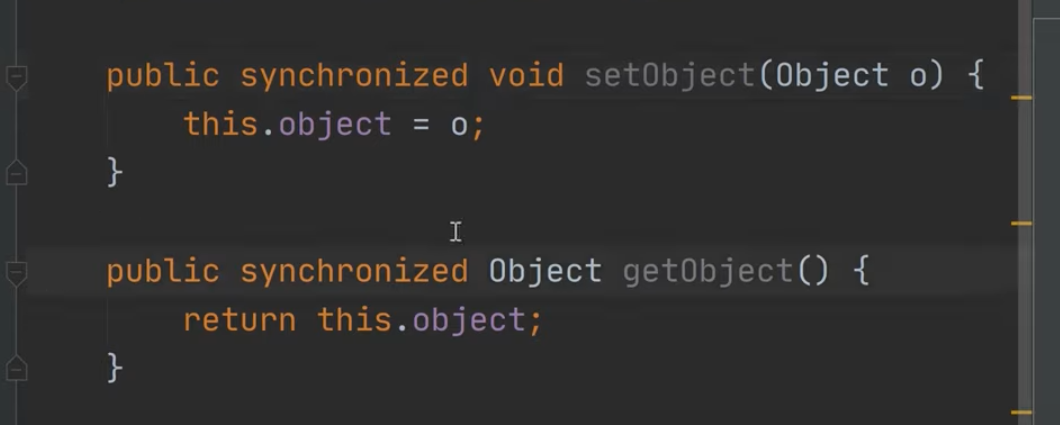
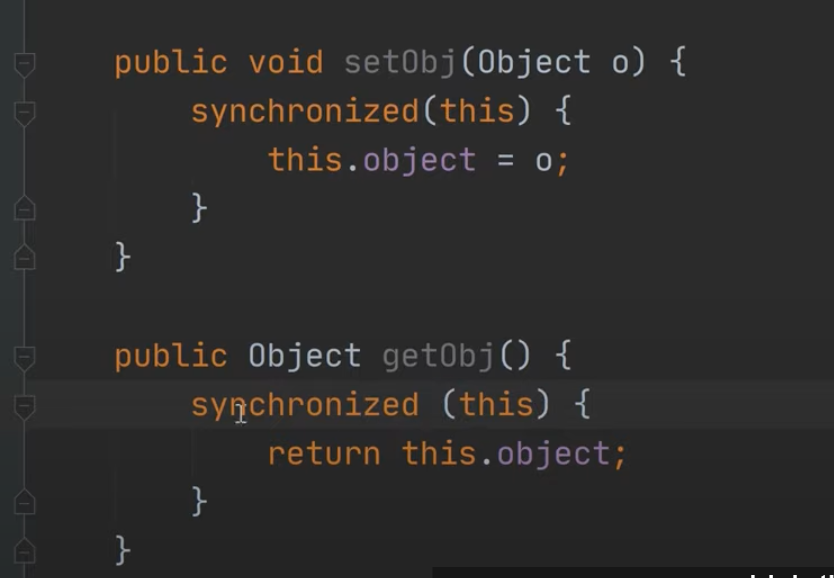
**Synchronize :**

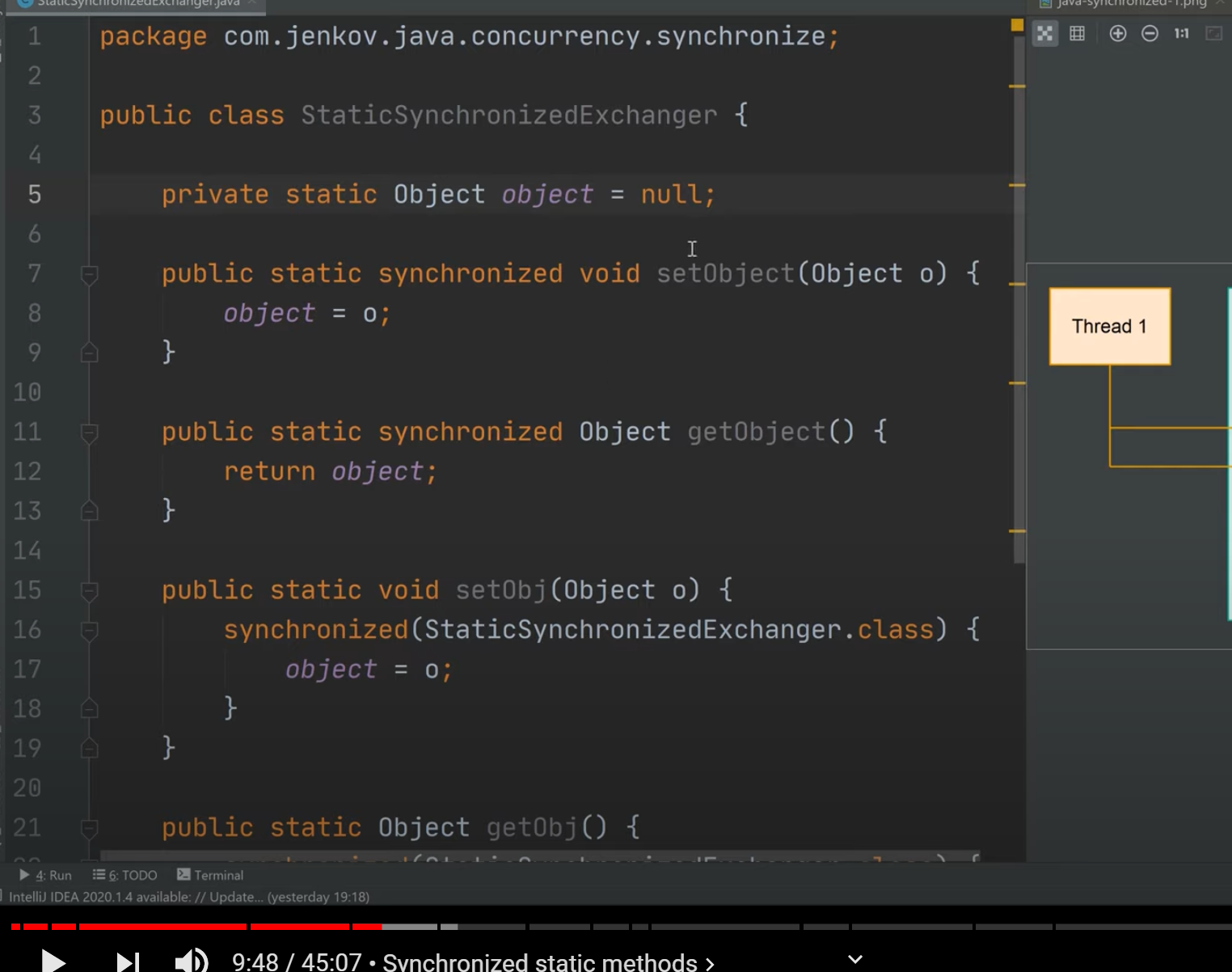
* Java synchronize blocks is a block of java code which can only be executed by one thread at a time.
* Synchronize method: …………………………………………………………….. be executed by one thread at a time on the given instance of that class.



