Table of Contents

Java's Synchronizing Mechanism

Our bathroom morning bathroom routine

In my house we have three family members (folks F1,F2,F3) who like to take showers in the morning! We have one bathroom (with one shower stall!). Here is how we cooperatively manage on a typical morning.

> The bathroom door has a **lock**. When closed, the door automatically **locks** from the outside; you cannot get in without the key. We have just ONE key which sits on a hook in the hallway outside the bathroom.

All family members have the same morning routine in this order

1. Perform morning chores
2. Have breakfast
3. Take a shower (requires the shared resource – the bathroom)
4. Get dressed

Example shower timeline

F1:

- time 7:00am: grabs the key from the hook
- time 7:00am: unlocks and enters the bathroom with the key
- starts shower

F2 :

- time 7:02am: goes to the hallway and sees NO key
- time 7:02am: waits in the hallway for the key!

F3 :

- time 7:04am: goes to the hallway and sees NO key
- time 7:04am: waits in the hallway for the key!

F1:

- time 7:10: leaves the bathroom and replaces the key on the hook

F2:

- time 7:10 grabs the key from the hook
- time 7:10am unlocks and enters the bathroom with the key
- start shower
- time 7:14: leaves the bathroom and replaces the key on the hook

F3:

- time 7:14 grabs the key from the hook
- time 7:14am unlocks and enters the bathroom with the key
- starts shower
- time 7:19: leaves the bathroom and replaces the key on the hook

Simulate our family morning routine

We write code to simulate our family morning routine.

First we code a logger for timestamped logging to console

```
package Log;

public class Logger {

	public static void log(String text) {
		long timeStamp = System.currentTimeMillis();
		System.out.println(timeStamp + " " + text);
	}

	public static void logThisJunk(String text) {
		// just ignore the text -- it is junk!!!
	}
}
```

This code uses up CPU. We simulate time taken to accomplish various morning routines like getting dressed or taking a show. The elapsed times do not vary much at all but that is OK for our simple simulation.

```
package simulation;

import Log.Logger;

public class CpuHog {
	public static void doIt() {

		final int big = 10000000;
		double junkA =1;
		for (int I = 0; I < big ; I ++) {
			junkA   =  junkA + Math.log(I+.004);
		}

		double junkB = 1; ;
		for (int I = 0; I < big ; I ++) {
			junkB   =  junkB + Math.sin(I);
			junkB = junkB - junkB/10;
		}
```

```
                double junkC =1;
                for (int I = 0; I < big ; I ++) {
                        junkC   =  junkC + Math.log(I+.004);
                }

                double junkD = 1; ;
                for (int I = 0; I < big ; I ++) {
                        junkD   =  junkD + Math.sin(I);
                        junkD = junkD - junkD/10;
                }

                Logger.logThisJunk("Junk -  ignore this! " + junkA + " " + junkB + " " + junkC + " " + junkD);
        }
}
```

Our bathroom

```
package bathroom;

import Log.Logger;
import simulation.CpuHog;

public class Bathroom {

        public void takeShower() {
                Logger.log("Thread: " + Thread.currentThread().getId() + " begin Shower");

                long start = System.currentTimeMillis();
                // use up some cpu to slow things down.. to simulate time taken to shower
                CpuHog.doIt();
                long done = System.currentTimeMillis();

                Logger.log("Thread: " + Thread.currentThread().getId() +
            " completed Shower in  " + (done-start)  + " ms");
        }
}
```

A Family member

```
package family;

import Log.Logger;
import bathroom.Bathroom;
import simulation.CpuHog;

public class FamilyMember implements Runnable {

	protected final Bathroom bathroom;

	public FamilyMember (Bathroom bathroom) {
		this.bathroom = bathroom;
	}

	@Override
	public void run() {

		performTask("My first morning task: Do my morning chores");
		performTask("Have breakfast");
		this.bathroom.takeShower();
		performTask("My last morning task: Get dressed");
	}

	Private void performTask (String task) {
		Logger.log("Thread  " + Thread.currentThread().getId() + " Start " + task);
		CpuHog.doIt();
		Logger.log("Thread  " + Thread.currentThread().getId() + " Completed " + task);
	}
}
```

Our controller creates and starts the threads. A thread represents one family member morning routine. Remember, all family members share the single bathroom.

```
package bathroomControl;

import resource.Bathroom;
import workers.FamilyMember;

public class BathroomController {

	private Thread[] threadArr;
	private final int numberFolks;

	public BathroomController(int numberFolks) {

		this.numberFolks = numberFolks;
		this.threadArr = new Thread[numberFolks];

		Bathroom bathroom = new Bathroom();
		for (int I = 0; I < numberFolks; I++) {
			threadArr[I] = new Thread(new FamilyMember(bathroom));
		}
	}

	public void startThreads() {
		for (int I = 0; I < numberFolks; I++) {
			threadArr[I].start();
		}
	}
}
```

Finally, we need a main.

```
package appControl;

import bathroom.BathroomController;

public class Main {
	public static void main(String[] args) {
		BathroomController bathroomController = new BathroomController(3);
		bathroomController.startThreads();
	}
}
```

We have a concurrency Issue

We execute a run and notice a concurrency issue.

1645820340108 Thread: 13 Start My first morning task: Do my morning chores
1645820340109 Thread: 14 Start My first morning task: Do my morning chores
1645820340110 Thread: 15 Start My first morning task: Do my morning chores
1645820341474 Thread: 14 Completed My first morning task: Do my morning chores
1645820341474 Thread: 14 Start Have breakfast
1645820341475 Thread: 13 Completed My first morning task: Do my morning chores
1645820341475 Thread: 13 Start Have breakfast
1645820341499 Thread: 15 Completed My first morning task: Do my morning chores
1645820341499 Thread: 15 Start Have breakfast
1645820342821 Thread: 14 Completed Have breakfast
1645820342821 Thread: 14 begin Shower
1645820342822 Thread: 13 Completed Have breakfast
1645820342822 Thread: 13 begin Shower
1645820342846 Thread: 15 Completed Have breakfast
1645820342846 Thread: 15 begin Shower
1645820344170 Thread: 14 completed Shower in 1348 ms
1645820344170 Thread: 14 Start My last morning task: Get dressed
1645820344174 Thread: 13 completed Shower in 1352 ms
1645820344174 Thread: 13 Start My last morning task: Get dressed
1645820344215 Thread: 15 completed Shower in 1369 ms
1645820344215 Thread: 15 Start My last morning task: Get dressed
1645820345518 Thread: 14 Completed My last morning task: Get dressed
1645820345519 Thread: 13 Completed My last morning task: Get dressed
1645820345564 Thread: 15 Completed My last morning task: Get dressed

Elapsed time 1645820345564 – 1645820340108 = 5456ms

Family members can and do perform their tasks at the same time – concurrently. The previous sentence applies to all tasks EXCEPT for the shower – the shared resource. The code clearly does **NOT** work! The threads run concurrently! BUT we do NOT want concurrent showering! We want a thread (family member) to WAIT when another family member is showering. We need a bathroom lock.

We need to synchronize!

We use the java synchronized mechanism to fix the issue. We add the bathroom lock that we discussed earlier.

```
package bathroom;
import Log.Logger;
import simulation.CpuHog;

public class Bathroom {

	public synchronized void takeShower() {
		Logger.log("Thread: " + Thread.currentThread().getId() + " begin Shower");
		long start = System.currentTimeMillis();
		// use up some cpu to slow things down.. to simulate time taken to shower
		CpuHog.doIt();
		long done = System.currentTimeMillis();
		Logger.log("Thread: " + Thread.currentThread().getId() +
		    " completed Shower in  " + (done-start)  + " ms");
	}
}
```

We execute a run.

```
1645821327270 Thread: 13 Start My first morning task: Do my morning chores
1645821327272 Thread: 14 Start My first morning task: Do my morning chores
1645821327286 Thread: 15 Start My first morning task: Do my morning chores
1645821328648 Thread: 14 Completed My first morning task: Do my morning chores
1645821328648 Thread: 15 Completed My first morning task: Do my morning chores
1645821328648 Thread: 14 Start Have breakfast
1645821328648 Thread: 15 Start Have breakfast
1645821328649 Thread: 13 Completed My first morning task: Do my morning chores
1645821328649 Thread: 13 Start Have breakfast
1645821329986 Thread: 14 Completed Have breakfast
1645821329987 Thread: 14 begin Shower
1645821329991 Thread: 13 Completed Have breakfast
1645821329992 Thread: 15 Completed Have breakfast
1645821331332 Thread: 14 completed Shower in  1345 ms
1645821331332 Thread: 14 Start My last morning task: Get dressed
1645821331332 Thread: 15 begin Shower
1645821332639 Thread: 15 completed Shower in  1307 ms
1645821332639 Thread: 15 Start My last morning task: Get dressed
1645821332639 Thread: 13 begin Shower
1645821332639 Thread: 14 Completed My last morning task: Get dressed
1645821333935 Thread: 15 Completed My last morning task: Get dressed
1645821333935 Thread: 13 completed Shower in  1296 ms
1645821333935 Thread: 13 Start My last morning task: Get dressed
1645821335217 Thread: 13 Completed My last morning task: Get dressed
```

Observe that showers are no longer concurrent; other tasks remain concurrent. This is good!

Shower times

1645821331332 Thread: 14 completed Shower in 1345 ms
1645821332639 Thread: 15 completed Shower in 1307 ms
1645821333935 Thread: 13 completed Shower in 1296 ms

Timing observations and arithmetic

The end-to-end time for our TWO runs.

> Case one: without the bathroom lock. The three showers are concurrent.
> Elapsed time 1645820345564 – 1645820340108 = 5456ms
>
> Case two: with the lock. The three showers are not concurrent.
> Elapsed time 1645821335217 – 1645821327270 = 7947ms

The locked version took 2491ms longer
7947 - 5456 = 2491ms

Combined locked shower times 1345 + 1307 + 1296 = 3948

Average locked shower time 3948/3 = 1316

Finally! If we subtract 2 average shower times from the lock elapsed time we get close to the concurrent elapsed time of 5456ms

7947 - 1316 – 1316 = 5315

Sorry, if I got carried away with the math

An uncooperative family member

We now introduce a fourth family member; an uncooperative family member Bailey

Our family mornings depend on all family members **cooperating**. Here is an example of an un-cooperative member who delays everyone else's shower!

Bailey grabs the bathroom key and instead of taking a quick shower decides to walk over to the local coffee shop! The other family members are locked out of the bathroom! Call a lock smith!

```
package family;

import Log.Logger;
import bathroom.Bathroom;
import simulation.CpuHog;

public class Bailey extends FamilyMember implements Runnable {

	public Bailey (Bathroom bathroom) {
		super(bathroom);
	}

	@Override
	public void run() {

		synchronized (bathroom)
		{
			Logger.log("Thread: " + Thread.currentThread().getId()
					+ " Bailey walks to coffee shop with the bathroom key!");
			int hogs = 20;
			while (hogs > 0) {
				CpuHog.doIt();
				hogs--;
			}
			Logger.log("Thread: " + Thread.currentThread().getId()
					+ " Bailey back from coffee shop about to return the key");
		}

		//Now do the family tasks
		super.run();
	}
}
```

The bathroom controller adds Bailey to the family.

```
package bathroom;

import family.Bailey;
import family.FamilyMember;

public class BathroomController {

	private Thread[] threadArr;
	private Thread threadBailey;

	private final int numberFolks;

	public BathroomController(int numberFolks) {

		this.numberFolks = numberFolks;
		this.threadArr = new Thread[numberFolks];

		Bathroom bathroom = new Bathroom();

		for (int I = 0; I < numberFolks; I++) {
			threadArr[I] = new Thread(new FamilyMember(bathroom));
		}

		Bailey bailey = new Bailey(bathroom);
		threadBailey = new Thread(bailey);
	}

	public void startThreads() {
		for (int I = 0; I < numberFolks; I++) {
			threadArr[I].start();
		}
		threadBailey.start();
	}
}
```

We execute a run.

1645824806618 Thread: 13 Start My first morning task: Do my morning chores
1645824806618 Thread: 15 Start My first morning task: Do my morning chores
1645824806618 Thread: 14 Start My first morning task: Do my morning chores
1645824806619 Thread: 16 Bailey walks to coffee shop with the bathroom key!
1645824808006 Thread: 13 Completed My first morning task: Do my morning chores
1645824808006 Thread: 13 Start Have breakfast
1645824808014 Thread: 14 Completed My first morning task: Do my morning chores
1645824808014 Thread: 14 Start Have breakfast
1645824808017 Thread: 15 Completed My first morning task: Do my morning chores
1645824808017 Thread: 15 Start Have breakfast
1645824809339 Thread: 13 Completed Have breakfast
1645824809343 Thread: 14 Completed Have breakfast
1645824809349 Thread: 15 Completed Have breakfast
1645824832595 Thread: 16 Bailey back from coffee shop about to return the key
1645824832595 Thread: 16 Start My first morning task: Do my morning chores
1645824832595 Thread: 15 begin Shower
1645824833910 Thread: 15 completed Shower in 1315 ms
1645824833910 Thread: 15 Start My last morning task: Get dressed
1645824833910 Thread: 14 begin Shower
1645824833916 Thread: 16 Completed My first morning task: Do my morning chores
1645824833916 Thread: 16 Start Have breakfast
1645824835255 Thread: 15 Completed My last morning task: Get dressed
1645824835257 Thread: 16 Completed Have breakfast
1645824835258 Thread: 14 completed Shower in 1348 ms
1645824835258 Thread: 14 Start My last morning task: Get dressed
1645824835258 Thread: 13 begin Shower
1645824836565 Thread: 13 completed Shower in 1307 ms
1645824836565 Thread: 16 begin Shower
1645824836565 Thread: 13 Start My last morning task: Get dressed
1645824836565 Thread: 14 Completed My last morning task: Get dressed
1645824837871 Thread: 16 completed Shower in 1306 ms
1645824837871 Thread: 16 Start My last morning task: Get dressed
1645824837872 Thread: 13 Completed My last morning task: Get dressed
1645824839157 Thread: 16 Completed My last morning task: Get dressed

Bailey held things up for 26 seconds - 1645824832595 - 1645824806619 = 25976

The end-to-end time for TWO runs.

