Multithreading

Josh Wilcox (jw14g24@soton.ac.uk)

March 17, 2025

Contents

1 Processes vs Threads

1.1 Process

- A program in execution
  - A series of activities that interact to produce a result
  - Includes the program code and its current activity
- An instance of a computer program that is executed by one or many threads
- Has its own memory space and system resources
- Can perform multiple tasks simultaneously through multithreading

1.2 Thread

- The smallest sequence of programmed instructions that can be managed independently by a scheduler
- In many cases, a thread is a **component of a process**
- Shares the process's resources, such as memory and open files
- Can run concurrently with other threads within the same process
- Useful for performing background tasks without interrupting the main program

1.3 Why use threading

- More efficient
- Allows for background processing
- Makes programming simpler

2 How Java handles Threading

2.1 Java support of Threading

- Java has built-in support for multithreaded applications:
  - The `Thread` class
  - Pre-defined packages like `java.util.concurrent`

2.2 Threading and the JVM

- Each thread has its own private memory to store the Java Stack and program counter
  - **Java Stack -** Stores frames, which hold local variables and partial results, and plays a part in method invocation and return.
  - **Program Counter (PC) -** A register that contains the address of the JVM instruction currently being executed. Each thread has its own PC register.
- Some components of Java are shared between all threads
  - **Metaspace -** The method area of Java, which stores class metadata, method data, and other information that is shared among threads. Unlike the older PermGen space, Metaspace can grow dynamically.
  - **Heap -** The runtime data area from which memory for all class instances and arrays is allocated. The heap is shared among all threads and is the primary area for garbage collection.

3 Java Thread Implementation

3.1 'Extends Thread' class

- For any class that `extends Thread`, can create threads.
- In the subclasses we create, we must overwrite the `run()` method - this is the key part of the thread
  - `run()` doesn't actually start the thread
  - Instead we should use `start()` to concurrently start the thread

3.1.1 Why we can't use run() to start a thread

- The `run()` method is just a normal method and calling it directly does not create a new thread of execution.
- When `run()` is called directly, it will execute in the current thread, blocking it until the method completes.
- The `start()` method, on the other hand, creates a new thread and calls the `run()` method within that new thread.
- Using `start()` allows the new thread to run concurrently with the existing thread, enabling true multithreading.

3.2 'Implements Runnable' Interface

- We can also make a class so it `implements Runnable`
- This interface only contains the `public void run();` abstract method

3.2.1 Problem Introduced by Runnable

- We are no longer extending `Thread`, therefore we need to find a new way of starting each thread
- This time, we create a `public Thread (Runnable target, String name)`

3.2.2 Why this is preferred

- Multiple Inheritance
  - Remember, a Java class can only **extend** from **one class** but can **implement many!**
  - This means, using the Runnable interface allows the extension of other classes other than just `Thread`

4 Thread Priorities

- Thread priority levels are used as **hints** to indicate which threads should be run first
- We use the `theThread.setPriority(int)` to set the priority of a thread
- You can use these constants as well - standard
  - `Thread.MIN_PRIORITY` - Value of 1
  - `Thread.NORM_PRIORITY` - Value of 5
  - `Thread.MAX_PRIORITY` - Value of 10

5 Yielding Threads

- Yielding is a way for a thread to voluntarily release the CPU, allowing other threads to execute.
- The `Thread.yield()` method is used to signal to the thread scheduler that the current thread is willing to yield its current use of the CPU.
- This does not guarantee that the current thread will stop executing immediately; it is merely a hint to the thread scheduler.
- Yielding is useful in scenarios where you want to give other threads a chance to execute, especially in a busy-wait loop.

5.1 Example of Yielding

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

    public static void main(String[] args) {
        YieldExample t1 = new YieldExample();
        YieldExample t2 = new YieldExample();
        t1.start();
        t2.start();
    }
}
```

- In this example, two threads are created and started.
- Each thread prints its name and a counter, then calls `Thread.yield()` to hint to the scheduler to allow other threads to execute.

6 Interrupting Threads

- Interrupting a thread is a way to signal that the thread should stop what it is doing and do something else.
- The `Thread.interrupt()` method is used to interrupt a thread.
- When a thread is interrupted, it sets the interrupt flag, which can be checked using `Thread.interrupted()` or `isInterrupted()`.
- Interrupting a thread does not stop it immediately; it is up to the thread to handle the interruption appropriately.

6.1 Handling Interruption

```java
public class InterruptExample extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + " - " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " was interrupted.");
        }
    }

    public static void main(String[] args) {
        InterruptExample t1 = new InterruptExample();
        t1.start();
        try {
            Thread.sleep(3000);
            t1.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- In this example, a thread is created and started.
- The thread prints its name and a counter, then sleeps for 1 second.
- After 3 seconds, the main thread interrupts the created thread.
- The created thread catches the `InterruptedException` and prints a message indicating it was interrupted.