Threads

Introduction

Interruptions

Synchronization

Atomic

volatile

Wait/Notify

Locks and Monitors

Executors and Thread Pools

Immutable Objects

Thread-Safe Collections

INTRODUCTION

Threads

- A *thread* is a separate computation process that might execute in parallel with other processes.
- You use threads all the time:
  - Having a video playing while you are typing an email
  - Reading a website while you have music playing
- Java programs can have multiple threads.

Java Memory: Heap and Stack

- Stack Memory
  - Local variables and object references are stored in stack memory
  - Method calls (*activation records*) are stored on the stack
  - Variables on the stack exist as long as the method that created them is running
- Heap Memory
  - The actual objects are stored on the heap
  - The *reference* points to the location on the heap

- Good article/pic here: https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

Java Threads

- Each thread has its own stack memory:
  - method activation records
  - local variables
  - parameters
- But threads share the heap.
  - objects are shared!
- Threads share resources (variables and data).
  - This happens so fast that it appears that the threads are operating in parallel.
- This is more efficient and allows for easy communication…
- But of course it’s risky to have shared resources.

Single Threaded Programs

- All Java programs use threads.
- By default, each program uses a single thread.
  - The *main thread*
- If you've never used threads directly… then you've been writing one-thread programs all along!

static Thread Methods

- Thread.sleep
  - Pauses the execution of the current thread for a specified number of milliseconds (ish)
  - Throws a checked exception (InterruptedException) that must be caught (or propagated)
- Thread.activeCount()
- Thread.enumerate(Thread[] activeThreadArray)
- Thread.currentThread()
- Can set a name on a Thread using super(name) or setName method, which is useful when distinguishing between threads

Practice

- Review the example that invokes methods on the main thread.

CREATING THREADS

Creating Your Own Threads

- You can create and manage threads in your program
- Two ways: extend Thread or implement Runnable

The Thread Class

- Your class extends `Thread`
  - Overrides the `public void run()` method
  - Can call super(name) to set a name for the thread

- You start by invoking the start method:

```
myThreadObject.start()
```

  - Do not call run- call start.
  - The start method has setup code it needs to execute before it invokes run.
  - Invoking run doesn't actually create a new thread!!

The Runnable Interface

- Your class implements `Runnable`
  - Overrides the `public void run()` method
- You start by sending an instance of your class to the Thread constructor and invoking start:

```
Thread thread = new Thread(myRunnableObject)
thread.start()
```

  - Again, do not call run- call start.
  - The start method has setup code it needs to execute before it invokes run.
  - Invoking run doesn't actually create a new thread!!

The Runnable Interface

- Runnable is a functional interface!
  - Queue the lambda entrance music!
- If you have a simple run method, you can skip creating an entire class that implements Runnable and just use a lambda:

```
new Thread(
        () -> myRunningMethod();
        ).start()
```

Practice

- Review the program that prints dates using Thread and Runnable.

When should I use concurrency?

- Tasks that will take a long time to execute.
- Any program where you want one thing to be able to run in the background and do other things too.

- GUI programs- you want the program to remain responsive while it is performing tasks.
  - **NOTE:** You cannot use the Thread and Runnable setup with JavaFX.
  - The scene graph is not thread-safe and can only be accessed and modified from the UI thread called the JavaFX Application thread.
  - The javafx.concurrent package supposed multithreaded GUIs.

INTERRUPTIONS

Ending Threads

- A thread terminates when the run method terminates by:
  - returning,
  - reaching the end, or
  - throwing an uncaught exception
- You can also request a thread to terminate with an *interrupt*
  - You can’t force it! But you can request it.

Thread Interruptions

- Each thread has a boolean flag that represents whether it is interrupted

Requesting a Thread Interruption

- void interrupt()
  - Sends an interrupt request to the thread
  - The flag of the thread is set to true
  - If the thread is *blocked* (it is sleeping or waiting), an InterruptedException is thrown

Responding to Thread Interruptions

- A thread *could* just ignore interruptions! But this isn't convention.
- Threads should occasionally check to see if they've been interrupted.
  - This code is not automatic- you have to write it!
- It's common to stop a thread when it's been interrupted.
  - But again, this isn't automatic!

