Race Conditions

Aaron Smith

COMP 301

May 5, 2021

Review

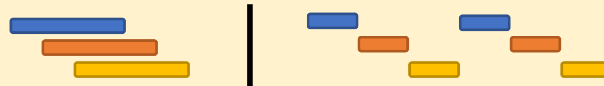
Models of Computing

Sequential Computing

Only one task can be executed at a time; the current task must complete before the next one can begin

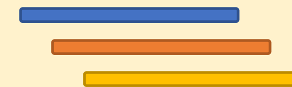
Concurrent Computing

Multiple tasks can be executed during overlapping time periods, either ***in parallel*** or by ***context switching***



Parallel Computing

Multiple tasks can simultaneously be executed on separate processing elements



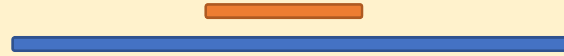
Models of Programming

Synchronous Programming

The program waits for a task to finish completely before continuing on

Asynchronous Programming

The program continues on without waiting for a method to finish



Poll Everywhere (1)

To answer, go to
https://pollev.com/onsmith

```java
public class Example {
  public static void main(String[] args) {
    System.out.print("a");

    Runnable task = () -> {
      System.out.print("b");
    };

    System.out.print("c");
    task.run();
    System.out.print("d");
  }
}
```

The task is run **synchronously**

Prompt: What will this program print?

Poll Everywhere (1)

To answer, go to
https://pollev.com/onsmith

```
public class Example {
  public static void main(String[] args) {
    System.out.print("a");

    Runnable task = () -> {
      System.out.print("b");
    };

    System.out.print("c");
    task.run();
    System.out.print("d");
  }
}
```

Does the output change if "c" is moved above the task definition?

Prompt: What will this program print?

Poll Everywhere (1)

To answer, go to https://pollev.com/onsmith

```
public class Example {
  public static void main(String[] args) {
    System.out.print("a");
    System.out.print("c");

    Runnable task = () -> {
      System.out.print("b");
    };

    task.run();
    System.out.print("d");
  }
}
```

Does the output change if "c" is moved above the task definition?

Prompt: What will this program print?

Poll Everywhere (2)

To answer, go to https://pollev.com/onsmith

```
public class Example {
  public static void main(String[] args) {
    System.out.print("a");

    Runnable task = () -> {
      System.out.print("b");
    };

    System.out.print("c");
    Thread thread = new Thread(task);
    thread.start();
    System.out.print("d");
  }
}
```

The task is run **asynchronously**

Prompt: What will this program print?

Waiting for a `Thread` to finish

The `join()` method

Waiting for a **Thread** to finish

```
public static void main(String[] args) {
  Runnable task =
      () -> {
        for (int i = 0; i < 10; i++) {
          System.out.print(i);
          System.out.print(" ");
        }
      };

  Thread thread1 = new Thread(task);
  thread1.start();

  Thread thread2 = new Thread(task);
  thread2.start();

  System.out.println("Finished!");
}
```

This task prints the numbers 1 – 10 with a space in between

```
0 1 2 3 4 5 6 7 8 9
```

Perform the task twice asynchronously

Print “Finished!”

Waiting for a **Thread** to finish

```
public static void main(String[] args) {
  Runnable task =
      () -> {
        for (int i = 0; i < 10; i++) {
          System.out.print(i);
          System.out.print(" ");
        }
      };

  Thread thread1 = new Thread(task);
  thread1.start();

  Thread thread2 = new Thread(task);
  thread2.start();

  System.out.println("Finished!");
}
```

```
Sample output:

Finished!
00  1 1 2 3 2 4 3 5 4 5 6 7 8 6 9 7 8 9
```

Will this line run after the threads complete?

No! It's asynchronous

Waiting for a **Thread** to finish

```java
public static void main(String[] args) {
    Runnable task =
        () -> {
          for (int i = 0; i < 10; i++) {
            System.out.print(i);
            System.out.print(" ");
          }
        };

    Thread thread1 = new Thread(task);
    thread1.start();

    Thread thread2 = new Thread(task);
    thread2.start();

    System.out.println("Finished!");
}
```

```
Sample output:

Finished!
00  1 1 2 3 2 4 3 5 4 5 6 7 8 6 9 7 8 9
```