> Case one: with the lock and without Bailey
> Elapsed time 1645821335217 – 1645821327270 = 7947
> Average shower time 1316
>
> Case two: with the lock and with Bailey
> Elapsed time 1645824839157 – 1645824806618 = **32539**
>
> 25976 + 7947 + 1316= **35239** amazing!!!!

Controlling the uncooperative family member: private synchronizing

Here we enhance our code and prevent Bailey from disrupting our morning routine. All code stays the same except for our bathroom class which now will protect itself privately (self-defense!). We do **not** change Bailey.

```
package bathroom;

import Log.Logger;
import simulation.CpuHog;

public class Bathroom {

	private final Object myLock = new Object();

	public  void takeShower() {
		synchronized(myLock) {
			Logger.log("Thread: " + Thread.currentThread().getId() + " begin Shower");
			long start = System.currentTimeMillis();
			// use up some cpu to slow things down.. to simulate time taken to shower
			CpuHog.doIt();
			long done = System.currentTimeMillis();
			Logger.log("Thread: " + Thread.currentThread().getId()+" completed Shower in”
			    + (done-start)  + " ms");
		}
	}
}
```

We execute a run.

1645826064219 Thread: 14 Start My first morning task: Do my morning chores
1645826064219 Thread: 15 Start My first morning task: Do my morning chores
1645826064221 Thread: 13 Start My first morning task: Do my morning chores
1645826064251 Thread: 16 Bailey walks to coffee shop with the bathroom key!
1645826065686 Thread: 14 Completed My first morning task: Do my morning chores
1645826065686 Thread: 14 Start Have breakfast
1645826065704 Thread: 15 Completed My first morning task: Do my morning chores
1645826065704 Thread: 15 Start Have breakfast
1645826065706 Thread: 13 Completed My first morning task: Do my morning chores
1645826065706 Thread: 13 Start Have breakfast
1645826067028 Thread: 15 Completed Have breakfast
1645826067028 Thread: 15 begin Shower
1645826067031 Thread: 14 Completed Have breakfast
1645826067034 Thread: 13 Completed Have breakfast
1645826068336 Thread: 15 completed Shower in 1308 ms
1645826068336 Thread: 15 Start My last morning task: Get dressed
1645826068336 Thread: 13 begin Shower
1645826069671 Thread: 13 completed Shower in 1335 ms
1645826069671 Thread: 14 begin Shower
1645826069671 Thread: 13 Start My last morning task: Get dressed
1645826069672 Thread: 15 Completed My last morning task: Get dressed
1645826071010 Thread: 14 completed Shower in 1338 ms
1645826071010 Thread: 14 Start My last morning task: Get dressed
1645826071010 Thread: 13 Completed My last morning task: Get dressed
1645826072318 Thread: 14 Completed My last morning task: Get dressed
1645826090291 Thread: 16 Bailey back from coffee shop about to return the key
1645826090291 Thread: 16 Start My first morning task: Do my morning chores
1645826091573 Thread: 16 Completed My first morning task: Do my morning chores
1645826091573 Thread: 16 Start Have breakfast
1645826092855 Thread: 16 Completed Have breakfast
1645826092855 Thread: 16 begin Shower
1645826094136 Thread: 16 completed Shower in 1281 ms
1645826094136 Thread: 16 Start My last morning task: Get dressed
1645826095418 Thread: 16 Completed My last morning task: Get dressed

1645826072318 - 1645826064219 = 8099ms

So, all the family members except Bailey complete their routine in 8099ms.

Notice: Bailey logged incorrectly! Bailey did not have the bathroom key!!!

Storm tracker POJO concerns

We are in the middle of a massive storm – a Nor'easter!

Class: Storm Tracking Data. This class is a POJO. Tracking a storm is complicated and requires a bunch of data. The class conforms to Java Bean standard: all members are private, and each member has s setter and getter pair etcetera. The class implements the Storm Tracking Read Only Data interface (see below). The actual related data changes over time as the storm location and conditions constantly change. We decide that our application will calculate a new set of data every 15 minutes. The Storm Tracking Data is available to reader threads via the get latest method. We do **NOT** want reader threads to have write access to the underlying Storm Tracking Data

Rule: readers cannot change the data

Class: Storm Track Calculator. The class method calculate data is invoked by a controlling class every 15 minutes

```
public class StormTrackCalulator implements StormTrackingReadOnlyData {

        private StormTrackingData dataPacket = null;
         void init(){
              calculateData();
         }
         void calculateData() {
            get input required data from external sources and sensors etc.
            perform calulations
            change  state of our dataPacket with the new results of the calculations using setters
         }
         public StormTrackingReadOnlyData getLatest() {
                  StormTrackingReadOnlyData returnData = dataPacket;
                  return returnData;
         }
}
```

Storm Tracking Read Only Data is an interface.

```
public interface StormTrackingReadOnlyData {
        Include here ALL of the StormTrackingData getters methods. Some examples…
         public int getHumidity();
         public double getBarometricPressure();
         public double getDewPoint();
         public WindData getWindData();
         etc…
}
```

Wind Data is a user defined java class that contains all wind related data. ALARM! This is a problem waiting to happen!

The above design is a mess. It has so many issues that we may need a long int just to count them.

Issue one

Timeline. Here the calculation takes **about 2 minutes**

Time	Controller	Reader A	Reader C
10:00	Invokes init		
10:02	Starts 3 reader threads		
10:03		get latest	
		latest.setHumidity(87)[1]	
			get latest
			WindData wd = latest.getWindData()[2]
			wd.setMaxWindGust(120);

All readers share the same Storm Tracking Data object- with the 10:00 related data

1: Our read only interface prevents reader A from setting (changing) the humidity. It will **not** compile!

2: However, reader C is allowed to break our rule regarding readers. The interface allows the line:

```
WindData wd = latest.getWindData().
```

Now, once the reader has a reference to the wind data it can invoke the wind data setters – here it changes the maximum wind gust. The problem is that change affects reader A (all current readers) – all readers share the same Storm Tracking Data object.

Issue two

Timeline

Time	Controller	Reader A
10:00	Invokes init	
10:02	Starts N reader threads	
10:15	Start: calculate Data	
10:16		get Latest (the ‘‘grab’)
10:17	Completed: calculate Data	

Issue. Our controller is in the middle of refreshing its private member Storm Tracking Data via its setters; during the refresh seters are invoked with refreshed data. At various points of time our data packet has data from the 10:00 run (not refreshed yet) **and** data from the current 10:15 run; the data is temporarily in an inconsistent state. The reader(s) in the meantime can grab a reference to the data – not good!

Can we synchronize like the following? Yes but we choose **not** to!

```
public class StormTrackCalulator implements StormTrackingReadOnlyData {

        private StormTrackingData dataPacket = null;

        void synchronized calculateData() {
            input required data from external sources and sensors
            perform calulations ➔ results
            change  state of dataPacket with new results via the StormTrackingData setters
        }

        public synchronized StormTrackingReadOnlyData getLatest() {
                StormTrackingReadOnlyData returnData = dataPacket;
                return returnData;
        }
}
```

This not acceptable. While a calculate Data invocation is executing all reader calls to get Latest will be held up for up to 2 minutes.

Switching

Alternative. Create a new data packet and after its state is set and consistent (all setter invoked) then SWITCH

```
public class StormTrackCalulator implements StormTrackingReadOnlyData {

        private StormTrackingData dataPacket = null;

        void calculateData() {
            input required data from external sources and sensors
            perform calulations
            StormTrackingData newDataPacket = new StormTrackingData()
            Call newDataPacketsetters with the new results of the calculations
            dataPacket = newDataPacket  // SWITCH
        }

        public StormTrackingReadOnlyData getLatest() {
                StormTrackingReadOnlyData returnData = dataPacket;
                return returnData;
        }
}
```

Do we need to add a private lock and **synchronize**? No. The following is **NOT** necessary for thread safety.

```
void calculateData() {
        ….
        synchronized(privateLock{
            dataPacket = newDataPacket  // SWITCH
        }
        …
}

public StormTrackingReadOnlyData getLatest() {
        StormTrackingReadOnlyData returnData;
        synchronized(privateLock{
             returnData = dataPacket;
        }
        return returnData;
}
```

If we synchronize then readers are competing to get the lock. The lock will by unlocked quickly but there would be a performance hit if there are frequent readers.

Alas!, there is NO issue here. We do NOT need the lock at all!. Get rid of it!

Google this “java assignments thread safety”

I found this-> https://www.oreilly.com/library/view/java-performance-tuning/0596000154/ch10s06.html

Cloning

Back to **Issue two** which we have not resolved. Remember, the issue was demonstrated earlier when a reader was able to change the Wind Data.

We can design our readers so that they cooperatively do not change the data. In this case, we do NOT programmatically prevent them; we depend on the readers being cooperative (like our family members showering in the morning).

One way to programmatically prevent readers from messing up other readers is to give each reader its own copy (clone) of the data. Of course, there is a performance and memory impact. In a 'close call' case the *dataPacketToClone* may be either one of two packets; this is when a reader reads just as we are performing our 15-minute refresh followed by our SWITCH! This is ok!

```
void calculateData() {
                //…
                // SWITCH
                dataPacket = newDataPacket
        }

public StormTrackingReadOnlyData getLatest() {
                StormTrackingData dataPacketToClone = this.dataPacket;
                StormTrackingData returnData = clone(dataPacketToClone)
                return returnData;
}

private StormTrackingData clone(StormTrackingData clonee){
                // etc…
}
```

Remember the two object reference java assignments above are thread-sage.

Miscellaneous synchronization examples and observations

Here we look at coding examples. In each case multiple running (concurrent) threads have access (have a reference) to an object of the class XXX.

Compare these two implementations of class XXX.

Example One

```
public class XXX {
        public  synchronized double xxx() {
                double result = 0;
                //  calculate result
                return result;
        }
}
```

```
public class XXX {
        public double xxx() {
                synchronized (this) {
                        double result = 0;
                        //  calculate result
                        return result;
                }
        }
}
```

Example two

Here **two** public methods are synchronized. The object's one monitor lock must be obtained before entering the methods. Our two methods cannot run concurrently. Is this the behavior you want? -- see example three just below

```
public class XXX {

        public  synchronized double xxx() {
                double result = 0;
                //  calculate result
                return result;
        }

        public  synchronized double yyy() {
                double result = 0;
                //  calculate result
                return result;
        }
}
```

Example three

Compare this example to **example two.**

```
public class XXX {
        private Object xxxLock = new Object();
        private Object yyyLock = new Object();

        public  double xxx() {
                double result = 0;
                synchronized (xxxLock) {
                        //  calculate result
                }
                return result;
        }

        public double yyy() {
                double result = 0;
                synchronized (yyyLock) {
                        //  calculate result
                }
                return result;
        }
}
```

Note: If your class uses multiple locks you may consider a separate class for each function, i.e., write multiple classes. Are the functions related? If the classes share functionality consider an abstract base class. Always follow the KISS rule.

Example four

This is a horror show. Notice the temporal ordering of obtaining the locks. **NEVER** do this! This is a **DEADLOCK** waiting to happen!.

```
public class XXX {
        private Object xxxLock = new Object();
        private Object yyyLock = new Object();

        public void xxx() {
                // ...
                synchronized (xxxLock) {
                    // ...
                        synchronized (yyyLock) {
                                //...
                        }
                }
        }

        public void yyy() {
                // ...
                synchronized (yyyLock) {
                    // ...
                        synchronized (xxxLock) {
                                //...
                        }
                }
        }
}
```

Example five

This is fancy BUT it will NOT cause a dreaded DEADLOCK.

```
public class XXX {
        private Object xxxLock = new Object();
        private Object yyyLock = new Object();

        public void xxx() {
                // ...
                synchronized (xxxLock) {
                        // ...
                        synchronized (yyyLock) {
                                //...
                        }
                        //...
                }
                //...
        }

        public void yyy() {
                // ...
                synchronized (yyyLock) {
                        // ...
                }
                //...
        }
}
```

Static Synchronization

An object (instance) has access to TWO monitors. Each instance has its own 'instance monitor' – we have been using this monitor in our previous examples. Additionally, all instances of a class share a second common 'static monitor.'

Here the class static monitor 'protects' the instance Counter. An instance of XXX has its **own** synchronized monitor - no sharing.

```
public class XXX {

        private static int instanceCounter = 0;

        private synchronized static void incrementInstanceCount() {
                instanceCounter = instanceCounter + 1;
        }

        public synchronized static int getInstanceCount() {
                return (instanceCounter);
        }

       // constuctor
        public XXX() {
                XXX.incrementInstanceCount();
        }

        public synchronized void doSomethingSynched() {

        }

       public synchronized void doSomethingElseSynched() {

        }
}
```

DB connection example

In this example we have multiple worker threads. A worker does work and then persists the work results to a DB.

```
public class Worker implements Runnable {

        protected final DBConnection conn;


        public Worker (DBConnection conn) {
                this. conn = conn;
        }

        @Override
        public void run() {
              // do a bunch of work
               WorkResulst results = ….
               // use conn to send results transactions to DB
               performTransaction(results)
         }

         private void performTransaction(WorkResulst results){
            try{
                conn.beginTran();
                 use conn to do a bunch of inserts and updates etc
                 conn.commitTran()
             } catch (Exception E){
                log the issue
                 conn.rollback();
             }
        }
}
```

A controller creates and starts the workers. Workers depend on the controller for an instance representing a DB connection.