Responding to Thread Interruptions

- **static** boolean interrupted()
  - Tests whether the *current* thread has been interrupted
  - Side effect: resets the flag to false (clears the interruption status)
- boolean isInterrupted
  - Tests whether a thread has been interrupted **(invoke on a Thread object)**
  - No side effects
  - Often use with: Thread.currentThread()
    - Thread.currentThread().isInterrupted()

Supporting Interruptions

- Inside a run that uses sleeps:

```
try {
    while(moreWorkToDo) {
        // do the work
        // sleep
    }
} catch(InterruptedException ex) {
    // thread interrupted during sleep/wait
    return;
}
```

Supporting Interruptions

- Inside a run that doesn't sleep:

```
while(!Thread.currentThread().isInterrupted())
        && moreWorkToDo)
            // do the work
}
// exit the run method
```

Supporting Interruptions

- Inside a run that doesn't sleep:

```
try {
    while(moreWorkToDo) {
      if(Thread.currentThread().isInterrupted()) {
           // throw new InterruptedException();
      } else {
           // do the work
      }
    }
} catch(InterruptedException ex) {
      // react to all interruption (perhaps return)
}
// exit the run method
```

Practice

- Interrupt the time print thread.
  - When it’s asleep
  - While it’s running
  - Change the behavior of how the thread responds to the interrupt

Join

- Invoking join makes the current thread wait until another thread finishes to continue.
- Example: invoking myThread.join() from main will make the main method wait until myThread finishes execution.

Practice

- Wait until the time programs are done to print a final message.

SYNCHRONIZATION

Shared Resources

- Multiple threads share access to the same data.
  - Objects are on the shared heap!
- If two threads each modify the state of the object, unpredictable or incorrect results can occur.
- This is called a *race condition*.

Race Conditions

- Race conditions can occur on any statement that is not *atomic* (meaning *happening all at once*).
- Even simple statements have multiple steps:
  - i++
  - i = i + 1;
  - get the current value of i
  - add one to it
  - store the new value of i back to the variable
- You could have different threads working at different steps!

Practice

- Review the simple counter example.
  - Counter, RaceConditionTest
- Review the bank account example (bonus/fee).
  - BankAccount, BankThread, BankTester

Synchronized

- The most straightforward way to prevent a race condition is to *synchronize*.
- Synchronizing allows only one thread to access a block of code at a time.
- You can synchronize an entire method:

```
public synchronized void myMethod() {
        … critical code
}
```

- Or a block of code:

```
public void myMethod() {
        … non critical code
        synchronized(someObject) { // could be “this”
                … critical code
        }
}
```

- You can also synchronize static methods.

Synchronized

- Note that this:

```
public synchronized void myMethod() {
        … critical code
}
```

- Is equivalent to this:

```
public void myMethod() {
        synchronized(this) {
                … critical code
        }
}
```

Synchronized

- When a thread reaches synchronized code, the JVM checks to see if anyone holds the lock associated with the sync object.
  - If no one does, the current thread gets the lock.
  - If someone else does, the current thread is placed in the waiting area.
- When a thread is done with synchronized code, it releases the lock.
  - The JVM then checks the waiting area to see if someone else needs the lock.

- Synchronizing works great… but it can lead to inefficiency, so it should be used cautiously.

Practice

- Update the bank example to avoid race conditions.

Shortcomings of synchronized

- It can be easy to make mistakes that make code appear to be synchronized, when it’s really not
  - To avoid this, make sure to choose an object common to all threads.
- Deadlock
  - Avoid this by being careful about how synchronized methods invoke other synchronized methods (or nested synchronized blocks).