Waiting for a Thread to finish

```
public static void main(String[] args) {
    Runnable task =
        () -> {
          for (int i = 0; i < 10; i++) {
            System.out.print(i);
            System.out.print(" ");
          }
        };

    Thread thread1 = new Thread(task);
    thread1.start();

    Thread thread2 = new Thread(task);
    thread2.start();

    thread1.join();
    thread2.join();

    System.out.println("Finished!");
}
```

```
Sample output:

0 1 2 3 0 1 2 3 4 5 64  7 8 9 5 6 7 8 9 Finished!
```

Pause the `main()` thread until thread1 and thread2 are finished

Race conditions

When the **timing of execution** affects the result

Race conditions

Race condition – A segment of concurrent code where the ***timing of execution*** affects **the result**

Race conditions occur when two or more threads **share memory**

- Multiple threads reading from or writing to ***the same object***

What can go wrong?

- Two threads write to a field at the same time
  - Who wins? It's a race!
- One thread reads a field, but then another thread overwrites it
  - Stale values

Example: A shared **Counter** class

```
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public void addOne() {
    value = getValue() + 1;
  }

  public void subtractOne() {
    value = getValue() - 1;
  }

  public int getValue() {
    return value;
  }
}
```

Encapsulates an integer

The integer starts at 0

Can “increment” the integer

Can “decrement” the integer

Can get the current integer value

Example: Using the **Counter** class

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

  Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.addOne();
    }
  });

  Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.subtractOne();
    }
  });

  thread1.start();
  thread2.start();

  thread1.join();
  thread2.join();

  System.out.println(counter.getValue());
}
```

One thread **increments** the counter 100,000 times

The other thread **decrements** the counter 100,000 times

Afterwards, print the value

Example: Using the **Counter** class

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

  Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.addOne();
    }
  });

  Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.subtractOne();
    }
  });

  thread1.start();
  thread2.start();

  thread1.join();
  thread2.join();

  System.out.println(counter.getValue());
}
```

What would you expect the answer to be?

```
Sample output:

-2782
```

A closer look at **Counter**

```
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public void addOne() {
    value = getValue() + 1;
  }

  public void subtractOne() {
    value = getValue() - 1;
  }

  public int getValue() {
    return value;
  }
}
```

This 1 line of code is actually 3 operations!

1. Get the value
2. Add 1 to that number
3. Set the value

A closer look at `Counter`

Imagine `value = 0`, when both `addOne()` and `subtractOne()` are called **concurrently**

```java
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public void addOne() {
    value = getValue() + 1;
  }

  public void subtractOne() {
    value = getValue() - 1;
  }

  public int getValue() {
    return value;
  }
}
```

Thread 1, `addOne()`

1. Get the value (0)
2. Add 1 to that number
3. Set the value (1)

Thread 2, `subtractOne()`

1. Get the value (0)
2. Subtract 1 from that number
3. Set the value (-1)

This is a **race condition**! Depending on which line executes first, `value` might be `1` or `-1`!

Example: Using the **Counter** class

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

  Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.addOne();
    }
  });

  Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.subtractOne();
    }
  });

  thread1.start();
  thread2.start();

  thread1.join();
  thread2.join();

  System.out.println(counter.getValue());
}
```

If both methods execute at the same time, there's a chance that only one will take effect

But which method takes effect is **completely unpredictable!**

Synchronized methods

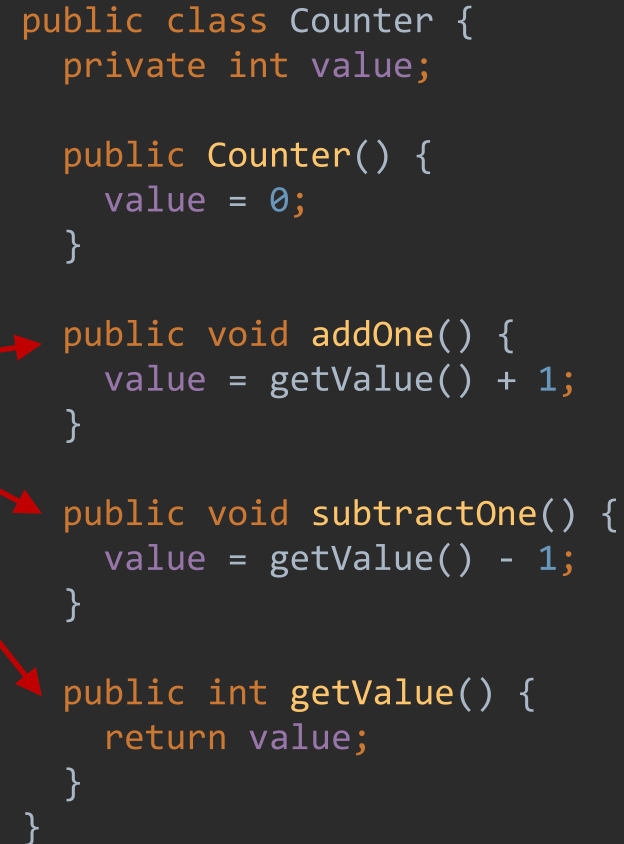
Enforcing **mutual exclusion** in Java

Mutual exclusion

These methods simply can't be executed at the same time

- Concurrency of these methods results in a **race condition**
- In general, this occurs *any time* you **read** or **write** to data that in memory shared between threads

We say that these methods must be made **mutually exclusive**