Here the controller creates N workers and N DB connections. Each worker gets their **own connection**. This is end-to-end concurrency; workers can both work and perform the DB transactions concurrently. This is like having each family member having their own bathroom – heaven!

```
public class WorkController {

        private Thread[] threadArr;
        private final int numberWorkers;

        public WorkController (int numberWorkers) {

                this. numberWorkers = numberWorkers;
                this.threadArr = new Thread[numberWorkers];

                for (int I = 0; I < numberWorkers; I++) {
                        threadArr[I] = new Thread(new Worker(new DBConnection());
                }
        }

        public void startThreads() {
                for (int I = 0; I < numberWorkers; I++) {
                        threadArr[I].start();
                }
        }
}
```

Now we discover the unwelcome news. Suppose there is a system constraint with which you (the designer) must live. Your masterpiece is allocated a single DB connection. You must redesign the above!

Sharing a single DB connection

We carry on. Here is our new controller. There is now a single shared DB connection.

```
public class WorkController {

        private Thread[] threadArr;
        private final int numberWorkers;

        public WorkController (int numberWorkers) {

                this. numberWorkers = numberWorkers;
                this.threadArr = new Thread[numberWorkers];
                DBConnection sharedConn = new new DBConnection();
                for (int I = 0; I < numberWorkers; I++) {
                        threadArr[I] = new Thread(new Worker(sharedConn));
                }
        }

        public void startThreads() {
                for (int I = 0; I < numberWorkers; I++) {
                        threadArr[I].start();
                }
        }
}
```

Our Worker needs a change. Multiple workers can no longer concurrently access the DB connection; the connection is now shared.

The following – of course - will **NOT** work! Each worker has its own monitor!

```
public void run() {
      // do a bunch of work
       WorkResulst results = ….
       Synronized (this){
          // use conn to send results transactions to DB
          performTransaction(results);
       }
 }
```

We work on!

Solution

Create a private lock like our bathroom – except here the lock must be static - shared by all worker instances. The workers 'know' that the shared resource connection is not (may not be) thread safe and so the workers take charge and force the DB work to be safe! Kudos to the workers!

```
public class Worker implements Runnable {
        private final  static Object staticPrivateLock = new Object()
        protected final DBConnection conn;

public void run() {
            // do a bunch of work
             WorkResulst results = ….
              synronized (Worker.staticPrivateLock){
                  // use conn to send results transactions to DB
                 performTransaction(results);
              }
        }
etc…
```

Mutex implements the following interface.

```
public void lock() throws Exception
public void unlock()
```

Note that our DB transactions are accomplished by a bunch of java **statements**. The group of statements is surrounded by a lock

```
Synchronized (XXX){    // lock monitor
   Perform java DB related statements
}  // unlock monitor
```

Something to be on the 'look for.' The running thread may experience a 'context switch' right in the middle of the **statements**. The thread is no longer running BUT has/keeps hold of the lock. As a rule, keep the code within a locked block as short and fast as possible. Unlocking quickly gives other threads quicker access to the lock and lessens the chances of a thread experiencing a 'context switch' while holding the lock.

So, KISS and KIQ - keep it simple AND keep it quick!

Two timelines

Example timeline where each worker has its own connection

Worker A	**Worker B**	**Worker C**	**Worker D**
Work: start		Work: start	
	Work: start		
Work: done	Work: done		
Send results to DB: start	Send results to DB: start	Work: done	
Send results to DB: done		Send results to DB: start	Work: start
	Send results to DB: done		Work: done
			Send results to DB: start
		Send results to DB: done	

Example Timeline with our constraint. The workers share a single connection.

Worker A	**Worker B**	**Worker C**	**Worker D**
Work: start		Work: start	
	Work: start		
Work: done	Work: done		
Send results to DB: start		Work: done	
Send results to DB: done			Work: start
	Send results to DB: start		Work: done
	Send results to DB: done		
		Send results to DB: start	
		Send results to DB: done	
			Send results to DB: start

Waiting in a synchronized block

The synchronized mechanism allows a thread that currently holds a monitor to enter a wait state.

Here is an example timeline for the simple case of TWO threads; a waiter (Thread A) and one notifier (Thread B). Both treads have a reference to a shared single Object xxx;

Thread A acquires the xxx monitor (enters the synched block)

Thread A calls the wait method in the synch block; this releases the hold on the monitor and puts the thread it a wait state (it stops running)

Thread B acquires the same xxx monitor and (while holding it) invokes the xxx notify method

Thread A reacquires the monitor and proceeds at the line of code right after the wait call

Note:

> Temporal ordering issue: The Thread B notify **MUST** be executed AFTER the Thread A wait! Such required timing (wait before notifying) may be tricky in a multi-threaded system.
>
> A thread that signals may:
>
> - Hope that another thread(s) is waiting
> - Know that another thread(s) is waiting
> - Send a signal **in case another** thread(s) is waiting – this is what our Mutex class will do! (more later)
> - Send a signal too early – there are NO waiting thread(s) now
> - Etcetera – you get the point!

We will explore the synchronized wait and notify mechanism again later in this article when we create our *h*ome brewed Mutex class. In the meantime, we took a look..

An example

Our worker thread waits in its run method until notified and then proceeds to do its work. We call the shared single Object the 'work Signal.'

Worker thread with a bunch of logging follows. The worker acquires the monitor and waits before doing its work.

The main thread creates and starts a worker. We pause in main via the call System.in.read() to simulate time passing before we signal our worker.

```
package workers;

import Log.Logger;

public class Worker implements Runnable {
	private final String myName;
	private final Object workSignal;
	public Worker(String name, Object workSignal) {
		myName = name;
		this.workSignal = workSignal;
	}

	@Override
	public void run() {

		Logger.log( myName + " Begin run. About to enter synch block ");
		 /// can do a bunch of pre-work stuff here before waiting

		synchronized (workSignal) {
			Logger.log(myName +
			    " In synch block.  About to wait for signal to start working");

			try {
				workSignal.wait();
			} catch (Exception e) {
				Logger.log(myName + " In synch block.  Wait was interrupted");
			}

			Logger.log(myName +
			     " In synch block.  About to return from wait.  Now can do some work...");
		}

		Logger.log(myName + " Exited synch block and run()");
	}
}
```

Controller

```
package resource;

import java.io.IOException;
import Log.Logger;
import workers.Worker;

public class WorkerController {
	private Thread[] workerThread;
	private final int numWorkers;
	private final  Object workSignal;

	public WorkerController(String[] workerNames) {
		workSignal = new Object();
		numWorkers = workerNames.length;
		workerThread = new Thread[numWorkers];
		for (int I = 0; I < numWorkers; I++) {
			workerThread[I] = new Thread(new Worker(workerNames[I], workSignal));
		}
	}

	public void startThreads() {
		Logger.log( "Work Controller: starting workers.  Count = " + numWorkers);
		for (int I = 0; I < numWorkers; I++) {
			workerThread[I].start();
		}
		Logger.log( "Work Controller: started " + numWorkers+ " workers.");
		Logger.log("Work Controller: pause before sending signal...  Hit enter to Signal");

		try {
			System.in.read();    // Stall before sending signal
		} catch (IOException e1) {
		}

		synchronized(workSignal) {
			workSignal.notify();
		}

		Logger.log("Work Controller sent signal and completed");
	}
}
```

A test run with ONE worker – Amy!

```
package appControl;

import Log.Logger;
import resource.WorkerController;

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

		String workerNameArr[] = {"Amy"} ;

		Logger.log("Main thread starting");
		WorkerController workerController = new WorkerController(workerNameArr);
		workerController.startThreads();
		Logger.log("Main thread started threads");

	}
}
```

A test run with ONE worker. I paused for about 8 seconds before allowing the controller to send signal.

1645298119830 Main thread starting
1645298119831 Work Controller: starting workers. Count = 1
1645298119831 Work Controller: started 1 workers.
1645298119831 Work Controller: pause before sending signal... Hit enter to Signal
1645298119831 Amy Begin run. About to enter synch block
1645298119832 Amy In synch block. About to wait for signal to start working
1645298128426 Work Controller sent signal and completed
1645298128426 Main thread started threads
1645298128426 Amy In synch block. About to return from wait. Now can do some work...
1645298128426 Amy Exited synch block and run()

Elapsed time: 1645298128426 – 1645298119832 = 8,594

We continue our example by adding more workers – total of three.

Main

```
Change this line

String workerNameArr[] = {"Amy"} ;

To this line

String workerNameArr[] = {"Amy", "Sammy", "Moe"} ;
```

A test run with THREE workers.

1645298429897 Main thread starting
1645298429898 Work Controller: starting workers. Count = 3
1645298429898 Work Controller: started 3 workers.
1645298429898 Work Controller: pause before sending signal... Hit enter to Signal
1645298429898 Sammy Begin run. About to enter synch block
1645298429898 Moe Begin run. About to enter synch block
1645298429898 Moe In synch block. About to wait for signal to start working
1645298429899 Sammy In synch block. About to wait for signal to start working
1645298429899 Amy In synch block. About to wait for signal to start working
1645298436665 Work Controller sent signal and completed
1645298436665 Main thread started threads
1645298436665 Moe In synch block. Returned from wait. Now can do some work...
1645298436665 Moe Exited synch block and run()

This did **NOT** work. All three workers entered a wait state. Our controller then sent out ONE notify which - in this case – signaled Moe. Amy and Sammy are left waiting!

Solution: we can have our controller send three notifies OR we can use the synchronized mechanism notify all method. We try the later.

Controller

Change this line

```
synchronized(workSignal) {
                workSignal.notify();
        }
```

To this line

```
        synchronized(workSignal) {
                workSignal.notifyAll();
        }
```

A successful test run with Three worker. The three threads all waited and were signaled.

1645299591636 Main thread starting
1645299591637 Work Controller: starting workers. Count = 3
1645299591637 Work Controller: started 3 workers.
1645299591637 Work Controller: pause before sending signal... Hit enter to Signal
1645299591637 Amy Begin run. About to enter synch block
1645299591638 Amy In synch block. About to wait for signal to start working
1645299591637 Sammy Begin run. About to enter synch block
1645299591638 Sammy In synch block. About to wait for signal to start working
1645299591638 Moe Begin run. About to enter synch block
1645299591638 Moe In synch block. About to wait for signal to start working

1645299599774 Work Controller sent signal and completed
1645299599774 Main thread started threads
1645299599774 Amy In synch block. Returned from wait. Now can do some work...
1645299599774 Amy Exited synch block and run()
1645299599774 Moe In synch block. Returned from wait. Now can do some work...
1645299599774 Moe Exited synch block and run()
1645299599774 Sammy In synch block. Returned from wait. Now can do some work...
1645299599774 Sammy Exited synch block and run()

The Mutex class

Constructors	
Constructor and Description	
Mutex() Constructs a Mutex.	

Method Summary

Methods	
Modifier and Type	**Method and Description**
`void`	lock() Acquires ownership (locks the Mutex) of the Mutex. The invoking Thread waits if another Thread currently holds the lock
`void`	unlock() Releases ownership (unlocks the mutex) of the mutex **if** the current owner is the invoking Thread

Home Brewed Mutex

Earlier we examined the synchronize wait and notify mechanism. We now discuss a Mutex which uses this mechanism. A mutex can be locked and unlocked. A lock indicates ownership of the mutex – an owner is always a thread.

We start with an example mutex usage timeline. This should remind you of our family showering routine. In the example we have a single instance of class Mutex. A mutex can be locked (i.e., obtained or owned) and can be unlocked. A mutex owner is a thread. At a point of time, a mutex has one or zero owners.

- Thread A calls the lock method and obtains the mutex. A is the current 'owner' (A has the bathroom key)
- Thread B calls the lock method and waits.
- Thread C calls the lock method and waits.
- Thread A calls the unlock method to release the lock causing a signal to be generated. Note: we have TWO waiting threads
- Thread B wakes and grabs the lock. C continues to wait.
- Thread B calls the unlock method to release the lock causing a signal to be generated.
- Thread C wakes up and grabs the lock.
- Thread C calls the unlock method to release the lock. The signal is generated and ignored as there are NO waiting threads <- this is perfectly fine!

Version one. When a thread is waiting to obtain the Mutex the thread may be **interrupted**. Note, the 'chance' and possibility of a thread being interrupted is up to you (the designers). Your code MUST explicitly interrupt a running thread where the interrupt is issued from a second thread. I personally never interrupt!

```
package threadSynch;
public class Mutex {
	private Thread owner = null;
	private Object privateSignal;

	public Mutex() {
		privateSignal = new Object();
	}

	public void lock() throws InterruptedException {
		synchronized(privateSignal) {
			Thread requester = Thread.currentThread();
			if (owner == null) {
				owner = requester;
				return;
			}
			if (owner == requester) {
				return;
			}
			while (owner != requester) {
				try {
					privateSignal.wait();
				}
				catch (InterruptedException E) {
					throw E;
				}
				catch (IllegalMonitorStateException E) {
				}

				if (owner == null) {  //  Point A
					owner = requester;  // obtain the lock
				}
			}
		}
	}
```

```
        public void unlock() {
                synchronized(privateSignal) {
                        if (owner == Thread.currentThread()) {
                                owner = null; // release lock
                                try {
                                        privateSignal.notify();
                                } catch (IllegalMonitorStateException  e) {
                                }
                        }
                }
        }
}
```

Note re the above code. A change. The signaling thread (who unlocks) signals ONE waiting thread which will wake up from its **wait**. Before signaling the signaler first assigns the **owner** member variable to null. The only way a waiter could find a non-null owner upon waking up (see Point A) is if the wait caused an exception to be thrown!

Compare the following with **version one** (above) and **version two** below!

```
                if (owner == requester) {
                        return;
                }

                while (owner != requester) {  //
                        try {
                                privateSignal.wait();
                        }
                        catch (InterruptedException E) {
                                throw E;
                        }
                        catch (IllegalMonitorStateException EE) {
                                throw EE
                        }

                        if (owner == null) {   // Point A
                                owner = requester;  // obtain the lock
                        } //  //
                } //  // while loop
```

Note: Keeping the **while loop** and the check at Point A may be a clever idea!!!

