week 9 - java memroy model, java monitors and conditional synchronisation

Overview
We have now covered fundamental aspects of concurrency including interleaving of threads, synchronization, and locking. This week we will learn about variable visibility and the Java Memory Model. We will then discuss the Monitor pattern and conditional synchronization that are often used to scale programs to larger concurrent systems.

References
Essential readings: GPBBHL 3 except 3.3, preamble of **16 till 16.1.1 (excluded), 4.2, 14.1, 14.2**
Further readings: GPBBHL 3.3, 16, 4.1, 4.3, 4.4

Chapter 3 - Sharing Objects

Ensure when a thread modifies the state of an objkect, other threads can actually see the changes that were made - not possible without synchronisation

3.1 Visibility

No guarantee that reading thread will see a value written by another thread on a timely basis, or even at all. Synchronisation must be used to ensure visibility of memory writes across threads

Example:

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Sharing variables withotu synchronisation - NoVisibility could loop forever because value of ready might never become visible to reader thread OR it could print zero because write to ready might be made visible to reader thread before the write to number - reordering

There is no guarantee that operations in 1 thread will be performed in the order given by program as long as reordering is undetectable from within that thread - even if that reordering is apparent to other threads

In absence of synchronization, compiler, processor and runtime can do some downright weird things to the order in which operations appear to execute. Attempts to reason about the order in which memory actions "must" happen in insufficiently synchronized multithreaded programs will almost certainly be incorrect

ALWAYS use the proper synchronization whenever data is shared across threads

Stale Data
Data that is **out-of-date** which is not all-or-nothing: a thread can see an up-to-date value of one variable but a stale value of another variable that was written first. This happens unless synchronisation is used

Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations and infinite loops which are serious safety or liveness failures.

Example:

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int  get() { return value; }
    public void set(int value) { this.value = value; }
}
```

Listing 3.3. Thread-safe Mutable Integer Holder.

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

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value; }
}
```

value field is accessed from both get and set withotu synchronisation - meaning it is suspectible to stale values where if one thread calls set, other threads calling get may or may not see the update

Thus you synchronise both getter and setter methods to eliminate stale values being used by threads

Out-of-thin-air safety

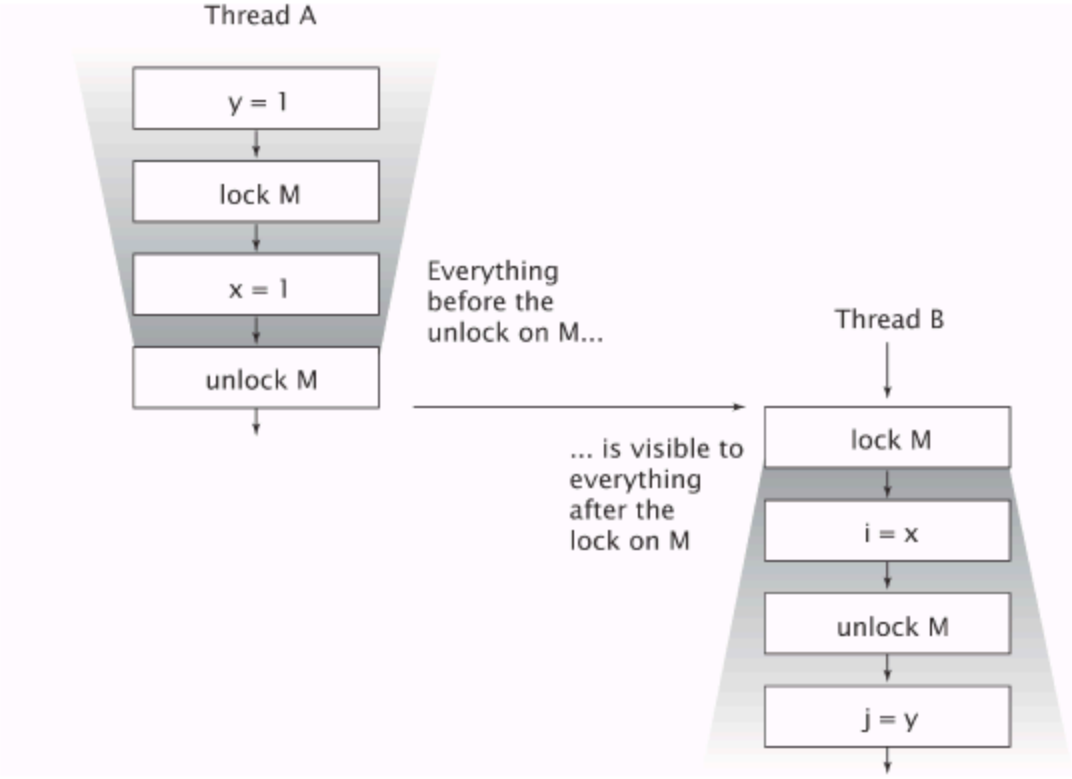
- safety guarantee that a stale value seen was placed there by some thread rather than a random value
- applies to all variables apart from 64-bit numeric variables (double/long) that are not declared `volatile` - because JMM requires all data types apart from double/long to have atomic fetch and store operations as JVM treats 64-bit read/write operations as two separate 32-bit operations
- you must declare double/long variables `volatile` or guard them witha. lock to ensure safety when using shared mutable long/double variables in multi-threaded programs