A thread can execute either method 1 or method 2 because both of them are synchronized methods.

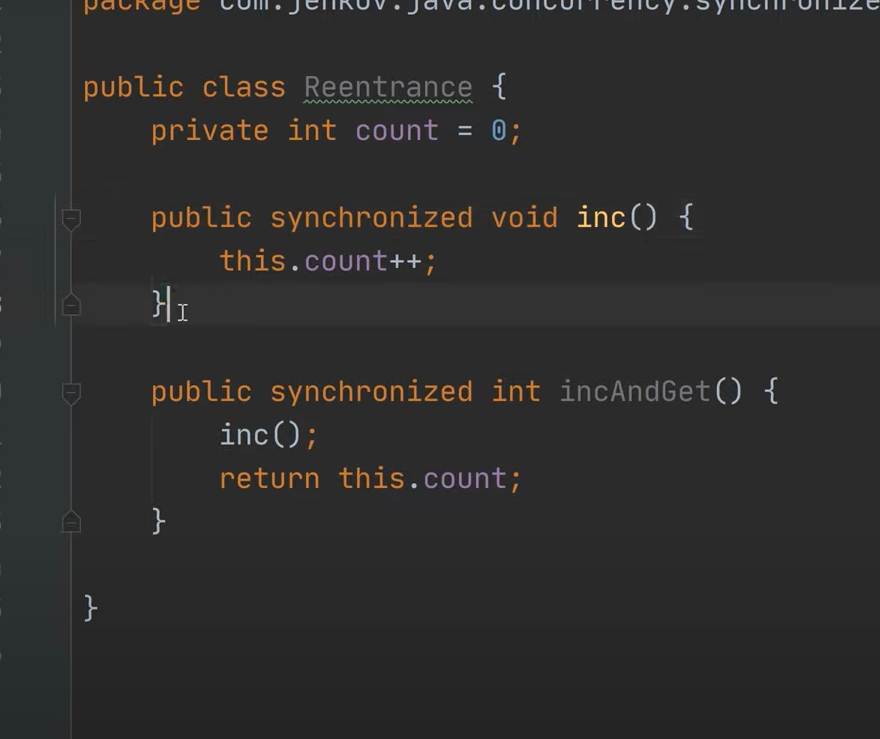
* 

This keyword is monitor object. Monitor object is the object on which these method or these blocks are synchronized



Synchronized Static Method: Static Synchronized method uses the class object of the class it belongs to as the monitor object. It means that instead of using the instance of the class, method it uses the class object. It doesn’t belong to any instance, it belongs to the class.

**Reentrance**

****

Reentrance means a thread that already hold the monitor object and it’s **allowed** to enter a new synchronized block

**Class Object**

**Wait**

Method này sẽ làm cho luồng đang sở hữu monitor của đối tượng b ( hay luồng đang khóa đối tượng b và nắm giữ đội tượng này) tạm thời nhưng hoạt động và trả monitor của b cho luồng khác. Sau khi trả monitor luồng 1 sẽ về trạng thái đợi.

**Notify**

Sau khi luồng 2 nắm giữ Monitor của b và xử lý xong những gì luồng 1 cần, thì luồng 2 sẽ gọi phương thức notify() hoặc notifyall() trên đối tượng b để đánh thức các luồng đang chờ monitor của b và ngay sau đó luồng 2 sẽ trả lại monitor của b.

* Lưu ý: Có thể xảy ra hiện tượng khóa chết (DEADLOCK) khi mà cả 2 process đều đợi nhau hoàn thành (LOOP)

