Java Concurrency

Thread-safety for intermediate developers

What’s in this talk?

> Some definitions and concepts

> Benefits and risks of concurrency

> Deep dive into race conditions

> One weird trick for threadsafe code!!

> Synchronization

> Alternatives to synchronization

A quick intro

Parallel and Concurrent

-

Threads

- A way for a single program to split execute pieces of code simultaneously
- Relatively lightweight
- Each has its own name, code stack, state, etc
- Can execute truly simultaneously on separate processors, or be interleaved with other threads on one processor
- They are managed by the same JVM instance and share the same resources

Synchronous and Asynchronous

- Synchronous means there's co-ordination between things
- Async means there's no coordination, each thread does its own thing at its own pace
- Async is the *default* for multi-threaded code in java. If you want things to be synchronized, you need to be explicit about it
- Async is *faster* because more things are happening all at once
- It's also less *safe* because there's no guarantee what order the threaded things happen in

Atomic

Atomic

- Atomic mean it can’t be broken down further
- An action is atomic if it can’t be interrupted or interleaved with other actions
- Atomic
  - Reading an int, boolean, etc
  - Writing an int, boolean, etc.
  - Compare-and-swap operation
- Non-atomic:
  - If (mylist.contains(user.getName()) i++
  - Reading a long (sometimes)
  - i++

Benefits

1. **Throughput**: utilize hardware more effectively to process more data in less time

2. **Responsiveness**: process user inputs as soon as you get them, instead of waiting until you’ve finished the previous task

3. **Simplicity**: having separate tasks in separate threads keeps things nice and encapsulated

Problems

1. **Safety**: nothing bad ever happens, e.g. no race conditions

2. **Liveness**: something good eventually happens, e.g. deadlocks

3. **Performance**: it doesn't take forever for something good to happen

What does multithreaded code look like?

```
new Thread.start(myRunnable)  //runnable can be a lambda

ExecutorService executor = new ThreadPoolExecutor(10)
executor.execute(myRunnable) //or .sumbit(myRunnable)

stream().parallel()
```

Also lifecycle management frameworks, like spring, can spin up extra threads, e.g. @PostConstructor, @Scheduled, REST api requestst

Race conditions

Race conditions

- **Race Conditions** are when the order of threads executing affects the outcome
- E.g. 2 threads modifying the same value at the same time
- A very easy mistake to make
- Hard to spot, debug and reproduce

Race conditions: a simple example

Thread 1

```
map.put("john", "Doe")
```

Thread 2

```
map.put("john", "Smith")
```

Race conditions: another example

Thread 1

```
println("hello")
```

Thread 2

```
println("hello")
```

Race condition 1: read-modify-write

Both threads are running one line of code: i++

Thread 1	Thread 2
`Get the value of i`	
	`Get the value of i`
`Calculate i+1`	
	`Calculate i+1`
`Write that value to i`	
	`Write that value to i`

Race condition 2: check-then-act

Thread 1	Thread 2	
`if (list.length() < 10)`		`size = 9`
	`if (list.length() < 10)`	`size = 9`
`list.add(x)`		`size = 10`
	`list.add(x)`	`size = 11 :(`

The simple solution

The simple solution

Race conditions affect **shared**, **mutable** **state**

Your data will always be thread safe if it is either:

- **not shared across multiple threads**
- **Immutable**
- **Make things stateless**

1) Don’t share data

If you can keep data encapsulated, and control who has access to it, you can make sure it’s only accessible from one thread

Some strategies:

- **Ad-hoc Stack confinement:** keep in a single code stack, don’t write it out to a class field
- **Make local copies** rather than using the shared object
- **ThreadLocal** is an object wrapper that stores one instance per thread
- **Pass-the-baton**: only make the object available to another thread when the first is done with it, e.g. blockingQueue

2) make it immutable

- A variable is immutable if there's no way to modify its state after it's created
- Final stops the variable being reassigned (important!), but doesn't stop it being changed
- Primitives and Strings are always immutable
- Other objects are immutable if:
    - All fields are final (or effectively final)
    - The class provides no way to mutate its fields and doesn't expose them (e.g. in getters), or its fields are immutable classes

3) go stateless

- An object is stateless if it doesn’t have any member fields
- It can process data in methods, but it doesn’t store anything
- Not always possible, but a good, easy approach is it is

The hard solution

Synchronization

- Synchronize between threads let us control the order of execution
- (In java at least) it's done by "acquiring a lock" on an object.
  - When you need to read or write to shared mutable data, you first lock it
  - Each lock can only be held once, so if you need a lock but another thread has it, you wait until it becomes available
  - When you're done, you unlock it so other threads can use it
- There are an infinite number of possible locks, and which one you use matters
  - If you access the same data from multiple methods, you need to use the same lock object
  - If you have two areas that require synchronization, but they don't share data, use different objects