Version two: ignores any interrupts while waiting!

```
package threadSynch;
public class Mutex {
	private Thread owner = null;
	private Object privateSignal;

	public Mutex() {
		privateSignal = new Object();
	}

	public void lock() throws InterruptedException {
		synchronized(privateSignal) {
			Thread requester = Thread.currentThread();
			if (owner == null) {
				owner = requester;
				return;
			}
			if (owner == requester) {
				return;
			}
			while (owner != requester) {
				try {
					privateSignal.wait();
				}
				catch (Exception E) {
					//    ignore E
				}

				if (owner == null) {  // Point A
					owner = requester;  // obtain the lock
				}
			}
		}
	}
```

```
        public void unlock() {
                synchronized(privateSignal) {
                        if (owner == Thread.currentThread()) {
                                owner = null; // release lock
                                try {
                                        privateSignal.notify();
                                } catch (IllegalMonitorStateException  E) {
                                        //
                                }
                        }
                }
        }
}
```

Notes re exception handling.

The lock method

In general, the java **wait** method throws two Exceptions

- Illegal Monitor State Exception − if the current thread is not the owner of the object's monitor.
- Interrupted Exception − if another thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Our Mutex logic first obtains ownership of the private Signal's monitor via the synchronized mechanism. It also checks that the invoking thread is indeed the mutex owner. So, the thread owns both the monitor and the mutex. It ignores the Illegal Monitor State Exception. **Version one** throws (back to the caller) the Interrupted Exception. *Version two* ignores the Interrupted Exception. The mutex is dumb – it does not (should not) know what to do when interrupted.

The unlock

In general, the java notify method throws one Exception

- Illegal Monitor State Exception − if the current thread is not the owner of the object's monitor

Here again, our Mutex logic first obtains ownership of the private Signal's monitor. It also checks that the invoking thread is indeed the mutex owner. So, the thread owns both the monitor and the mutex. It ignores the Illegal Monitor State Exception

If when unlocking, our Mutex logic detects that the invoking thread is NOT the mutex owner, it simple does nothing! The Mutex could throw an exception to an uncooperative thread who tries to unlock a mutex that they do not own; this implementation does NOT do this. If it did throw the exception the cooperative threads would also need to check for the exception – an 'unfair' burden.

Version three: I change the lock method and rename it.

```
public boolean acquireLock() {

        synchronized(privateSignal) {
            Thread requester = Thread.currentThread();
            if (owner == null) {
                owner = requester;
                return;
            }
            if (owner == requester) {
                return;
            }
            while (owner != requester) {
                try {
                    privateSignal.wait();
                }
                catch (Exception E) {
                    return false;
                    break;
                }

                if (owner == null) {  // Point A
                    owner = requester;  // obtain the lock
                } //
            } // end loop
        }
        return true;
    }
```

Pardon the interruption

A note about Interrupts: If your design calls for using interrupts then go right ahead – I dare you! It can make you code a bit tricky. One use is to atom bomb all your threads as part of an application shutdown. In any case, the Mutex wait may be interrupted. We keep the Mutex as dumb as possible.

Warning: opinion alert --> this may be considered a subjective option! !!! → NEVER use the interrupt!

A Mutex class test

The shared resource. A bunch of workers will share the resource that performs a unit of work (A thru F).

```
package resources;

public class SharedResource {

	public void  doIt() {

		System.out.println(Thread.currentThread().getId() + " A");
		System.out.println(Thread.currentThread().getId() + " B");
		System.out.println(Thread.currentThread().getId() + " C");

		double d = eatUpCPU();
		System.out.println(Thread.currentThread().getId() + " " +  d);
		System.out.println(Thread.currentThread().getId() + " D");
		System.out.println(Thread.currentThread().getId() + " E");
		System.out.println(Thread.currentThread().getId() + " F");

	}

	private double eatUpCPU() {
		double result = 0;
		int big = 10000000;
		double arr[] = new double[big];

		for (int I = 0; I < big ; I ++) {
			double x  =  Math.sin(I+.002) * Math.log(I+.004);
			x  = Math.cos(x+.009);
			arr[I] = x;
		}


		for (int I = 0; I < big ; I ++) {
			result = result + arr[I]/big;
		}

		return result;
	}
}
```

```
package workerThread;

import Log.Logger;
import resources.SharedResource;

public class Worker implements Runnable {

	private final SharedResource sharedResource;

	public Worker (SharedResource sharedResource) {
		this.sharedResource = sharedResource;
	}

	@Override
	public void run() {

		Logger.log("Worker " + Thread.currentThread().getId() + " Begin Work");
		try {
			sharedResource.doIt();
		} catch (Exception e) {
			Logger.log("Worker " + Thread.currentThread().getId()
				+ " Caught  work related exception: " + e.getMessage());
		}
		Logger.log("Worker " + Thread.currentThread().getId() + " Completed work");
	}
}
```

The controller creates one shared resource and provide it to the workers.

```
package workerThread;

import resources.SharedResource;

public class WorkerController {

	private Thread[] workerThread;
	private final int numWorkers=8;

	public WorkerController() {

		workerThread = new Thread[numWorkers];

		SharedResource sharedResource = new SharedResource();
		for (int I = 0; I < numWorkers; I++) {
			workerThread[I] = new Thread(new Worker(sharedResource));
		}
	}

	public void startThreads() {
		for (int I = 0; I < numWorkers; I++) {
			workerThread[I].start();
		}
	}

}
```

Main

```
package main;

import Log.Logger;
import workerThread.WorkerController;


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

		Logger.log("Main thread starting");
		WorkerController controller = new WorkerController();
		controller.startThreads();
		Logger.log("Main thread started threads");

	}
}
```

We perform a run and notice the concurrency. The threads are concurrently sharing the shared resource!

```
Main thread starting
Main thread started threads
Worker 14 Begin Work
14 A
14 B
14 C
Worker 13 Begin Work
13 A
13 B
13 C
Worker 15 Begin Work
15 A
15 B
15 C
Worker 17 Begin Work
17 A
17 B
17 C
etcetera
```

Suppose we do not want the above concurrency. We now introduce a Mutex so that the shared resource work is **NOT** concurrent; the threads execute the resource's work one thread at a time. If the work produces an exception we throw it back up.

```
package resources;
import threadSynch.Mutex;
public class SharedResource {
	private final Mutex mutex;
	public SharedResource (Mutex mutex){
		this.mutex = mutex;
	}
	public void  doIt() throws Exception {
		Exception exception = null;
		boolean gotLock = true;
		try {
			mutex.lock();
		} catch (Exception e) {
			gotLock = false;
		}

		if (gotLock) {
			try {
				System.out.println(Thread.currentThread().getId() + " A");
				System.out.println(Thread.currentThread().getId() + " B");
				System.out.println(Thread.currentThread().getId() + " C");
				double d = eatUpCPU();
				System.out.println(Thread.currentThread().getId() + " " +  d);
				System.out.println(Thread.currentThread().getId() + " D");
				System.out.println(Thread.currentThread().getId() + " E");
				System.out.println(Thread.currentThread().getId() + " F");
			} catch (Exception e) {
				exception = e;
			} finally {
				if (gotLock) {
					mutex.unlock();
				}
			}
		} else {
			System.out.println(Thread.currentThread().getId() +
			    " cannot work.  Failed to obtain Lock!!!");
		}
		if (exception != null) {
			throw exception;
		}
	}
```

```
        private double eatUpCPU() {
                double result = 0;
                int big = 10000000;
                double arr[] = new double[big];

                for (int I = 0; I < big ; I ++) {
                        double x  =  Math.sin(I+.002) * Math.log(I+.004);
                        x  = Math.cos(x+.009);
                        arr[I] = x;
                }

                for (int I = 0; I < big ; I ++) {
                        result = result + arr[I]/big;
                }

                return result;
        }
}
```

The controller controls things!

- The controller provides the mutex for protecting the *Shared Resource*.
- The controller provides the *Shared Resource* for the workers.

We add the mutex creation to the controller and use it to construct our shared resource.

```
package workerThread;

import threadSynch.Mutex;
import resources.SharedResource;

public class WorkerController {

	private Thread[] workerThread;
	private final int numWorkers=8;

	public WorkerController() {

		workerThread = new Thread[numWorkers];
		Mutex mutex = new Mutex();

		  SharedResource sharedResource = new SharedResource(mutex);
		  for (int I = 0; I < numWorkers; I++) {
			workerThread[I] = new Thread(new Worker(sharedResource));
		  }
	}

	public void startThreads() {
		  for (int I = 0; I < numWorkers; I++) {
			workerThread[I].start();
		  }
	}

}
```

Perform a run and notice the results of mutex. The threads do NOT concurrently execute the shared resource' work. Looks like the mutex is doing its job.

Subset of output.

```
Worker 13 Begin Work
13 A
13 B
13 C
Worker 14 Begin Work
Worker 15 Begin Work
Worker 16 Begin Work
Worker 17 Begin Work
Worker 18 Begin Work
Worker 19 Begin Work
Worker 20 Begin Work
13 -0.05361478989054746
13 D
13 E
13 F
Worker 13 Completed work
14 A
14 B
14 C
etcetera
```

We look at a work-related exception handling case. The worker throws an exception in the middle of doing its work!

We reduce the number of threads to three

```
public class WorkerController {

        private Thread[] workerThread;
        private final int numWorkers=3;
```

We add a dummy exception that is thrown in the middle of the work

```
public class SharedResource {
    …
        public void  doIt() throws Exception {
            …
            System.out.println(Thread.currentThread().getId() + " C");
            double d = eatUpCPU();
            if (d < 0) {
                throw new Exception ("Dummy force exception!!!!");
            }
```

The work should stop just after successfully executing the line:

```
System.out.println(Thread.currentThread().getId() + " C");
```

Perform a run

```
Main thread starting
Main thread started threads
Worker 13 Begin Work
Worker 14 Begin Work
13 A
Worker 15 Begin Work
13 B
13 C
14 A
14 B
Worker 13 Caught work-related exception: Dummy force exception!!!!
14 C
Worker 13 Completed work
15 A
15 B
Worker 14 Caught work-related exception: Dummy force exception!!!!
Worker 14 Completed work
15 C
Worker 15 Caught work-related exception: Dummy force exception!!!!
Worker 15 Completed work
```

The Mutex wait

Our Mutex class forces users to wait forever when attempting to obtain the lock via the Mutex lock method. If an uncooperative thread holds (is the owner of) the lock and does NOT release it then another thread(s) waits forever! In the lock method we coded a 'forever wait.' – the wait method has no timeout argument.

```
try {
privateSignal.wait();
}
catch (Exception E) {
        throw (E);
}
```

We could include a time out on the wait. There are folks have the opinion that a thread should NEVER wait forever! Warning: opinion alert --> I disagree!

Let say the Mutex constructor included a long TIMEOUT argument. The Mutex could make sure the TIMEOUT is ZERO (wait forever) or greater than ZERO and use it in the wait. We will revisit this in a later chapter.

```
// In constructor validate the TIMEOUT input
If (TIMEOUT < 0) TIMEOUT = 0;

try {
        privateSignal.wait(TIMEOUT);
        // Was I signaled or did my wait timeout?
}
catch (Exception E) {
// I am a simple Mutex I do NOT  know what to do! So I throw
         throw (E);
}
```

What is the Mutex logic do if there is a timeout? Does the Mutex even know a timeout occurred (versus being signaled). Is it the Mutex's responsibility to handle the timeout case? …or pass the situation up to the invoker?

Notice what our Mutex does after the wait completes. It checks (an extra check!) to see if it can become the owner else it goes back to waiting – it is in a loop!! The Mutex cannot know if it woke up via signal or via a timeout. So why use the timeout at all? Well! there are situations where your application logic needs to know if a timeout occurred, i.e., where your application does NOT wait forever! Round and round we go!!

```
try {
        privateSignal.wait();
}
catch (Exception E) {
        throw (E);
}

if (owner == null) {  // Point A
        owner = requester;  // obtain the lock
}
```

If your design causes your system to wait forever (no signal is ever generated) AND this is a problem then maybe you should re-design your system.

Professionally, I have never used a Mutex with a time out.

We will revisit the Mutex implementation in a later chapter and include the time-out feature! I promise!

Home Brewed Conditional Variable

Now we get a bit more sophisticated; we introduce a Conditional Variable.

Class CondVar

Constructors

Constructor and Description
CondVar() Constructs a CondVar.

Method Summary

Methods

Modifier and Type	Method and Description
void	wait(Mutex mutex) Causes the invoking Thread to wait until ***a condition*** is satisfied, i.e., waits until the condition is true
void	**signal**() Signals a single waiting Thread that ***the condition*** has been satisfied, i.e., that the condition is true

The conditional variable implements two public interfaces

```
package threadSynch;

 public interface CondVarWaiter {
        public void wait(Mutex mutex);
}
```

```
package threadSynch;

public interface CondVarSignaler {
        public void signal();
}
```

This is another class (like our Mutex) where the public interface does NOT tell the whole story. The proper use of a conditional variable depends on the software engineer (you!) following certain design conventions. We discuss the conventions in the following pseudo code example.

Notice that the conditional variable's wait method has an argument of type Mutex. This is one advantage of making a lock into its own class – our Mutex class.

Conditional variable conventions.

- We have two groups of threads: signalers and waiters. There are one or more threads in a group.
- **All** the threads share
  1. a single resource
  2. a mutex
  3. a conditional variable.
- The mutex is cooperatively used to protect the shared resource - like our bathroom example.
- A waiter MUST provide a reference to the mutex when invoking the condition variable wait method. The mutex **MUST** be locked before invoking the wait method.

A pseudo code example

The above two interfaces and the list of conventions is enough to provide an example. We will code up the conditional variable after the example – I promise!

We consider seven classes

1. Work. This represents work that needs to be done
2. A single Work Store. Stores work objects. The class is NOT thread safe. The store has three methods:

    - public Work getWorkItem()
    - public void addWorkItem(Work work)
    - public Boolean isEmpty()

3. A single Mutex to protect the store.
4. A single Conditional Variable. This implements our two interfaces
5. Producer. A thread. This creates Work items and places them in the Work Store
6. Consumer. A Thread. This grabs (one at a time) a Work items from the Work Store and performs the work.
7. Controller. This ties everything together. It MUST follow the conditional variable conventions.

