Java Basic-Jan 31

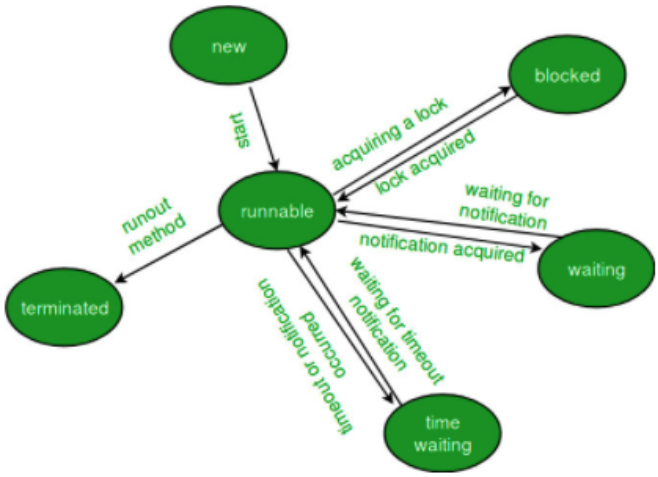
19. Multithread

process vs. thread - the difference between process and thread

- process
  - independent execution of instruction with independent memory space, stack, heap, and OS resources
  - two processes will not share something
- Thread (each process can have many thread)
  - shared memory space(heap and method area, and Native Method stack) (not share: stack, pc register)
  - private stack, program counter, register

*Thread state (6 states)

- new
  - thread created,(not yet start)
- runnable
  - executing in JVM
- blocked
  - wait for a monitor lock to entry synchronized block or method
- waiting (by calling wait() method)
  - Object.wait with no timeout (trigger waiting)
  - Thread.join with no timeout (trigger the waiting)
  - park();
- timed_waiting
  - thread sleep
  - Object.wait with timeout
  - Thread.join with timeout
  - park with timeout
- terminated
  - thread has completed
  - or interrupted exception happened



Life Cycle of a Thread

1. New State - new thread, after new, a thread will be created
2. call start(), thread will jump into runnable
    - When we've created a new thread and called the *start()* method on that, it's moved from *NEW* to *RUNNABLE* state.
3. Runnable State
    - **Threads in this state are either running or ready to run, but they're waiting for resource allocation from the system.**
4. Blocked State
    - A thread tries to access some data that is locked by some other thread, and then it will enter this state and be waiting.
    - until other thread release lock, this thread can access the data/object
5. Waiting State
    - When we call wait(), the thread will be in the waiting state
    - any thread can enter this state by calling any one of the following three methods:
        1. *object.wait()*
        2. *thread.join()* or
        3. *LockSupport.park()*
6. Timed_waiting
    - if we set a time for waiting, it will enter timed_waiting state
7. Terminated

- everything is down
- some exceptions happened thread will also jump into Terminated

Thread creation

create method:

1. extends Thread (do not prefer)
    - not use normally, cuz Java is single thread and extends only one class. After extends Thread, it will not extend others.
2. implements runnable (recommanded)
3. implements callable (recommanded)
4. thread pool

