# Introduction to Multithreaded Programming in Java

Affan Rauf

National University of Computer and Emerging Sciences

December 18, 2024

# What is a Thread?

- A thread is the smallest unit of execution in a program.
- Threads allow a program to perform multiple tasks simultaneously.
- Threads within a process share memory, open files, and other resources.

# Overview of Memory Segments in a Program

- When a C program is loaded into memory, it is divided into four main segments:
  - **Code Segment (Text Segment):** Contains the executable code.
  - **Data Segment:** Holds global and static variables initialized at compile time.
  - **Heap Segment:** Used for dynamically allocated memory during runtime.
  - **Stack Segment:** Stores local variables, function calls, and return addresses.

# Memory Sharing Among Threads

- Not all memory segments are shared between threads. Here's how they differ:

| Memory Segment | Shared by Threads? |
|---|---|
| Code Segment | Yes |
| Data Segment | Yes |
| Heap Segment | Yes |
| Stack Segment | No |

# Advantages Of This Memory Design?

- **Code Sharing:**
  - Reduces memory usage by allowing threads to share instructions.
- **Data/Heap Sharing:**
  - Enables threads to collaborate on shared data structures.
- **Private Stacks:**
  - Ensures independent function calls and local variable usage without conflicts.

# Extending Thread Class

- One way to create a thread is by extending the Thread class.

```java
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread running: " + i);
            try {
                Thread.sleep(500); // Pause for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e.getMessage());
            }
        }
    }
}
public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();

        for (int i = 0; i < 5; i++) {
            System.out.println("Main thread: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Main thread interrupted: " + e.getMessage())
                    ;
            }
        }
    }
}
```

# Implementing Runnable Interface

- Another way to create a thread is by implementing the `Runnable` interface.

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable running: " + i);
            try {
                Thread.sleep(500); // Pause for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Runnable interrupted: " + e.getMessage());
            }
        }
    }
}
public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
        for (int i = 0; i < 5; i++) {
            System.out.println("Main thread: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Main thread interrupted: " + e.getMessage())
                    ;
            }
        }
    }
}
```

# What is Thread Interleaving?

- Threads in a multithreaded program can execute in an arbitrary order.
- This behavior is known as **thread interleaving**.
- Interleavings are determined by the JVM scheduler and the underlying operating system.
- Without proper synchronization, interleaving can lead to **race conditions**.

# Example: Incrementing a Shared Counter

**Scenario:**

- Two threads increment a shared counter 1,000 times each.
- Shared variable: `int count = 0;`

**Code:**

```
class Counter {
    int count = 0;
    public void increment() {
        count++;
    }
}
```

**Expected Result:** `count = 2000` **Actual Result:** Can be less than 2000 due to interleaving!

# Breaking Down `count++`

**Steps in `count++`:**

1. Read the value of `count`.
2. Increment the value.
3. Write the updated value back to `count`.

**Java Bytecode for `count++`:**

```
load count    // Read count from memory
increment     // Increment value
store count   // Write back to memory
```

**Problem:** These steps are **not atomic**, meaning they can be interleaved.

# Possible Thread Interleavings

**Example Interleaving: Two Threads (T1 and T2) Incrementing** `count`

**Expected Execution:**

1. T1: Read `count = 0`.
2. T1: Increment to 1.
3. T1: Write 1.
4. T2: Read `count = 1`.
5. T2: Increment to 2.
6. T2: Write 2.

**Actual Interleaving:**

1. T1: Read `count = 0`.
2. T2: Read `count = 0`.
3. T1: Increment to 1.
4. T2: Increment to 1.
5. T1: Write 1.
6. T2: Write 1.

**Result:** Final `count` is 1, not 2!

# Why Does This Happen?

- The `count++` operation is not atomic (composed of multiple steps).
- Threads interleave arbitrarily between these steps.
- Updates to `count` are overwritten due to simultaneous reads and writes.
- This is called a **race condition**.

# Fixing the Problem: Synchronization

**Solution: Make `count++` Atomic**

- Use the `synchronized` keyword to ensure only one thread executes `increment()` at a time.

**Fixed Code:**

```java
class Counter {
    int count = 0;
    public synchronized void increment() {
        count++;
    }
}
```

**Result:** Correct final `count = 2000`.

# Need for Synchronization

- Multiple threads accessing shared resources can cause issues (e.g., race conditions).
- Java provides `synchronized` keyword for thread safety.

# What is `synchronized`?

- The `synchronized` keyword is used to ensure thread safety in Java.
- It prevents multiple threads from executing a critical section of code simultaneously.
- Achieves **mutual exclusion** by using a monitor (or intrinsic lock).

## Key Benefits

- Avoids **race conditions**.
- Protects **shared mutable resources**.
- Ensures **predictable behavior** in multithreaded programs.

# How It Works

1. A thread entering a `synchronized` block or method acquires the **monitor lock**.
2. Other threads attempting to enter the same synchronized block/method are **blocked**.
3. The thread releases the lock upon exiting the synchronized block or method.

### Monitor Lock

Each object in Java has a monitor lock that can be acquired or released.

# Types of Synchronization

1. **Synchronized Methods:** Entire methods are synchronized.
   - Instance methods: Lock on the instance.
   - Static methods: Lock on the class object.
2. **Synchronized Blocks:** Finer-grained control by synchronizing specific blocks of code.

# Synchronized Method

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
}
```

