Java Multithreading, Concurrency & Performance Optimization

- Udemy https://www.udemy.com/course/java-multithreading-concurrency-performance-optimization/learn/lecture/10187964#overview
- Github https://github.com/jgregorio0/java-multithreading

Introduction

Why to use multithreading?

Responsiveness / Concurrency

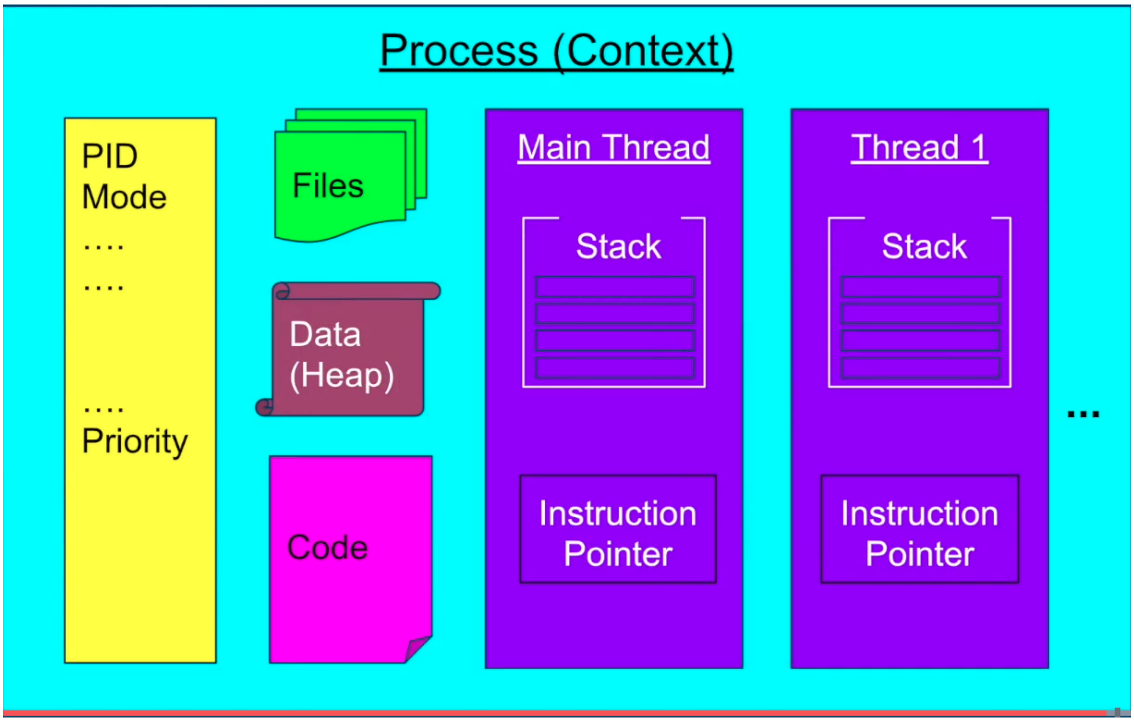
- Serve multiple users simultaneously
- Critical in user interface, ie: video player
- One core creates illusion of multiple tasks in parallel

Performance / Parallelism

- Multiple cores run tasks in parallel
- Higher performance >> more work in same time and less machines

Threads structure in OS

1. Users run an app >> OS create an instance (process / context) of the app from HD to memory which is completely isolated from the other processes
2. Each **process** contains
    a. metadata (PID, prority, mode,...)
    b. files
    c. code
    d. heap (data for our app)
    e. One (Main Thread) or more threads
3. **Thead** contains
    a. Stack: local variables are stored
    b. Instruction pointer: address of the next instruction
4. All but the stack and instruction pointer is shared



Context switch

- When switching between threads we need to
  - stop a current thread execution
  - schedule it out
  - schedule new thread in
  - start new thread

Take into account that:

- **Thrashing** happends when
  - management thread > productive work
- Threads resources < process resources
  - The costs of changing thread from the same resources < threads from diff process

