Concurrency

- How to Create Thread - Runnable - Method 1
- How to Create Thread - Method 2
- How to Catch any Error from Thread
- How to Interrupt a Thread
- How to run a thread and still end the application
- How To Make a Thread wait and go in Order
- What Lives in Heap ? What is shared between threads ?
- What Lives in Stack
- Synchronized
- Atomic ?
- Volatile ?
- Data Race vs Race Condition?
- Dead Lock
- Right Remedy for Deadlocks ?
- ReEntrant Lock
    - Lock Interruptibility Method
    - Lock Interruptibility Use Cases:
    - Try Lock Method
    - TryLock method use cases
- REVELATION
- ReEntrant Read Write lock
    - How to Create a ReEntrant Read Write Lock
    - What is mutual Exclusion and What's so special about this lock
    - Use Case :
- What is Semaphore ?
    - What is Binary Semaphore ?
    - Use Case:
- Condition and AWAIT() and SIGNAL()/SIGNALALL()
- Wait - Notify - NotifyAll
- Lock Free Operations
    - What must be our goal to become LockFreeness?
    - CompareAndSet method
    - AtomicReference

How to Create Thread - Runnable - Method 1

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
        System.out.println("Vinodh");
    }
});

thread.start();
```

- Instantiating Thread Class alone will not suffice
- It needs to be a given a Runnable Implementation and the `start` method does what we need
- the magic happens when `.start()` is called

How to Create Thread - Method 2

- Extend any class with Thread class - `Thread class implements Runnable Interface`
- Override the Run method in there

- Then wherever you want call the thread.run method

```
public static void main(String[] args) {
    ThreadClass threadMain = new ThreadClass();
    threadMain.start();
}

public class ThreadClass extends Thread {
    @Override
    public void run() {
        super.start();
        System.out.println("From here");
    }
}
```

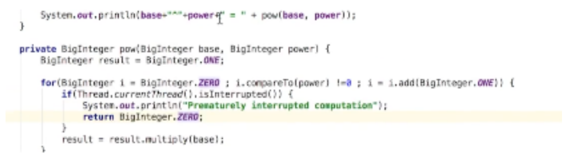
How to Catch any Error from Thread

```
thread.setUncaughtException // This will catch any error from within the thread that was not caught and can be used for analysis
```

How to Interrupt a Thread

First lets talk about why,

- A thread is running for a long time and we want to interrupt ourselves we can `thread.interrupt` on the thread and if the thread has caught the InterruptedException and we break the thread.
- Sometime when we start a thread we know few hotspots in which the the code will be long running at certain loops or so , so in those places we need to run a if clause to check if a interrupt has been issued from the external world and end the thread.



```
    System.out.println(base+"^"+power+" = " + pow(base, power));
}

private BigInteger pow(BigInteger base, BigInteger power) {
    BigInteger result = BigInteger.ONE;

    for(BigInteger i = BigInteger.ZERO ; i.compareTo(power) !=0 ; i = i.add(BigInteger.ONE)) {
        if(Thread.currentThread().isInterrupted()) {
            System.out.println("Prematurely interrupted computation");
            return BigInteger.ZERO;
        }
        result = result.multiply(base);
    }
```

```
mythreadName.interrupt();