- Here, the `increment()` method is synchronized, ensuring mutual exclusion.
- Only one thread can execute `increment()` at a time.

# Synchronized Block

```java
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) {
            count++;
        }
    }
}
```

- Here, only the critical section (`count++`) is synchronized.
- Non-critical operations in the same method can execute concurrently.

# Synchronized Static Method

```java
class StaticCounter {
    private static int count = 0;
    public static synchronized void increment() {
        count++;
    }
}
```

- The `increment()` method locks the class object, ensuring mutual exclusion across all threads accessing static data.

# Notes and Best Practices

- Synchronize only the code that truly needs to be thread-safe.
- Be mindful of **deadlocks**, where two threads are waiting for each other to release locks.
- Avoid synchronizing large blocks of code as it can degrade performance.

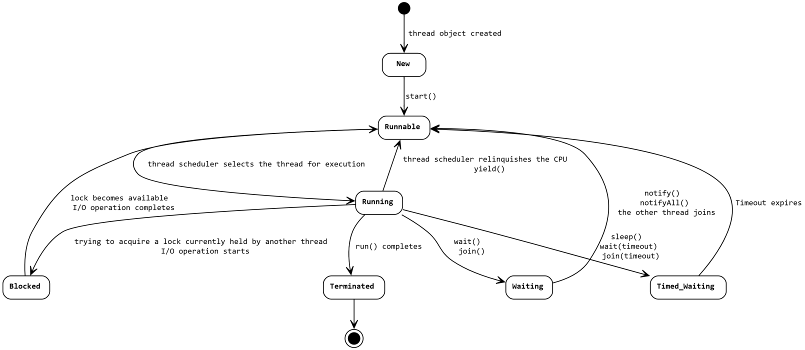
# Key Takeaways

- Use `synchronized` to protect shared resources in multithreaded environments.
- Understand the difference between synchronizing methods and blocks.
- Be cautious of potential performance issues and deadlocks.
- Always design for thread safety when shared mutable state is involved.

# Thread Lifecycle in Java

- A Java thread goes through multiple states during its lifecycle:
  - **NEW**: Thread is created but not started.
  - **RUNNABLE**: Ready to run but waiting for CPU time
  - **RUNNING**: Actively executing.
  - **BLOCKED**: Waiting for resources
  - **WAITING**: Waiting for a signal from another thread
  - **TIMED_WAITING**: Waiting for timer to expire
  - **TERMINATED**: Finished execution.

# Thread Lifecycle



# Methods Controlling Thread Lifecycle

- `start()` - Starts the thread. Moves thread from NEW to RUNNABLE state.
- `run()` - Entry point of the thread. Defines the thread's behavior.
- `sleep(ms)` - Puts the thread to sleep. Moves thread to TIMED_WAITING state.
- `join()` - Causes the current thread to wait for another thread to finish.
- `wait()`, `notify()`, `notifyAll()` - For inter-thread communication.
- `yield()` - Temporarily pauses the thread to allow others to execute.
- `interrupt()` - Interrupts a thread in certain states.

# start() and run()

`start()`:

- Transitions thread from NEW to RUNNABLE.
- Calls `run()` internally.

`run()`:

- Defines the thread's task.
- Invoked by the JVM when the thread starts.

**Example:**

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

public class Main {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.start(); // Calls run()
    }
}
```

# sleep()

- Pauses the thread for a specified duration.
- Transitions thread to TIMED_WAITING state.

**Example:**

```java
class SleepExample extends Thread {
    public void run() {
        try {
            System.out.println("Thread sleeping for 2 seconds");
            Thread.sleep(2000); // Sleep for 2 seconds
            System.out.println("Thread awake!");
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        new SleepExample().start();
    }
}
```

# join()

- Causes the current thread to wait for another thread to finish.
- Useful for coordinating threads.

**Example:**

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

public class Main {
    public static void main(String[] args) {
        Thread t = new JoinExample();
        t.start();
        try {
            t.join(); // Wait for t to finish
            System.out.println("Main thread continues");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# yield()

- Suggests that the thread scheduler give another thread an opportunity to execute.
- The scheduler may or may not honor this request.
- May move the thread back to RUNNABLE state.

**Example:**