**Lock interface**

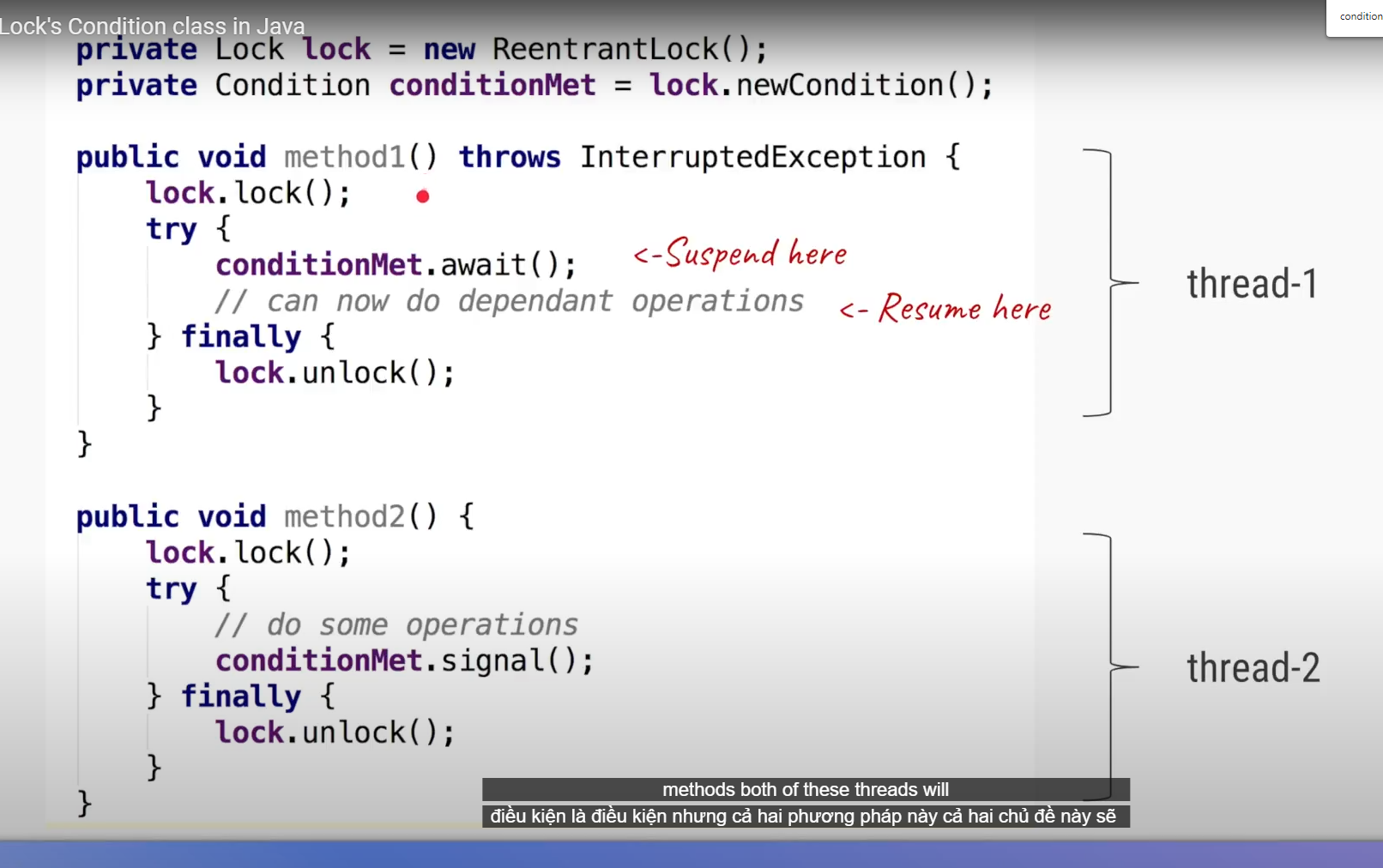
Java Lock is an interface, so we cannot create an instance of Lock directly. We must create an instance of a class that implements the Lock interface.. The java.util.concureent.locks package has the following implementations of the Lock interface:

* Java.util.concurrent.locks.ReentrantLock
* Lock lock = new ReentrantLock();

Obviousy all threads must share the same Lock instance. If each thread creates it own Lock instance, then they will be locking on different locks, and thus not be blocking each other from access.

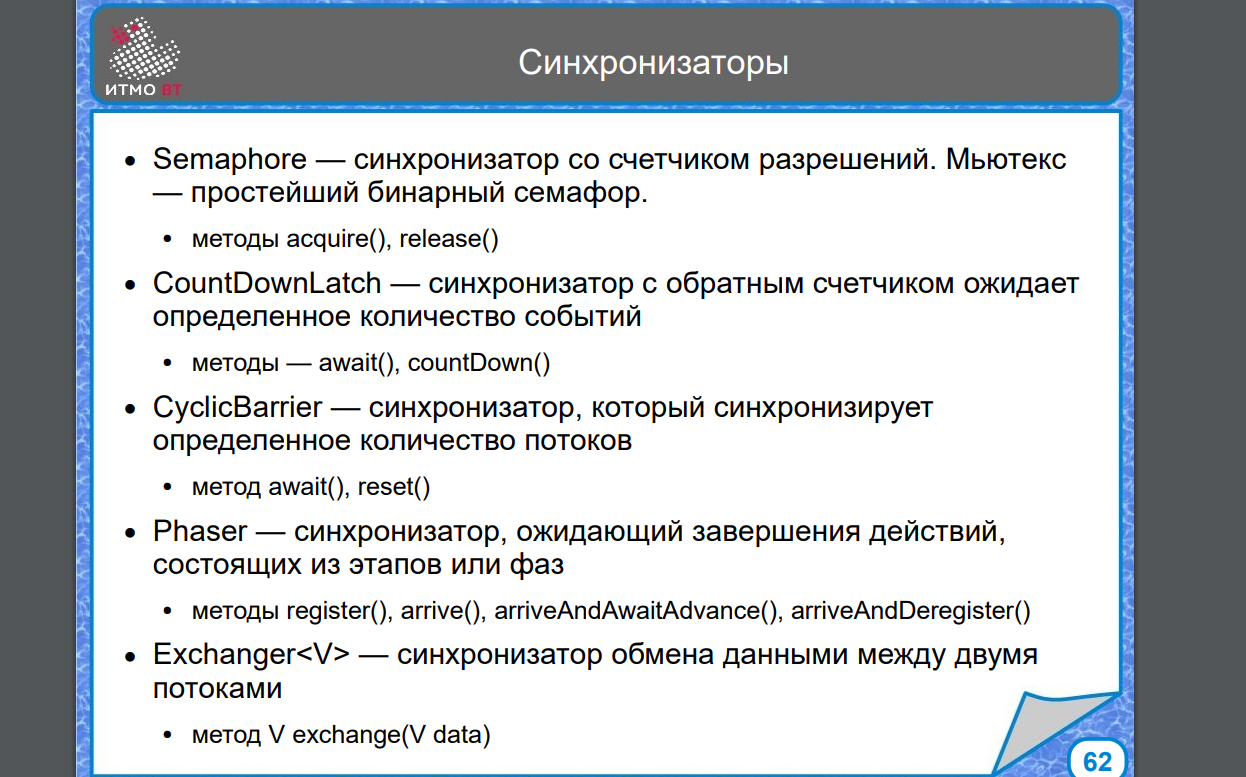
**Condition interface**

Java Condition interface provides a thread ability to suspend its execution, until the given condition is true.