The controller logic

The controller has three private members

1. private final Work Store
2. private final mutex. This protects the store which is not thread-safe
3. private final ConditionalVar condVar

The controller has two methods

1. Method add work used by producer threads. The method add/inserts a work instance to the work store via the store's add work item method.
2. Method get work used by consumer threads. The method grabs a work item from the work store via the store's get work item method.

This logic is simple. The controller follows the convention: use the mutex to protect our shared resource – the store that is shared by N threads.

For threads who produce work – producers create work instances and the calls the controller's add work method; the work is added to the store. After a work item is safely in the store we **signal** any waiting consumers.

```
// Controller method to add work

public void addWork (Work work) {
        lock the mutex;
        store.addWortkItem(work);
        unlock the mutex
        condVar.signal();
}
```

The mutex protects the store. The purpose of the signal is to wake up ONE thread that is waiting for work to do. One or zero threads gets wake up. If there are NO *waiting* threads the signal is ignored. A thread (a Consumer) is *waiting* for work when it invokes the controller get work method (just below) AND the store happens to be empty at the time!

Consumers look for work; if there is no work they **wait** until there is! It uses the controller's get Work method.

```
// Controller method to get work

public void getWork {
        lock the mutex;
        while (store.isEmpty(){
                condVar.wait(mutex)
        }
        Work work = store. getWorkItem();
        unlock the mutex
        return work
}
```

This is where it gets a bit tricky. As my third-grade teacher would say "so! pay attention"

The convention: The controller locks the mutex when invoking any of the three store methods - is empty, get work item and add work Item. BUT!!! Notice that we pass a reference to our locked mutex to the conditional variable wait method. Looks dangerous BUT this is 'by design'; the controller is following the conditional variable conventions. The point is that the conditional variable wait method will eventually release the lock to allow signalers. It will release the lock when the 'time is right.' It regains the lock for the invoker before returning.

The conditional variable's wait and signal pseudocode logic without exception handling follows. It has two methods (see the two interfaces above). Here they are again.

1. public void wait(Mutex mutex)
2. public void signal()

Wait

The wait method expects the mutex to be locked (conditional variable convention). The wait method unlocks the mutex, but first obtains a lock on the second mutex – the conditional variable's private lock. The mutex is unlocked allowing other threads access to the store. These other threads can be producers and/or consumers. Unlocking the mutex is a MUST since the consumer invoking the wait method (via the controller's get work method) has encountered an empty store – it is waiting for a producer to add to the store; we allow both producers and consumers access to the store during the wait. Finally, the wait method restores the lock on the original mutex on behalf of the invoking consumer thread. You may want to revisit the controller methods above.

Note: re-ordering the wait method and unlocking the mutex before getting the private lock would be a design **flaw**! We discuss this a bit later.

Signal

This conditional variable method simply notifies (signals) via the conditional variables private lock. Who gets the signal ? One waiting consumer thread OR no thread – of course! Producers should always send a signal after adding work to the store.

The Conditional Variable implementation

As promised the conditional variable code. Here we use the Mutex version that does NOT throw Interrupted Exceptions

```
package threadSynch;
public class CondVar implements CondVarWaiter,CondVarSignaler {

	private final Object myPrivateLock;
	public CondVar() {
		myPrivateLock = new Object();
	}

	public void wait(Mutex mutex) {

		// Convention: we expect the Mutex to
		// be currently locked, i,e, held by the
		// current (invoking) thread

		synchronized(myPrivateLock) {
			mutex.unlock();

			try {
				myPrivateLock.wait();
			}
			catch (Exception E) {
				//
			}
		}

		mutex.unlock();
	}

	public void signal() {
		synchronized(myPrivateLock) {
			myPrivateLock.notify();
		}
	}
}
```

Producer and consumer example timeline

1. The producers invoke the controller add work method populating the store. No consumers are running.
2. The above producers continue adding work and a consumer starts.

At step (1) the producers compete for access to the store. The controller mutex protects the store and work items are safely added.

After step (2) the consumer – at a point – locks the mutex. The store is NOT empty, and the consumer removes one work item and unlocks the mutex.

3. The above producers continue adding work and a second consumer thread starts

After step (3) a consumer at a point in time locks the mutex. The store is NOT empty, and the consumer gets one work item and unlocks the mutex. If the consumers are faster (I doubt it!) then the producers, the store may become periodically empty – causing consumers to wait.

4. The above producers strop adding work and the consumer threads continue

After step (4) a consumer at a point in time locks the mutex. The store is NOT empty, and a consumers gets one work item at a time and unlocks the mutex allowing other consumers access to work. Eventually the store gets emptied, and all consumers get into the condition variable's wait. Nothing is happening.

5. A producer creates ONE work item.

The producer has access to the store since the conditional variable wait method always releases the controller mutex. So, the store NOW has one item. The producer calls (via the controller) the conditional variable signal. One waiting consumer wakes up and consumes the work item leaving the store empty again. The other consumers continue to wait

6. Producers all create work items.

The controller mutex keeps the store safe – one reader xor writer at a time. In the meantime, Java guarantees that conditional variable private lock synch, wait notify mechanism "works" and all is fine!

Analysis. The controller and the conditional variable work together cooperatively. One advantage of our Mutex class is that fact that it is a class! We can create (new) a Mutex instance and pass it around to various classes. Our controller creates the mutex and locks and unlocks it. It also sends the mutex reference to the conditional variable's wait method; it must first lock the mutex by convention.

The conditional variable logic involves two locks - the **mutex** and the conditional variable's private **monitor** lock. You may ask why the controller does not keep the mutex private and unlock it before calling the conditional variable wait method! The reason is that in the sequence of consumer events the private monitor lock MUST be obtained before the mutex lock is released – thread safety-ness demands it! In the wait method we have:

```
…
synchronized(myPrivateLock) {  // get monitorlock
        mutex.unlock();          // release mutex lock
        myPrivateLock.wait();
}
mutex.lock();             // re-obtain the mutex lock
…
```

Timelines.

When attempting to visualize multiple concurrent threads it is tempting to do something like this – here for three threads(A, B and C) and two available CPUs. Reading the table - top to bottom - represents time – row N occurs before row N+1

CPU One	**CPU two**
Thread A	**Thread B**
Task a1	Task b1
Task a2	Task b2
Task a3	Task b3
etcetera	etcetera

The above approach is too simplistic to be a valuable visual aid except for the simplest cases.. It assumes that the tasks ALL take the same amount of time – NO way!. Also , the above timeline does not consider context switches

We try again

	CPU One	**CPU two**
	Thread A	**Thread B**
	Task a1	Task b1
	Task a2	Task b2
	Task a3	
CPU One Context switch here…		
	Thread C	
	Task c1	Task b3
	etc.	etc.

Here, Task b2 takes more time to complete compared to Task a2. Also, after Thread A completes Task a3, Thread C takes over CPU One!

The timeline gets more complicated when threads share synchronized resources – like the shared Mutex in our consumer/producer example! Say a consumer has the mutex locked (owns it) and a second thread is waiting to obtain the mutex – in this case, the second thread is a prime target to lose the CPU - especially if a third thread is ready to take over the CPU!

It gets more complicated still as our consumer/produce example has TWO shared 'locks'

1. The controller's **mutex**
2. The conditional variable's private lock (**monitor**)

Example scenario: The store is initially empty. A **consumer** accesses these two locks in the following temporal order. Remember, locking may necessitate waiting.

A consumer thread timeline

Lock Access	***Java class***	***Notes and actions***
Lock mutex	Controller	Need access to store. Discovers that the store is empty
	Controller	Invoke Cond var wait method
Lock monitor	Cond var	Enter monitor synch block
Unlock mutex	Cond var	The store is unlocked
Unlock monitor	Cond var	Wait for a signal – this unlocks monitor - allows a producer to access the monitor
Lock monitor	Cond var	Wait ends. Got a signal - generated by a producer who added a **work item** to store and then signaled. The producer found both locks UNLOCKED
Lock the mutex	Cond var	The store is locked
Unlock monitor	Cond var	End of monitor synch block. Cond var wait method about to return
	Controller	Get **the above work item** from the store
Unlock Mutex	Controller	The store is unlocked

Another

	Lock Access	***Java class***	***Notes and actions***
1	Lock mutex	Controller	Need access to store. Discovers that the store is empty
2		Controller	Invoke Cond var wait method
3	Lock monitor	Cond var	Enter monitor synch block
4	Unlock mutex	Cond var	The store is unlocked
5			Producer adds a **work item** to store. The producer found one lock (the mutex) UNLOCKED – but cannot signal yet since monitor is locked
6	Unlock monitor	Cond var	Wait for a signal – this unlocks monitor - allows a producer to access the monitor
7	Lock monitor	Cond var	Wait ends. Got a signal - generated by a producer who added a **work item** to store and then signaled. The producer was waiting on monitor since step 5
8	Lock the mutex	Cond var	The store is locked
9	Unlock monitor	Cond var	End of monitor synch block. Cond var wait method about to return
10		Controller	Get **the above work item** from the store
11	Unlock Mutex	Controller	The store is unlocked

A simple example

Here we have one consumer thread and one producer thread.

Two work items will be processed. The two work items occur 20 minutes apart (things are slow!).

The first work item enters the system. The consumer and producer are in a race to get the lock on the store. The producer wants to add the work item to the store and the consumer wants to check the store for work. They race for the mutex.

Case one producer wins the race.

The producer has the mutex. The consumer waits for the mutex.

The producer adds the work item in the store and then releases the mutex lock. This allows the consumer access to the mutex lock.

The producer signals.

The consumer locks the mutex and sees one item in the store. It does NOT invoke the conditional variable's wait - the above signal is ignored which is OK!)

The consumer grabs the work item unlocks the mutex and processes it.

The consumer grabs the mutex and discovers an empty store. It invokes the conditional variable wait and waits for 20 minutes. Both mutex and monitor are unlocked

After 20 minutes the producer gets the mutex adds the second work item to the store unlocks the mutex and signals!

The consumer wakes up and…

Case two: consumer wins the race.

The consumer has the mutex. The producer waits for the mutex.

The consumer discovers an empty store and invokes the conditional variable wait. The mutex is still locked.

The conditional variables wait method takes over!

The private monitor is locked

The mutex is unlocked.

The consumer enters the monitors wait (this releases the monitor lock) – Both locks are no unlocked

When the mutex was unlocked this allowed the producer to grab the mutex and to add the work item to the store and unlocks the mutex. The producer is the able **to signal** since the monitor is also unlocked.

All locks are now unlocked.

The producer signal wakes up the consumer and it proceeds to process the work item.

The consumer proceeds and waits on the empty store.

A second work item arrives in 20 minutes with the consumer waiting. All locks are unlocked. The producer can proceed to add to the store and signal.

We plant a BUG

We now introduce an error in our logic. We switch the wait methods locking/unlocking order in the conditional variable's wait method.

```
public class CondVar implements CondVarWaiter,CondVarSignaler {

        private final Object myPrivateLock;
        public CondVar() {
                myPrivateLock = new Object();
        }

        public void wait(Mutex mutex) throws InterruptedException   {

                mutex.unlock();
                /// Interval XXX – our logic is in jeopardy here
                synchronized(myPrivateLock) {    // lock the monitor

                        mutex.unlock();

                        try {
                                myPrivateLock.wait();
                        }
                        catch (Exception E) {
                        }
                }

                try {
                        mutex.lock();
                }
                catch (InterruptedException  E) {
                        throw E;
                }

        }

        public void signal() {
                synchronized(myPrivateLock) {
                        myPrivateLock.notify();
                }
        }
}
```

Notice the window labeled **Interval XXX** - our logic fails here. The interval is deadly! Once the mutex is unlocked both locks are unlocked in the interval. A producer can grab the lock and proceed to signal. The jeopardy!

This timeline illustrates the issue. Three threads: a consumer, a produce and thread XX

CPU A	CPU B
Consumer has CPU	**Producer has CPU**
Lock mutex	
Discover empty store	Waiting for mutex
Unlock mutex Both locks are unlocked	
Consumer is in Interval XXX **Thread XX takes CPU**	Lock mutex
Thread XX is working	Adds work item to store. Store now has two items.
	Unlock mutex
	Lock monitor
Thread XX is working	Signal (A) -- OH NO! there is NO waiter to get the signal
Consumer takes CPU	Unlocks monitor
Invoke cond var wait method	
Lock monitor	
Wait for signal (unlocks monitor)	
	Disaster !! 20 minutes pass with no activity!
	Finally, work arrives!!!
	Locks mutex
	Adds work item to store. Store now has two items
	Locks mutex
	Locks monitor
	Signal (B)
	Unlocks monitor
Wait is over!!!!! Via signal (B) **The consumer waited 20 minutes**	

Make sure that you are convinced that the above issue CANNOT occur with the meticulously designed controller and conditional variable!

BUG Note

We introduce the above BUG as an example to help understand our locking logic. We had to concoct an elaborate timeline to demonstrate the effects of the bug because the BIG would occur (i.e., manifest itself) very infrequently in the 'real world' – the locking/unlocking logic would seldom cause a problem.

Is a bug (red) that occurs (manifests itself) very infrequently a 'better bug' than a bug (green) that manifests itself frequently?

Question: If you HAD to HAVE a bug which one would you prefer!? <- That is a joke!

The above question is a bit of a conundrum.

I prefer the green bug (the one that happens frequently!). Because of its frequency the bug will hopefully be detected in software testing, and it will be fixed! – there will NO bug in software release! The red bug may go undetected in testing, get released into production and not manifest itself until it does!!! Ouch!

A Safe FIFO

Heads up: you should NOT be designing and coding classes like our Safe Q. Instead examine and use java.util.concurrent (ReentrantLock, BlockingQueue, . . .).