```java
class YieldExample extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            Thread.yield(); // Hint scheduler to switch
        }
    }
}

public class Main {
    public static void main(String[] args) {
        new YieldExample().start();
        new YieldExample().start();
    }
}
```

# interrupt()

- Sends an interrupt signal to a thread.
- Does not forcibly stop the thread but sets an interrupt flag.
- Commonly used to stop threads in WAITING, TIMED_WAITING, or BLOCKED state. Throws an `InterruptedException`.

**Example:**

```java
class InterruptExample extends Thread {
    public void run() {
        try {
            Thread.sleep(5000); // Sleep for 5 seconds
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new InterruptExample();
        t.start();
        t.interrupt(); // Interrupt the thread
    }
}
```

# wait(), notify(), notifyAll()

- Used for inter-thread communication.
- `wait()` pauses a thread until another thread invokes `notify()` or `notifyAll()`.
- Must be used in a synchronized block or method.

**Example:**

```java
class Shared {
    synchronized void produce() throws InterruptedException {
        System.out.println("Producing...");
        wait(); // Wait for notification
        System.out.println("Resumed production");
    }
    synchronized void consume() {
        System.out.println("Consuming...");
        notify(); // Notify waiting thread
    }
}
public class Main {
    public static void main(String[] args) {
        Shared shared = new Shared();
        new Thread(() -> {
            try {
                shared.produce();
            } catch (InterruptedException e) { }
        }).start();
        new Thread(shared::consume).start();
    }
}
```

# wait(), notify(), and notifyAll() in Java

- These methods are part of the ‘Object‘ class.
- Used for **inter-thread communication** within a synchronized context.
- Allows one thread to pause execution and wait until another thread signals it to continue.

# Why wait(), notify(), and notifyAll() are in the Object Class

- **Object-Level Locking:**
  - Every Java object has an intrinsic lock (or monitor).
  - `wait()`, `notify()`, and `notifyAll()` operate on this lock.
- **Synchronization is Object-Oriented:**
  - Multiple threads may coordinate using a shared object.
  - The shared object is the entity that ensures proper thread communication.
- **Thread Communication:**
  - `wait()`, `notify()`, and `notifyAll()` are designed for inter-thread communication via shared resources.
  - These methods naturally belong to the object being shared, not the thread itself.

# wait()

- **Definition:** Makes the current thread wait until another thread invokes ‘notify()‘ or ‘notifyAll()‘ on the same object.
- Must be called inside a `synchronized` block or method.
- Releases the lock on the object temporarily, allowing other threads to acquire it.
- Syntax:

```
synchronized (object) {
    while (condition) {
        object.wait();
    }
}
```

# notify()

- **Definition:** Wakes up a single thread that is waiting on the object's monitor.
  - The thread to wake up is chosen by the JVM scheduler (not guaranteed to be FIFO).
- The thread awakened must re-acquire the lock before resuming execution.
- Syntax:

```java
synchronized (object) {
    object.notify();
}
```

- Useful when only one waiting thread needs to be notified.

# notifyAll()

- **Definition:** Wakes up all threads that are waiting on the object's monitor.
- Threads compete to acquire the lock and proceed based on thread scheduling.
- Syntax:

```java
synchronized (object) {
    object.notifyAll();
}
```

- Useful in scenarios where all waiting threads must proceed.

# Rules for Using wait(), notify(), notifyAll()

- Must be called from within a `synchronized` block or method.
- Always ensure a proper condition check (e.g., `while` loop) when using `wait()`.
- Avoid relying on thread priorities; behavior depends on the JVM scheduler.
- Use `notifyAll()` in complex scenarios to avoid deadlocks.
- Avoid "spurious wakeups" by rechecking conditions after `wait()`.

# Example: Producer-Consumer with wait() and notify()

```java
// Shared object
class Queue {
    private List<Integer> buffer = new ArrayList<>();
    private final int SIZE = 5;

    public synchronized void produce(int value) throws InterruptedException {
        while (buffer.size() == SIZE) wait(); // Try using if
        buffer.add(value);
        notifyAll(); // Notify consumers
    }

    public synchronized int consume() throws InterruptedException {
        while (buffer.isEmpty()) wait();
        int value = buffer.remove(0);
        notifyAll(); // Notify producers
        return value;
    }
}
```

# When to Use notify() vs notifyAll()

- Use `notify()` when:
  - Only one waiting thread should proceed.
  - There is a single condition to check.
- Use `notifyAll()` when:
  - Multiple threads may be waiting for different conditions.
  - You want to avoid potential deadlocks.
- Example:
  - A producer-consumer system with multiple producers and consumers benefits from `notifyAll()`.

# Exercise: Sum of Array

- Write a multithreaded program to calculate the sum of an array using:
  - Multiple threads
  - Proper synchronization