Practice

- Review an incorrect bank account examples that synchronizes the thread's method, not the Bank Account methods.
- Review the deadlock example
  - MyObject, DeadlockThread

VARIABLES

Atomic

- An *atomic* action happens all at once
  - It cannot stop in the middle
  - It either happens completely or not at all
- i++ is **not** atomic- there are multiple steps

Atomic Variables

- java.util.concurrent.atomic contains classes that support atomic actions on variables
- Example: AtomicInteger
  - get
  - set
  - incrementAndGet
  - decrementAndGet
  - updateAndGet(IntUnaryOperator) (IntUnaryOperator: param int, return int)
- Using an atomic variable can help avoid unnecessary synchronization

Practice

- Review the revised Counter example that uses atomic integer
  - AtomicCounter, AtomicTest

LongAdder

- Java 8 added a new class LongAdder designed represent a collective sum.
- Methods
  - add(long)
  - increment
  - sum

volatile

- If one thread changes the value of a variable, you expect other threads to see that change.
- But that's not always the case!
  - The JVM allows values to be stored temporarily in working memory (like a cache), rather than main memory (the heap).
  - There is no guarantee that the heap version has been updated- the updated value could still be stored in cache.
- So one thread could change the value and that change is stored in cache. But then another thread reads from the heap and gets the wrong value.

volatile

- Declaring a variable volatile enforces that all updates will be written back to heap memory immediately.

- It also ensure that reads/writes are atomic (all at once).
  - Reads and writes are atomic for reference variables and for most primitives (NOT for long and double).
  - Reads and writes are atomic for *all* variables declared volatile (including long and double).

volatile- NOT synchronized!!

- volatile does **not** mean atomic overall!
  - myVolatileVariable++ is still not atomic!
  - volatile is **not** the same as synchronized
- volatile may be good enough if you have only one thread updating the variable and many others only reading it
  - if you have multiple threads updating, you need to synchronize!

WAIT/NOTIFY

Locks

- Every object has an *intrinsic lock*.
  - Also called a *monitor lock* or just *monitor*.
- A thread can *acquire* a lock.
  - It then *owns* the lock.
  - As long as it owns the lock, no other thread can acquire the lock.
  - When the thread it done, it can *release* the lock.

Synchronization

- If thread A tries to enter a synchronized block and thread B is already executing that block (meaning it *holds the lock on the synchronized object*), thread A is placed into a waiting area.
- When thread B finishes, it releases the lock.
- The JVM then removes A from the waiting area and assigns the lock to thread A.

- This is one specific application of the wait/notify mechanism.

wait/notify/notifyAll

- A thread *waits* on a condition to continue executing.
- Some other thread updates that condition (*notifies the thread*) when it can execute.

- notify wakes up a single thread (decided by the ThreadScheduler)
  - We don’t automatically know which one will be notified- only good for “massively parallel” operations where all threads doing the same job.
- notifyAll wakes up *all* threads

Waiting and Notifying

```
synchronized(object) {
    while(conditionThatMustBeTrue) {
          try {
            object.wait();
          } catch(InterruptedException ex) {}
     }
    object.doTheWork();
}
```

```
synchronized(object) {
    // work is ready to be done
    conditionThatMustBeTrue = true;
    object.notifyAll();
}
```

Waiting

- You have to own the lock in order to invoke wait.
  - Putting the wait() call inside a synchronized block is one way to ensure this.
  - Invoking wait() releases the lock
  - When the thread is notified, it will have to continue waiting until the lock is released
- Always wait inside of a loop that checks the condition.
  - There is no guarantee that the condition has changed as a result of the notification.
  - So you want to recheck it before moving on.

Practice

- Review the deposit/bonus methods of the bank account classes.
  - Multiple threads will make random deposits.
  - There are “bonus” threads that wait to add a $1 “savings bonus” when the balance passes a certain amount.
  - Classes: WNBankAccount, WNBankThread, WNBankTester

Producer/Consumer Relationship

- An application that shares data between two threads: a producer and a consumer
- A producer thread creates some data/item/object that a consumer thread uses.
  - Example: read an object from a database and pass it off for processing
  - Example: read input from a user and hand it off to be analyzed
- Coordination is essential!
- The consumer cannot use item unless it's been produced.
  - It has to wait.
- The producer cannot create an item if the consumer is still busy or hasn't retrieved the old item.
  - It has to wait.

