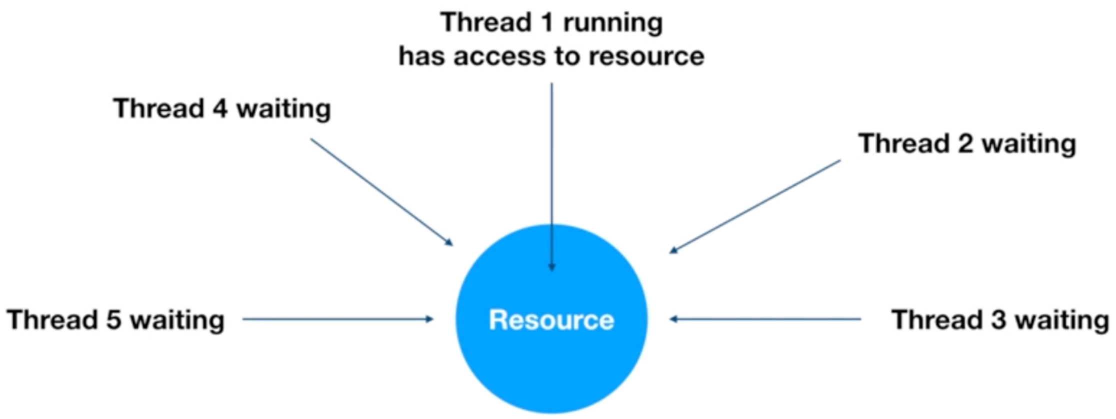
Synchronization

It helps to achieve the access of one resource by only one thread at a time.

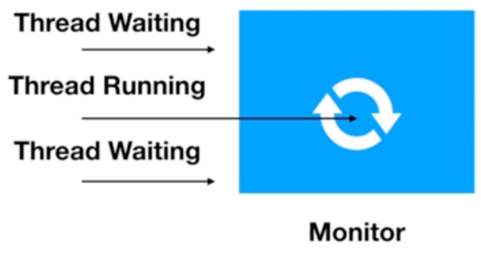


Concept of Monitor

- In java the synchronization take place with the help of Monitor.
- The concept of monitor is similar to that of an ATM machine.

Monitor

- A Monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- All the other threads attempting to enter the locked Monitor will be suspended until the first threads exits the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.



Monitor Synchronization

- All object have their own implicit monitor associated with them.
- To enter an objects monitor we call the methods modified with synchronized keyword.
- To exit the monitor the owner thread simply returns from the synchronized method.
- Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
- Non synchronized methods on that instance will continue to be callable.

Synchronization

```java
synchronized public void IAmSynchronized(){
    // method implementation
}

static synchronized public void IAmStaticSynchronized(){
    // method implementation
}

public void IAmBlockSynchronized(){
    synchronized (this){
        // something here
    }
}
```

When to use synchronized

When you have a method or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the synchronized keyword to guard that state from the race condition.

https://youtu.be/IdY1IButi_E

YouTube
Tutorial: Java And Android Multithreading, ThreadPoolExecutor, Callable and F
0:14 / 4:12

Project Setup

- Install gradle: https://gradle.org/install
- Download IntelliJ IDEA CE: https://www.jetbrains.com/idea
- Open terminal run following:
- mkdir ~/Desktop/mindorks-multithreading-2
- cd ~/Desktop/mindorks-multithreading-2
- gradle init --type java-library
- IntelliJ IDEA > Open > Desktop > mindorks-multithreading
- OK
- Create classes: App with main method
- Select App and right click and select run App.main()

Tutorial: Java And Android Multithreading, ThreadPoolExecutor, Callable and Future: Part 2

Dead Lock

- It occurs when two threads have a circular dependency on a pair of synchronized objects.
- It is a difficult error to debug because it occurs rarely when the time slice is just right and also it may involve more that two threads and two synchronized objects.

