Some of the important methods are lock() to acquire the lock, unlock() to release the lock, tryLock() to wait for lock for a certain period of time, newCondition() to create the Condition etc.

2. **Condition**: Condition objects are similar to Object wait-notify model with additional feature to create different sets of wait. A Condition object is always created by Lock object. Some of the important methods are await() that is similar to wait() and signal(), signalAll() that is similar to notify() and notifyAll() methods.
3. **ReadWriteLock**: It contains a pair of associated locks, one for read-only operations and another one for writing. The read lock may be held simultaneously by multiple reader threads as long as there are no writer threads. The write lock is exclusive.
4. **ReentrantLock**: This is the most widely used implementation class of Lock interface. This class implements the Lock interface in similar way as synchronized keyword. Apart from Lock interface implementation, ReentrantLock contains some utility methods to get the thread holding the lock, threads waiting to acquire the lock etc.
synchronized block are reentrant in nature i.e if a thread has lock on the monitor object and if another synchronized block requires to have the lock on the same monitor object then thread can enter that code block. I think this is the reason for the class name to be ReentrantLock. Let's understand this feature with a simple example.

```
public class Test{

public synchronized foo(){
    //do something
    bar();
  }

  public synchronized bar(){
    //do some more
  }
}
```

If a thread enters foo(), it has the lock on Test object, so when it tries to execute bar() method, the thread is allowed to execute bar() method since it's already holding the lock on the Test object i.e same as synchronized(this).

Java Lock Example – ReentrantLock in Java

Now let's see a simple example where we will replace synchronized keyword with Java Lock API.

Let's say we have a Resource class with some operation where we want it to be thread-safe and some methods where thread safety is not required.

```
package com.journaldev.threads.lock;

public class Resource {

        public void doSomething(){
                //do some operation, DB read, write etc
        }

        public void doLogging(){
                //logging, no need for thread safety
        }
}
```

Now let's say we have a Runnable class where we will use Resource methods.

```
package com.journaldev.threads.lock;

public class SynchronizedLockExample implements Runnable{

        private Resource resource;

        public SynchronizedLockExample(Resource r){
                this.resource = r;
        }

        @Override
        public void run() {
                synchronized (resource) {
                        resource.doSomething();
        }
}
```

Notice that I am using synchronized block to acquire the lock on Resource object. We could have created a dummy object in the class and used that for locking purpose.

Now let's see how we can use java Lock API and rewrite above program without using synchronized keyword. We will use ReentrantLock in java.

```
package com.journaldev.threads.lock;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ConcurrencyLockExample implements Runnable{

        private Resource resource;
        private Lock lock;

        public ConcurrencyLockExample(Resource r){
                this.resource = r;
                this.lock = new ReentrantLock();
        }

        @Override
        public void run() {
                try {
                        if(lock.tryLock(10, TimeUnit.SECONDS)){
                        resource.doSomething();
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }finally{
                        //release lock
                        lock.unlock();
                }
                resource.doLogging();
        }

}
```

try-finally block to make sure lock is released even if doSomething() method call throws any exception.

Java Lock vs synchronized

Based on above details and program, we can easily conclude following differences between Java Lock and synchronization.

1. Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.
2. Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
3. synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.
4. synchronized keyword doesn't provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
5. We can create different conditions for Lock and different thread can await() for different conditions.

That's all for Java Lock example, ReentrantLock in java and a comparative analysis with synchronized keyword.

FILED UNDER: JAVA

About Pankaj
If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.
Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

« Core Java Interview Questions and Answers Java 8 Features with Examples »

Comments

Riteeka says
JUNE 10, 2017 AT 9:12 PM
Hi Pankaj,