Locking and visibility

- Intrinsic locking used to guarantee one thread sees the effects of another in predictable manner where when thread A executes a synchronized block and then thread B enters a synchronized block that is guarded by the same lock, the values of variables that were visible to A prior to releasing the lock are guaranteed to be visible to B upon acquiring the lock.
- Basically everything A did in or prior to synchronized block is visible to B when it executes a synchronized block guarded by the same lock - there is NO guarantee if the block isnt synchronized

- ALlt hreads to synchronize ont he same lock when accessing a shared mutable variable - to guarantee that values written by 1 thread are made visible to other threads, otherwise if a thread reads a variable without holding the approriate lock, it might see a stale value

Figure 3.1. Visibility Guarantees for Synchronization.



Visibility guarantees for synchronization

locking isnt all about mutual excluision - it is about memory visiblity. to ensure all threads see the most up-to-date values of shared mutable variables, the reading/writing threads must synchronize on a common lock

Volatile Variables

- Via `volatile` keyword in Java when declaring variables to indicate its a volatile variable
- Ensures updates to a variable are propagated predictably to other threads
- when a field is declared `volatile`, the compiler and runtime are put on notice that this variable is shared and that operations on it should not be reordered with othert memory operations
- Volatile variables a4re not cached in registers or in caches where they are hidden from other processes
- A read of a volatile variable always returns the most recent write by any thread
- Writing to a volatile variable is like exiting a synchronized block, and reading from a volatile variable is like entering a synchronized block

- example: when thread A writes to volatile variable and subseqently B reads that same variable, the values of all variables that were visible to A prior to writing to the volatile variable becomes visible to B after reading the volatile variable
- however code that rely on volatile variables for visiblity of arbitrary state is more fragile/harder to understand than code that uses locking
- **use `volatile` variables only when they simplify implementing/verifying your synchronization policy; avoid using `volatile` variables when verifying correctness would require subtle reasoning about visibility. Good uses of `volatile` variables include ensuring the visibility of their own state, that of the object they refer to, or indicating that an important life-cycle event (e.g. initialisation/shut down) has occurred.**
- volatile variables are convenient but have limitations
- uses of volatile variables:
  - completion
  - interruption
  - status flag
    - example: asleep flag

      ```
      volatile boolean asleep;
      ...
          while (!asleep)
              countSomeSheep();
      ```

- volatile does not guarantee atomicity

Locking guarantees visibility and atomicity; volatile variables only can guarantee visibility

You use volatile variables only when all the following criteria are met:

- writes to that variable do not depend on the current value, or you can ensure that only a single thread ever updates the variable
- the variable doesn't participate in invariants (doesn't affect what is considered correct) with other state variables
- locking is not required for any other reason while the variable is being accessed

3.2 Publication and Escape

Publishing an object = making the object available to code outside of its intended scope. This happens when you:

- store a reference to the object where other code can find it
- returning the object from a nonprivate method, or
- passing the object to a method in another class

Once published, other threads can access the object, potentially leading to thread safety issues

Ensure that objects and their internals are not published, but we might want to publish an object for general use, but with synchronsiation especially in a thread-safe manner.