Here, as a learning exercise, we roll our own mutex, condition variable and safe Q as a leaning exercise. Using the built-in java provided classes is the only way to program in the real world in your 'professional programmer' role. Here you are preparing for that role by 'rolling our own.'

Class SafeQ<T>

Constructors

Constructor and Description
SafeQ() Constructs a SafeQ

Method Summary

Public Methods

Modifier and Type	Method and Description
void	push(T t) Adds the item t to the end of the underlying FIFO
void	priorityPush (T t) Adds the item t to the front of the underlying FIFO.
T	popWait() Removes and returns the first item in the underlying FIFO. If the FIFO is empty the invoking Thread waits.
T	pop () Removes and returns the first item in the underlying FIFO. If the FIFO is empty NULL is returned
boolean	isEmpty() Returns true if the underlying FIFO is empty else returns false.
int	size () Returns the size of the underlying FIFO.

void	clear() Clears the underlying FIFO.
int	`backLog()` Returns the size of the underlying FIFO. Same as size() method.
T	peek() Returns (does NOT remove) the first item in the underlying FIFO. If the FIFO is empty NULL is returned

Back to our consumer/producer example

In our previous consumer/producer example the Store was not thread safe. Consumers and producers shared a stateless controller that took care of our safety concerns by using a mutex and conditional variable. Consumers invoked the get work method. Producers invoked the add work method.

```
// Controller method to add work
public void addWork (Work work) {
        lock the mutex;
        store.addWortkItem(work);
        unlock the mutex
        condVar.signal();
}
```

```
// Controller method to get work
public void getWork {
        lock the mutex;
        while (store.isEmpty(){
                condVar.wait(mutex)
        }
        Work work = store. getWorkItem();
        unlock the mutex
        return work
}
```

We remove our controller and replace the **Store** with a Safe Q <Work>. Consumers and producers share the safe Q. Both consumer and producer deal directly with a single shared Safe Q.

The producer now invokes the Safe Q push(work) method.
The consumer now invokes the Safe Q pop Wait() method.

Safe Q implementation

```
package threadSynch;
import java.util.LinkedList;

public class SafeQ<T> {

	private final Mutex mutex;
	private final CondVar condVar;
	private LinkedList<T> list;

	public SafeQ() {
		mutex = new Mutex();
		condVar = new CondVar();
		list = new LinkedList<T>();
	}

	public void push(T t)  {
		mySafePush(t, false);
	}

	public void priorityPush(T t) {
		mySafePush(t, true);
	}

	private void mySafePush(T t, boolean priorityPush) {

		mutex.lock();

		if (priorityPush) {
			list.addFirst(t);
		} else {
			list.addLast(t);
		}

		mutex.unlock();
		condVar.signal();
	}
```

```
public T popWait()  {

        mutex.lock();

        while (list.isEmpty()) {
                condVar.wait(mutex);
        }
        T t = list.removeFirst();

        mutex.unlock();
        return t;
}

public T pop()   {

        T t = null;
        mutex.lock();

        if (!list.isEmpty()) {
                t = list.removeFirst();
        }

        mutex.unlock();
        return t;
}

public boolean isEmpty() throws InterruptedException {
        mutex.lock();
        boolean isMT = list.isEmpty();
        mutex.unlock();
        return isMT;
}

public int size() throws InterruptedException {
        mutex.lock();
        int         size = list.size();
        mutex.unlock();
        return size;
}

public int backLog() throws InterruptedException {
        return this.size();
}
```

```
        public T peek() {
                mutex.lock();
                T t = list.peekFirst();
                mutex.unlock();
                return t;
        }

        public void clear() {
                mutex.lock();
                list.clear();
                mutex.unlock();
        }
}
```

We use a primitive Java list as our FIFO queue(our private Q) which is NOT thread safe. We repeatedly use the private mutex to protect the list.

The two push methods are the signalers – a signal announces that an item was just added to the Q.

The pop wait method is the waiter. It will wait to be signaled while the Q is empty.

The functionality of the various methods should be obvious!

We provide an example in the following chapter.

Thread returns from run: a logger thread

A thread implements runnable – a simple concept. Simple is good! When the run method returns the threads work is done; it cannot be restarted.

We can keep a thread's run method going and going and going by introducing a loop inside the run method. Our singleton logger thread will do just that.

You should never have to write your own logger – like we are doing here. There are plenty of "off the self' provided Loggers around!. This is a leaning exercise to examine how a Safe Q can be used and how to keep a thread's run method active for a "long time"!

We want the logging thread

1. To be a singleton
2. To run in a single thread – its 'own' thread
3. To run all the time in the background
4. To be accessible to ALL threads
5. To be tread-safe
6. To log all log entries to a single "location" – a file, a DB, a console etcetera
7. To be kind to users – i.e., FAST. We do not want logging to slow down our application threads. We want to minimize the case where Thread A logging blocks Thread B logging. Also, a thread should NOT have to wait while "location" related I/O is taking place. The always slow I/O should happen in the single thread
8. To hide its implementation details from other Threads.

We jump right into the code!

A log entry. An entry consists of a time stamp, one of five categories and text. Notice that nothing here is public. The test can be a simple string or a list of strings. This all stays inside the Log package

```
package Log;

import java.util.ArrayList;
import java.util.List;

class LogEntry  {

	enum Category {
		ERROR, WARNING, INFO, MISC, SHUTDOWN;
	}

	private List<String> textLines;
	private Category category;
	private long timeStamp;

	List<String> getTextLines() {
		return textLines;
	}

	Category getCategory() {
		return category;
	}

	long getTimeStamp() {
		return timeStamp;
	}

	LogEntry(Category category, String text){
		textLines = new ArrayList<String>();
		textLines.add(text);
		commonInit(category);
	}

	LogEntry(Category category, List<String> textLines){
		this.textLines = textLines;
		commonInit(category);
	}
```

```
        private void commonInit(Category category) {
                timeStamp = System.currentTimeMillis();
                if (category != null) {
                        this.category = category;
                } else {
                        this.category = Category.MISC;
                }
        }

        boolean isShutdownRequest() {
                return this.category == Category.SHUTDOWN;
        }

        static LogEntry buildShutdownRequest() {
                return new LogEntry(Category.SHUTDOWN, "Shut down requested");
        }

}
```

Notice what may seem ODD – we include a shutdown related log entry. We will use this to log the shutdown message AND to signal the logging thread to stop. When the shutdown is first requested, the logging thread will first process backlogged log entries then log the shutdown entry and then shutdown (return from run method).

We need the functionality to persist our entries to a 'location."

```
package Log;

interface LogEntryPersister {
        public void persist(LogEntry logEntry);
}
```

For our example we log to the console (our log's "location). Typically, this would be a file and the logger would require external props for the file name, location, etc. This suffices for our learning example.

```
package Log;

public class LogEntryConsolePersister implements LogEntryPersister{

        LogEntryConsolePersister(){
        }

        @Override
        public void persist(LogEntry logEntry) {

                for (String text : logEntry.getTextLines()) {
                        System.out.println(""+ logEntry.getTimeStamp() +
                                ": " + logEntry.getCategory().name() + " " +text);
                }
        }
}
```

Finally, our logger thread. It exposes these public methods to its fellow threads.

- public void logError(String text)
- public void logInfo(String text)
- public void logWarning(String text)
- public void logError(List<String> text)
- public void logInfo(List<String> text)
- public void logWarning(List<String> text)
- public void shutDown()

The implementation of our singleton Logger

```
package Log;

import java.util.List;
import threadSynch.SafeQ;

public class Logger {
	public static Logger logger;
	static {
		logger = new Logger();
		logger.start();
	}

	private final SafeQ<LogEntry> sharedLogEntryQ;
	private final Thread myThread;

	private Logger() {
		LogEntryPersister logItemPersister = new LogEntryConsolePersister();
		sharedLogEntryQ = new SafeQ<LogEntry>();
		LoggerRunnable myRunnable =
		    new LoggerRunnable(sharedLogEntryQ, logItemPersister);
		myThread = new Thread(myRunnable);
	}

	private void start() {
		myThread.start();
	}

	public void logError(String text) {
		LogEntry entry = new LogEntry(LogEntry.Category.ERROR, text);
		sharedLogEntryQ.push(entry);
	}

	public void logWarning(String text) {
		LogEntry entry = new LogEntry(LogEntry.Category.WARNING, text);
		sharedLogEntryQ.push(entry);
	}

	public void logInfo(String text) {
		LogEntry entry = new LogEntry(LogEntry.Category.INFO, text);
		sharedLogEntryQ.push(entry);
	}
```

```
        public void logError(List<String> text) {
                LogEntry entry = new LogEntry(LogEntry.Category.ERROR, text);
                sharedLogEntryQ.push(entry);
        }

        public void logWarning(List<String> text) {
                LogEntry entry = new LogEntry(LogEntry.Category.WARNING, text);
                sharedLogEntryQ.push(entry);
        }

        public void logInfo(List<String> text) {
                LogEntry entry = new LogEntry(LogEntry.Category.INFO, text);
                sharedLogEntryQ.push(entry);
        }

        public void shutDown() {
                sharedLogEntryQ.push(LogEntry.buildShutdownRequest());
        }
}
```

Notice the private constructor. In here we create a persister and a safe Q which we provide for our runnable – which follows.

```
package Log;

import threadSynch.SafeQ;

class LoggerRunnable implements Runnable {

	private final SafeQ<LogEntry> logEntryQ;
	private final LogEntryPersister persister;

	LoggerRunnable(SafeQ<LogEntry> logEntryQ, LogEntryPersister persister){
		this.logEntryQ = logEntryQ;
		this.persister = persister;
	}

	@Override
	public void run() {

		LogEntry entry;
		boolean shutdownRequested  = false;

		while (!shutdownRequested) {
			entry = logEntryQ.popWait();
			persister.persist(entry);
			shutdownRequested = entry.isShutdownRequest();
		}

	}
}
```

The (see above) Logger class provides a user the mechanism for shutting down the logging thread via

```
public void shutDown() {
	sharedLogEntryQ.push(LogEntry.buildShutdownRequest());
}
```

We could have coded the following – but we did not!

```
public void shutDown() {
	sharedLogEntryQ.priorityPush(LogEntry.buildShutdownRequest());
}
```

Notice how the run method handles the shutdown entry.

A silly test. A worker thread that logs.

```
package workerThread;

import java.util.ArrayList;
import java.util.List;
import Log.Logger;
import simulation.CpuHog;

public class LoggingWorkerBee implements Runnable {
	private final Logger myLogger;
	private final List<String> myStats;
	public LoggingWorkerBee () {
		myLogger = Logger.logger;
		myStats = new ArrayList<String>();
	}

	@Override
	public void run() {
		long myID = Thread.currentThread().getId();
		myLogger.logInfo ("Worker BEE Thread: " + myID + " Starting up");
		myLogger.logError("Worker BEE Thread: " + myID + " logged an error message");
		myStats.add("Worker BEE Thread: " + myID + " Stats");
		CpuHog.doIt();
		myStats.add("    Number of sales: 1299");
		myStats.add("    Number of refunds: 23");
		myStats.add("    Number of Frauds detected: 2");
		myStats.add("    Average sales amount $239.89");
		myLogger.logInfo(myStats);
		myLogger.logError("Worker BEE Thread: " + myID + " logged an error message");
		myLogger.logError("Worker BEE Thread: " + myID + " logged a warning message");
		myLogger.logInfo ("Worker BEE Thread: " + myID + " returning  from run");
	}
}
```

The logger is statically available in our constructor. We use our CPU Hog to simulate the worker doing actual work!

Main will be our test driver.

```
package main;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import Log.Logger;
import workerThread.LoggingWorker;
import workerThread.LoggingWorkerBee;

public class Main {
	public static void main(String[] args) {
		Logger myLogger = Logger.logger;
		myLogger.logInfo("Main thread starting");
		List<Thread> threads = new ArrayList<Thread>();

		int I = 0;
		while (I < 9) {
			Thread t = new Thread(new LoggingWorkerBee());
			threads.add(t);
			I++;
		}

		for (Thread t: threads) {
			t.start();
		}

		myLogger.logInfo("Main thread started threads");

		System.out.println("We stall before shutting down...Hit key to exit main");
		try {
			System.in.read();
		} catch (IOException e) {
		}

		myLogger.logInfo("Main thread stopping logger and exiting");
		myLogger.shutDown();
	}
}
```

A test run

1646776294987: INFO Main thread starting
1646776294988: INFO Main thread started threads

We stall before shutting down...Hit key to exit main

1646776294989: INFO Worker BEE Thread: 15 Starting up
1646776294989: INFO Worker BEE Thread: 14 Starting up
1646776294989: ERROR Worker BEE Thread: 14 logged an error message
1646776294989: ERROR Worker BEE Thread: 15 logged an error message
1646776294989: INFO Worker BEE Thread: 16 Starting up
1646776294989: ERROR Worker BEE Thread: 16 logged an error message
1646776294989: INFO Worker BEE Thread: 17 Starting up
1646776294990: ERROR Worker BEE Thread: 17 logged an error message
1646776294990: INFO Worker BEE Thread: 18 Starting up
1646776294990: INFO Thread: 14 CPU hog Start
1646776294990: INFO Thread: 16 CPU hog Start
1646776294990: INFO Thread: 15 CPU hog Start
1646776294994: INFO Thread: 17 CPU hog Start
1646776295027: ERROR Worker BEE Thread: 18 logged an error message
1646776295027: INFO Worker BEE Thread: 19 Starting up

etcetera

1646776298077: INFO Worker BEE Thread: 22 returning from run
1646776303652: INFO Main thread stopping logger and exiting
1646776303652: SHUTDOWN Shut down requested

Notice that the threads ran concurrently. Also note that the stats related entries for a thread are always adjacent to each other in the log.

Trick question. Suppose we increased the stats text lines – say to 200 lines. Would the change adversely affect other threads who are concurrently logging? Answer: NO, it would have ZERO effect. A logging thread deposits a reference to a log entry in the sage Q and continues – all the logging work is done in our logger runnable.

