Multithreading

Josh Wilcox (jw14g24)

May 11, 2025

Table of Contents

Creating and Using Threads in Java

Why use threads

- Efficiency through concurrency
  - Facilitates using multiple core CPUs that are available
- Makes computer programs simpler to right
  - Separating complex tasks into parallel operations
  - Handling multiple user interactions simultaneously
- Prevents the inefficiency caused by sequential processes when they can be parrallelisable

---

Threading in the JVM

- In the JVM, each thread has its own **Private Memory Space** to store its own Java Stack and a program counter
- The metaspace (where method details are stored) and the heap are **shared between threads**

The Methods of Creating Threads

- There are two main ways to create threads in Java:
  - `class MyThread extends Thread`
  - `class MyRunner implements Runnable`

Extends Thread Class

Information on the thread Class

- The `Thread` class is defined in `java.lang`
- Any class that extends the `Thread` class can be created as its own thread
  - This is done by **overriding** the `run()` method which details how the main process the thread should achieve
  - **IMPORTANT** - `run()` does not actually start the class! To start it use `start()`
    - `run()` is automatically called by `start()`
    - `start()` only allows the thread to start if it has the right status

Example

```java
public class MyThread extends Thread {
    private String name;
    public MyThread(String name) {
        this.name = name;
    }

    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(name + " is executing, i = " + i);
        }
    }
}

public class ThreadDemo01 {
    public static void main (String[] args) {
        MyThread mt1 = new MyThread("Thread A");
        MyThread mt2 = new MyThread("Thread B");
        // Start each thread - They will now run in parrallel
        mt1.start();
        mt2.start();
    }
}
```

Implements Runnable Interface

Information on the Runnable Interface

- The `Runnable` interface only has one abstract method:
  - `public abstract void run();`
- This means that a new class that **implements** the `Runnable` class does **not** have it's own `start()` class manually
- So we still do use the `Thread` class in order to start the thread.

```
public class MyThread implements Runnable {
    @Override
    public void run(){
        // Run Logic Here
    }
    public static void main(String[] args){
        // Create the class with the run logic
        MyThread myThread = new MyThread();
        // Encapsulate the class in a Thread Object
        Thread threadRunner = new Thread(myThread);
        // Start the Thread
        threadRunner.start();
    }
}
```

Why use a Runnable Interface over a Thread Abstract Class

- A class can implement **Multiple** interfaces
- A class can only extend **One** abstract class
- Therefore using the `Runnable` interface allows our threads to both extend an abstract class and implement other interfaces
  - This provides more oppurtunity for polymorphism

Thread Methods

`Thread.sleep(long n)`

- Makes the current thread pause for n milliseconds
- During this time, the thread enters a ”sleeping” state
- Must be in a try-catch block as it throws InterruptedException

```java
try {
    Thread.sleep(1000); // Sleep for 1 second
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Thread Priorities

- Threads have priorities from 1 (lowest) to 10 (highest)
- Set using `setPriority(int priority)`
- Constants available: `MIN_PRIORITY`, `NORM_PRIORITY`, `MAX_PRIORITY`
- Higher priority threads are generally executed first

`Thread.yield()`

- Temporarily pauses current thread to let other threads execute
- Thread moves from ”running” state to ”ready” state
- No guarantee which thread will execute next
- Primarily used for thread scheduling optimization
  - Helps prevent any single thread from monopolizing CPU time
  - Particularly useful in busy-wait scenarios
  - Can improve overall system responsiveness
- Important considerations:
  - Behavior is highly platform-dependent
  - Only a scheduling hint - may be ignored by the JVM
  - Not recommended for synchronization purposes
- Example usage:

```java
if(threadNeedsToWait) {
    Thread.yield();  // Give other threads a chance
}
```

Interrupting Threads

- Use `interrupt()` to interrupt a thread
- Interrupted threads throw InterruptedException
- Check if interrupted using `isInterrupted()`
- Useful for stopping long-running or sleeping threads

States and Names of a Java Thread

What are the States of a Thread

- A thread in Java exist in any of six states
- Each state is stored as a key in an `Enum` object
  - `NEW` - Thread that has not yet started
  - `RUNNABLE` - Thread that is **executing** in the JVM
  - `BLOCKED` - Thread that is vlocked and waiting for a **monitor lock**
  - `WAITING` - Thread is waiting **indefinitely** for another thread to perform a particular action
  - `TIMED_WAITING` - Like `WAITING`, but includes a maximum waiting time
  - `TERMINATED` - The Thread has exited

Setters and Getters for Thread Names

- We can use constructors to define names for our threads
  - `publicThread (String name)`
  - `public Thread(Runnable target, String name)`
- Normally set threads names before execution, but we can change them afterwards using `setName(String name)`
- You can get the name using `getName()`

Daemon Threads

- Deamon threads are low-priority threads whose only role is to **provide services to user threads**
  - User threads are the normal threads we are used to
- Main Properties of Daemon Threads:
  - Won't prevent JVM from exiting while running
    * Normally, JVM only terminates when **all user threads** have finished executing
  - Threads created in a daemon threads will also be daemon
- Use `setDaemon(boolean on)` to true to make a thread a daemon

Useful Methods for Java Threads

`isAlive()`

- Determines whether a thread has been started or has been terminated
- Returns true if the thread has been started and has not yet died
- Returns false if either:
  - The thread hasn't been started yet
  - The thread has completed its execution
  - The thread has been terminated

`join()`

- The `join()` method is used to make one thread wait for another thread's completion
- When `threadA.join()` is called from `threadB`:
  - `threadB` will pause its execution
  - `threadB` will wait until `threadA` completes
  - This ensures sequential execution when needed
- Has three variants:
  - `join()` - waits indefinitely
  - `join(long millis)` - waits for specified milliseconds
  - `join(long millis, int nanos)` - adds nanosecond precision
- Common use cases:
  - Waiting for background tasks to complete
  - Ensuring proper order of operations
  - Coordinating dependent thread activities

Life Cycle of a thread

Thread States and Transitions

- New State
  - Thread is created but not yet started
  - Transitions to Runnable when `start()` is called
- Runnable State
  - Thread is executing or ready to execute
  - Can transition to:
    - Running - when selected by scheduler
    - Blocked/Waiting - when requesting resources
    - Terminated - when execution completes
- Blocked/Waiting States
  - Thread is temporarily inactive
  - Caused by:
    - sleep() calls
    - I/O operations
    - Lock acquisition
    - join() calls
- Terminated State
  - Thread has completed execution
  - Cannot be restarted

Synchronisation

The need for Syncronisation

- All threads in the JVM **share the heap**
  - Any fields in an object can be accessible to a number of threads at the same time
- After a thread has upated a field with a value, it's possible another thread comes along and updates the same field with a different value
- When one thread is trying to read the value from a field, another thread may be trying to change that value
- This can lead to a lot of funkiness to do with value change order
- Even simple operations like incrementing are not atomic:
  - A statement like `z = z + 1` compiles to multiple bytecode instructions
  - These include loading from heap, incrementing on stack, and storing back
  - Threads can be interrupted between any of these steps
  - This creates race conditions even for seemingly atomic operations

Incrementation Example

- See below, the final output for z should be 4

```java
public class UnsafeCounter {
    private int z = 2;  // Initial value is 2
    void incr() {
    int x = z;    // Thread 1: reads z=2
             // Thread 2: reads z=2 (before Thread 1 updates z)
    x = x + 1;    // Thread 1: x becomes 3
             // Thread 2: x becomes 3 (both threads working with original z value)
    z = x;        // Thread 1: sets z=3
             // Thread 2: sets z=3 (overwrites Thread 1's update)
    }                 // Final z=3, even though we wanted z=4
}
public class ThreadDemo {
    public static void main(String[] args) {
    UnsafeCounter counter = new UnsafeCounter();
    Thread t1 = new Thread(() -> counter.incr());
    Thread t2 = new Thread(() -> counter.incr());
    t1.start();
    t2.start();
    // After both threads finish, z might still be 3 instead of 4!
    }
}
```

Synchronising and Locks in Java

Locks

- If some thread is holding the lock of a syncronised block or method, no other thread can access that block!
- We can use locks to ensure **only one thread** can access code at any one point
  - This kind of lock is known as **mutex lock** which provides **mutual exclusion**
- Every object in Java has a mutex lock associated with it

Synchronised Blocks

- You can **lock** a defined section of code using a **synchronised block**:

```
synchronized(lockObject){
    // Code Here...
}
```

Synchronised Methods

- Methods can be marked as synchronised using the `synchronized` keyword
- This automatically synchronises the entire method body
- Example:

```
public synchronized void increment() {
    count++;  // This operation is now thread-safe
}
```

- Key points about synchronised methods:
  - Only one thread can execute a synchronized method at a time
  - The lock is on the entire object for instance methods
  - For static methods, the lock is on the Class object

The problem with Synchronisation

- Using synchronised blocks is very inefficient in many circumstances
- There are heavy runtime overheads in using locks and blocking threads
  - Try to synchronise the **minimum** number of instructions you can

'this' lock and multiple locks

- Using `synchronized(this)` or declaring a method as `synchronized` locks the entire object, preventing any other synchronized method/block on the same object from executing concurrently.
- For finer control, use dedicated lock objects (e.g., `private final Object lock = new Object();`) to synchronize only specific data or operations.
- Static synchronized methods lock on the `Class` object, so they do not interfere with instance-level locks.
- Multiple locks can be used within a class to protect independent resources, increasing concurrency and reducing contention.
- Always document which locks protect which data to avoid confusion and bugs.

Synchronising in Java - **Example of using Multiple Locks**

Synced Bank account using Sync Blocks

```java
public class BankAccount {
    private int savingsBalance = 0;
    private int checkingBalance = 0;

    // Separate lock objects for each account type
    private final Object savingsLock = new Object();
    private final Object checkingLock = new Object();

    // Deposit to savings account
    public void depositToSavings(int amount) {
        synchronized (savingsLock) {
            savingsBalance += amount;
        }
    }

    // Deposit to checking account
    public void depositToChecking(int amount) {
        synchronized (checkingLock) {
            checkingBalance += amount;
        }
    }

    // Get savings balance
    public int getSavingsBalance() {
        synchronized (savingsLock) {
            return savingsBalance;
        }
    }

    // Get checking balance
    public int getCheckingBalance() {
        synchronized (checkingLock) {
            return checkingBalance;
        }
    }
}
```

- This example allows threads to safely update or read the savings and checking balances independently.
- A thread updating the savings account does not block another thread updating the checking account, increasing concurrency.

Deadlocks

What is a Deadlock?

- A deadlock occurs when two or more threads are waiting for each other to release resources
- Each thread holds a resource that another thread needs
- None can proceed because they're all waiting
- Real-life example:
  - Two cars meet on a narrow bridge from opposite directions
  - Neither can proceed because both need the other to back up
  - Both are waiting for the other to move first

Code Example of Deadlock

```java
public class DeadlockExample {
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void method1() {
        synchronized(lock1) {
            System.out.println("Method 1 has lock1");
            try { Thread.sleep(100); } catch(InterruptedException e) {}
            synchronized(lock2) {
                System.out.println("Method 1 has lock2");
            }
        }
    }

    public void method2() {
        synchronized(lock2) {
            System.out.println("Method 2 has lock2");
            try { Thread.sleep(100); } catch(InterruptedException e) {}
            synchronized(lock1) {
                System.out.println("Method 2 has lock1");
            }
        }
    }
}
```

Preventing Deadlocks

- Always acquire locks in the same order
- Use timeouts when acquiring locks
- Avoid nested locks where possible
- Use `tryLock()` methods from `java.util.concurrent.locks`
- Implement deadlock detection mechanisms

Communicating Between threads

The need for communication handling

- Sometimes threads need to communicate or coordinate
  - For example, a consumer thread may need to **wait** for a producer to put data in the shared buffer
  - Java has methods that allow threads to pause execution until a certain criterion is met
    - Threads can also **notify** other threads when such condition has changed

A note on Object Monitors

- A monitor is a mechanism that ensures:
  - Only one thread can execute a synchronized method/block at a time
  - Other threads must wait for the active thread to finish
  - Provides a way for threads to coordinate through wait/notify
- When a thread enters a synchronized block:
  - It acquires the object's monitor
  - Other threads cannot acquire the same monitor
  - Monitor is released when thread exits the synchronized block

wait

- The `wait()` method causes **the thread it is called in** to release the lock it holds and **wait** until another thread calls `notify()` or `notifyAll()`
- Must be called inside a **synchronized block** or **method**
- Can also include a timeout by adding numbers in milliseconds to the constructor
- The `wait()` method is called on the same object as the lock in the synchronised block

notify

- The `notify()` method wakes up a **single** thread that has previously been **waiting** on the object's **monitor**
- The awakened thread only proceed if the synchronised block is **unlocked** - it will still wait until this is true before proceeding
- The `notify()` method is called on the same object as the lock in the synchronised block

notifyAll

- The `notifyAll()` method has the same core functionality as `notify()` but wakes up **all** threads waiting on the object's **monitor**
- Only one thread will acquire the lock and proceed; others will continue waiting for the lock.
- The `notifyAll()` method is called on the same object as the lock in the synchronised block