* Lock.newCondition() to create instance
* 

Thread 1 đến method await sẽ tạm đỉnh chỉ cho đến khi thread 2 thông báo tín hiệu ( signal ())

**Synchronizer classes from package java.util.concurrency**

****

CountDownLatch:

<http://tutorials.jenkov.com/java-util-concurrent/countdownlatch.html>

CyclicBarrier:

<http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html>

Threads đợi các thread khác bằng cách gọi await() method trong CyclicBarrrier. Khi tất cả N threads cùng đợi CyclicBarrrier, thì tất cả các threads sẽ đc giải phóng và có thể tiếp tục chạy.

Exchanger:

<http://tutorials.jenkov.com/java-util-concurrent/exchanger.html>

Exchanger class cung cấp một cái như là “điểm hẹn” nơi mà 2 threads có thể trao đổi object.

Semaphore:

<http://tutorials.jenkov.com/java-util-concurrent/semaphore.html>

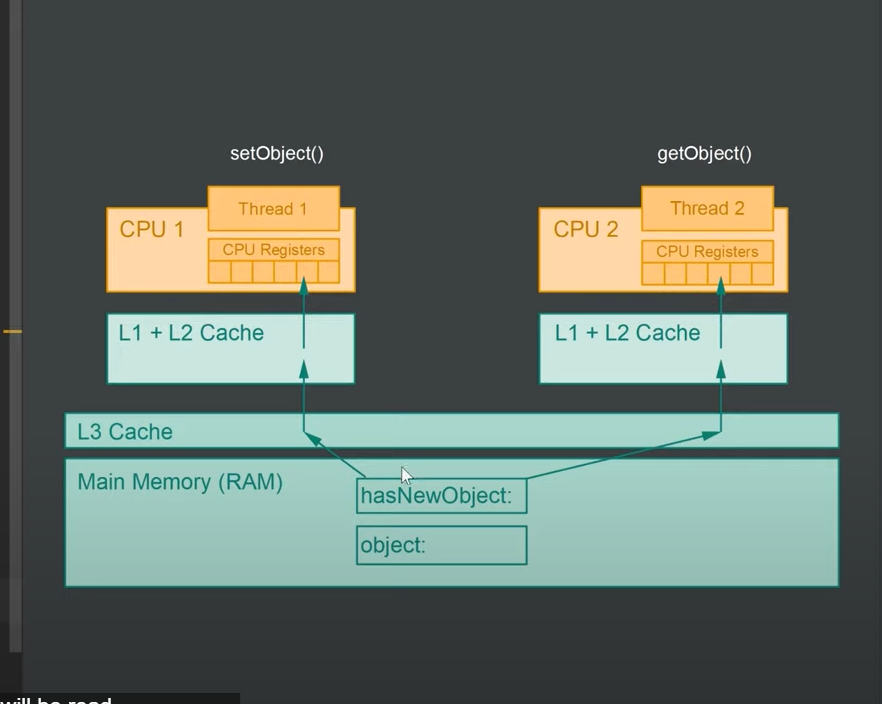
Là một class đếm số lượng tín hiệu. Có 2 method chính:

* Acquire()
* Release()

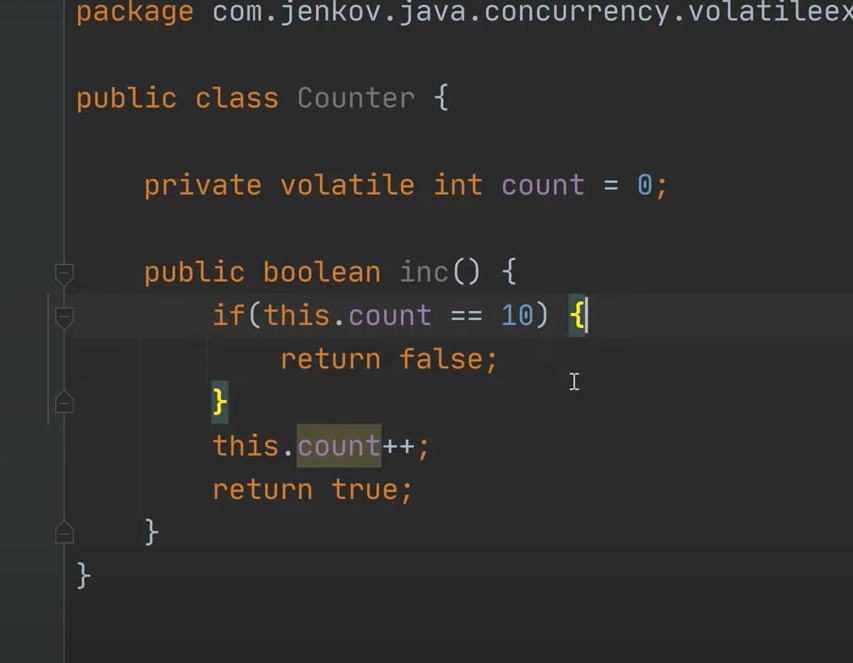
Semaphore được dung để giới hạn số lượng N threads thực thi cùng lúc

**Volatile. Atomic data type and operation**

The java volatile keyword is a special keyword which can be applied when declaring a variable. By declaring a variable volatile you tell the JVM that this variable must always be read directly into main memory (RAM), when the valuable is changed, the value must be written directly back to RAM immediately.