Practice

- Review the producer/consumer example.
  - The *number box* holds a single number and can either be filled or empty.
  - Classes: ProducerThread, ConsumerThread, ProducerConsumerTest, NumberBox

Using wait/notify

- Most people don’t directly use wait/notify anymore.
- As Joshua Block (Effective Java) states:
  - “using wait and notify directly is like programming in “concurrency assembly language,” as compared to the higher-level language provided by java.util.concurrent. **There is seldom, if ever, a reason to use wait and notify in new code.**”
- Instead, use high-level constructors, including BlockingQueue.

Blocking Queue

- A special queue that supports the producer/consumer pattern.
  - Threads must wait for a queue to be non-empty to retrieve.
  - Threads must wait for space to add.
- Different methods based on the action you want to take when an add/remove cannot be handled immediately.

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Practice

- Review the producer/consumer example using a BlockingQueue.
  - ProducerThreadBQ, ConsumerThreadBQ, ProducerConsumerTestBQ

LOCKS

Locks

- A *lock* ensures that only one thread can access a block of code at a time.
- Use the Lock interface and ReentrantLock class

```
myLock.lock();
try {
    // code to be protected
} finally {
    myLock.unlock(); // must be in finally!
}
```

Locks

- Locks can provide similar functionality to synchronized, and they have internal wait/notify mechanisms built in.
- They also have additional benefits:
  - tryLock method (backs out if a lock is not available)
  - lockInterruptibly (backs out if interrupted while waiting)
  - Better performance under high *thread contention*
  - Can include multiple condition variables
  - Can split locking across methods (e.g., get a lock in one method and release it in another)

Practice

- Review the example of the BankAccount with fee and bonus, using locks instead of synchronized
  - Classes: LBankAccount, LBankThread, LBankTester

Conditions

- You can add multiple conditions to a lock.
  - This can replicate wait/notify, but is more flexible since you can have multiple conditions.

```
private Condition myCondition;

myCondition = myLock.newCondition();

myCondition.await();
myCondition.signal(); // or signalAll()
```

Locks and Conditions

```
private Lock myLock = new ReentrantLock();
private Condition myCondition = myLock.newCondition();

myLock.lock();
try {
    while (!someConditionThatMustBeTrue) {
      myCondition.await ();
    }
    // do the work that requires the condition and lock
} catch (InterruptedException e) {}
} finally {
    myLock.unlock();
}
```

```
myLock.lock();
try {
    change someConditionThatMustBeTrue;
    myCondition.signalAll();
} finally {
  myLock.unlock();
}
```

Practice

- Review the BankAccount deposit/bonus example that uses locks and conditions.
  - A bonus is given only when the balance is over $5000.
  - Classes: LCBankAccount, LCBankThread, LCBankTester

Practice

- Write a program to simulate the one-lane bridge problem using locks and conditions.

- There is a one-lane bridge where a car can only cross in one direction at a time.

- Write a simple bridge that doesn't consider the cars' directions.
  - Any car can go next, just one car at a time.
- Write a bridge that does take direction into .
  - The direction of the current car is given priority for who gets to go next.

EXECUTORS AND THREAD POOLS

Executors

- A downside of low-level thread code is that it links our task to our task-execution policy (e.g,. create one thread for every task).
- Also, it is **expensive** to create new threads.
- *Executors* allow you to run multiple threads without manually managing them.
  - They also re-use threads, rather than always creating new ones.

Thread Pools

- Thread pools are groups of threads, known as *worker threads*.
  - Worker threads sit idle and ready to run.
- Using a thread pool can reduce the expense of creating threads.
  - Important for large-scale applications
- Tasks are submitted to the pool and queued up if all threads in the pool are busy.
- Example:
  - Consider a website that can handle 10 requests at a time. (Hope they don't get too popular!)
  - Without a thread pool, when the 11^{th} request comes in, an 11^{th} thread is created, and the system then freezes- for **all** threads.
  - With a pool, the 11^{th} task is queued up and waits for one of the 10 threads to be free.