- 2 ways to catch
  - Thread.currentThread.isInterrupted and return
  - Always  try catch InterruptException and return
```

How to run a thread and still end the application

In this scenario we make the thread daemon and set the property as true. Think about the potential it has with specific use cases.

How To Make a Thread wait and go in Order

We need to use `.join()` method or `.join(2000)` (Next thread will wait only for 2 seconds). This join call will make the thread fall in line and will execute and keep going one by one. It is for making thread go in an order. Example Thread A can be sending input to Thread B so we need to join ThreadA and B like threadA.join() and then threadB.join(). So that the B will start only when A is done

What Lives in Heap ? What is shared between threads ?

First thing to know is that all the items in the Heap are shared between threads,

- Objects - Thats why we are able to get the value out of recursive functions as these are shared

- Member variables - Doesn't matter if they are primitives but they are still shared
- Static variables

What Lives in Stack

- Local variables
- Local references

No surprise here

Synchronized

- This is applied per object
    - When multiple threads are going to try to call these `SYNCHRONIZED` methods on the same object of this class only one thread would be able to execute either of those methods

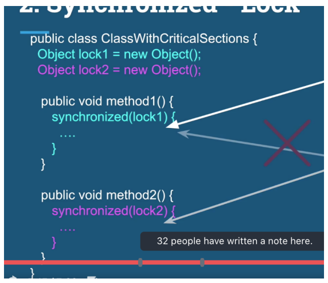
  So even though you apply synchronized to a method the whole Object is basically locked for any other Synchronized method execution. Which means,

    - Not only you are preventing other threads from executing the method you are executing but also the other synchronized methods of the object. (notice , we didn't say class we said object). Because this happens only when two threads work on same object.

We can protect a method from Thread Execution in 2 ways,

- making the method `synchronized`
- or adding a `synchronized(this)` block and placing the code that needs to be thread safe. ? Remember Singleton Pattern ?

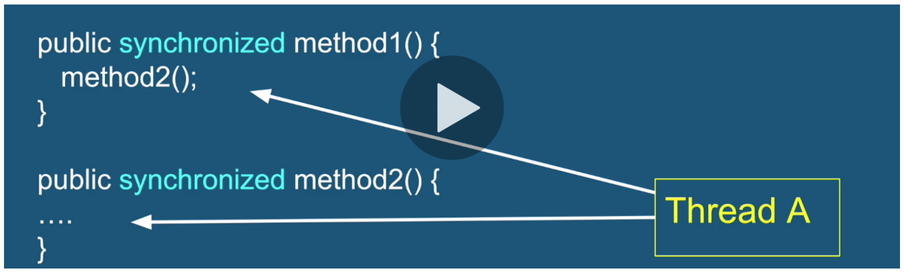
HACK : Instead of THIS keyword you can also place a dummy object name within the synchronized block and make two threads access different blocks at the same time.



```
public class ClassWithCriticalSections {
  Object lock1 = new Object();
  Object lock2 = new Object();

   public void method1() {
     synchronized(lock1) {
       ....
     }
   }

   public void method2() {
     synchronized(lock2) {
       ....
     }
   }
}
```

Catch :

When Synchronized is placed, 2 threads acting on same object cannot call any synchronized method but if the executing thread in a synchronized method can call another synchronized method without any problem. RE_ENTRANT LOCK.



```
public synchronized method1() {
   method2();
}

public synchronized method2() {
….
}
```

Atomic ?

What operations are thread safe.

- All getters and setter of member variables
- All primitives except **long and double**

Volatile ?

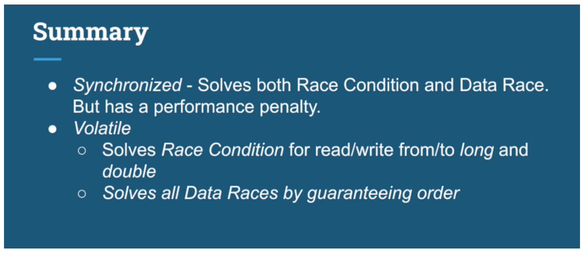
data types long and double store 64 bit data so 2 operations are needed so when we add volatile key word they become threadsafe aka atomic

VOLATILE KEYWORD - makes a variable THREAD SAFE and ATOMIC

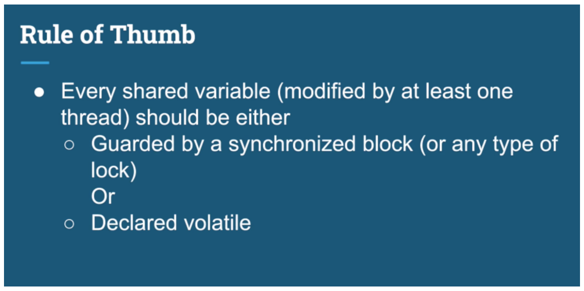
Data Race vs Race Condition?

The compiler sometimes rearranges the instructions in such a way that 2 independent instructions can be executed in any order. But if those instructions are shared variable incrementing function for example then we might land a wrong answer. This might lead to data race

```
The below condition is called Data Race
class DataRaceExample {

long x;
long y;

public void someMethod() {
x++;
y++;
}

}

in the above case if our logic is based on the values of x and y , we must not assume that at any point during thread intervention x

Remedy always make shared resources VOLATILE

class DataRaceExample {

volatile long x;
volatile long y;

public void someMethod() {
x++;
y++;
}

}
```


Summary

- *Synchronized* - Solves both Race Condition and Data Race. But has a performance penalty.
- *Volatile*
  - Solves *Race Condition* for read/write from/to *long* and *double*
  - *Solves all Data Races by guaranteeing order*


Rule of Thumb

- Every shared variable (modified by at least one thread) should be either
  - Guarded by a synchronized block (or any type of lock)
    Or
  - Declared volatile

Dead Lock

This is the scenario we an simulate this,

- Create a class
- Add 2 member variables
- So now these variables become shared resource for any thread on this object as they sit on the heap.
- Create a method inside a class named `WowDeadLock`
  - Make a sync block within the method (notice not the whole method itself only the block) using 1 member variable like `synchronized(member1)`
  - Then within then block acquire a lock on the other member