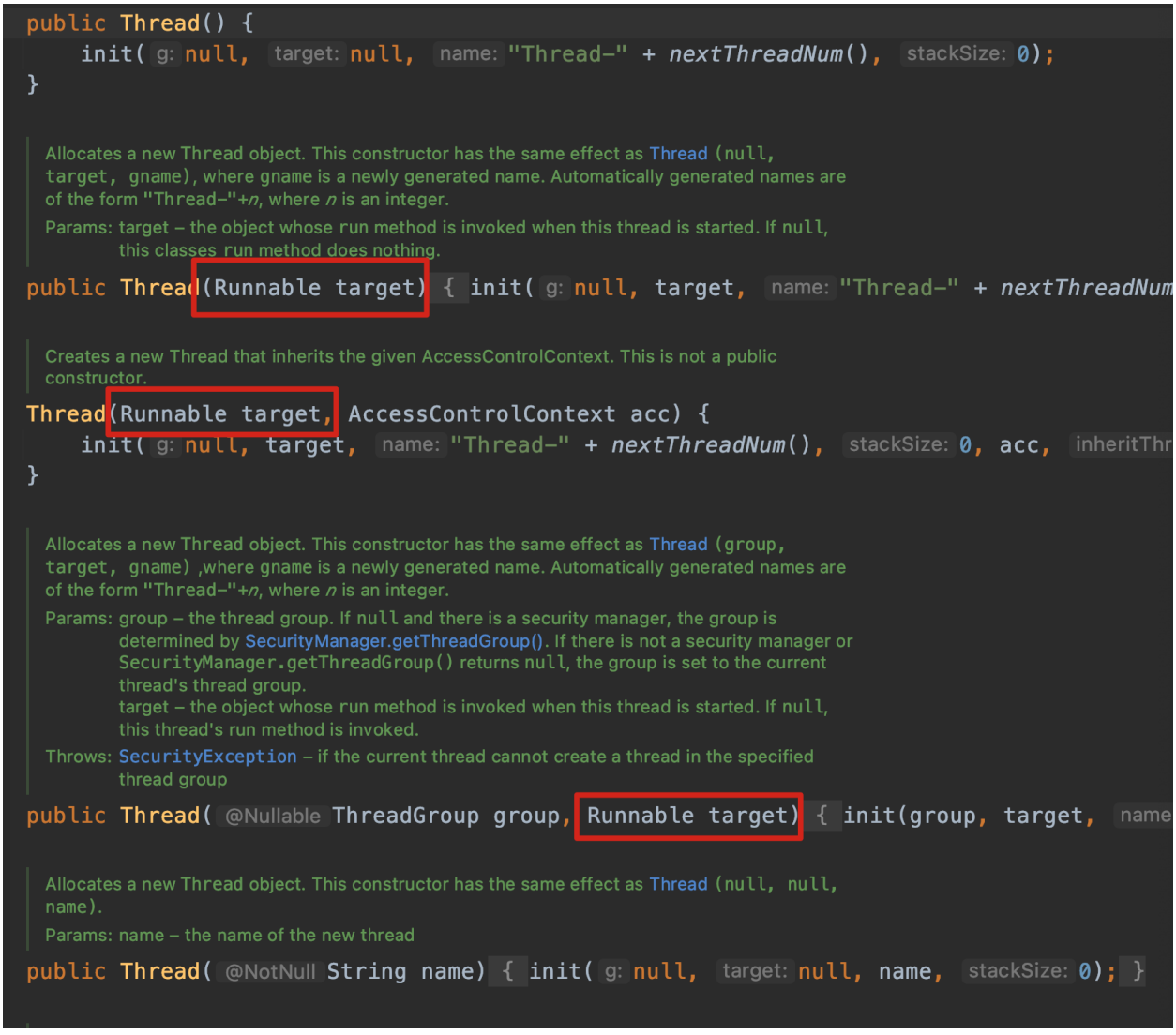
runnable vs. callable

- runnable doesn't have return, callable has return (has return means, it has get() method)
- runnable cannot throw exception, callable can throw exception
- runnable override run(), callable override call()

```
public class ThreadCreation {
   public static void main(String[] args) throws ExecutionException,
                                  InterruptedException {
       Thread t1 = new FromThread();
       t1.start();

       Thread t2 = new Thread(new FromRunnable()); // using Runnable by passing
                                       //the instance into Thread constructor
       t2.start();

      //Create Thread implemented by Callable
       FutureTask ft = new FutureTask(new FromCallable());
      // FutureTask implement RunnableFuture, (like a middleware)
      //and RunableFuture extends Runnable and Future
//** Interface can extend multiple Interfaces
       Thread t3 = new Thread(ft);
       t3.start();
       System.out.println(ft.get()); // get(), callabe has return
                                    // which wil/l return somthing depends on call() method
                             // get() return a Object named comeout in Class FutureTask
  }
}
// extends Thread
class FromThread extends Thread {
   public void run() {
       System.out.println("extends Thread class, the current thread is " + Thread.currentThread().getName());
   }
}
// implement Runnable
class FromRunnable implements Runnable {
```

```
    @Override
    public void run() {
        System.out.println("Implements Runnable interface, the current thread is " + Thread.currentThread().getName());
    }
}
// implement Callable
class FromCallable implements Callable {

    @Override
    public Integer call() throws Exception {
        System.out.println("Implements Callable interface, the current thread is " + Thread.currentThread().getName());
        return 200; //
    }
}
```

Attention: Thread() only accepts Runnable Interface and not accept Callable Interface



If need to use Callable, we should use other stuff, the FutureTask

```
 FutureTask ft = new FutureTask(new FromCallable());
//** Interface can extend multiple Interfaces
       Thread t3 = new Thread(ft);
       t3.start();
       System.out.println(ft.get());
```

Thread Pool

In Java, the operating system, the JVM resources is limited like Memory, CPU source, core number.