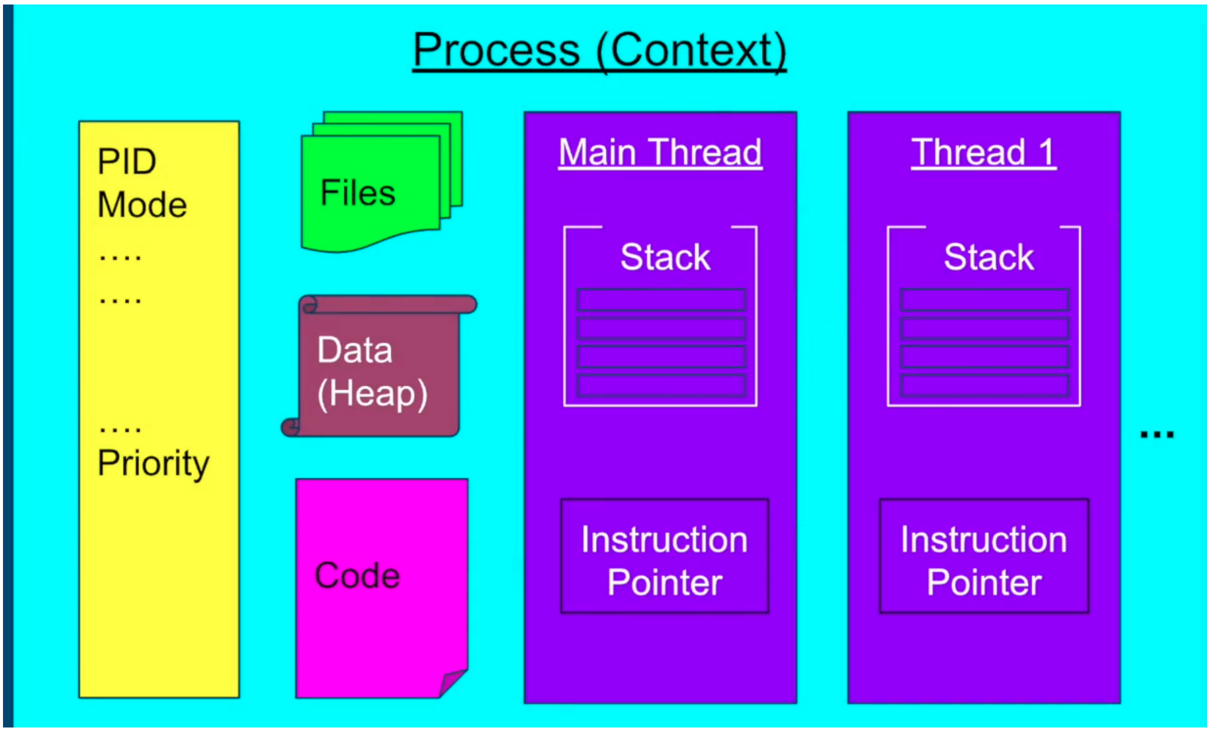
Thread scheduling

- Who run first?
  - Long threads can cause **starvation** over short threads
    - Short threads can cause long threads are never executed
  - Solving starvation by dividing time into **Epochs** and adding bonus to threads did not complete in the previous epoch
    - Epoch (time slice) in which a thread uses CPU
    - dynamic priority = static priority + Bonus
      - static priority
        - UI will be executed first improving UX
      - Bonus

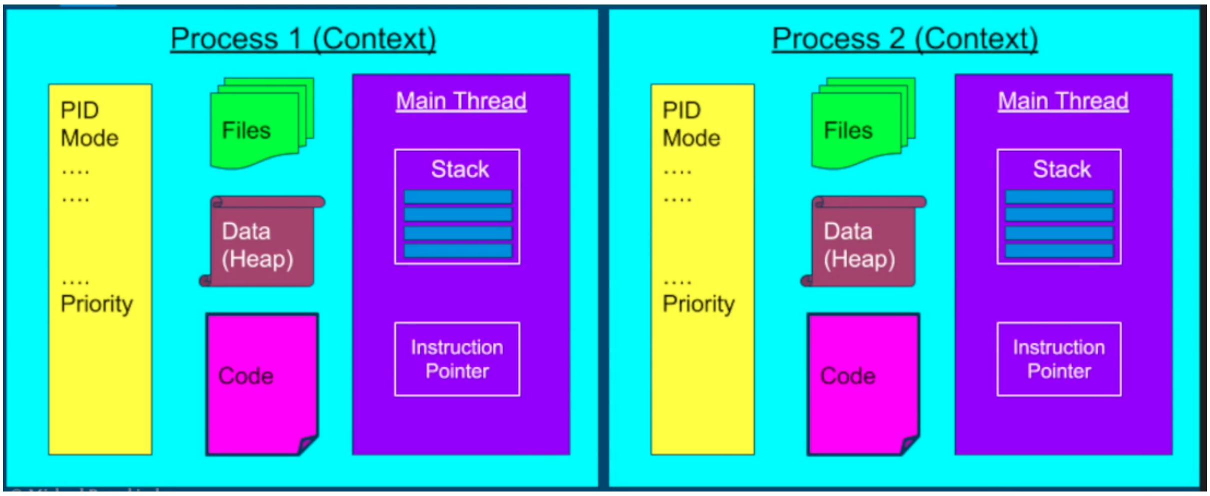
    - Icrease priority for threads that did not complete in last epoch prevents starvation