Using Executors

- ExecutorService object created through factory method of the Executors class:

```
ExecutorService executor =

        Executors.newSingleThreadExecutor();
        // all tasks execute on a single thread

        or
        Executors.newFixedThreadPool(poolSize);
        // all tasks execute on pool of threads

        or
        Executors.newCachedThreadPool();
        // all tasks execute on pool of threads
```

Using Executors

- Execute the executor

```
executor.execute(Runnable r);
executor.execute( () -> runnable code );
```

  - You can call execute multiple times for repeated tasks.
- You can also send in an object of type Callable<V>
  - Callable is analogous to Runnable, but it returns a value!
- Stop the executor (must do!!):

```
executor.shutdown();
```

Practice

- Review the dice example to compare low-level thread control to executor control.
  - Class: DiceTester

IMMUTABLE OBJECTS

Immutable Objects

- An object is *immutable* if its state cannot change once it is constructed.
- Example- the String class
  - String s = “Jessica”;
  - s.toUpperCase(); // does not change s!
  - String s2 = s.toUpperCase();
  - s = s.toUpperCase();

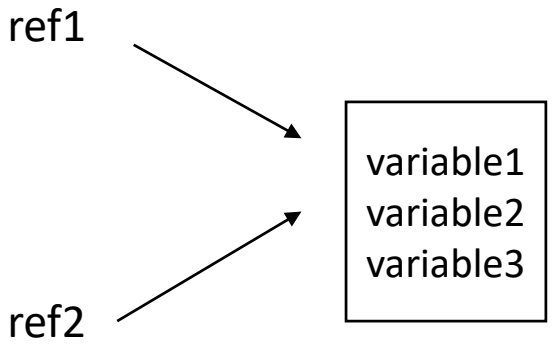
Immutable Objects

- An object is *immutable* if its state cannot change once it is constructed.
- Immutable objects are great for use in multi-threaded programs.
- Because their state cannot change, you don’t have to be worried about corruption through thread interference or inconsistent states.
- Immutable objects are thread-safe and can be used outside of synchronized blocks.

Objects in Memory- Modifying an Object

- NOT POSSIBLE with an immutable object!

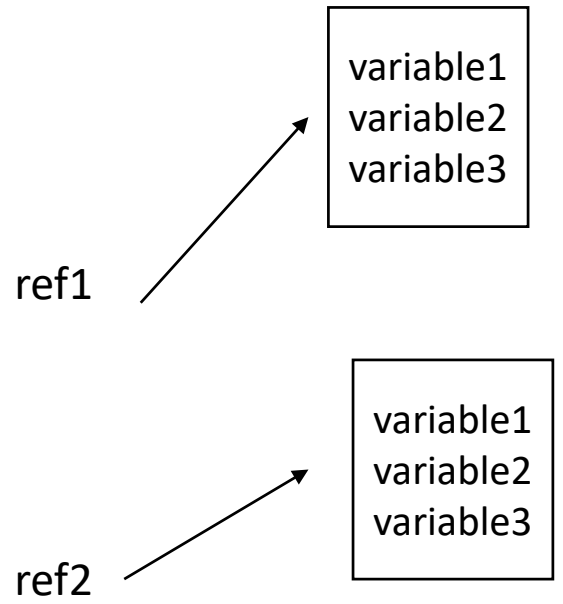
```
MyObject ref1 = new MyObject(…);
MyObject ref2 = ref1;

ref1.setVariable2(…);
 // not allowed if MyObject is immutable!
```



Objects in Memory- Changing a *Reference*

- Still allowed with an immutable object! You are not changing an object!

```
MyObject ref1 = new MyObject(…);
MyObject ref2 = ref1;
ref1 = new MyObject(…);
```