Q: For example, 1000 requests enter in JVM, how does the JVM handle these requests at same time?

It will not create 1000 threads at same time, because we have only limited core number.

The max number of Threads running at the same time depends on the number of cpu core number. Usually, one Core, one thread. (or Intel HT?超线程)

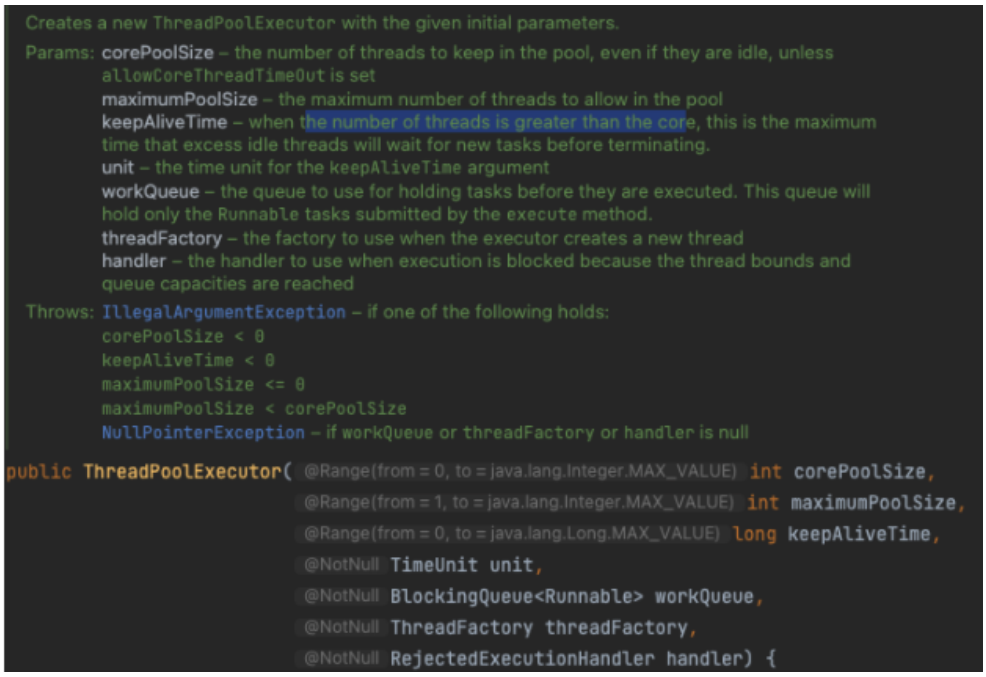
i.e

physically: 8 core → 8 threading (running at the same time)

Solution:

- Create a thread pool, and store the threads into that pool. Get a thread from thread pool to use, when complete using, turn it back into the pool.

How to customized our own thread pool:

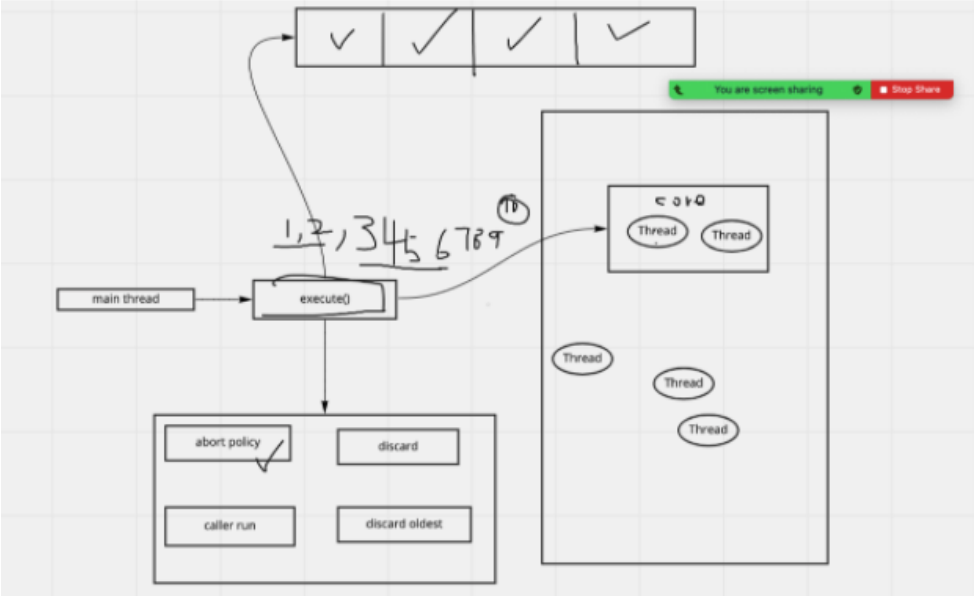


```
Creates a new ThreadPoolExecutor with the given initial parameters.
Params: corePoolSize – the number of threads to keep in the pool, even if they are idle, unless
        allowCoreThreadTimeOut is set
        maximumPoolSize – the maximum number of threads to allow in the pool
        keepAliveTime – when the number of threads is greater than the core, this is the maximum
        time that excess idle threads will wait for new tasks before terminating.
        unit – the time unit for the keepAliveTime argument
        workQueue – the queue to use for holding tasks before they are executed. This queue will
        hold only the Runnable tasks submitted by the execute method.
        threadFactory – the factory to use when the executor creates a new thread
        handler – the handler to use when execution is blocked because the thread bounds and
        queue capacities are reached
Throws: IllegalArgumentException – if one of the following holds:
        corePoolSize < 0
        keepAliveTime < 0
        maximumPoolSize <= 0
        maximumPoolSize < corePoolSize
        NullPointerException – if workQueue or threadFactory or handler is null
public ThreadPoolExecutor( @Range(from = 0, to = java.lang.Integer.MAX_VALUE) int corePoolSize,
                           @Range(from = 1, to = java.lang.Integer.MAX_VALUE) int maximumPoolSize,
                           @Range(from = 0, to = java.lang.Long.MAX_VALUE) long keepAliveTime,
                           @NotNull TimeUnit unit,
                           @NotNull BlockingQueue<Runnable> workQueue,
                           @NotNull ThreadFactory threadFactory,
                           @NotNull RejectedExecutionHandler handler) {
```

ThreadPoolExecutor

- corePoolSize
  - the number of threads to keep in the pool, even if they are idle
- maximumPoolSize
  - the max number of threads to allow in the pool
- keepAliveTime
  - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.
  - When corePoolSize < the number of threads < maxPoolSize, how long the idle threads can wait for new tasks
- unit - time unit (year/day/second...)
- workQueue - blockingQueue used to catch all the task, all the request
  - holding tasks before they executed. The queue will hold only the Runnable tasks submitted by the execute method
- threadFactory
  - Factory design pattern for creating a new thread
- handler - rejection policy, when thread bounds and queue capacities are readed
  - AbortPolicy (default)
  - CallerRunPolicy
  - DiscardPolicy

- DiscardOldestPolicy



- main thread → create some tasks
- want to get some threads from thread pool to run those tasks
- for thread pool, there are two arguments to define the thread pool

core: 2

max: 5

queue 4

- at the beginning the execute() get thread from Core area
- when the extra request comes in (the total number > core number), it will put into Queue
  - all the requests in the Queue are waiting for the core threads are released (not be used)
  - when for example, the 1 and 2 requests finish the tasks, they will return the threads to the thread pool, and all the requests in the Q can use these threads again

- When core has no threads can be used and Q is full, a request comes in, what will happen?
  - i.e. No.7 request comes in core threads are all occupied, and blockingQ is full
  - S: cuz max thread pool is 5, and we only create 2 by core, it(thread factory?) will create another one
    - When the requests that are accepted up to the max pool size and a new request comes in, it will trigger handler (rejection policy)

- They are 4 policies: abort policy, discard, caller run, and discard oldest
- We can define which policy we can use
  - **abort policy:** is default, and also it will throw the rejection exception
  - **discard:** just discard the new coming request
  - **discard oldest**: will discard the oldest request in the blocking queue, and put the new request into the queue
  - caller run: the request will be handled by local thread, not the thread pool

```
```