Multithreaded vs Multi Processes

- Multi Threads
    - share a lot of data
    - Threads are faster to create and destroy
    - Shorter context switches



- Multi Processes
    - Security and stability is higher
    - Task are unrealated



Thread fundamentals

Create thread implementing Runnable

```
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("We are in thread " +
Thread.currentThread().getName());
        });
        t.start();
    }
}
```

- setName
- setPriority
    - Thread.MAX_PRIORITY
- Thread.sleep(milliseconds)
- setUncaughtExceptionHandler

Create thread extending Thread

```
public class SimpleThread2 {
    public static void main(String[] args) throws InterruptedException {
        Thread rt = new RunableThread();
        rt.start();
    }

    static class RunableThread extends Thread {
        @Override
        public void run() {
            System.out.println("We are in thread " +
Thread.currentThread().getName());
        }
    }
}
```

Interrupt thread

- Threads consume resources
    - Memory
    - kernel resources
    - CPU
    - cache
- Thread is finished but app is still running >> Clean up threads' resources
- Thread is misbehaving
- App will not stop if one thread is still running >> close all threads gracefully

InterruptedException

- InterruptedException: Thread executes a method that throws InterruptedException

```
private static class BlockingTask implements Runnable {
   @Override
   public void run() {
       try {
           Thread.sleep(50000);

       } catch (InterruptedException e) {

           System.out.println("Exiting blocking task");
       }
   }
}
```

isInterrupted

- isInterrupted: Theads code is handling interrupt signal

```
@Override
public void run() {
   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);
   }
   return result;
}
```

Daemon threads

- Background tasks that should not block our app
    - File saving
- Code could not listen to interrupt
    - External library

setDaemon

- **setDaemon(true)**

    - Thread will be **finished** even if not throwing **InterruptedException** or checking **isInterrupted**

Thread coordination

Join

- join method sleep thread A until thread B is finished
    - timeout parameter throws InterruptedException after x miliseconds
-

Performance optimization

- Latency: Time to complete a task. Time units
- Throughput: Amount of task in a period. Task/time units

Latency

- Latency = T/N
    - T time to execute original task
    - N number of subtasks
- Dividing a task into N tasks to run in parallel
    - N = number of cores
    - each core running 1 thread
    - if threads are runable without interruption (IO blocking, sleep...)
    - No other tasks are consuming CPU
    - Hyperthreading: virtual cores share hardware
- Cost of parallelization
    - Breaking task into multiple tasks
    - Thread creation and passing task to threads
    - Time to schedule a task
    - Time until last task finishes and signals
    - Time until aggregator taks runs
    - Aggreegation of the results
- It is not posible to divide a task always:
    - Parallelizable tasks
    - Sequential tasks
    - PArtially parallelizable / partially sequential

Throughput

- Throughput = N/T (rendimiento)
    - N number of subtasks
    - T time to execute original task
- Dividing tasks into N tasks
    - Latency = T/N
- Runing tasks in parallel

    - Each task in different thread
    - Improve throghput by N
    - N = Threads = Cores
- Thread pooling
    - Reusing threads minimize creation and schedule tasks
    - Executor.new**FixedThreadPool**