Figure - 1

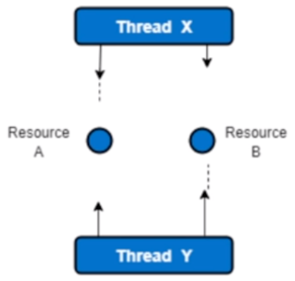
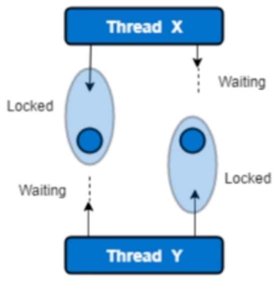


Figure - 2



Lock

- Lock provide a alternative to the use of synchronized methods
- Before accessing a shared resource, the lock that protects that resource is acquired. When access to the resource is complete, the lock is released.
- If the second thread attempts to acquire the lock that is in use by another thread, the second thread will suspend until the lock is released.

ReadWriteLock

- This interface specifies a lock that maintain separate locks for read and write access.
- This enables multiple locks to be granted for readers of a resource as lock as the resource is not being written.
- ReeinterantReadWriteLock provide an implementation of the ReadWriteLock.

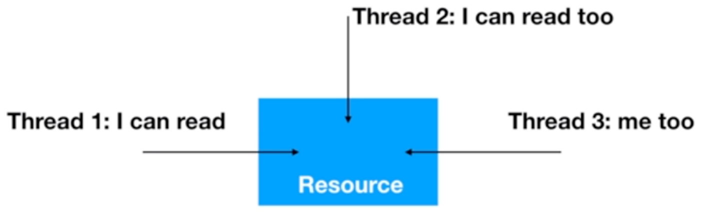


fig 1

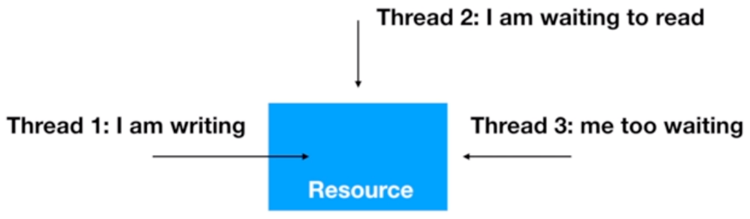
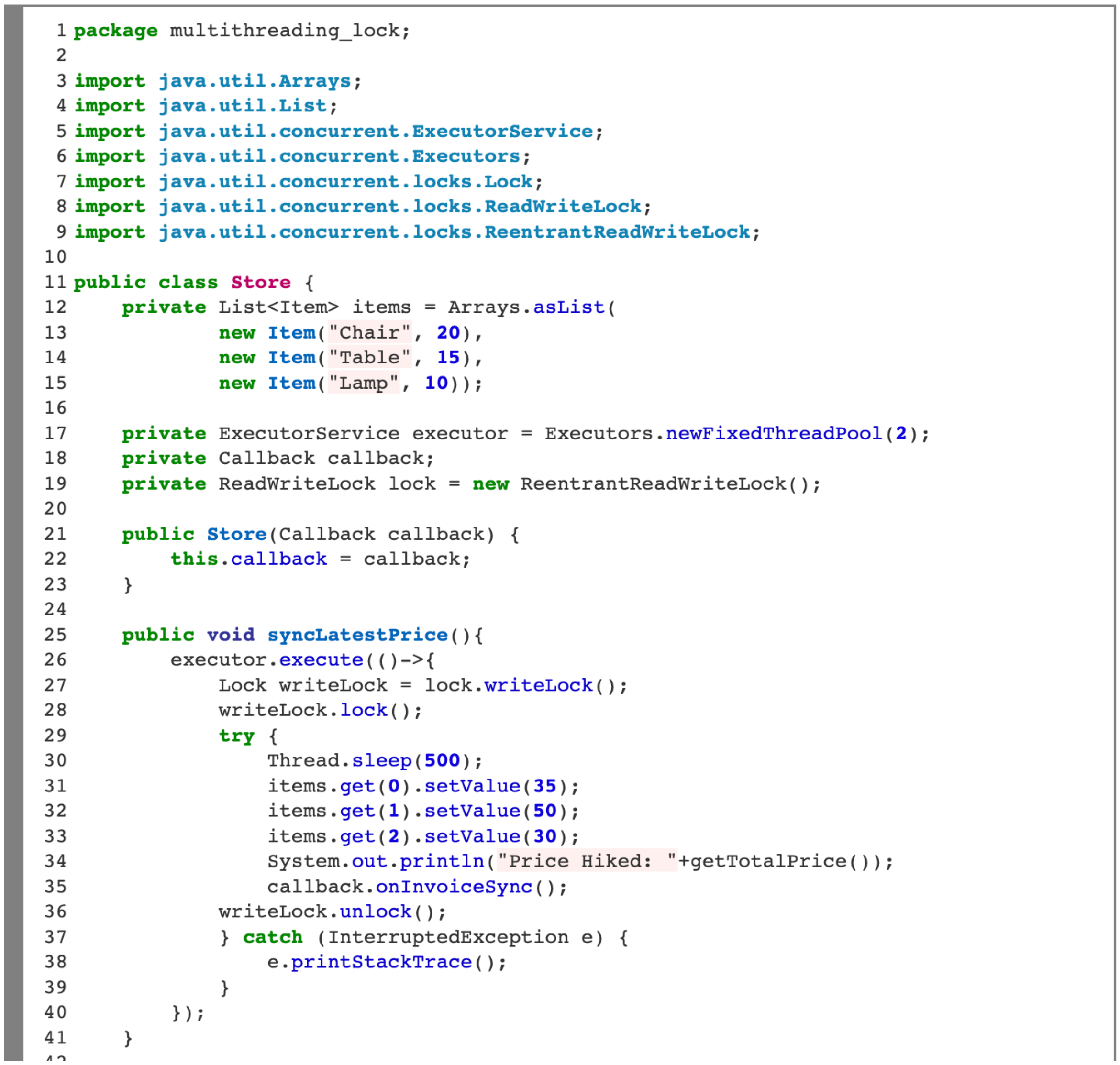