Predefined Thread pool

- FixedThreadPool
  - core == max
- SingleThreadExecutor
  - core == max == 1
- CachedThreadPool
  - core == 0 and max == Integer.Max_VALUE
- ScheduledThreadPool
  - implements by FixedThreadPool(that's why define a core size at the beginning)

```
ExecutorService threadPool1 = Executors.newFixedThreadPool(5);
// core == 5, max == 5
ExecutorService threadPool2 = Executors.newSingleThreadExecutor();
// core 1, max 1
ExecutorService threadPool3 = Executors.newCachedThreadPool();
// core 0, max = Integer.MAX_VALUE;
ExecutorService threadPool4 = Executors.newScheduledThreadPool(3);
// Creates a thread pool that can schedule commands to run after
//a given delay, or to execute periodically
// use some methods like schedule(Runnable, delay)
```

Lock

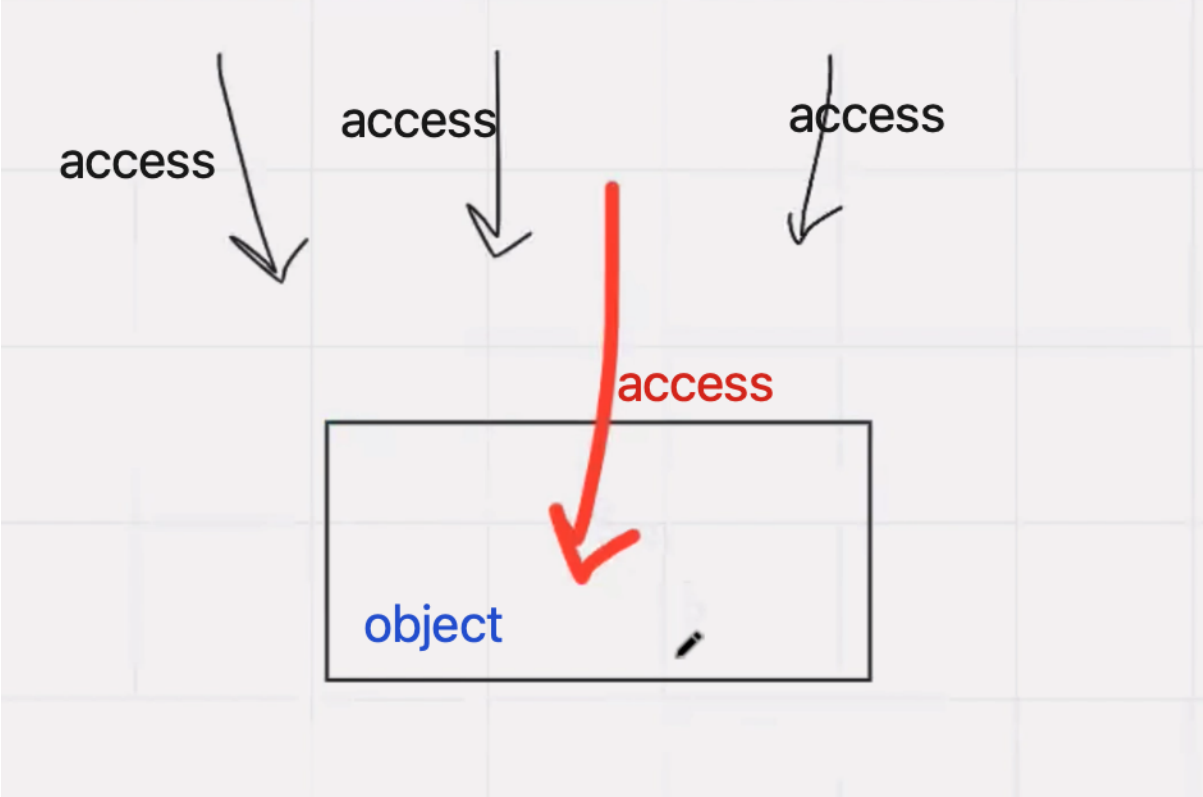
When a block/method/class is locked, only one thread can access it.

Synchronized (internal lock?)