```
public class WowDeadLock {


public boolean something() {

synchronized(member1) {
-------
-------
synchronized(member2) {
-------
--------
}
--------
-------
}


}
```

  - Now create another method in the same class and do the opposite

```
public class WowDeadLock {


public boolean somethingOppo() {

synchronized(member2) {
```

```
-------
-------
synchronized(member1) {
-------
--------
}
--------
-------
}

}
```

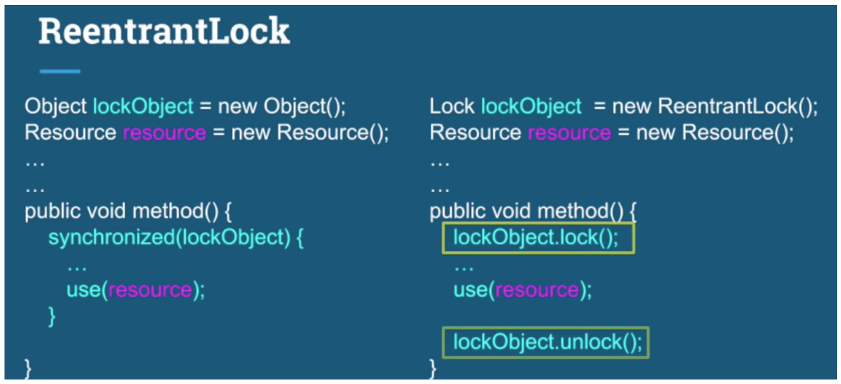
- Now create other different classes `Dummy1` and `Dummy2` with `Thread` interface and make sure that both of them constructor inject `WowDeadlock` . Inside Dummy's `RUN` call `something` and `somethingOppo` respectively.
- In the main class, create just 1 `WowDeadlock` and inject Dummy1 and 2 objects and start the thread
- We have the Deadlock

Right Remedy for Deadlocks ?

- Always acquire lock in a specific order and maintain the same order within the class.
- In the above example we an always maintain `member1` followed by `member2` lock always

ReEntrant Lock

- You need to manually lock and unlock the object
  - You cannot just create a dummy object instead you need to create something called as `ReentrantLock` Object.



ReentrantLock

```
Object lockObject = new Object();
Resource resource = new Resource();
...
...
public void method() {
   synchronized(lockObject) {
      ...
      use(resource);
   }

}
```

```
Lock lockObject  = new ReentrantLock();
Resource resource = new Resource();
...
...
public void method() {
   lockObject.lock();
   ...
   use(resource);