Objects in Memory

- Invoking a method with the dot operator most likely changes the object itself
  - The state or characteristic of the actual object changes.
- Assigning a new value wchanges what the variable/reference points to
- ith the equals sign just This “changes the memory arrows” but does not change the actual object

How to Create Immutable Classes

- Make the class final
  - Or make the constructor private and provide static factory method
- No setter methods
- Make all instance data “variables” final and private
- For instance variables that are **mutable** objects:
  - Ensure there are no methods that would modify those objects
  - Do not share references to those objects!
    - When passed an object, make a copy before storing it as instance data
      - Example: in a constructor!
    - When returning an object, make a copy to return
      - Example: in getter methods!

Practice

- Review the EmployeeIM example.

COLLECTIONS

Multiple Threads and Collections

- It is easy to corrupt a data structure if accessed by multiple threads.
- You should not use a collection with no thread support in a concurrent program.
- Instead, use a:
  - Synchronized Collection
  - Concurrent Collection

Practice

- Look at an example of filling and emptying a list with threads using a normal non-thread-safe collection.
  - Classes: ListAdder, ListRemover, ConcurrentCollectionsTest

Synchronized Collection Classes

- All **methods** in the class are synchronized.
- Classes
  - Vector
  - Hashtable (**not** HashMap or TreeMap)
- Create your own (preferred to above):
  - Collections.synchronizedList(myList)
  - Collections.synchronizedMap(myMap)
  - Collections.synchronizedSet(mySet)

Practice

- Re-run the add/remove list example with a synchronized list.

Thread Safety- Important!!

- Using a synchronized collection assures no corruption from invoking a method.
- But it doesn't mean you're off the hook for thread safety overall!
- Combinations of methods are not atomic!
- Example:

```
if(!testList.isEmpty())
   testList.remove(0);
```

- Fix:

```
synchronized(testList) {
   if(!testList.isEmpty())
      testList.remove(0);
}
```

Practice

- Re-run the add/remove list example with a synchronized list AND thread-safe code.

Iterators

- A ConcurrentModificationException is thrown if one thread modifies a synchronized collection while an iterator is retrieving an item.
- You'd have to synchronize access to the block of code iterating to prevent this.
  - Costly!

Concurrent Collection Classes

- More sophisticated support
  - Thread safe and scalable!
- These classes allow concurrent access to different parts of a single data structure.
- Classes
  - ConcurrentHashMap
    - More efficient than Collections.sychronizedMap(myMap)
  - ConcurrentLinkedQueue

Iterators

- The concurrent collection classes return *weakly consistent iterators*
- These iterators will **not** throw a ConcurrentModificationException
- The iterator may or may not reflect all modifications made since construction
  - It will **not** return an item twice
  - If an element was removed before iteration starts, it won't be returned
  - If an element is added after iteration starts, it may or may not be returned

ConcurrentHashMap

- Retrieval operations do not block
- Supports multiple concurrent write operations
- Use traditional Map methods, plus atomic versions of compound operations:
  - putIfAbsent(key, value)
  - compute(key, BiFunction<K, V> remapFunction)
    - BiFunction: parameter K and V, returns V
  - computeIfPresent(key, BiFunction)
  - computeIfAbsent(key, BiFunction)

ConcurrentHashMap

- Bulk operations
  - search- search elements
  - reduce- accumulate elements
  - forEach- perform an action on each element
- Each can be applied to Keys, Values, both, or Map.Entry
- Each takes a parallelismThreshold and a function or bi function
  - Threshold: if the map has more elements than the threshold, it will be parallelized

Thread Safety- Important!!

- Again, using a concurrent collection only assures no corruption from invoking a method.
- But it doesn't mean you're off the hook for thread safety overall!
- Combinations of methods are still not atomic!

Practice

- Review the scrabble words example.
  - Make a map of the number of words that start with each letter.
  - Make a map of the sum of the score of words that start with each letter.
    - Classse: LetterStartThread, LetterStartThreadSafe, WordScoreSumThreadSafe, ConcurrentMapTestingScrabble