Shared Mutable State

SWEN-220
The Ultimate Culprit - Shared, Mutable State

- Most of your development has been in imperative languages.
- The fundamental operation is assignment to change state.
  - Assignable variables are mutable.
  - May be exposed as public (bad karma).
  - May be exposed via interface methods (medium warm karma).
  - Things get tricky very fast when > 1 thread can invoke a mutating function.
Most of your development has been in imperative languages.

The fundamental operation is assignment to change state.

- Assignable variables are mutable.
- May be exposed as public (bad karma).
- May be exposed via interface methods (medium warm karma).
- Things get tricky very fast when > 1 thread can invoke a mutating function.

```java
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

If we call `increment()` 10,000 times and then call `getCount()`, what value is returned?
The Ultimate Culprit - Shared, Mutable State

• Most of your development has been in imperative languages.
• The fundamental operation is assignment to change state.
  – Assignable variables are mutable.
  – May be exposed as public (bad karma).
  – May be exposed via interface methods (medium warm karma).
  – Things get tricky very fast when > 1 thread can invoke a mutating function.

```java
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

If we call `increment()` 10,000 times and then call `getCount()`, what value is returned?

If the Counter object is accessed sequentially by one thread – then 10,000.

If multiple threads are accessing it concurrently – then ???
The Perils of Interleaving

What “we humans” see as instructions:

```java
public class Counter {
    private int count = 0 ;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count ;
    }
}
```

What the Java VM is actually executing:

```java
public class Counter {
    public Counter();
    Code:
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init">:()V
    4: aload_0
    5: iconst_0
    6: putfield  #2                  // Field count:I
    9: return

    public void increment();
    Code:
    0: aload_0
    1: dup
    2: getfield  #2                  // Field count:I
    5: iconst_1
    6: iadd
    7: putfield  #2                  // Field count:I
    10: return

    public int getCount();
    Code:
    0: aload_0
    1: getfield  #2                  // Field count:I
    4: ireturn
}
```

Viewing Java bytecodes from a class file: `javap -c Counter > Counter.bc`
Mind the Critical Section

What “we humans” see as instructions:

```java
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

What the Java VM is actually executing:

```java
public class Counter {
    public Counter();
    Code:
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init>"():V
    4: aload_0
    5: iconst_0
    6: putfield    #2                  // Field count:I
    9: return

    public void increment();
    Code:
    0: aload_0
    1: dup
    2: getfield    #2                  // Field count:I
    5: iconst_1
    6: iadd
    7: putfield    #2                  // Field count:I
    10: return

    public int getCount();
    Code:
    0: aload_0
    1: getfield    #2                  // Field count:I
    4: ireturn
}
```

Source Code view of the Critical Section
Mind the Critical Section

What “we humans” see as instructions:

```java
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

Source Code view of the Critical Section

VM view of the Critical Section

What the Java VM is actually executing:

```java
public class Counter {
    public Counter() {
        Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: aload_0
        5: iconst_0
        6: putfield #2                  // Field count:I
        9: return
    }

    public void increment() {
        Code:
        0: aload_0
        1: dup
        2: getfield #2                  // Field count:I
        5: iconst_1
        6: iadd
        7: putfield #2                  // Field count:I
        10: return
    }

    public int getCount() {
        Code:
        0: aload_0
        1: getfield #2                  // Field count:I
        4: ireturn
    }
}
```
Mind the Critical Section(s)

What “we humans” see as instructions:

```java
class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

What the Java VM is actually executing:

```java
class Counter {
    public Counter();
    Code:
    0: aload_0
    1: invokespecial #1                  // Method java/lang/Object."<init>":()V
    4: aload_0
    5: iconst_0
    6: putfield #2                  // Field count:I
    9: return

    public void increment();
    Code:
    0: aload_0
    1: dup
    2: getfield #2                  // Field count:I
    5: iconst_1
    6: iadd
    7: putfield #2                  // Field count:I
    10: return

    public int getCount();
    Code:
    0: aload_0
    1: getfield #2                  // Field count:I
    4: ireturn
}
```
The Ultimate Culprit - Shared, Mutable State

• Most of your development has been in imperative languages.
• The fundamental operation is assignment to change state.
  – Assignable variables are mutable.
  – May be exposed as public (bad karma).
  – May be exposed via interface methods (medium warm karma).
  – Things get tricky very fast when > 1 thread can invoke a mutating function.

• Three basic approaches:
  – Provide *mechanisms* to support controlled access to shared, mutable state. (Locks – semaphores)
  – Hide shared state behind sequential access. (Messaging – channels)
  – Make things immutable (functional programming)

(We’ll look at the first two approaches in this course)
The Ultimate Culprit - Shared, Mutable State

• Provide *mechanisms* to support controlled access to shared, mutable state.
• Java uses the keyword *synchronized* to provide exclusive access to a method or block of code.
• First thread to enter a synchronized section of code obtains that object’s lock, and releases it when exiting.
• Other threads will become blocked until the lock is released.

```java
public class Counter {
    private int count = 0;  // Shared Mutable Resource

    public synchronized void increment() {
        count++;  // Critical Section protected
        // by the object’s lock
    }

    public synchronized int getCount() {
        return count;
    }
}
```