fig 2

```java
package multithreading_lock;

public class Item {
    private String name;
    private int value;

    public Item(String name, int value) {
        this.name = name;
        this.value = value;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "Item{" +
                "name='" + name + '\'' +
                ", value=" + value +
                '}';
    }
}
```



```java
package multithreading_lock;

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Store {
    private List<Item> items = Arrays.asList(
            new Item("Chair", 20),
            new Item("Table", 15),
            new Item("Lamp", 10));

    private ExecutorService executor = Executors.newFixedThreadPool(2);
    private Callback callback;
    private ReadWriteLock lock = new ReentrantReadWriteLock();

    public Store(Callback callback) {
        this.callback = callback;
    }

    public void syncLatestPrice(){
        executor.execute(()->{
            Lock writeLock = lock.writeLock();
            writeLock.lock();
            try {
                Thread.sleep(500);
                items.get(0).setValue(35);
                items.get(1).setValue(50);
                items.get(2).setValue(30);
                System.out.println("Price Hiked: "+getTotalPrice());
                callback.onInvoiceSync();
            writeLock.unlock();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }

    public void prepareInvoice(){
        executor.execute(()->{
            Lock readLock = lock.readLock();
            readLock.lock();
            int total = getTotalPrice();
            System.out.println("Your invoice is for the amount: "+total);
            callback.onInvoicePrepared(total);
            readLock.unlock();
        });
    }

    public int getTotalPrice(){
        int total = 0;
        for (Item i : items) {
            total += i.getValue();
        }
        return total;
    }

    public interface Callback{
        public void onInvoiceSync();
        public void onInvoicePrepared(int total);
    }
}
```

```java
package multithreading_lock;

import java.util.concurrent.atomic.AtomicBoolean;

public class Resource {
    private AtomicBoolean disallow = new AtomicBoolean(false);

    public void setDisallow(){
        disallow.set(true);
    }

    public void process(){
        if (!disallow.get()){
            try {
                Thread.sleep(2000);
                System.out.println("I process because It was allowed!: "+Thread.curren
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }else {
            System.out.println("I could not process because It was not allowed! :"+Thr
        }
    }
}
```

```java
package multithreading_lock;

import java.util.concurrent.atomic.AtomicBoolean;

public class Main {

    public static void main(String[] args) {

        AtomicBoolean syncing = new AtomicBoolean(true);
        AtomicBoolean preparing = new AtomicBoolean(true);
        Store store = new Store(new Store.Callback() {
            @Override
            public void onInvoiceSync() {
                 syncing.set(false);
            }

            @Override
            public void onInvoicePrepared(int total) {
                preparing.set(false);
            }
        });
        store.syncLatestPrice();
        store.prepareInvoice();

        while (syncing.get() || preparing.get()){
            //running
        }
        System.out.println("Program Terminated!");
    }
}
```