- modify block: object level lock
- modify method

- modify static method - class level lock, it will lock all the object who try to access the method
- modify class

```
class ClassName {
public void method () {
  synchronized(ClassName.class) { //class level
  // todo
  }
}


public synchronized void method() { // object level
  // lock whole object
}

public synchronized static void method() {  //class level
  // will lock all the objects who have the mehtod
}

public void method () {
  synchronized(this) { // to lock object
                       // object level
  }
  }
}
```



only one access can come in limited by **synchronized** keyword

When we finish the logic, JVM will automatically release lock.

Lock Interface - lock the method

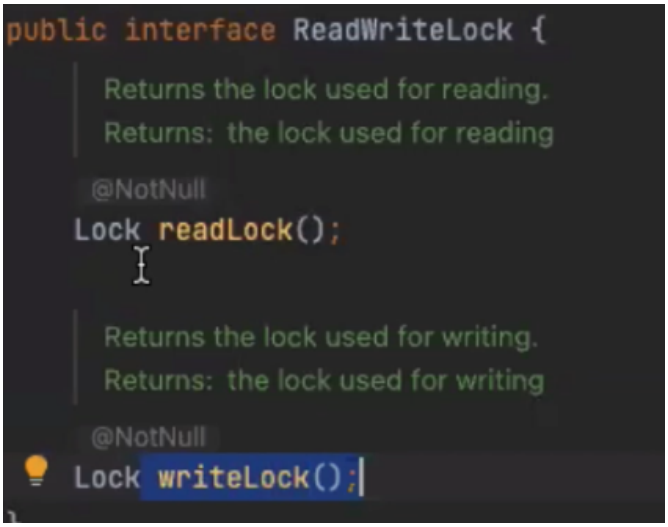
- lock(), unlock(), newCondition(), tryLock(), lockInterruptibly()...
  - if use lock, need to define these interface
- **only one** class implement these interface, which is

  ReentrantLock

ReadWriteLock interface

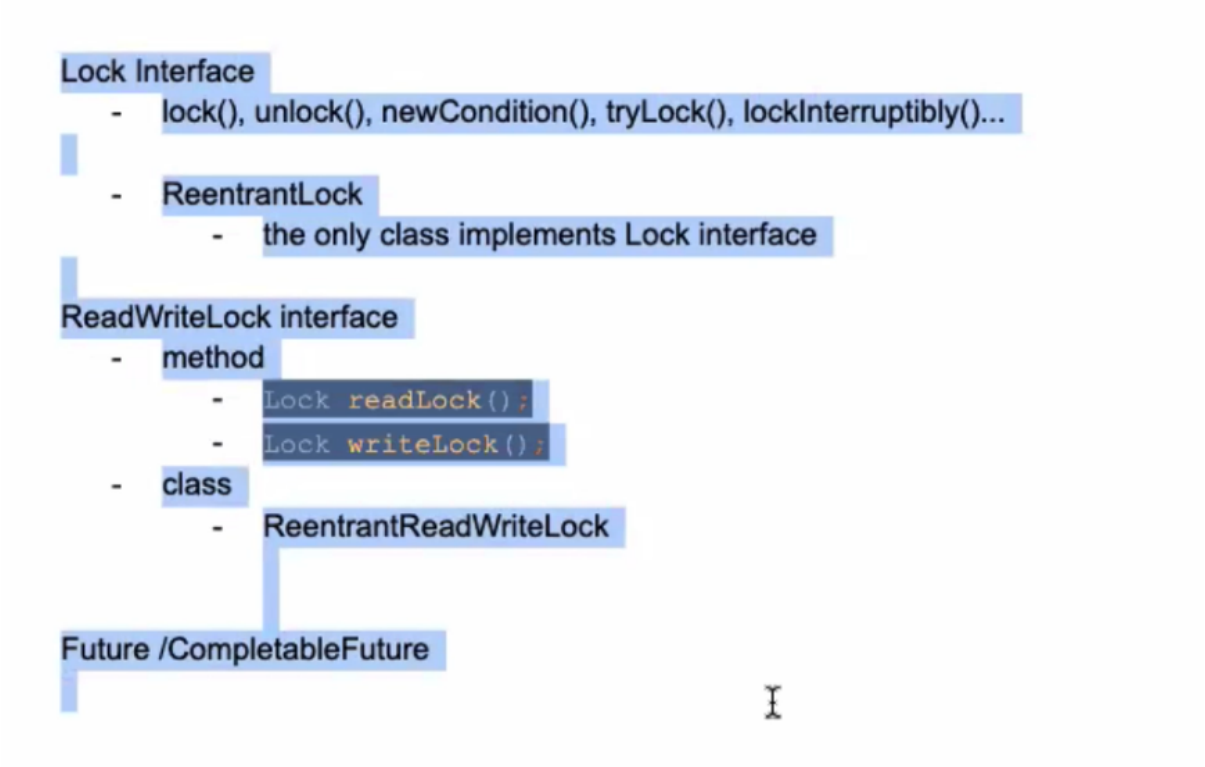
- method
  - Lock readLock();
    - **Read lock:** If there is no thread that has requested the write lock and the lock for writing, then **multiple threads** can lock the lock **for reading**. It means multiple threads can **read the data at the very moment**, as long as there's no thread to write the data or to update the data.
    - **Write Lock:** If no threads are **writing or reading**, **only one** thread at a moment can lock the **lock for writing**. Other threads have to **wait** until the lock gets released. It means, only one thread can write the data at the very moment, and other threads have to wait.
  - Lock writeLock();
- class who implements ReadWriteLock
  - **ReentrantReadWriteLock**
  - ReentrantReadWriteLock.WriteLock/ReadLock
    - static nested class
      - A static class is a class that is created inside a class, is called a static nested class in Java.

```
public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }
```




```
private static ReentrantLock lock = new ReentrantLock();

private static void accessResource() {

    lock.lock();

    // access the resource

    lock.unlock();
}

public static void main(String[] args) {

    Thread t1 = new Thread(() -> accessResource()); t1.start();
    Thread t2 = new Thread(() -> accessResource()); t2.start();
    Thread t3 = new Thread(() -> accessResource()); t3.start();
    Thread t4 = new Thread(() -> accessResource()); t4.start();
```