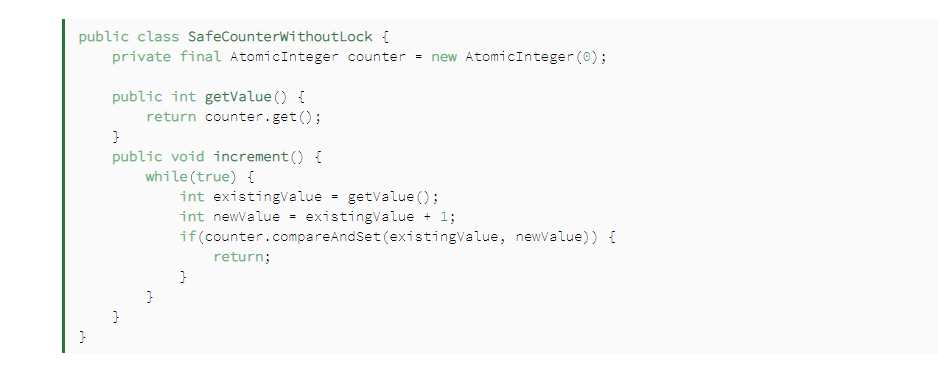
****

* Khi setObject thì nó có thể không update trực tiếp vào Main Memory mà chỉ lưu vào CPU registers, điều đó không đảm bảo visibility problems
* Using Volatile is not always enough.



**Atomic variable and operation:**

* AtomicBollean
* AtomicInteger
* AtomicLong
* AtomicReference
* AtomicStampedReference
* AtomicIntegerArray
* AtomicLongArray
* AtomicReferenceArray



**Atomic Operation:** Those operations that always execute together is known as the atomic operations or atomic action. All the atomic operations either execute effectively happens all at once or it does not happen at all.

Get()

Set()

incrementAndGet()

compareAndSet(expected, value)

addAndGet()

getAndAdd()

decrementAndGet()

getAndDecrement()

**Collection from package Java.util.concurrent**

BlockingQueue / BlockingDeque

+ put and take

+ for deque, putFirst, put Last, takeFirst, takeLast

+ ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue

+ DelayQueue

+SynchronousQueue

ConcurrentMap / CocurrentNavigableMap

+ ConcurrentHashMap

+ ConcurrentSkipListMap, ConcurrentSkipListSet

ConcurrentLinkedQueue

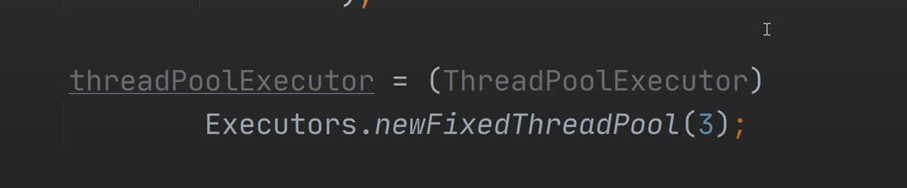
CopyOnWriteArrayList / CopyOnWriteArraySet

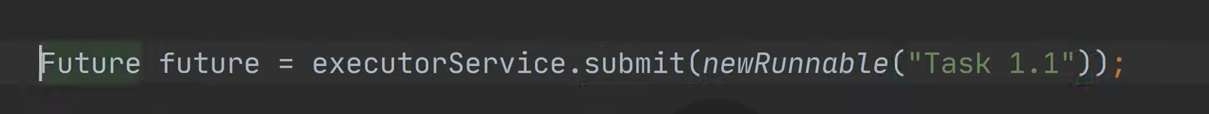
**Interface Executor ExecutorService Callable Future**

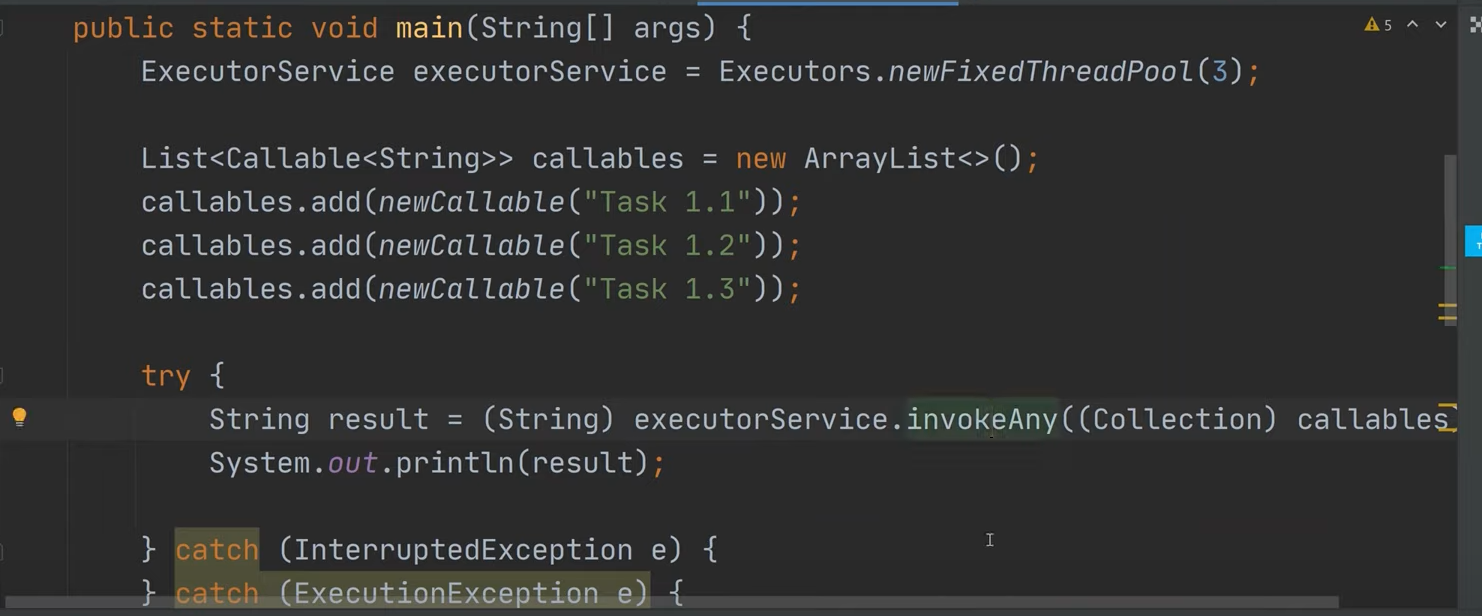
* ExecutorService is an interface that means that in order to use it you must create an instance of the class which implements ExecutorService interface
* Java has two built-in implementations of the ExecutorService interface:

+ ThreadPoolExecutor

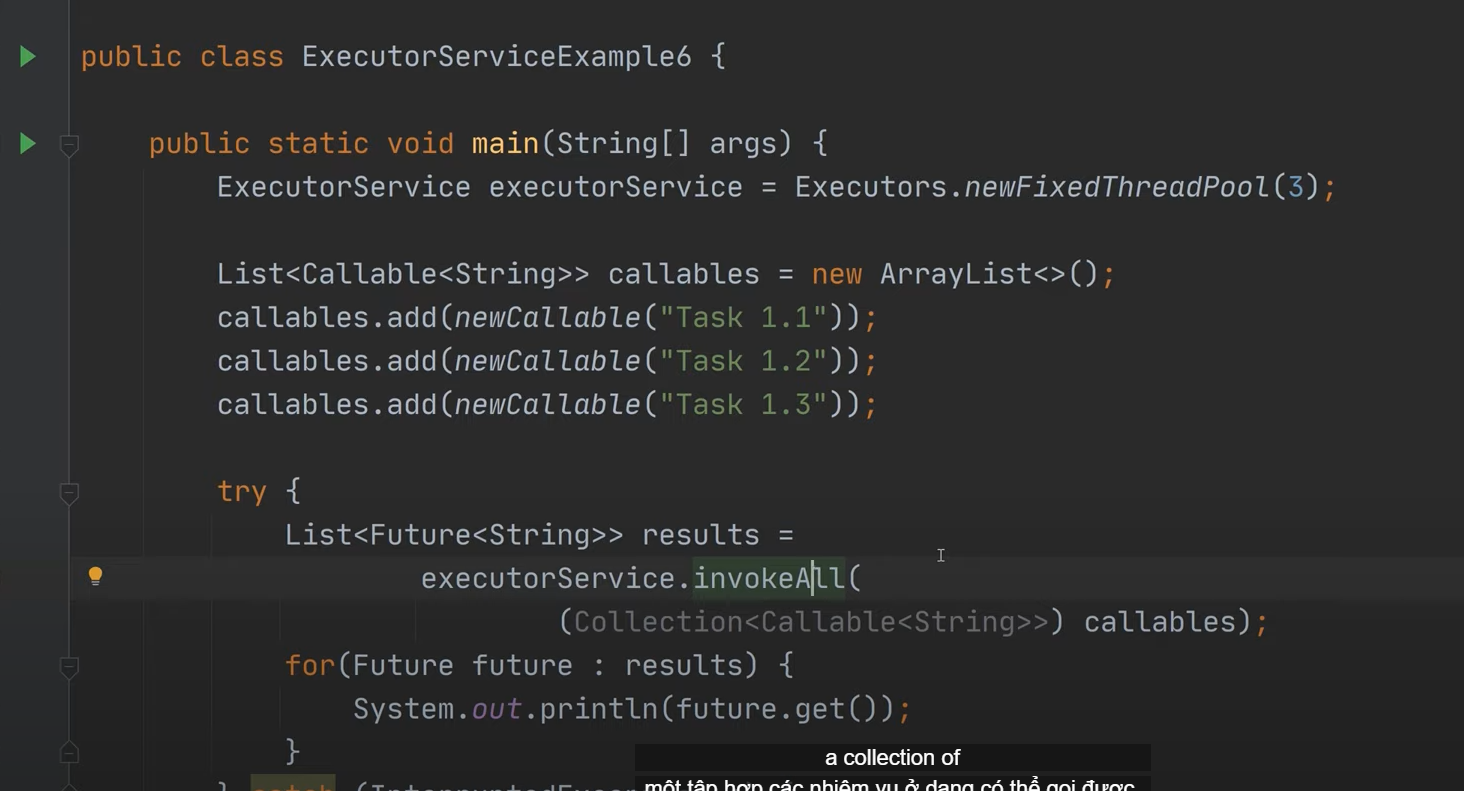
+ ScheduledThreadPoolExecutor

* The Java ExecutorService is a thread pool to which you can submit tasks for concurrent execution.
* Create ExecutorService by using: Executors.newFixedThreadPool(10) for example
* 
* executorService.execute( Runnable implementation )
* executorServer.submit(Runnable implementation) but it returns a future



* future.get() đợi cho đến khi task được hoàn thành completely
* ExecutorService.submit( Callable implementation) , it returns an Object whereas runnable could not ( return void )
* 

Method invokeAny sẽ gọi bất kỳ 1 task nào có trong collection



Method invokeall() will execute all of the tasks in the collection of tasks. Method này trả lại 1 Collection chứa các Future

**ThreadPool**

**JDBC**

JDBC – Java DataBase Connectivity

JDBC API – là một interface cấp cao để kết nối với database dưới dạng bảng

JDBC Driver API là một interface cấp thấp cho drivers

**DriverManager**

Is a class manager lists of drivers.

Method:

+ getConnection()

+ getConnection(String url)

+ getConnection(String url, String username, String passwd)

**Connection interface**

Connection là một interface mà đối tượng của nó được trả về bởi method getConnection của class DriverManager.

Đối tượng connection dung để tạo các statement:

+ Statement createStatement()

+ PreparedStatement prepareStatement(String sql)

+ CallableStatement prepareCall(String sql)

+ DatabaseMetaData getMetaData()

**Statement PreparedStatement ResultSet RowSet interface**

* **Statement** là một interface, tạo bằng cách : connection.createStatement()

Dùng để thực thi các query tường minh!

