# week 8 - basics of multi-threaded java programs

**Overview**
This week we will start looking at how to write multi-threaded Java programs. In particular, we will cover some basic aspects of multi-threading in Java, including creation and lifecycle of Java threads, thread interference, and mutual exclusion through the keyword `synchronized`.

**References**
**Essential readings: GPBBHL 1.3, 2**
Further readings: OO "Thread Objects", OO "Synchronization", GPBBHL 1.4

---

# Chapter 1.3 - Risks of Threads

**Risks of Threads** (Thread safety issues)
What are the hazards / issues of thread safety that can arise:

**Safety Hazards**
Can result from ordering of operations in multiple threads being unpredictable if there is no synchronisation between threads - interleavings

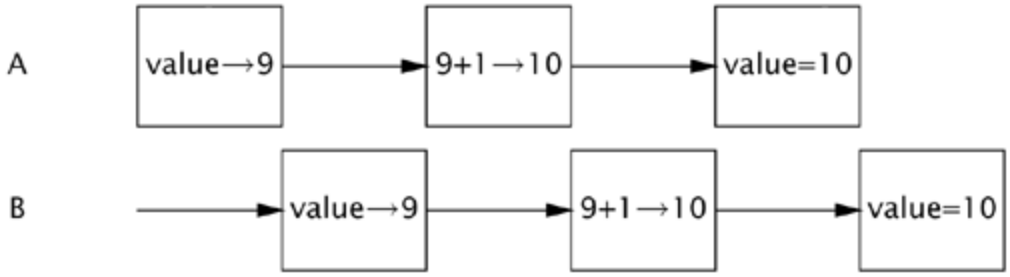
SOme code can behave correctly in a single-threaded environment BUT not in a multi-threaded environment

For example:

```
tents
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
        return value++;
    }
}
```

**Figure 1.1. Unlucky Execution of `UnsafeSequence.Nextvalue`.**




(Worst case of interleaving - can potentially happen if we incorrectly assume things will happen in a particular order)

Happens if two threads call getNext function and get the same value. THe increment notation `value++` involves 3 separate operations (read the value, add one to it, and write out the new value).

As operations in multiple threads can be interleaved by runtime, it can be possible for two threads to perform all 3 operations at the same time, resulting in the same sequence number being returned from multiple calls even in different threads.

Whether or not the function `getNext`, which is called from multiple threads, returns a unique value mainly depends on how runtime interleaves the thread operations

The diagram above is an example of a **race condition** which occurs as we mihg tnot know the order in which threads will attempt to access the shared data - nondeterminism

**Race conditions**: common concurrency bug that occurs when two or more threads access shared data and try to change it simultaneously

**How to resolve**

- Ensure **access to shared variables are properly coordianted** so that threads do not interfere with one another - Java provides synchonization mechanisms to coordinate the access.
- Make methods `synchronized` to prevent the unfortunate interaction where two threads call the same method at the same time (so same value won't be obtained - eliminating the race condition)

e.g.

```
@ThreadSafe
public class Sequence {
    @GuardedBy("this") private int value;

    public synchronized int getNext() {
        return value++;
    }
}
```

**Liveness Hazards**
Use of threads can introduce additional forms of liveness failures that do not occur in single threaded programs

Liveness failures occur when an activity gets into a state such that it is permanently unable to make forward progress or complete its intended work

They depend on relative time of events in different threads, thus dont always manifest themselves in development or testing

Example of liveness failure in programs is infinite loops where code following the loop does not get executed, where if there are two threads A and B, if thread A is waiting for a resource that thread B is holding exclusively and thread B doesn't release it due to an infinite loop, thread A would have to wait forever.

Forms of liveness hazards:

- deadlock
- starvation
- livelock

**Performance Hazards**
Performance hazards can cause problems like poor sevice time, responsiveness, throughput, resource consumption or scalability

Threads can carry some degree of runtime overhead

Context switches occur when scheduler suspends an active thread temporarily so another thread can run.
They happen more frequently in applications with many threads, and it can contribute to significant costs: saving and restoring execution context, loss of scalability and CPU time spent scheduling threads instead of running them.

Synchronisation methods introduced for threads sharing data can contribute to additional performance costs.

# Chapter 2

**Thread Safety** - aim is to **protect data from uncontrolled concurrent access** where thread safe code has to manage threads' access to shared, mutable states (data stored in state variables) in memory.

