** due to an exception.