Publishing internal state variables can compromise encapsulation and make it harder to preserve invariants, anbd publishing objects before they are fully constructed can affect thread safety.

Escaped object = An object that has been published when it shouldnt have been

Listing 3.5. Publishing an Object.

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

Blatant form of publication: Storing a reference in public state field, where any class and thread could see it. `initialize` method instantiates a new hashset and publishes it by storing a reference to it in knownSecrets

Publisihng one object may indirectly publish others. Also returning a reference from a non-private method also publishes the returned object.

Listing 3.6. Allowing Internal Mutable State to Escape. ***Don't do this.***

```
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```

YOu must not puiblish states as any caller can modify its contents - in the example, the `states` array has escaped its intended scope, as it is made public while it is meant to be private, as the method `getStates` returned a direct reference to the internal `states` array.

Publishing an object also publishes any objects referred to by its non-private fields where any object that is reachable from a published object by following some chain of non-private field references and method calls has also been published

Passing an object to an alien method (method with a unspecified behaviour by a class) can be considered as publishing that object - you can't tell if that method would publish the object or retain a reference to it (could be used from another thread)

Encapsulation helps to analyse porograms for correctness and harder to accidentally violate design constrraints

Publishing an inner class instance can potentially publish an object or its internal state

Listing 3.7. Implicitly Allowing the `this` Reference to Escape. *Don't do this.*

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```

Inner class instances contain a hidden reference to the enclosing ThisEscape instance, which then causes that instance to be published when ThisEscape publishes EventListener, and this might publish the not yet fully constructed ThisEscape object;

Safe construction practices

- An object is not properly constructed if the `this` reference escapes during construction
- **do not allow the `this` reference to escape during construction**
  - don't start threads from a constructor as an object always shares its `this` reference with a newly created thread if the thread was created from its constructor\
  - don't register listeners in constructors
  - don't pass `this` to methods called from constructors
  - new thread might be able to see the owning object before it is fully constructed
  - it is fine to create new threads in constructor but DO NOT start them in the constructor immediately
- solution is to expose a `start` or `initialize` method that starts the owned thread
- to register an event lisner or start a thread from constructor, you should use a private constructor and public factory method as shown in example below, to ensure safe publication

Listing 3.8. Using a Factory Method to Prevent the `this` Reference from Escaping During Construction.

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

- This approach ensures the object is fully constructed before it's published:

1. The constructor only initializes fields
2. The factory method creates the object first
3. Only after construction completes does it register the listener
4. Then it returns the safely constructed object

By following these practices, you ensure objects are fully constructed before they become visible to other threads, preserving thread safety and encapsulation.

3.4 Immutability

Immutable object:

- object whose state cannot be changed after construction
- **are always thread-safe**
- only can be in one state that is carefully controlled by the constructor
- buggy/malicious code cannot modify immutable objects (unlike mutable objects where their states can be modified or their reference retained and their states modified later from another thread)
- safe to share and publish freely without needing to make defensive copies
- can still use mutable objects internally to manage their state

An object is **immutable** if:

- its state cant be modified after constructiobn
- all its fields are final
- it is **properly constructed** (the `this` reference doesnt escape duriing construction)

Listing 3.11. Immutable Class Built Out of Mutable Underlying Objects.

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

it is impossible to modify the Set after construction as `stooges` reference is `final`, so all object state is reached through a `final` field.

there is a difference between an object being immutable and the reference to that object being immutable.

program state stored in immutable objects can still be updated by "replacing" immutable objects with a new instance holding new state

Final fields
`final` keyword supports construction of immutable objects as `final` fields cant b emodified. this gaurantees initialisation safety that lets immutable objects be freely accessed and shared without synchronisation

Specifying an object as `final` even if they are mutable can ssimplify reasoning about its stae as limiting mutability of an object restricts its set of possible states

just as it is good practice to make all fields `private` unless they need greater visibility, it is also good practice to make all fields `final` unless they need to be mutable

Example: using `volatile` to publish immutable objects
immutable objects can provide a weak form of atomicity.

Listing 3.12. Immutable Holder for Caching a Number and its Factors.