JMeter

Automate test performance

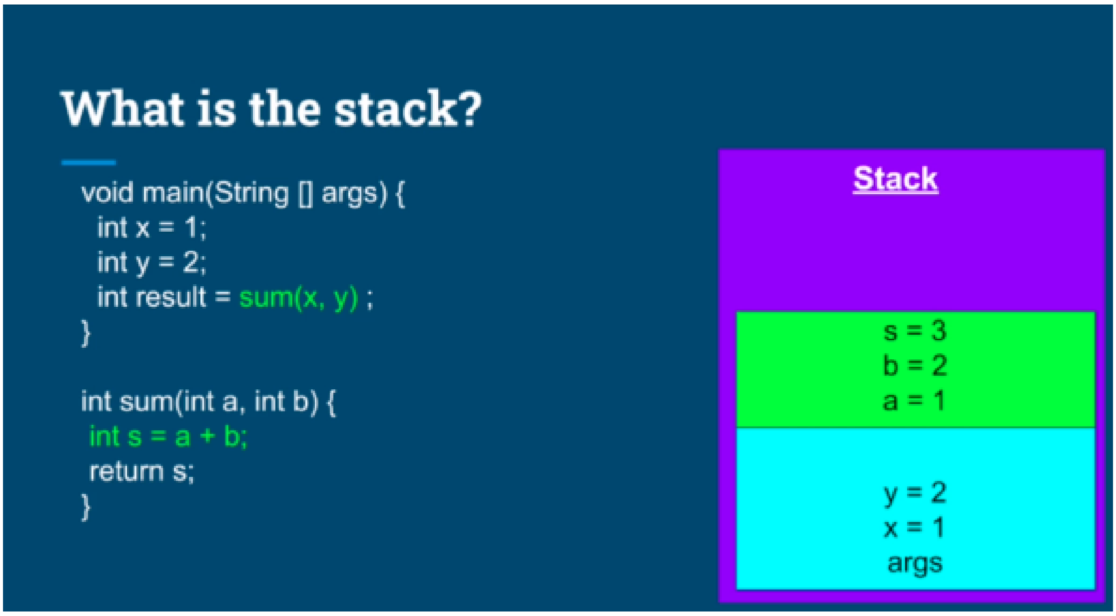
1. Add Thread Group
    a. 200 threads
2. Add Logic Controler / While Controller
3. Add Config Element / CSV Data Set Config
    a. Filename
        i. search_words.csv
    b. Variable Names
        i. WORD
    c. Delimeter
        i. \n
    d. Recycle on EOF
        i. false
    e. Stop thread on EOF
        i. true
4. While Controller Condition
    a. ${__javaScript("${WORD}" !=== "<EOF>")}
5. Add Sample / HTTP Request
    a. protocol
        i. http
    b. server
        i. localhost
    c. port
        i. 8000
    d. Method
        i. GET
    e. Path
        i. /search?word=${WORD}
6. Add listener / Summary Report
7. Add listener / View Results Tree