An object's state can be any data that can affect its behaviour, where its state may include instance (static) variables or fields from other dependent objects, which can be potentially be accessed and modified by multiple threads during its runtime.

Whenever more than 1 thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronisation, where you can use teh `synchronized` keyword in Java (ensure only 1 thread can execute a synchronised block at a time) - provides exclusive locking. Synchronisation in Java can also use the `volatile` variables, explicit locks and atomic variables

if multiple threads access the same mutable state variable without appropriate synchronisation, your program is broken. How to fix it:

- don't share the state variable across threads
- make state variable immutable
- or use synchronisation whenever accessing the state variable

it is easier to design a class to be thread-safe rather than retrofit it for thread safety later - less code that has access to a particular variable => easier to ensure they have proper synchronisation => easier to reason about conditions under which a given variable might be accessed.

you can store states in public (static) fields or publish a reference to an internal object but it is better to encapsulate these states.

Designing thread-safe classes involve OO techniques e.g. encapsulation, immutability, and clear specification of invariants (some set of conditions or assertions that need to hold (must be true) throughout the life of an object of a class)

## 2.1 Thread Safety

A class is **thread-safe** if it behaves correctly when accessed from multiple threads, regardless of scheduling and interleaving of the execution of those threads by runtime environment and with no additional synchronisation or other coordination on the part of the calling code.

No set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state

Thread-safe clients encapsulate any needed synchronization so that clients need not provide their own

example - servlet
Very often thread safety requirements stem not from decision to directly use threads but from decisions to use a facility like servlet framework

What are stateless objects:

- objects that doesnt contain any instance or class (static) variables
- objects that doesnt store data between invocations such that each method call is indepedent of previous calls
- objects that has no mmeory of past operations

**Stateless objects are always thread safe** as they have no fields and doesnt reference any fields from other classes. Actions of a thread accesing a stateless object cannot affect correctness of operations in other threads as more than 1 thread do not share the same state thus cannot influence result of another thread accessing the same stateless object.

## 2.2 Atomicity

Atomic operations - operations that execute as a single, indivisble operation.

Example `++count` is not an atomic operation as it consist of 3 distinct steps: fetch current value, add one to it, write the new value back (*read-modify-write* operation)

race conditions - possibility of incorrect results due to unlucky timing of more than two threads (jtnterleavings)

**Race Conditions**
Occurs when correctness of computation depends on relative timings (interleavings) of multiple threads by runtime, where getting the right results depends on lucky timing and incorrect computations occur in specific interleavings (e.g. unlucky timings)

**Most common type of race condition**: check-then-act (where a potentially stale observation is used to make a decision on what to do next) - where you observe something to be true, and then take action based on that observation; but that obersavtion could have become invalid between the time you observed it and the time you acted on it, causing a problem
e.g. you see file X doesn't exist so you create file X. But in the meantime, someone else created file X, resulting in unexpected exception, overwritten data or file corruption.

Another example of check-then-act is during **lazy initialisation** (objects are only initialised if they are needed and they are initialised only once)

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

Can cause race condition as if threads A and B execute getInstance at the same time, A sees that instance is null (depends on timing) so creates new instance of ExpensiveObject. B can also examine it not having an instance (instance == null) so creates a new instance. The check for instance depends on timing based on scheduling and how long A takes to instantiate the object and set the instance field (the state)

This reuslts in two callers to getInstance receiving two different results, even though getInstance is supposed to return the same instance consistently. Having two different instances can cause problems e.g. lost registrations or multiple activites to have inconsistent views of the set of registered objects

Race conditions don't always result in failure: some unlucky timing is also required, but it can cause serious problems

**Compound actions**
To avoid race coniditons, there must be a way to prevent other threads from using a variable while in midst of modifying that variable, so other thrads can observe or modify the state only before a thread starts/after that thread finishes

**Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has. An atomic operation is atomic with respect to all operations, including itself, that operate ont he same state**

To ensure thread safety, check-then-act operations (e.g. lazy initialisation) and read-modify-write operations (e.g. increment) must always be atomic where they are referred as compound actions where they are sequence of operations that must be executed atomically to remain thread-safe.