```
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

you can use an immutable object to hold all variables to eliminate race conditions.

immutable holder object ensures that other threads won't modify its state once a thread acquires a reference to that object.

3.5 Safe publication

We need to safely publish objects so they can be shared across threads

Improper publication can allow another thread to observe a *partially constructed object* which can stem from visibility problems where an object could appear to another thread to be in an inconsistent state

Improper publication: when good objects go bad

Listing 3.15. Class at Risk of Failure if Not Properly Published.

```
public class Holder {
    private int n;

    public Holder(int n) { this.n = n; }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError("This statement is false.");
    }
}
```

Holder is deemed not properly published as synchronisation was not used to make Holder visible to other threads

Other threads could see a stale value for holder field and thus see a null reference or other older value even thougha. value has been placed in holder. Alternative scenario is that other

threads could see an up-to-date reference for holder reference but stale values for the state of the holder.

A thread could see a stale value the first time it reads a field and then a more up to date value next time, hence method `assertSanity()` could throw an error.

Immutable objects and initialization safety
Object reference being visible to another thread doesnt necessarily mean that the state of that object is visible to the consuming thread - synchronisation is needed to guarantee a consistent view of the object's state

Immutable objects can be used safely by any thread without additional synchronization, even when synchronization is used to publish them.

Final fields that refer to mutable objects still require synchronization to access the state of objects they refer to

Safe publication idioms
Not immutable objects must be **safely published** where it usually involves synchronization by both publishing/consuming thread

To publish an object safely, both the reference to object and object's state MUST be made visible to otehr threads simultaneously. A properly constructed object can be safely published by:

- initializing an object reference from a static initialiser
  - e.g. `public static Holder holder = new Holder(42);`
- storing a reference to it into a `volatile` field or `AtomicReference`
- storing a reference to it into a `final` field of a properly constructed object OR
- storing a reference to it into a field that is guarded by a lock

safe publication guarantees in Java

manner has no *explicit* synchronization. The thread-safe library collections offer the following safe publication guarantees, even if the Javadoc is less than clear on the subject:

- Placing a key or value in a `Hashtable`, `synchronizedMap`, or `Concurrent-Map` safely publishes it to any thread that retrieves it from the `Map` (whether directly or via an iterator);
- Placing an element in a `Vector`, `CopyOnWriteArrayList`, `CopyOnWrite-ArraySet`, `synchronizedList`, or `synchronizedSet` safely publishes it to any thread that retrieves it from the collection;
- Placing an element on a `BlockingQueue` or a `ConcurrentLinkedQueue` safely publishes it to any thread that retrieves it from the queue.

Other handoff mechanisms in the class library (such as `Future` and `Exchanger`) also constitute safe publication; we will identify these as providing safe publication as they are introduced.

Effectively immutable objects
Safe publication is sufficient for other threads to safely access objects that are not going to be modified after publication without additional synchronization.

Effectively immutable objects: objects that are not technically immutable but their states will not be modified after publication

Safely published effectively immutable objects can be used safely by any thread without additional synchronization

Mutable objects
Synchronization must be used not to publish a mutable object but ensure visiboility of subsequent modifications that happen every time the object is saccsessed.

Publication requirements for an object depend on its mutability:

- immutable objects can be **published through any mechanism**
- effectively immutable objects **must be safely published**
- mutable objects **must be safely published** and **must be either thread-safe or guarded by a lock**

Sharing objects safely
When you get a reference to an object, you should know what you are allowed to do with it. When you publish an object, you should document how the object can be accessed. Not knowing the rules of engagement for a shared object can lead to concurrency errors

Policies for using/sharing objects in a concurrent program:

- **Thread confined** - thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread
- **Shared read-only** - a shared read-only thread can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by one thread. They include both immutable and exclusively immutable objects
- **Shared thread-safe** - a thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without additional synchronization
- **Guarded** - a guarded object can only be accessed by a specific lock held. They include objects that are encapsulated within other thread-safe objects and published objects that are known to be guarded by a specific lock

Chapter 4

4.2 Instance Confinement

Non thread-safe objects can still be used safely in a multi threaded program but have to ensure that it is only accessed from a single thread (thread confinement) or that all access to that object is guarded by a lock

Encapsulating data within an object confines access to the data to the object's methods, making it easier to ensure that the data is always accessed with the appropriate lock held.