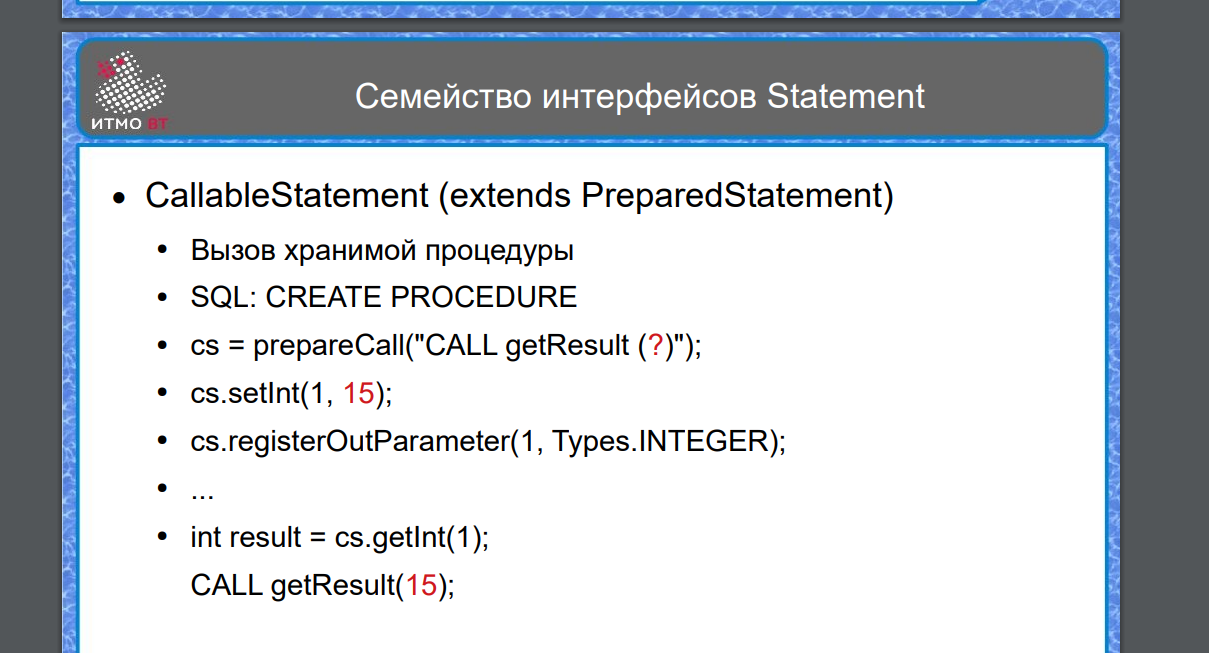
* **PreparedStatement** (kế thừa Statement) là một dynamic request with parameter, có thể truyền tham số vào để thực thi.

Cần phải set parameter cho prepareStatement , lưu ý đến thứ tự của dấu ? và kiểu dữ liệu truyền vào



* **CallableStatement** là một interface kế thừa PreparedStatement dung để thực thi một procedure/function mà được tạo ra ( theo ý muốn ) .

Cũng cần phải lưu ý truyền tham số theo thứ tự của dấu ? và kiểu dữ liệu truyền vào



* Method:
  + executeQuery(String sql )để thực thi lệnh SELECT trả về 1 **ResultSet**
  + executeUpdate (String sql) để thực thi lệnh INSERT, UPDATE, DELETE trả về kiểu **int** là số hang bị thay đổi
  + execute (String sql) để thực hiện một câu lệnh bất kỳ, trả về một buổi **boolean**
* Transation

Connection

+ setAutoCommit(true/false)

+ commit()

+ rollback()

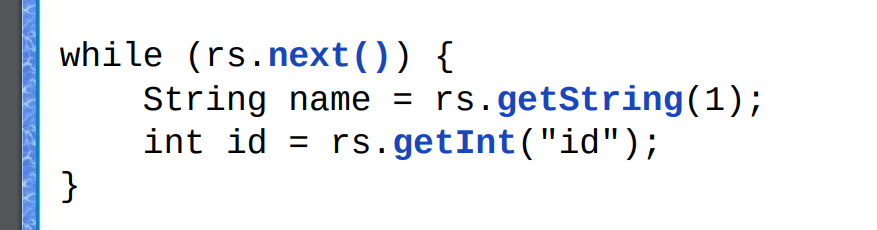
Statement

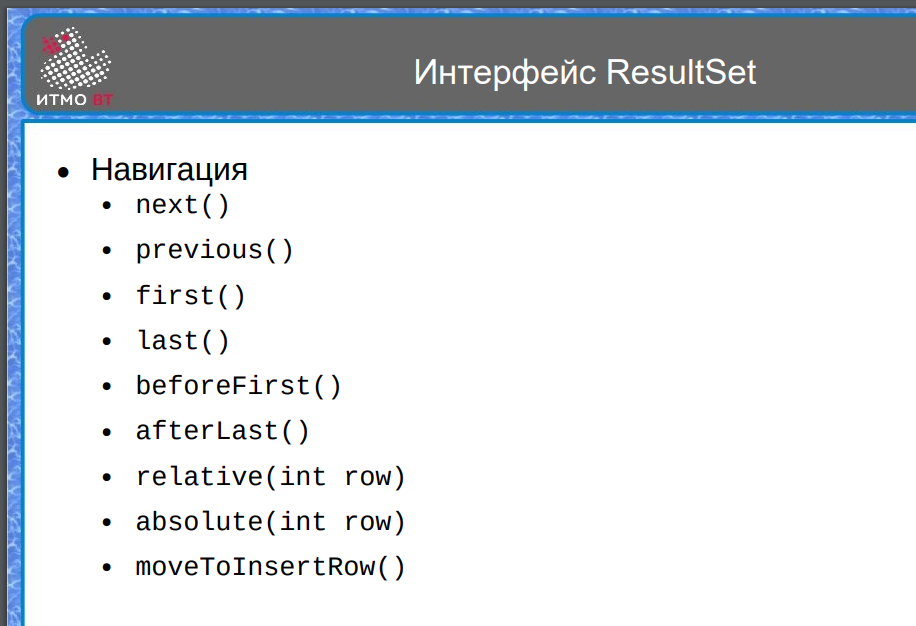
+ addBatch(String sql)