   lockObject.unlock();
}
```

- Caution when using this LOCK interface always add `finally` block

Lock Interruptibility Method

- So you have a resource and using `Reentrant` Lock you locked the critical portion and a resource is being used within it.
- Now another thread comes in for executing the critical section but it needs to wait/sleep, so instead of wasting and avoiding wait time we can use the `lockObject.lockInterruptibiliy()` instead of just `lockObject.lock()`
- This method usage will force you to throw InterruptException and you can call `cleanUpAndExit` and exit when the resource is busy/locked
- So the interrupted thread happens when interrupt method is called on the thread,
  - when just `.lock()` is used nothing changes but
  - when `.lockInterruptibily` is called we can clean and exit

Lock Interruptibility Use Cases:

- When closing app call interrupt method on the idle thread and close all waiting threads gracefully
- Implement a WATCH-DOG algo - which detects DEADLOCK and recovers it

Try Lock Method

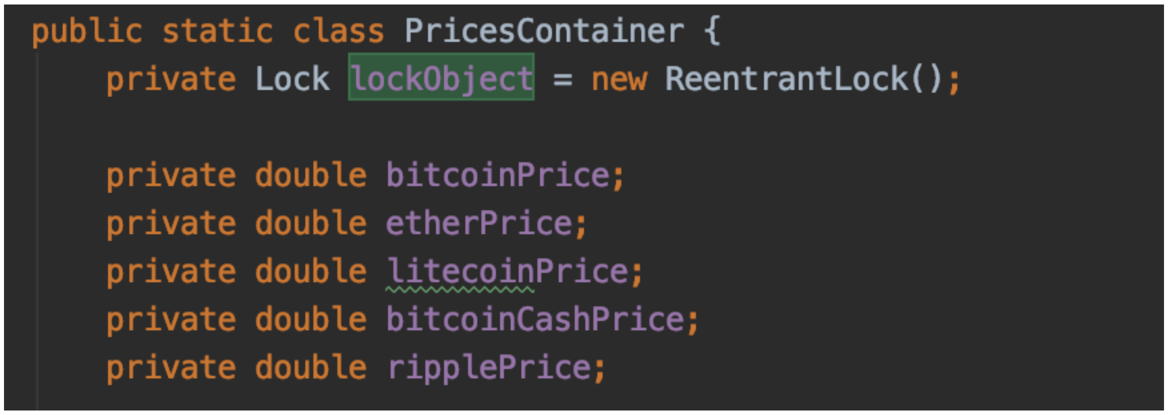
- Use this method instead of just `lock()`
- So what will happen here
  - If not for this method, the thread will check if the block is locked
  - It will then go to sleep until the existing lock releases the lock
  - So we can use the `.interrupt` method on the thread and wake it up when using the `.interruptibility` method and use the thread for more useful purpose instead of wasting resources
  - Even better when we use the `tryLock()` method, we check if the block is locked and then if true continue with something else, its more automated than `interruptibility` method

TryLock method use cases

- Never waste resources kind of business
  - Trading
  - Banking

REVELATION

WHEN YOU SHIP AN OBJECT AND IF YOU WANT IT TO BE THREAD SAFE , SHIP it WITH A LOCK (Especially REENTRANT LOCK.



```
public static class PricesContainer {
    private Lock lockObject = new ReentrantLock();

    private double bitcoinPrice;
    private double etherPrice;
    private double litecoinPrice;
    private double bitcoinCashPrice;
    private double ripplePrice;
```

ReEntrant Read Write lock

What if a critical section of code is often read and not modified that often ??

- So we can allow multiple threads to read from shared resource

How to Create a ReEntrant Read Write Lock

```
ReentrantReadWriteLock obj = new ReentrantReadWriteLock();

// Create a Read Lock
Lock readLock = obj.readLock();j

// Create a Write Lock
Lock writeLock = obj.writeLock();
```

We can use these locks in the place where we want to lock critical conditions just like any other lock.

What is mutual Exclusion and What's so special about this lock

- If the code is covered within a read lock , `multiple threads are allowed` but no thread is allowed to enter a code block guarded by `writeLock` .
- Like wise, `Only one thread can enter` a write lock protected code , and during that time no thread is allowed into a block protected by `readLock`

Now thats called Mutual Exclusion. You can't write when you read and you can't read when you write. Thats all is needed to understand this.

Use Case :

- Use this when you have `heavy read and minimal writes`.

What is Semaphore ?

First of all Semaphore is not a great replacement for a lock.

Remember Semaphore as an option for Inter Thread Communication

- Semaphore is more of a Authority to issue a lock to a thread that is requesting it.
- When creating a semaphore we can specify the number of lock it can issue.
- Then Semaphore keeps track of the issued locks and can block threads if it doesn't have any more locks.
- You can consider created with only 1 lock to issue to any other lock we have learnt till now
- Useful APIs are
  - `acquire(SOME_NUMBER)`
  - `release(SOME_NUMBER)`

What is Binary Semaphore ?

Its more of a technique, this is how it goes

- You create 2 semaphores
  - 1 with 0 (yes it cannot issue any lock for any thread) - Lets call it `ZERO`
  - 1 with just 1 token - Lets call this `UNO`
- Imagine a Pub-Sub model with a shared resource named `item`
- Pub will keep producing altering the item
- Sub will be processing the item
- We need to make sure that there is not dirty read.
- Start Sub with a while loop
  - Acquire `ZERO` and it has `0` semaphores
  - So now that thread goes to sleep

    - Once it wakes up it will process the `item` (shared resource)
- Meanwhile Start the Pub while loop
    - Acquire `UNO` , so now no other thread can process the PUB
    - Create the `item`
- The MAGIC
    - once that is done , `here is the twist` → Release `ZERO` (So PUB has released a SUB thread's lock)
    - So now ZERO will go ahead and process the shared resource `item`
    - Once ZERO's thread is done it will release `UNO` and this goes on without any dirty issues.
    - Hence this is called Binary Semaphore