Confined objects must not escape their intended scope, where an object may be confiend to a class instance, a lexical scope, or a thread.

example

Listing 4.2. Using Confinement to Ensure Thread Safety.

```
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```

PersonSet shows how confinement and locking can work together to make a class thread-safe even when its component state variables are not. The state of PersonSet is managed by a HashSet, which is not thread-safe. But because mySet is private and not allowed to escape, the HashSet is confined to the PersonSet. The only code paths that can access mySet are addPerson and containsPerson, and each of these acquires the lock on the PersonSet. All its state is guarded by its intrinsic lock, making PersonSet thread-safe.

The example makes no assumption about thread-safety of Person

Instance confinement

- one of the easiest method to build thread-safe classes
- allows flexibility in the choice of locking strategy
- allows different state variables to be guarded by different locks
- works by:
    - encapsulating objects - keep mutable objects private fields within a class
    - controlling access - managing all interactions with confined objects through enclosing class's methods
    - applying synchronization - using synchronization in all methods that access confined objects

It is also possible to violate confinement by publishing a supposed confined objectl if an object is intended to be confined to a specific scope, then letting it escape from that scope is a bug.

Confinement makes it easier to build thread-safe classes because a class that confines its state can be analysed for thread safety without having to examine the whole program

Java monitor pattern

- an object that follows the java monitor patterns encapsulate all its mutable state and guards it with the object's own intrinsic lock
- iit is used by many library classes
- just a convention as any lock object could be used to guard an object's state so long as it is used consistently

Example: private lock

Listing 4.3. Guarding State with a Private Lock.

```
public class PrivateLock {
    private final Object myLock = new Object();
    @GuardedBy("myLock") Widget widget;

    void someMethod() {
        synchronized(myLock) {
            // Access or modify the state of widget
        }
    }
}
```

advantages of using a private lock object compared to using intrinsic lock:

- encapsulates the loick so that client code cnanto acquire it, whereas a puiblicly accessible lock allows client code to participate in its synchronization policy - correctly or incorrectly

Verifying that a publicly accessible lock is properly used requires examining the whole program rather than a single class as clients that improperly acqurie another object's lock could cause liveness problems

Thread safety can be maintained by copying mutable data before returning it to client - performance issues could happen if a set of data is large

Chapter 14 - Building Custom Synchronizers

Class libraries include **state-dependent classes** which have operations with **state-based preconditions**

Preconditions: constraints or requirements that must be **satisfied** before a method can be executed correctly - they define the valid states/input values that a method expects before it begins execution (technically validation at the beginning of methods, but with responsibility on the caller)

You can create **custom synchronisers** which **manage state dependence** via **condition queues**, suspending and resuming threads based on state changes relevant to their specific waiting conditions. This can be done by `wait`, `notify`, `notifyAll`

14.1 Managing state dependence

A condition will never be true if **state-based precondition** (e.g. connection pool is nonempty must hold before invoking the method correctly) **doesn't hold when a method is called in a single threaded program**

State-based conditions could change through actions of other threads: a connection pool that was empty a few instructions ago can become nonempty because another thread returned an element.

clases in sequential programs can be coded to fail when their preconditions do not hold - but their state-based conditions can change through actions of other threads in concurrent programs.

state-dependent methods on concurrent objects can sometimes get away with failing when their preconditions are not met, but it is better to WAIT for the precondition to become true first

state-dependent operations that block until the operation can proceed are more convenient and less error-prone than those that fail.

blocking is done via the built-in **condition queue** mechanism which allows threads to block until an object has entered a state that allows progress and to wake blocked threads when they may be able to make further progress.

Blocking state-dependent actions

Listing 14.1. Structure of blocking state-dependent actions.