Synchronized

- There are several ways to do locking in Java, several different lock classes
- The simplest and most common is the ‘synchronized’ keyword
- Synchronized blocks have a “Monitor”
    -
    - This can be set implicitly or explicitly
    - Anything object can be a monitor

Synchronized

Synchronized can wrap a method

```
Public void synchronized addToQueue(Foo item) {
     if (list.length() < 10)
          list.add(item)
}
```

The monitor in this case is `this` (or the the Class for static methods)

Synchronized

Or Synchronized can wrap a block

```
Synchronized (this) {
    if (list.length() < 10)
        list.add(item)
}
```

```
You can use any object here to synchronize on, not just this

But using `this` is common
```

Race condition 1: check-then-act

Lock | Thread 1 | Thread 2

```
1    Public void synchronized () {
1    if (list.length() < 10)Public void synchronized () {
1    list.add(x)
1    }


2                                   if (list.length() < 10)
2                                   list.add(x)
```

- No matter which thread reaches it first, the other has to wait until the first thread is fully done
- The check-and-add action is now atomic. The threads can't be interleaved

Locking pitfalls:

Performance, liveness and safety

Performance

- Locks work by making some threads wait
- The more locks you have, the slower your code is
- Totally defeats the point of making your code concurrent in the first place

Liveness

```java
public class Deadlock {

    private final Object lockA = new Object();

    private final Object lockB = new Object();

    public void acquireAThenB() {
        synchronized (lockA) {
            synchronized (lockB) {
                // Do something...
            }
        }
    }

    public void acquireBThenA() {
        synchronized (lockB) {
            synchronized (lockA) {
                // Do something...
            }
        }
    }

}
```

Safety

- Locks are easy to misuse, particular if you don’t totally understand what you’re doing

Atomicity: multivariable states

- Sometimes multiple variables combine to form a single state, and when one changes the other needs to change as well
  - E.g.
- Modifying these needs to be wrapped in a single synchronized block
- A good pattern is to encapsulate multi-variable states in their own class, and synchronize the methods on that class

Atomicity: compound actions

- Some actions take multiple steps, and you need to make sure those are atomic, too
- E.g. check-then-act

Alternative locking strategies

Alternatives to synchronized

- Synchronized is good: it’s clear and widely used
- Sometimes you need something more powerful or flexible

ReentrantLock

- ReentrantLock is very similar to syncronized, but more power
- Reentrant means that one thread can acquire the same lock multiple times.
    - Syncrhonized is reentrant, but not all locking libraries are
- Example

```
ReentrantLock lock = new ReentrantLock();
lock.lock()
lock.unlock()
```

- These locks are more flexible
    - They are used by calling lock.lock() and lock.unlock(), which lets you break out of the nested block structure.
    - Configurable fairness policies, timeouts on waiting, interruptability, try but don't wait for the lock
    - Visibility on what else is waiting on the lock
    - Higher performance

ReadWriteLock

- ReadWriteLock lets multiple threads read, as long as no one is writing
- Example

```
ReadWriteLock lock = new ReentrantReadWriteLock();
lock.writeLock().lock()
```

StampedLock

- Supports optimistic locking
  - Rather than blocking, take a version number
  - That can be used to check if you lock is still valid

Other

- Stamped lock supports optimistic locking
    - Rather than blocking, take a version number
    - That can be used to check if you lock is still valid
    - Can be more efficent if lots of threads are reading a value that rarely changes
- Semaphor limits the number of threads that can concurrently access a lock

Atomic classes

- Atomic classes exist for all primitives, e.g. AtomicBoolean, AtomicLong
- These wrap up common multi-step operations into atomic functions, so you don't have to, e.g

  updateAndGet
- Concurrent collections (e.g. concurrentHashmap, concurrentArrayList) do a similar thing for collections

“Thread safe” classes

- Some classes have thread safety built in:
    - AtomicLong, AtomicBoolean, AtomicReference
    - ConcurrentList, ConcurrentHashmap
- Be careful in using these for mult-variable or multi-step processes

Volatile

- Java threads store variable values in their own cache
- If one thread modifies a value, then another thread accesses it, it can read a “stale” value
- The volatile key stops the value being cached, ensuring no reads will be stale
- Great if you have multiple threads reading and only one thread writing
- Doesn’t protect you against simultaneous write race conditions

Recap

- The simplest ways to make multi-threaded code safe are:
  - Keeping data contained in one thread
  - Immutability
  - Statelessness
- If you can't do that, use locking
  - Be consistent with your lock: use the same lock when accessing the same data
  - Synchronized is a good basic choice
  - Other options can improve performance