Mutex with a timeout

An example synchronized wait time out

```
private final Object privateSignal = new Object();

public void waitXXX (long timeOutMilli) {
if (timeOutMilli < 0) {
        timeOutMilli = 0;   // avoid IllegalArgumentException
}

long timeOutNano = 1000000*timeOutMilli;
long startWaitTimeNano = System.nanoTime();

try {
        privateSignal.wait(timeOutMilli);
}
catch (Exception E) {
}

long elapsedWaitTimeNano = System.nanoTime() - startWaitTimeNano;
boolean timedOut = timeOutMilli > 0 && elapsedWaitTimeNano >= timeOutNano;

// etc…
}
```

The Mutex implementation

Our Mutex with a Timeout follows.

```
package threadSynch;
public class TimedMutex {
	private Thread owner = null;
	private Object privateSignal;
	private final long timeOutMilli; // ZERO implies NO timeout
	public enum AcquireLockStatus {ACQUIRED, TIME_OUT, INTERRUPTED};

	public TimedMutex(long timeOutMilli) {
		privateSignal = new Object();
		this.timeOutMilli = Math.abs(timeOutMilli); // make sure this is valid
	}
	public long getTimeout() {
		return this.timeOutMilli;
	}
	public AcquireLockStatus acquireLock()  {
		AcquireLockStatus lockStatus = null;
		synchronized(privateSignal) {
			Thread requester = Thread.currentThread();
			if (owner == null) {
				owner = requester;
				lockStatus = AcquireLockStatus.ACQUIRED;
			} else if (owner == requester) {
				lockStatus = AcquireLockStatus.ACQUIRED;
			}
			if (lockStatus == null) {
				try {
					privateSignal.wait(timeOutMilli);
				}
				catch (InterruptedException IE) {
					lockStatus = AcquireLockStatus.INTERRUPTED;
				}
				catch (IllegalArgumentException IAE) {
					// we checked - our timeOutMilli is legal
				}
				catch (IllegalMonitorStateException IMSE) {
					// we are in a synch block
				}
			}
```

```
                        if (lockStatus == null) {

                                if (owner == null) {
                                        owner = requester;  // We were signaled
                                        lockStatus = AcquireLockStatus.ACQUIRED;
                                } else {
                                        lockStatus = AcquireLockStatus.TIME_OUT;
                                        // i.e., we woke up but we were NOT signaled
                                        //  AND no wait exceptions occurred!
                                }
                        }
                }

                return lockStatus;
        }


        public void unlock() {
                synchronized(privateSignal) {
                        if (owner == Thread.currentThread()) {
                                owner = null; // release lock
                                try {
                                        privateSignal.notify();
                                } catch (IllegalMonitorStateException IMSE) {
                                        // // we are in a synch block
                                }
                        }
                }
        }
}
```

Notes:

We did not have to perform any fancy timeout detection arithmetic! When we wake up (return from) wait we check if the mutex owner is null – if so we were notified and we acquire the lock. If the owner is NOT null we timeout out.

Scenarios where we return from wait. Look at the code an convince yourself that we 'covered' the scenarios.

1. The wait was interrupted
2. The wait timeout argument was invalid
3. An Illegal Monitor State Exception was thrown
4. We timed out
5. We did **not** time out and we were signaled

Test Example

Worker threads share this.

```
package resources;

import Log.Logger;
import threadSynch.TimedMutex;
import threadSynch.TimedMutex.AcquireLockStatus;

public class SharedResource {
	private final TimedMutex mutex;

	public SharedResource (TimedMutex mutex){
		this.mutex = mutex;

	}

	public void  doIt()  {
		Logger.log("T:"+ Thread.currentThread().getId() + " about to acquire lock...");

		AcquireLockStatus lockStatus = mutex.acquireLock();

		if (lockStatus == TimedMutex.AcquireLockStatus.ACQUIRED) {
			Logger.log("T:"+ Thread.currentThread().getId() +
					" 1  - GOT the LOCK Start work");
			Logger.log("T:"+ Thread.currentThread().getId() + " 2 ");
			Logger.log("T:"+ Thread.currentThread().getId() + " 3 ");

			double d = eatUpCPU();

			Logger.log("T:"+ Thread.currentThread().getId() + " 4 " +  d);
			Logger.log("T:"+ Thread.currentThread().getId() + " 5 ");
			Logger.log("T:"+ Thread.currentThread().getId() + " 6 ");
			Logger.log("T:"+ Thread.currentThread().getId() + " 7 Finished work");

			mutex.unlock();
		} else{
			Logger.log("T:"+ Thread.currentThread().getId() +
					" Failed to acquire lock:  " +  lockStatus.name());
		}
	}
```

```
	private double eatUpCPU() {
		double result = 0;
		int big = 10000000;
		double arr[] = new double[big];

		for (int I = 0; I < big ; I ++) {
			double x  =  Math.sin(I+.002) * Math.log(I+.004);
			x  = Math.cos(x+.009);
			arr[I] = x;
		}

		for (int I = 0; I < big ; I ++) {
			result = result + arr[I]/big;
		}

		return result;
	}

}
```

A worker

```
package workerThread;

import resources.SharedResource;

public class Worker implements Runnable {

	//private final static Object lock = new Object();

	private final SharedResource sharedResource;

	public Worker (SharedResource sharedResource) {
		this.sharedResource = sharedResource;
	}

	@Override
	public void run() {
		 sharedResource.doIt();
	}
}
```

Worker controller

```
package workerThread;

import resources.SharedResource;
import threadSynch.TimedMutex;

public class WorkerController {

	private Thread[] workerThread;
	private final int numWorkers=8;

	public WorkerController() {

		workerThread = new Thread[numWorkers];
		//TimedMutex mutex = new TimedMutex(5000);
		//TimedMutex mutex = new TimedMutex(7000);
		TimedMutex mutex = new TimedMutex(2000);

		SharedResource sharedResource = new SharedResource(mutex);

		for (int I = 0; I < numWorkers; I++) {
			workerThread[I] = new Thread(new Worker(sharedResource));
		}
	}

	public void startThreads() {
		for (int I = 0; I < numWorkers; I++) {
			workerThread[I].start();
		}
	}

}
```

Perform various runs and change the timeout in the controller to **various values.** Change the controller and make the timeout value a function of the number of worker threads!

Main

```
package main;

import Log.Logger;
import workerThread.WorkerController;


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

		Logger.log("Main thread starting");
		WorkerController controller = new WorkerController();
		controller.startThreads();
		Logger.log("Main thread started threads");

	}
}
```

Example run with 2000 (2 seconds) timeout

1647366690761 Main thread starting
1647366690763 T:13 about to acquire lock...
1647366690764 Main thread started threads
1647366690764 T:14 about to acquire lock...
1647366690764 T:15 about to acquire lock...
1647366690764 T:16 about to acquire lock...
1647366690764 T:13 1 - GOT the LOCK Start work
1647366690764 T:13 2
1647366690764 T:13 3
1647366690764 T:17 about to acquire lock...
1647366690764 T:18 about to acquire lock...
1647366690765 T:19 about to acquire lock...
1647366690765 T:20 about to acquire lock...
1647366691543 T:13 4 -0.05361478989054746
1647366691543 T:13 5
1647366691543 T:13 6
1647366691543 T:13 7 Finished work
1647366691543 T:16 1 - GOT the LOCK Start work
1647366691543 T:16 2
1647366691543 T:16 3
1647366692278 T:16 4 -0.05361478989054746
1647366692278 T:16 5
1647366692278 T:16 6
1647366692278 T:16 7 Finished work
1647366692279 T:14 1 - GOT the LOCK Start work
1647366692279 T:14 2
1647366692279 T:14 3
1647366692779 T:19 Failed to acquire lock: TIME_OUT
1647366692779 T:18 Failed to acquire lock: TIME_OUT
1647366692779 T:15 Failed to acquire lock: TIME_OUT
1647366692779 T:20 Failed to acquire lock: TIME_OUT
1647366692779 T:17 Failed to acquire lock: TIME_OUT
1647366692998 T:14 4 -0.05361478989054746
1647366692998 T:14 5
1647366692998 T:14 6
1647366692998 T:14 7 Finished work

Thread 16 waited 779ms for the lock. Subtract 1647366691543 – 1647366690764 = 779

1647366690764 T:16 about to acquire lock...
1647366691543 T:16 1 - GOT the LOCK Start work

Thread 19 timed out. Subtract 1647366692779 - 1647366690765 = 2014

1647366690765 T:19 about to acquire lock...
1647366692779 T:19 Failed to acquire lock: TIME_OUT

CountDown class

Constructors

Constructor and Description
CountDown(String name, int countDown) Constructs a CountDown with the given name. When the countdown reaches ZERO a signal is generated. Implements: CountDowner

Method Summary

Methods

Modifier and Type	Method and Description
void	countDown() Decrements the counter and signals when the counter reaches ZERO
void	waitUntilDone () Waits for s signal generated via the countDown method

```
package threadSynch;

public interface CountDowner {
	public void countDown();
}
```

This waits forever.

```
package threadSynch;

import threadSynch.TimedMutex.AcquireLockStatus;

public class CountDown implements CountDowner {

	private final Object privateSignal;

	private boolean allDone = false;
	private int currentCountDown;

	public CountDown(int countDown) {
		privateSignal = new Object();
		currentCountDown=countDown;
		if (currentCountDown < 1) {
			currentCountDown = 1;
		}
	}

	public void countDown() {
		synchronized(privateSignal) {
			if (!allDone) {
				currentCountDown = currentCountDown -1;
				allDone = currentCountDown == 0;
				if (allDone) {
					privateSignal.notify();
				}
			}
		}
	}
```

```
        public AcquireLockStatus waitUntilDone() {
                AcquireLockStatus lockStatus = null;
                synchronized(privateSignal) {
                        if (allDone)
                        {
                                lockStatus = AcquireLockStatus.ACQUIRED;
                        } else {
                                try {
                                        privateSignal.wait();
                                } catch (InterruptedException IE) {
                                        lockStatus = AcquireLockStatus.INTERRUPTED;
                                }
                                catch (IllegalArgumentException IAE) {
                                        // we checked - our timeOutMilli is legal
                                }
                                catch (IllegalMonitorStateException IMSE) {
                                        // we are in a synch block
                                }
                                if (lockStatus == null) {
                                        lockStatus = AcquireLockStatus.ACQUIRED;
                                }
                        }
                        return lockStatus;
                }
        }
}
```

When a **Count Downer** invokes the countdown we simple decrement our counter; when the counter reaches ZERO we signal. The signal indicates that the countdown has completed – 10, 9, 8…2,1,0 Blast OFF!

Example

Here is an example Count Downer

```
package workerThread;

import java.util.ArrayList;
import java.util.List;
import Log.Logger;
import simulation.CpuHog;
import threadSynch.CountDowner;

public class LoggingWorkerBee implements Runnable {

	private final Logger myLogger;
	private final List<String> myStats;

	private CountDowner countDown;

	public LoggingWorkerBee (CountDowner countDown) {
		myLogger = Logger.logger;
		myStats = new ArrayList<String>();
		this.countDown = countDown;
	}

	@Override
	public void run() {
		long myID = Thread.currentThread().getId();
		myLogger.logInfo ("Worker BEE Thread: " + myID + " Starting up");

		CpuHog.doIt();

		myStats.add("Worker BEE Thread: " + myID + " Stats");
		myStats.add("    Number of sales: 1299");
		myStats.add("    Number of refunds: 23");
		myStats.add("    Number of Frauds detected: 2");
		myStats.add("    Average sales amount $239.89");
		myLogger.logInfo(myStats);
		myLogger.logInfo ("Worker BEE Thread: " + myID + " returning  from run");
		countDown.countDown();
	}
}
```

Here is a count down with a time out.

```
package threadSynch;

import threadSynch.TimedMutex.AcquireLockStatus;

public class TimedCountDown implements CountDowner {

	private final Object privateSignal;
	private final long timeOutMilli;

	private boolean allDone = false;
	private int currentCountDown;

	public TimedCountDown(int countDown) {
		this(countDown, 0L);
	}
	public TimedCountDown(int countDown, long timeOutMilli) {
		privateSignal = new Object();
		this.timeOutMilli = Math.abs(timeOutMilli);   // make sure it is valid
		currentCountDown=countDown;
		if (currentCountDown < 1) {
			currentCountDown = 1;
		}
	}

	public void countDown() {

		synchronized(privateSignal) {
			if (!allDone) {
				currentCountDown = currentCountDown -1;
				allDone = currentCountDown == 0;
				if (allDone) {
					privateSignal.notify();
				}
			}

		}
	}
```

```
        public AcquireLockStatus waitUntilDone() {
                AcquireLockStatus lockStatus = null;

                synchronized(privateSignal) {
                        if (allDone)
                        {
                                lockStatus = AcquireLockStatus.ACQUIRED;
                        } else {

                                long startWaitTimeNano = System.nanoTime();
                                try {
                                        privateSignal.wait(timeOutMilli);
                                }catch (InterruptedException IE) {
                                        lockStatus = AcquireLockStatus.INTERRUPTED;
                                }
                                catch (IllegalArgumentException IAE) {
                                        // we checked - our timeOutMilli is legal
                                }
                                catch (IllegalMonitorStateException IMSE) {
                                        // we are in a synch block
                                }

                                if (lockStatus == null) {
                                        long elapsedWaitTimeNano =
                                              System.nanoTime() - startWaitTimeNano;

                                        boolean timedOut = timeOutMilli > 0 &&
                                                elapsedWaitTimeNano >= 1000000*timeOutMilli;

                                        if (timedOut) {
                                                lockStatus = AcquireLockStatus.TIME_OUT;
                                        } else {
                                                lockStatus = AcquireLockStatus.ACQUIRED;
                                        }
                                }
                        }

                        return lockStatus;
                }
        }
}
```