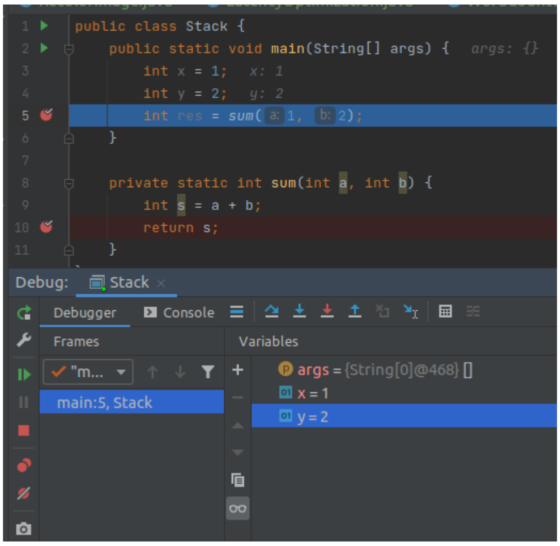
Thread Data Sharing

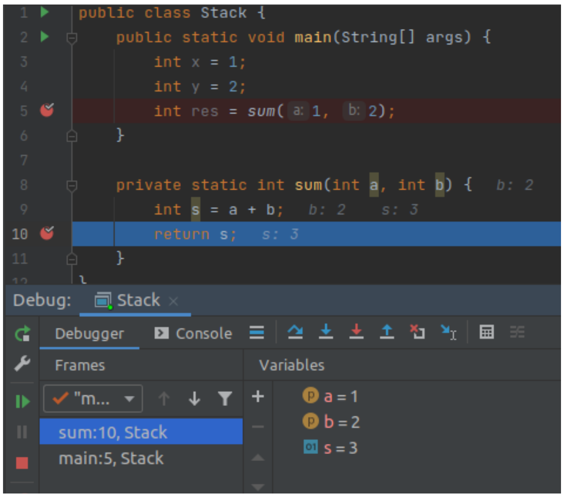
Stack

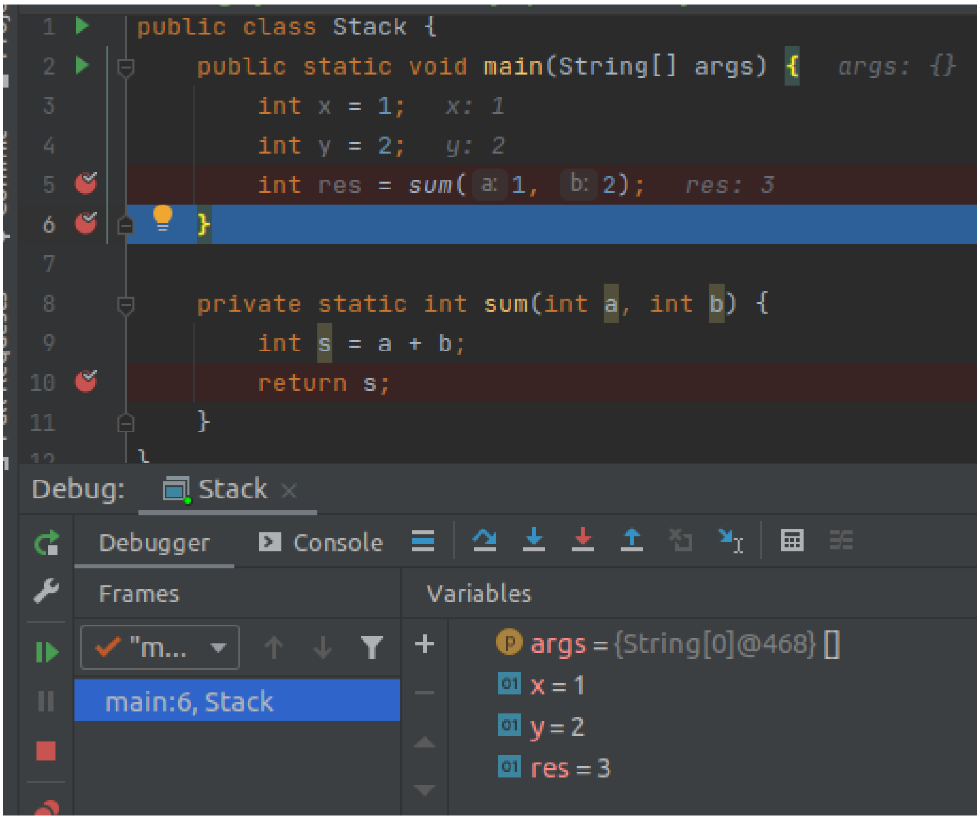
- Primitive types declared locally will be stored on the stack only
- Other threads have no access to the stack

- The following example shows how
    - First stack frame contains x and y
    - second stack frame contains a, b and s (return s)
    - second stack is invalidated and its result is allocated in first frame
    - First stack frame contains result of second frame and its invalidated when finished









Heap

- Static variables, objects and primitive types that objects contains will be stored on the heap
- Heap is shared between threads
- Governed by Garbage Collector
    - It should remove objects when there are no references to the objects
    - Static variables stay forever

References

- Reference is a pointer to an Object
- Local reference is allocated in the Stack
- but If they are member of a class they will be allocated in the Heap

Concurrency Solutions

Atomic operations

- Ocurred at once
- Single state, all or nothing, without intermediate states
- What is atomic?
    - Assignament to
        - References, including getters and setters
    - Assignament to
        - int
        - short
        - byte
        - float
        - char
        - boolean
        - ~~long (64 bits)~~
        - ~~double (64bits)~~
        - volatile long
        - volatile double
        - java.util.concurrent.atomic