```
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public void addOne() {
    value = getValue() + 1;
  }

  public void subtractOne() {
    value = getValue() - 1;
  }

  public int getValue() {
    return value;
  }
}
```

Synchronization

Solution: add the `synchronized` keyword to all methods that must be made **mutually exclusive**

- Usually, every method that reads or writes field values should be synchronized

What does this do?

```
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public synchronized void addOne() {
    value = getValue() + 1;
  }

  public synchronized void subtractOne() {
    value = getValue() - 1;
  }

  public synchronized int getValue() {
    return value;
  }
}
```

Java ensures that no two **`synchronized`** methods of a given instance will ever be executed at the same time by different threads

Using synchronized methods

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

  Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.addOne();
    }
  });

  Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
      counter.subtractOne();
    }
  });

  thread1.start();
  thread2.start();

  thread1.join();
  thread2.join();

  System.out.println(counter.getValue());
}
```

With **synchronization**, the output is predictable

```
Sample output:

0
```

Synchronization is achieved using locks

```
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public synchronized void addOne() {
    value = getValue() + 1;
  }

  public synchronized void subtractOne() {
    value = getValue() - 1;
  }

  public synchronized int getValue() {
    return value;
  }
}
```

How does Java enforce **mutual exclusion** of **synchronized methods**?

Internally, the JVM creates a **lock** for every instance of the class that is synchronized (e.g. Counter)

From Oracle's documentation: A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first.

Read about the Lock interface: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html

Manually using Java's Lock interface

```
public class Counter {
  private int value;

  public Counter() {
    value = 0;
  }

  public synchronized void addOne() {
    value = getValue() + 1;
  }

  public synchronized void subtractOne() {
    value = getValue() - 1;
  }

  public synchronized int getValue() {
    return value;
  }
}
```

Illustration of how synchronized is implemented

Every Counter instance has its own lock

ReentrantLock is the lock implementation used for synchronized

Acquire the lock, waiting if necessary until it is available

Critical section occurs once the lock is acquired

Release lock after critical section finishes

```
public class Counter {
  private int value;
  private Lock lock;

  public Counter() {
    value = 0;
    lock = new ReentrantLock();
  }

  public void addOne() {
    lock.lock();
    value = getValue() + 1;
    lock.unlock();
  }

  public void subtractOne() {
    lock.lock();
    value = getValue() - 1;
    lock.unlock();
  }

  public int getValue() {
    lock.lock();
    int v = value;
    lock.unlock();
    return v;
  }
}
```

Best practice: `unlock()` in `finally`

```
public void subtractOne() {
  lock.lock();
  value = getValue() - 1;
  lock.unlock();
}
```

```
public int getValue() {
  lock.lock();
  int v = value;
  lock.unlock();
  return v;
}
```

Need to ensure the lock is always released! What if an exception is thrown and the method never finishes?

If the thread never releases the lock, other threads will never be able to acquire it! This is called **deadlock**

Best Practice

```
public void subtractOne() {
  lock.lock();
  try {
    value = getValue() - 1;
  } finally {
    lock.unlock();
  }
}
```

```
public int getValue() {
  lock.lock();
  try {
    return value;
  } finally {
    lock.unlock();
  }
}
```

Bottom line

Parallelization can speed up your job by doing multiple tasks at once

- Must have tasks that inherently can be parallelized

Concurrent read/write to shared memory causes **race conditions**

- Program behavior is unpredictable because it depends on timing

Methods that read or write shared state must be **synchronized**

- Forces the methods to be executed with **mutual exclusion**
- This behavior is enforced with a **lock**

Deadlock occurs when a thread can't acquire the lock it needs to finish

Learn more at https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html