We don't need to consider all partial states that could occur during execution as each atomic operation completely finishes before another one can begin

Locking is a mechanism in Java to ensure atomicity

It is recommended to **use existing thread-safe objects** e.g. AtomicLong **to manage your class's state as its easier to reason about possible states and state transitions for existing thread-safe objects than it is for arbitrary state variables, and this makes it easier to maintain/verify thread safety**

example:

---

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

---

AtomicLong ensures all actions that access the counter state are atomic as state of the servlet is state of the counter, resulting in the counter and hence the servlet being thread-safe. AtomicLong is an existing thread-safe object used to manage the counter state

## 2.3 Locking

**To preserve state consistency, update related state variables in a single atomic operation**

**Intrinsic locks**
Via Java's `synchronized` block which has two parts:

1. reference to an object that will serve as the **lock**
2. block of code to be guarded by that lock

A `synchronized` method is a shorthand for a `synchronized` block that spans an entire method body,a dn whose lock is the object on which the method is being invoked.

```
synchronized (lock) {
    // Access or modify shared state guarded by lock
}
```

Every java object can implicitly act as a lock for purpose of synchronization where these built-in locks are known as **intrinsic locks** or monitor locks.

Lock is automatically acquired by the executing thread before entering a `synchronized` block and automatically released when control exits the `synchronized` block, whether by normal control path or throwing an exception out of the block

In Java, intrinsic locks ensure that at most 1 thread may own the lock. When thread A attempts to acquire the lock held by thread B, thread A has to wait (blocked) until thread B releases that lock. So in the case thread B never releases that lock (maybe due to infinite loop), thread A has to wait forever

Synchronized blocks guarded by the same lock execute atomically with respect to one another (as only 1 thread at a time can execute that block guarded by a given lock)

No thread executing a synchronized block can observe another thread to be in the middle of a `synchronized` block guarded by the same lock

---

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,
                                     ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}
```

---

Example: service is made a `synchornized` method so only 1 thread can enter service at a time. This makes the servlet thread-safe

**Reentrancy**
Threads can acquire locks they previously held, provided they have already fully released thoes locks (Java locks are mutually exclusion locks - mutexes)

As intrinsic locks are reentrant, if a thread tries to acquire a lock that it already holds, the request succeeds. Locks are acquired on a per-thread rather than per-invocation basis.

Reentrancy is implemented by associating with each lock an acquisition count (tracks how many times the lock has been acquired by the owning thread) and an owning thread whrre:

- lock is released/unheld if count is 0
- count is set to 1 if a thread acquires a previously unheld lock
- count is incremented if same thread acquires that lock again
- count gets decremented when owning thread exits the synchronized block

## 2.4 Guarding state with locks

Locks can be used to guarantee exclusive access to shared states as they enable serialised access to code paths that they guard

Compound actions e.g. read-modify-write/check-then-act actions must be made **atomic** by holding a lock for the entire duration of the compound action

If synchronisation is used to coordinate access to a variable, synchronisation is needed **everywhere that variable is accessed**

**For each mutable state variable that may be accessed by more than one thread, all accesses to that variable must be performed with the same lock held - we can say that the variable is guarded by that lock**

**Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.**

Common method: encapsulate all mutable states within an object and protect it from concurrent access by synchronising any code path that access mutable state using the object's instristic lock

Note: ONLY mutable shared data accessed by multiple threads needs to be guarded by locks

**For every invariant that involves more than 1 variable, all the variables involved in the invariant MUST be guarded by the same lock** - allows access/updates to these variables in a single atomic operation to preserve the invariant

While synchronised methods can make individual operations atomic, additional locking is required when multiple operations are combined into a compound action.

Synchronising every method can lead to liveness or performance problems

## 2.5 Liveness and performance

Poor concurrency - when multiple requests arrive and they have to queue and are handled sequentially where the number of simultaneous invocations is limited by structure of the application, not by availability of processing resources.

To solve that, you can narrow the scope of the `synchronized` block where you should exclude long-runnibng operations that do not affect shared states from being inside a `synchronized` block so other threads are not prevented from accessing the shared state while that operation is in progress

Access to local variables that are not shared by threads do not require synchronisation

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() { return hits; }
    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

Synchronised blocks are used for mutable state variables. Atomic variables are useful for effecting atomic operations on a single variables but not required in example as two different synchronised blocks are used to construct atomic operations. Example shows a balance between simplicity (synchronising the entire method) and concurrency (synchronising the shortest code paths).

Size of synchronised blocks depend on tradeoff between safety, simplicity and performance. **When implementing a synchronisation policy, resist the temptation to prematurely sacrifice simplicity (potentially compromise safety) for the sake of performance**

Whenever you use locking, you need to be aware of what the code in the block is doing and how likely it is to take a long time to execute. Holding a lock for a long time can introduce the risk of liveness or performacne problems

**Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O**