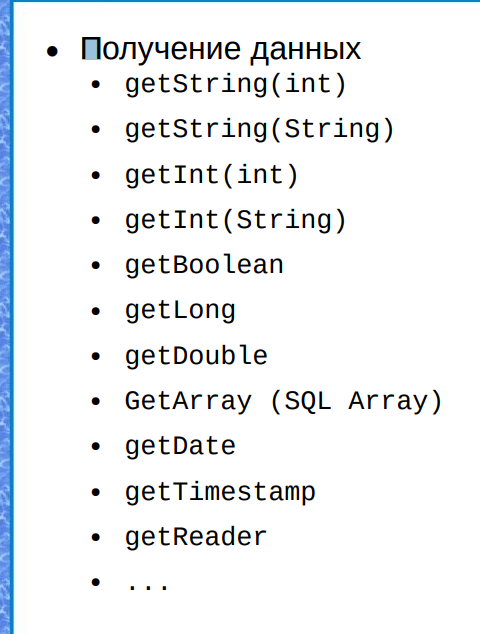
+ clearBatch()

+ executeBatch()

**Interface ResultSet**

Lấy dữ liệu từ ResultSet : 





**RowSet:** Là một interface cho tất cả các operations. RowSet extends ResultSet

RowSet có các phương thức như:

setURL

setUserName

setPasssword

setCommand (“ SELECT \* from …” )

execute(0

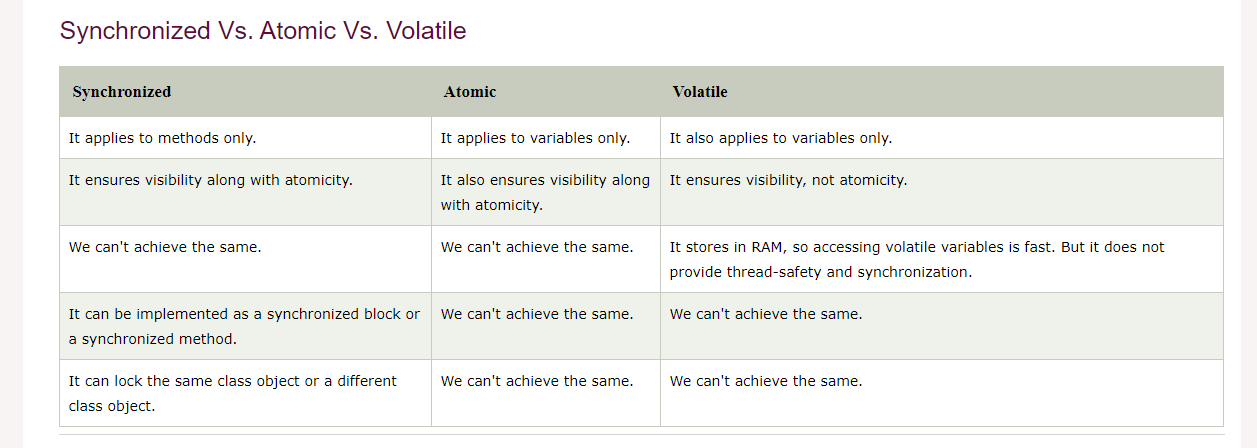
next()

getXXX() for example rs.getInt(“id”)

rs.updateString(“name”, “Popkin”);

Các loại rowSet

* JdbcRowSet
* CachedRowSet
* WebRowSet
* FilteredRowSet

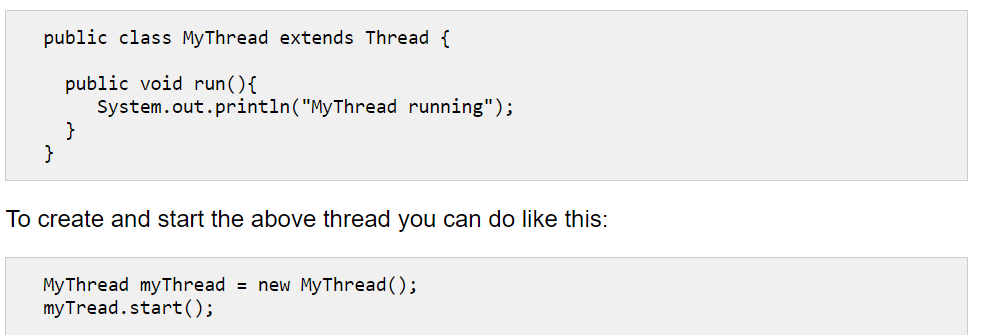
****

**ORM**

Taking a step further, an even more programmatic approach to SQL is the use of [Object-Relational Mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) (ORM). Using ORM means mapping your DB tables to your objects, allowing you to read, write and query entire objects. Since ORM further reduces your use of explicit SQL, it is also a good way to avoid SQL Injection.

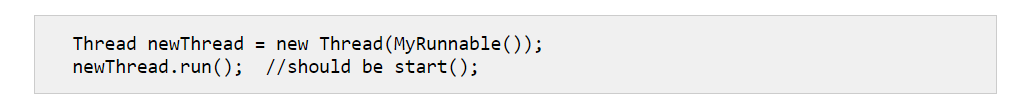
What difference between run() and start()?

**Thread Subclass**



The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU. When the run() method executes it will print out the text "MyThread running".

## Common Pitfall: Calling run() Instead of start()



At first you may not notice anything because the Runnable's run() method is executed like you expected. However, it is NOT executed by the new thread you just created. Instead the run() method is executed by the thread that created the thread. In other words, the thread that executed the above two lines of code. To have the run() method of the MyRunnable instance called by the new created thread, newThread, you MUST call the newThread.start() method.