```
acquire lock on object state
while (precondition does not hold) {
    release lock
    wait until precondition might hold
    optionally fail if interrupted or timeout expires
    reacquire lock
}
perform action
release lock
```

lock is released and reacquired in the middle of the operation. state variables that make up the precondition must be guarded by the object's lock, so that they can remain constant while testing preconditions. but if precondition doesnt hold, the lock must be released so another

thread can modify the object state - otherwise precondition will never become true. then lock must be reacquired before testing precondition again.

example of preconditions - bounded buffer (put/take operations)

- cannot take an element from an empty buffer
- cannot put an element into a full buffer

how to deal with precondition failure (wrong state) in state dependent operations?

- throw an exception
- return an error status (making it the caller's problem), or
- blocking until object transitions to the right state

Note: **callers must deal with precondition failures themselves**

Crude Blocking

Listing 14.5. Bounded Buffer Using Crude Blocking.

```
@ThreadSafe
public class SleepyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public SleepyBoundedBuffer(int size) { super(size); }

    public void put(V v) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isFull()) {
                    doPut(v);
                    return;
                }
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }

    public V take() throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isEmpty())
                    return doTake();
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }
}
```

In this example, the caller needs to deal with the `InterruptedException` (catch that exception if precondition has not been met)

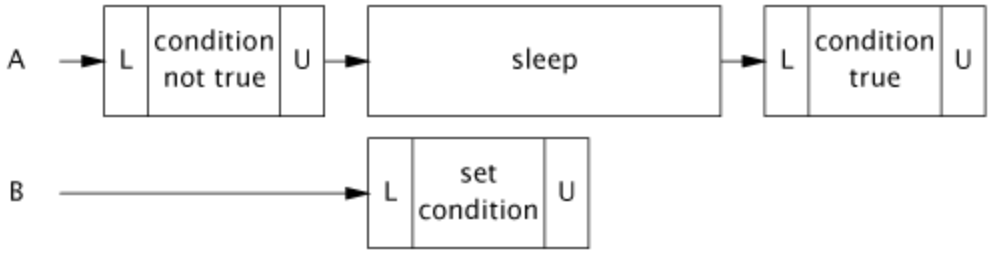
when a method blocks waiting for a condition to become true, it is good to provide a cancellation mechanism (can be done by interruption, returning early and throwing the exception if interrupted)

Condition queues:

- a method to suspend a thread but ensuring it is awaken promptly when a certain condition beocmes true
- gives a group of threads - called the **wait set** - a way to wait for a specific condition to become true
- elements of a condition queue are threads that are waiting for the condition
- makes it easier/more efficient to **express and manage state dependence** tho it doesnt let you do anything you cant do with sleeping and polling

Similar to the "toast is ready" bell in toaster (if you heard the bell, you are notified promptly by toaster when your toast is ready so you can get your toast, but if you are not listening for it you could miss the notification but you can observe the new state upon returning to the kitchen and either retrieve the toast if its finished or start listening for the bell again if its not)

Figure 14.1. Thread Oversleeping Because the Condition Became True Just After It Went to Sleep.



Each object can also act as a condition queue where an object's intrinsic lock and its intrinsic condition queue are similar: to call any of the condition queue methods on object X, you must hold the lock on X/

mechanism for waiting for state-based conditions is tightly bound to the mechanism for preserving state consistency where you cant wait for a condition unles you can examine the state, and you cant release another thread from a condition wait unless you can modify the state.

Listing 14.6. Bounded Buffer Using Condition Queues.