```
Semaphore ZERO = new Semaphore(0);
Semaphore UNO = new Semaphore(1);
Item sharedItem = null;

//SUB
while(true) {
  ZERO.acquire(); // Thread goes to sleep instantly
  doSomethingWithThisSharedItem(sharedItem);
  UNO.release();
}

// PUB
while(true) {
  UNO.acquire(); // The available 1 lock is given and no more threads can enter
  sharedItem = produceThatItem();
  ZERO.release(); // Wake the sleeping SUB Thread
}
```

Use Case:

- The above example can be modified from just 1 shared item to a queue of shared item and scale it to multiple PUB-SUB
- Semaphore is a good fit for PUB-SUB models and `Inter Thread Communication`

Condition and AWAIT() and SIGNAL()/SIGNALALL()

So if you want to communicate between threads `condition` is an important tool. Here is how it goes,

- Create the following,
    - A Reentrant Lock
    - A Condition object on the lock - yes thats a thing

    ```
    Lock reentLock = new ReentrantLock();
    Condition condition = lock.newCondition();
    ```

    - Now imagine a thread,
        - Acquire a LOCK
        - start something and check if the condition you are looking for is met.
            - Here you have 2 cases,
                - Yes its met → Then just proceed then release the lock
                - No its not met → call the `condition.await()`

- So what just happened, what can the await do ?
    - `Await` does 2 things here
        - `release the lock` its holding
        - `makes the thread to sleep`
- Another thread which can make this condition true by executing a complete different set of code gets this lock, does its thing and when it believes it is done will call `.signal()` method
- So what just happened, what can the signal do ?
    - `Signal` can
        - Wake up the thread we put to sleep before
            - Checks if the condition we are looking for is met ?
                - Yes → continue its execution and explicitly release the lock
                - No → Again call the AWAIT method and implicitly release the lock (because await method method will release the lock implicitly)

```
Lock reentLock = new ReentrantLock();
Condition condition = lock.newCondition();

// For this example assume that a someMethod() would definitely be called before someOtherMethod() so that the signal() actually

someMethod() {
  // AWAIT EXAMPLE
  reentLock.lock();
  try{
    while(sharedVariable1.val == null || sharedVariable12.val) {
      condition.await(); // Thread goes to sleep but the lock is released
    }
  } finally {
    reentLock.unlock();
  }
}

someOtherMethod() {
// SIGNAL EXAMPLE
  reentLock.lock();
  try{
    sharedVariable1.val = "NOT NULL ANYMORE";
    sharedVariable12.val = " NOT NULL DUDE";
    condition.signal(); // now we wake up the above block but we release first
  // using the below finally block
  } finally {
    reentLock.unlock();
  }
}
```

SignalAll() - is another useful API for broadcasting, always use this ??

Wait - Notify - NotifyAll

Another bunch of APIs to make inter thread communication possible.

Every class in Java extends Object class by default and that class has the following 3 methods,

- wait()
- notify()
- notifyAll()

So we can say that all objects in java has these 3 methods.

This gives a possibility of making any object as CONDITION as well as LOCK - a combo compared to CONDITION+REENTRANT lock as seen before.

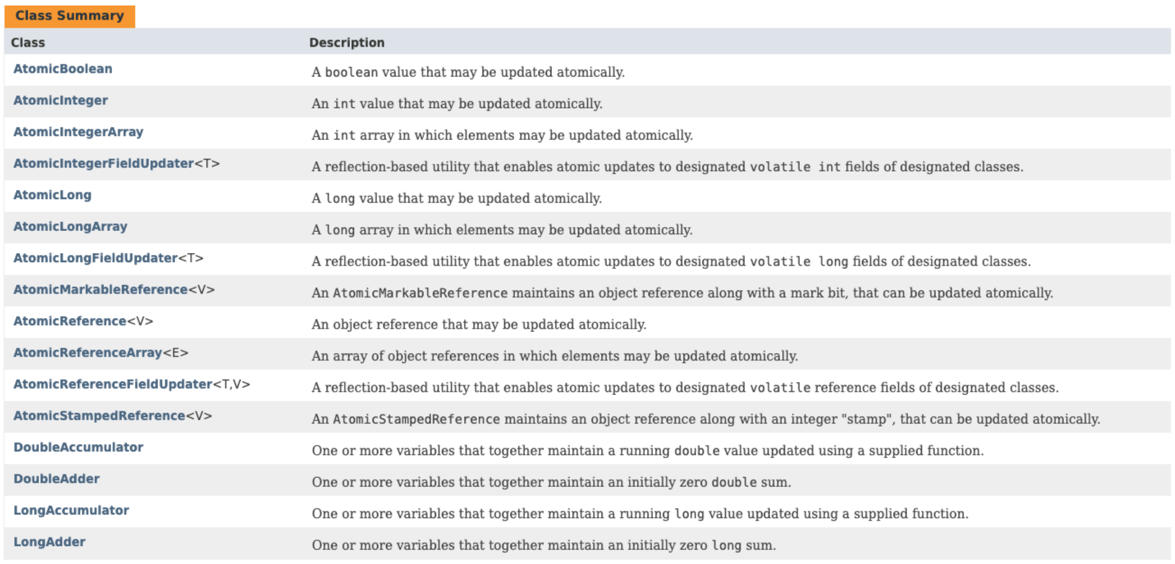
```
// Sample using COMBO lock and a condition
Lock reentLock = new ReentrantLock();
Condition condition = lock.newCondition();