Synchronized

- Lock mechanism
- Locks all synchronized methods of the object
- Use sychronized blocks instead of synchronized methods lock only the block instead of all methods

Data Race

- A shared resource
    - is accessed by multiple threads
    - is modified by at least one of those thread
- Timing of threads scheduling may cause incorrect results
- **Problem**: Non atomic operations performed
- **Solution:**
    - Using **synchronized** to atomize a critical section
    - Using **volatile** to atomize long and double assignaments
- **Example**:

Thread1 { x++; y++}
Thread2 { if (y > x) throws new Exception("Data Race detected!!"); }

Race condition

- A shared resource
  - is accessed by multiple threads
  - is modified by at least one of those thread
- Compiler & CPU may execute code out of order to increase performance and utilization maintaining logical correctness
- **Problem**: Reorder code in one thread results in unexpected behaviour for the other
- **Solution**:
  - Using **synchronized** to atomize a critical section
  - Using **volatile** to guarantee order on the previous and next instruction
- **Example**:

Thread1 { i++}
Thread2 { i-- }
Result i != 0

Summary

- **synchronized**
  - **atomize** a critical section
  - performance decrease
- **volatile**
  - **atomize** long and double assignments
  - **guarantee order** on the previous and next instruction
- **Rule**
  - Any **variable** used by **multiple threads** and **modified by one** at least must be in **synchronized** block **or** be declared as **volatile**

Lock

- Coarse grain locking
  - one lock for all shared resources
  - Decrease paralelism
- Fine grain locking
  - many locks for each shared resource
  - Deadlock

Deadlock

Step	Thread 1	Thread 2
1	lock A	
2		lock B

3	lock B > BLOCKED	
4		lock A > BLOCKED
DEADLOCK		

Deadlock condition

- **Mutual Exclusion**
  - Only one thread can have **exclusive access to a resource**
- **Hold and Wait** - At least one thread is holding a resource and is waiting for another resource
- **Non-preemptive allocation** - A resource is released only after the thread is done using it.
- **Circular wait** - A chain of at least two threads each one is holding one resource and waiting for another resource

Deadlock solution

- Avoid circular wait
  - Strict **order of locking** shared resources

Step	Thread 1	Thread 2
1	lock A	
2		lock A > WAIT
3	lock B	
4	unlock B	
5	unlock A	
6		lock A
7		lock B
8		unlock B
		unlock A

- Deadlock detection - Whatchdog
- Thread interruption
- tryLock operations

ReentrantLock

- same functionalities than synchronized method
- plus:
  - check lock status
  - lockInterruptibly
    - Allow to **interrupt** thread **waiting** for lock
  - tryLock
    - thread is not suspended forever
    - **wait** until **timeout** is achieved and return false if !lock

ReentrantReadWriteLock

- same than ReentrantLock but
  - it allow **many** threads to **read**
  - only **one write**.
  - It also block readers when writing

Semaphore

- Restrict how many threads access to shared resources
- **Lock**
  - allows access to **single thread** only
  - is **reentrant**
- **Semaphore**
  - allows access to a **multiple threads**
  - **Not reentrant**

Inter-thread communication

Condition Variables

- Thread can wait for condition that release other thread using Lock.newCondition()
- java.util.concurrent.locks.Lock
  - lock
  - unlock
- java.util.concurrent.locks.Condition
  - await
  - signal
  - signalAll

Object

- java.lang.Object
  - synchronized(object)
  - wailt

    - notify
    - notifyAll
    - current thread waits until other thread wakes it up
- notify
    - wakes up 1 thread waiting on that object
- notifyAll
    - wakes up all threads waiting on that object

Lock-Free Algorithms, Data-Structures & Techniques

Atomic Objects

- PROS
    - simplicity
    - No need for locks or synchronization
    - No race conditions or data races
- CONS
    - Only the operation is atomic
    - Race condition between 2 separate atomic operations could appear

AtomicReference

- Wraps a reference and allow to perform atomic operations on the reference
- if currentValue == expectedValue > assign newValue
- else if currentValue != expectedValue > nothing