```
@ThreadSafe
public class BoundedBuffer<V> extends BaseBoundedBuffer<V> {
    // CONDITION PREDICATE: not-full (!isFull())
    // CONDITION PREDICATE: not-empty (!isEmpty())

    public BoundedBuffer(int size) { super(size); }

    // BLOCKS-UNTIL: not-full
    public synchronized void put(V v) throws InterruptedException {
        while (isFull())
            wait();
        doPut(v);
        notifyAll();
    }

    // BLOCKS-UNTIL: not-empty
    public synchronized V take() throws InterruptedException {
        while (isEmpty())
            wait();
        V v = doTake();
        notifyAll();
        return v;
    }
}
```

Object.wait() atomically releases the lock and asks OS to suspend the current thread, to allow other threads to acquire the lock and therefore modify the object state. Upon waking, it reacquires the lock before returning. Calling wait technically is like "i want to go to sleep, but wake me up when something intersting happens" and calling notification methods means that "something interesting happened"

14.2 Using condition queues

They make it easier to build efficient/responsive state-dependent classes but can be easy to use incorrectly as there are a lot of rules on how to use condition queues properly that are not enforced by compiler or platofrm.

Condition preddicate
to use condition queues correctly, you must identify the **condition predicaters** that the object may wait for

it is the precondition that makes an operation state-dependent in the first place
e.g. in a bounded buffer (examples above), the condition predicate for `take` is "the buffer is not empty", which `take` must test for before proceeding. condition predicate for `put` is "the buffer is not full".

Condition predicates are expressions constructed from state variables of the class where for a bounded buffer, it can check for a buffer being either full or empty by comparing count to 0 or to the buffer size, respectively

document the condition predicate(s) associated with a condition queue and the operations that wait on them

there is an important three-way relationship in a condition wait involving locking, wait method and a condition predicate

condition predicates involve state variables which are guarded bu a lock, so before testing the condition predicate, we must hold the lock where the lock object and condition queue object (the object on which `wait` and `notify` are invoked) must also be the same object

if a condition predicate is not true, the state dependent operation must wait until another thread makes the condition predicate true by modifying the object's state where `wait` is called on the intrinsic condition queue, which requires holding the lock on a condition queue object.

the `wait` method:

- releases the lock, blocks the current thread, and waits until a specific timeout expires, the thread is interrupted or the thread is awakened by a notification
- after thread wakes up, it reacquires the lock before returning

a thread waking up from `wait` gets no special priority in reacquring the lock, but it still has to contend for the lock just like any other thread trying to enter a `synchronized` block

every call to `wait` is implicitly associated with a specific condition predicate. when calling `wait` regarding a particular condition predicate, the caller must already hold the lock assocated with the condition queue, and that lock must also guard the state variables from which the condition predicate is composed

what happens when the thread wakes up too soon
a single intrinsic condition queue may be used with more than 1 condition predicate, thus another thread can call `notify` or `notifyAll` even if the condition precidate that you are waiting for is not changed to true - this can wake threads up too soon

it is common to have multiple condition predicates using the same condition queue

it is essential to **test the condition predicate again when a thread wakes up from any `wait` call due to a `notify` or `notifyAll` call from other threads**, and if the predicate still doesnt hold, the thread needs to go back to waiting or fail.

threads can wake up repeatedly without their condition predicate being true - so you need to always call `wait` from within a loop, testing the condition predicate in each iteration

e.g.

Listing 14.7. Canonical Form for State-dependent Methods.