someMethod() {
  -----
  -----
  reentLock.lock();
  try {
    if(!somebooleanexpression is not met) {
      condition.await();
    }
  } finally {
    reentLock.lock();
  }

  -----
  -----

}

// Sample using Sync

someMethod() {
  -----
  -----
  synchronized(this) { // lock on the object which is the shared resource

      if(checkIfConditionIsMet) {
        this.wait(); // puts the thread to sleep and releases the lock just like await
      }

  }

}

someOtherMethod() {
  -----
  -----
  synchronized(this) { // lock on the object which is the shared resource

      checkIfConditionIsMet = true;
      this.notify(); // Invokes one of the thread that is sleeping because of wait
      this.notifyAll(); // Invokes all the threads that are sleeping because of wait

  }

}
```

Lock Free Operations

Forget all the above then ? No you still need them but the following will kind of save your time and complex code. Or consider them as additional tools in your bet.

What must be our goal to become LockFreeness?

- Make sure that all our operations are atomic. Which means that `the thing we want need to be done in one hardware operation`
- i++ involves
  - Get the value of i
  - Cal new val
  - Store new val in i
- So as already learnt this,
  - Make the variable `VOLATILE`
  - Also look at `java.util.concurrent.atomic` and try use them



Class Summary

Class	Description
AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
AtomicStampedReference<V>	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.
LongAccumulator	One or more variables that together maintain a running long value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero long sum.

CompareAndSet method

All Atomic classes comes with a `compareAndSet` method that takes 2 params. So when setting a value of a shared resource, it does 2 things.

- It will check if the current value of the reference you are trying to change is the expected value and only then proceed to set the new value

```
private static AtomicReference<Integer> count = new AtomicReference<>(0);

   public static void main(String[] args) {
       count.compareAndSet(0, 2); // 2
       log.info("count:{}", count.get());
       count.compareAndSet(0, 1); // no
       log.info("count:{}", count.get());
       count.compareAndSet(1, 3); // no
       log.info("count:{}", count.get());
       count.compareAndSet(2, 4); // 4
       log.info("count:{}", count.get());
       count.compareAndSet(3, 5); // no
       log.info("count:{}", count.get());
       count.compareAndSet(5, 6); // no
       log.info("count:{}", count.get());
       count.compareAndSet(4, 7); // 7
       log.info("count:{}", count.get());
       count.compareAndSet(6, 8); // no
       log.info("count:{}", count.get());
       count.compareAndSet(7, 10); // 8
       log.info("count:{}", count.get());
       count.compareAndSet(9, 13); // no
       log.info("count:{}", count.get());
       count.compareAndSet(11, 13); // no
       log.info("count:{}", count.get());
       count.compareAndSet(12, 13); // no
       log.info("count:{}", count.get());
       count.compareAndSet(10, 13); // 13
       log.info("count:{}", count.get());
   }
```

All you need to remember is , if you want to go Lock free use the Atomic package classes and make sure. This makes sure that all multiple hardware operations are bundled into 1 machine instruction, there by avoiding dirty read/writes by thread on an invalid state.

AtomicReference

If you want to make any of your shared resource operation atomic, make it a AtomicReference and do the thing.

```
Employee emp; // shared resource

SomeConstructor(Employee emp) {
  this.emp = emp;
}

AtomicReference<Employee> empAtomic; // atomic shared resource

SomeConstructor(AtomicReference<Employee> empAtomic) {
  this.empAtomic = empAtomic;
}

Employee fromAtomic = empAtomic.get();
```