Concurrency

Synchronization

- Two problems
  - Mutually exclusive execution
  - Visibility of changes

Visibility example

Broken!

```
public class SynchronizedMemory {
    private static boolean stop;

    public static void main(String[] args) throws InterruptedException{
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while(!stop) {
                    i++;
                }
            }
        });

        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stop = true;
    }
}
```

Runs forever

Visibility problems

- VM optimizations might cause reading stale data
  - Hoisting
  - Caching
- Always test using the server VM
  - -server option

Synchronize

- Synchronize makes changes visible to all threads
- Mutual exclusive execution
- Both read and write should be synchronized!
- Expensive

```java
public class SynchronizedMemory {
    private static boolean stop;

    private synchronized static boolean shouldStop() {
        return stop;
    }

    private synchronized static void stop() {
        stop = true;
    }
    public static void main(String[] args) throws InterruptedException{
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while(!shouldStop()) {
                    i++;
                }
            }
        });

        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stop();
    }
}
```

volatile

- Ensures visibility
  - Prevents reading stale data
- Does not ensure thread safety!
- Only slightly more expensive

```java
public class SynchronizedMemory {
    private static volatile boolean stop;

    public static void main(String[] args) throws InterruptedException{
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while(!stop) {
                    i++;
                }
            }
        });

        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stop = true;
    }
}
```

Atomicity

Broken!

```
public class AtomicExample {
    private long hits = 0;

    public void hit() {
        hits++;
    }
}
```

++ is not atomic

Is this code thread safe?

Atomicity

- Operations that consists of multiple steps
- e.g. ++
  - read
  - modify
  - write

AtomicLong

- Wraps a long
- Ensures atomicity

```
public class AtomicExample {
    private AtomicLong hits = new AtomicLong(0);

    public void hit() {
        hits.incrementAndGet();
    }
}
```

Broken!

```
private final AtomicLong lastNumber = new AtomicLong();
private final AtomicReference<Long[]> storedFactors =
        new AtomicReference<Long[]>();

public Long[] factor(long number) {
    if (lastNumber.get() == number) {
        return storedFactors.get();
    } else {
        Long[] factors = calculate(number);
        storedFactors.set(factors);
        lastNumber.set(number);
        return factors;
    }
}
```

Is this code thread safe?

Operation not atomic

Multi value atomicity

- Ensuring atomicity per value is not enough
  - A race condition might occur
- Preserve state consistency in atomic operations

Lazy initialization

Broken!

```
public class SingletonExample {
    private static SingletonExample instance;

    private SingletonExample() {
    }

    public static SingletonExample getInstance() {
        if(instance == null) {
            instance = new SingletonExample();
        }

        return instance;
    }
}
```

Possible race condition

Is this code thread safe?

Publication

Broken!

```
public class UnsafePublication {
    public Holder nrHolder;

    public void init() {
        nrHolder = new Holder(10);
    }
}

class Holder {
    private int nr;

    Holder(int nr) {
        this.nr = nr;
    }

    public int getNr() {
        return nr;
    }
}
```

Visibility not guranteed

Safe publication

- ☐ Initialize from a static initializer
- ☐ Reference stored as volatile or AtomicReference
- ☐ Store in final field
- ☐ Guard with lock

Safe publication examples

```
public volatile Holder nrHolder;
```

```
public final Holder nrHolder = new Holder(10);
```

```
public static Holder nrHolder = new Holder(10);
```

```
public class Publication {
    private Holder nrHolder = new Holder(10);

    public synchronized void init() {
        nrHolder = new Holder(10);
    }

    public synchronized Holder getHolder() {
        return nrHolder;
    }
}
```

Immutable

- State cannot be modified after construction
- All fields are final
- Properly constructed
  - This reference doesn't escape during construction

Immutable example

```
public class ImmutableHolder {
    private final int nr;

    public ImmutableHolder(int nr) {
        this.nr = nr;
    }

    public int getNr() {
        return nr;
    }
}
```

Immutable

- ☐ Always thread safe
- ☐ No synchronization required
- ☐ No visibility problems