```
void stateDependentMethod() throws InterruptedException {
    // condition predicate must be guarded by lock
    synchronized(lock) {
        while (!conditionPredicate())
            lock.wait();
        // object is now in desired state
    }
}
```

The method to structure condition waits ^ (via a guarded wait pattern)

When using condition waits (`Object.wait` or `Condition.await`):
you must have the following:

- always have a condition predicate - some test of object state that must hold before proceeding
- always test the condition predicate before calling wait, and again after returning from `wait`
- always call `wait` in a loop
- ensure state variables making up the condition predicate are guarded by the lock associated with the condition queue
- hold the lock associated with the condition queue when calling `wait` , `notify` , or `notifyAll`
- do not release the lock until after checking the condition predicate but before acting on it

Missed Signals
A form of **liveness failure**

Occurs when a thread must wait for a specifc condition that is already true, but fails to check the condition predicate before waiting. Now thread is waiting to be notified of an event that has already happened. technically *they did not get (missed) the notification and thus didn't wake up when the state its waiting for has changed*)

They are result of coding errors e.g. failing to test the condition predicate before calling `wait` - thus it is important to structure your condition waits as per **Listing 14.7** (canonical form for state-dependent methods)

Notifications
Whenever you wait on a condition, make sure that someone will perform a notification whenever the condition predicate becomes true

Two notification methods in condition queue API:

- `notify` - JVM selects one thread waiting on that condition queue to wake up

- `notifyAll` - wakes up all the threads waiting on that condition quue

but you must hold the lock associated with the queue object before being able to call any of these methods.

the notifying thread must release the lock as quickly to ensure waiting threads are unblocked as soon as possible, as waiting threads cannot return from `wait` without reacquiring the lock and that you mujst hold the lock on condition queue object when calling notification methods

it can be dangerous to use `notify` instead of `notifyAll` as multiple threads could be waiting in the same condition queue but for different condition predicates, and that a single notification can result in missed signals on waiting threads

use `notify` instead of `notifyAll` only when both conditions hold:

- **uniform waiters** - only 1 condition predciate is associated with the condition queue, and each thread executes the same logic upon returning from `wait`
- **one in, one out** - a notification on the condition variable enables at most one thread to proceed

it is easier to ensure your classes behave correctly when using `notifyAll` instead of `notify` even it its inefficient

Conditional notification
Optimisation to notification handling where a thread can be released from wait only if their conditions are met and are notified when those conditions become true %

Listing 14.8. Using Conditional Notification in `BoundedBuffer.put`.

```
public synchronized void put(V v) throws InterruptedException {
    while (isFull())
        wait();
    boolean wasEmpty = isEmpty();
    doPut(v);
    if (wasEmpty)
        notifyAll();
}
```

```java
// Consumer thread
synchronized (lock) {
    while (!conditionIsMet) {
        lock.wait(); // Release lock and wait
    }
    // Process the condition that's now true
}

// Producer thread
synchronized (lock) {
    // Change state to make condition true
    conditionIsMet = true;
    lock.notifyAll(); // Wake up waiting threads
}
```

Limitations of conditional notifications

- can introduce constraints that can complicate subclassing - you must structure your class so that their subclasses can add the appropriate notifcation on behalf of the base class if it is subclassed in a way that violates one of the requirements for a single or conditional notification

example of conditional notifcation:

Listing 14.9. Recloseable Gate Using `Wait` and `Notifyall`.

```
@ThreadSafe
public class ThreadGate {
    // CONDITION-PREDICATE: opened-since(n) (isOpen || generation>n)
    @GuardedBy("this") private boolean isOpen;
    @GuardedBy("this") private int generation;

    public synchronized void close() {
        isOpen = false;
    }

    public synchronized void open() {
        ++generation;
        isOpen = true;
        notifyAll();
    }

    // BLOCKS-UNTIL: opened-since(generation on entry)
    public synchronized void await() throws InterruptedException {
        int arrivalGeneration = generation;
        while (!isOpen && arrivalGeneration == generation)
            wait();
    }
}
```

in this example, ThreadGate needs to notify in both `open` and `close`, thus addition of new state dependent operation may require moidfying many code paths that modify the object state so that appropriate notifications can be performed

A state depndent class should either fully expose (and document) its waiting and notification protocols to subclasses, or prevent subclasses from participating in them at all.

designing a state-dependent class requires exposing condition queues and locks and documenting condition predicates and synchronization policy; and may also require exposing underlying state variables

Encapsulating condition queues
It is best to encapsulate condition queues so it is not accessible outside the class hierarchy in which it is used (keep it local and private without any publishing, i.e. exposed by public methods/fields)

it is hard to enforce uniform waiters requirement for a single notification unless the condition queue object is inaccessible to code you don't control

Entry and exit protocols
operations with state dependency should have a defined and docucment entry/exit portocol

entry protocol: object's condition predicate
exit protocol: examining any state variables modified by the operation to see if they may have caused some other condition predicate to be true, and if so, notifying on associated condition queue

Chapter 16

Java Memory Model (until 16.1.1)

JMM specifies the minimal guarantees the JVM must make about when writes to variables become visible to other threads. It was designed to balance the need for predictability and ease of program development with the realities of implementing high-performance JVMs on a wide range of popular processor architectures.