Future / CompletableFuture



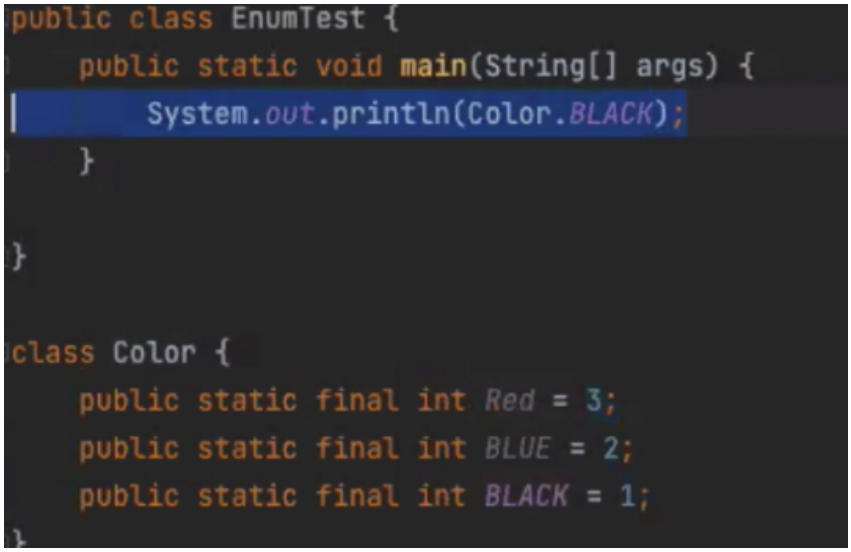
Attention: P**otential evaluation stuff above keywords**

20. Enum

An enum type is ***a special data type that enables for a variable to be a set of predefined constants****.*

i.e. color

before using constant - complex



print: 1

use enum to make everything easy and simple

```
public class EnumTest {
   public static void main(String[] args) {

//        System.out.println(EColor.BLUE.ordinal()); // index of BLUE
//        for (EColor val: EColor.values()) { // .vlaues() give all the values of enum
//            System.out.println(val);
//        }
       System.out.println(EColor.RED.getNum());
   }

}

enum EColor { // first letter should be capitalized for using enum
   RED(3), BLUE(2),BLACK(1);

   int value;
   EColor(int value) {
      this.value = value;
   }

   public int getNum() { // define method like normal class for enum
       return value;
   }
}

class Color {
   public static final int Red = 3;
   public static final int BLUE = 2;
   public static final int BLACK = 1;
}
```