We will use main as our controller who owns the Count Down.

```
package main;

import java.util.ArrayList;
import java.util.List;
import Log.Logger;
import threadSynch.CountDown;
import threadSynch.TimedCountDown;
import threadSynch.TimedMutex.AcquireLockStatus;
import workerThread.LoggingWorkerBee;

public class Main {
	public static void main(String[] args) {
		Logger myLogger = Logger.logger;
		myLogger.logInfo("---> Main thread starting");

		List<Thread> threads = new ArrayList<Thread>();
		final int numThreads = 5;

		// Test cases
		//TimedCountDown countDown = new TimedCountDown(numThreads, 1800);
		TimedCountDown countDown = new TimedCountDown(numThreads, 1600);
		// CountDown countDown = new CountDown(numThreads);

		int I = 0;
		while (I < numThreads) {
			Thread t = new Thread(new LoggingWorkerBee(countDown));
			threads.add(t);
			I++;
		}

		for (Thread t: threads) {
			t.start();
		}

		myLogger.logInfo("---> Main thread started threads  ... we wait here");
		AcquireLockStatus status = countDown.waitUntilDone();
		myLogger.logInfo(
		   "--->  Main. Bees are done...Main thread stopping logger and exiting  status: "+status);

		myLogger.shutDown();
	}
}
```

A run with `TimedCountDown countDown = new TimedCountDown(numThreads, 1600);`

1647382038833: INFO ---> Main thread starting
1647382038836: INFO Worker BEE Thread: 14 Starting up
1647382038836: INFO ---> Main thread started threads ... we wait here
1647382038836: INFO Worker BEE Thread: 15 Starting up
1647382038836: INFO Worker BEE Thread: 16 Starting up
1647382038837: INFO Worker BEE Thread: 18 Starting up
1647382038837: INFO Worker BEE Thread: 17 Starting up
1647382038837: INFO Thread: 17 CPU hog Start
1647382038839: INFO Thread: 18 CPU hog Start
1647382038841: INFO Thread: 16 CPU hog Start
1647382038849: INFO Thread: 14 CPU hog Start
1647382038873: INFO Thread: 15 CPU hog Start
1647382040468: INFO ---> Main.. Bees are done...Main thread stopping logger and exiting. status: **TIME_OUT**
1647382040468: SHUTDOWN Shut down requested

Elapsed time before timeout – we did NOT give the workers enough time
1647382040468 – 1647382038833 = 1,635ms

The workers were still working when we **timed out**. Hummm! be careful here

A run with `TimedCountDown countDown = new TimedCountDown(numThreads, 1700);`

```
1647382679500: INFO ---> Main thread starting
1647382679502: INFO Worker BEE Thread: 15 Starting up
1647382679502: INFO Worker BEE Thread: 14 Starting up
1647382679502: INFO ---> Main thread started threads  ... we wait here
1647382679502: INFO Worker BEE Thread: 16 Starting up
1647382679502: INFO Worker BEE Thread: 17 Starting up
1647382679502: INFO Worker BEE Thread: 18 Starting up
1647382679503: INFO Thread: 17  CPU hog Start
1647382679503: INFO Thread: 16  CPU hog Start
1647382679503: INFO Thread: 18  CPU hog Start
1647382679508: INFO Thread: 15  CPU hog Start
1647382679503: INFO Thread: 14  CPU hog Start
1647382681210: INFO Thread: 17  CPU hog Completed
1647382681210: INFO Worker BEE Thread: 17 Stats
1647382681210: INFO     Number of sales: 1299
1647382681210: INFO     Number of refunds: 23
1647382681210: INFO     Number of Frauds detected: 2
1647382681210: INFO     Average sales amount $239.89
1647382681210: INFO Worker BEE Thread: 17 returning  from run
1647382681214: INFO --->  Main.. Bees are done...Main thread stopping logger and
exiting. status: TIME_OUT
1647382681214: SHUTDOWN Shut down requested
```

One worker finished

Try this and re-run main. In main add a pause.

```
        . . .
        myLogger.logInfo("--->  Main.. Bees are done...Main thread stopping logger and exiting. status: "+status);

        System.out.println("Hit key to exit program...");
        try {
                System.in.read();
        } catch (IOException e) {
                //
        e.printStackTrace();
        }

        myLogger.shutDown();

} //exit main
```

A run with TimedCountDown countDown = **new** TimedCountDown(numThreads, 2000);

1647382146879: INFO ---> Main thread starting
1647382146881: INFO Worker BEE Thread: 14 Starting up
1647382146881: INFO Worker BEE Thread: 16 Starting up
1647382146881: INFO ---> Main thread started threads ... we wait here
1647382146881: INFO Worker BEE Thread: 15 Starting up
1647382146881: INFO Worker BEE Thread: 18 Starting up
1647382146881: INFO Worker BEE Thread: 17 Starting up
1647382146882: INFO Thread: 16 CPU hog Start
1647382146882: INFO Thread: 14 CPU hog Start
1647382146882: INFO Thread: 15 CPU hog Start
1647382146882: INFO Thread: 17 CPU hog Start
1647382146882: INFO Thread: 18 CPU hog Start
1647382148566: INFO Thread: 14 CPU hog Completed
1647382148566: INFO Worker BEE Thread: 14 Stats
1647382148566: INFO   Number of sales: 1299
1647382148566: INFO   Number of refunds: 23
1647382148566: INFO   Number of Frauds detected: 2
1647382148566: INFO   Average sales amount $239.89
1647382148566: INFO Worker BEE Thread: 14 returning from run
1647382148588: INFO Thread: 15 CPU hog Completed
1647382148588: INFO Worker BEE Thread: 15 Stats
1647382148588: INFO   Number of sales: 1299
1647382148588: INFO   Number of refunds: 23
1647382148588: INFO   Number of Frauds detected: 2
1647382148588: INFO   Average sales amount $239.89
1647382148588: INFO Worker BEE Thread: 15 returning from run
1647382148605: INFO Thread: 17 CPU hog Completed
1647382148605: INFO Worker BEE Thread: 17 Stats
1647382148605: INFO   Number of sales: 1299
1647382148605: INFO   Number of refunds: 23
1647382148605: INFO   Number of Frauds detected: 2
1647382148605: INFO   Average sales amount $239.89
1647382148605: INFO Worker BEE Thread: 17 returning from run
1647382148623: INFO Thread: 16 CPU hog Completed
1647382148623: INFO Worker BEE Thread: 16 Stats
1647382148623: INFO   Number of sales: 1299
1647382148623: INFO   Number of refunds: 23
1647382148623: INFO   Number of Frauds detected: 2
1647382148623: INFO   Average sales amount $239.89
1647382148623: INFO Worker BEE Thread: 16 returning from run
1647382148628: INFO Thread: 18 CPU hog Completed
1647382148628: INFO Worker BEE Thread: 18 Stats
1647382148628: INFO   Number of sales: 1299
1647382148628: INFO   Number of refunds: 23
1647382148628: INFO   Number of Frauds detected: 2
1647382148628: INFO   Average sales amount $239.89
1647382148628: INFO Worker BEE Thread: 18 returning from run
1647382148628: INFO ---> Main.. Bees are done...Main thread stopping logger and exiting. status: ACQUIRED
1647382148629: SHUTDOWN Shut down requested

Total elapsed time 1647382148629 - 1647382146879 = 1750ms

A run with CountDown countDown = **new** CountDown(numThreads);

1647382256169: INFO ---> Main thread starting
1647382256171: INFO Worker BEE Thread: 14 Starting up
1647382256171: INFO ---> Main thread started threads ... we wait here
1647382256171: INFO Worker BEE Thread: 15 Starting up
1647382256171: INFO Worker BEE Thread: 16 Starting up
1647382256171: INFO Worker BEE Thread: 17 Starting up
1647382256171: INFO Worker BEE Thread: 18 Starting up
1647382256172: INFO Thread: 17 CPU hog Start
1647382256172: INFO Thread: 16 CPU hog Start
1647382256172: INFO Thread: 18 CPU hog Start
1647382256172: INFO Thread: 15 CPU hog Start
1647382256172: INFO Thread: 14 CPU hog Start
1647382257885: INFO Thread: 17 CPU hog Completed
1647382257885: INFO Worker BEE Thread: 17 Stats
1647382257885: INFO   Number of sales: 1299
1647382257885: INFO   Number of refunds: 23
1647382257885: INFO   Number of Frauds detected: 2
1647382257885: INFO   Average sales amount $239.89
1647382257885: INFO Worker BEE Thread: 17 returning from run
1647382257893: INFO Thread: 15 CPU hog Completed
1647382257893: INFO Worker BEE Thread: 15 Stats
1647382257893: INFO   Number of sales: 1299
1647382257893: INFO   Number of refunds: 23
1647382257893: INFO   Number of Frauds detected: 2
1647382257893: INFO   Average sales amount $239.89
1647382257893: INFO Worker BEE Thread: 15 returning from run
1647382257894: INFO Thread: 18 CPU hog Completed
1647382257894: INFO Worker BEE Thread: 18 Stats
1647382257894: INFO   Number of sales: 1299
1647382257894: INFO   Number of refunds: 23
1647382257894: INFO   Number of Frauds detected: 2
1647382257894: INFO   Average sales amount $239.89
1647382257894: INFO Worker BEE Thread: 18 returning from run
1647382257899: INFO Thread: 16 CPU hog Completed
1647382257899: INFO Worker BEE Thread: 16 Stats
1647382257899: INFO   Number of sales: 1299
1647382257899: INFO   Number of refunds: 23
1647382257899: INFO   Number of Frauds detected: 2
1647382257899: INFO   Average sales amount $239.89
1647382257899: INFO Worker BEE Thread: 16 returning from run
1647382257911: INFO Thread: 14 CPU hog Completed
1647382257911: INFO Worker BEE Thread: 14 Stats
1647382257911: INFO   Number of sales: 1299
1647382257911: INFO   Number of refunds: 23
1647382257911: INFO   Number of Frauds detected: 2
1647382257911: INFO   Average sales amount $239.89
1647382257911: INFO Worker BEE Thread: 14 returning from run

1647382257912: INFO ---> Main.. Bees are done...Main thread stopping logger and exiting. status: ACQUIRED
1647382257912: SHUTDOWN Shut down requested
Total elapsed time 1647382257912- 1647382256169= 1,743ms

We again emphasize that our exercises here (roll our own) are for learning purposes. The java language already provides a countdown mechanism which is what you should use in your professional work!

Here it is in use!

```
package main;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import Log.Logger;
import workerThread.LoggingWorkerBee;

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

		Logger myLogger = Logger.logger;
		myLogger.logInfo("---> Main thread starting");

		List<Thread> threads = new ArrayList<Thread>();
		final int numThreads = 5;

		// CountDown countDown = new CountDown(numThreads);
		CountDownLatch countDown = new CountDownLatch(numThreads);

		int I = 0;
		while (I < numThreads) {
			Thread t = new Thread(new LoggingWorkerBee(countDown));
			threads.add(t);
			I++;
		}
		for (Thread t: threads) {
			t.start();
		}

		myLogger.logInfo("---> Main thread started threads  ... we wait here");
		try {
			countDown.await() ;
		} catch (InterruptedException e) {
			myLogger.logInfo("--->  Main..Interrupted while waiting....");
		}
		myLogger.logInfo("--->  Main.. Bees are done...Main  exiting.");
		myLogger.shutDown();
	}
}
```

A run1647383382696: INFO ---> Main thread starting

1647383382697: INFO Worker BEE Thread: 14 Starting up
1647383382697: INFO ---> Main thread started threads ... we wait here
1647383382698: INFO Worker BEE Thread: 15 Starting up
1647383382698: INFO Thread: 14 CPU hog Start
1647383382698: INFO Thread: 15 CPU hog Start
1647383382698: INFO Worker BEE Thread: 17 Starting up
1647383382698: INFO Thread: 17 CPU hog Start
1647383382730: INFO Worker BEE Thread: 16 Starting up
1647383382730: INFO Thread: 16 CPU hog Start
1647383382730: INFO Worker BEE Thread: 18 Starting up
1647383382730: INFO Thread: 18 CPU hog Start
1647383384408: INFO Thread: 17 CPU hog Completed
1647383384408: INFO Worker BEE Thread: 17 Stats
1647383384408: INFO Number of sales: 1299
1647383384408: INFO Number of refunds: 23
1647383384408: INFO Number of Frauds detected: 2
1647383384408: INFO Average sales amount $239.89
1647383384408: INFO Worker BEE Thread: 17 returning from run
1647383384420: INFO Thread: 14 CPU hog Completed
1647383384420: INFO Worker BEE Thread: 14 Stats
1647383384420: INFO Number of sales: 1299
1647383384420: INFO Number of refunds: 23
1647383384420: INFO Number of Frauds detected: 2
1647383384420: INFO Average sales amount $239.89
1647383384420: INFO Worker BEE Thread: 14 returning from run
1647383384422: INFO Thread: 16 CPU hog Completed
1647383384422: INFO Worker BEE Thread: 16 Stats
1647383384422: INFO Number of sales: 1299
1647383384422: INFO Number of refunds: 23
1647383384422: INFO Number of Frauds detected: 2
1647383384422: INFO Average sales amount $239.89
1647383384422: INFO Worker BEE Thread: 16 returning from run
1647383384428: INFO Thread: 18 CPU hog Completed
1647383384428: INFO Worker BEE Thread: 18 Stats
1647383384428: INFO Number of sales: 1299
1647383384428: INFO Number of refunds: 23
1647383384428: INFO Number of Frauds detected: 2
1647383384428: INFO Average sales amount $239.89
1647383384428: INFO Worker BEE Thread: 18 returning from run
1647383384454: INFO Thread: 15 CPU hog Completed
1647383384454: INFO Worker BEE Thread: 15 Stats
1647383384454: INFO Number of sales: 1299
1647383384454: INFO Number of refunds: 23
1647383384454: INFO Number of Frauds detected: 2
1647383384454: INFO Average sales amount $239.89
1647383384454: INFO Worker BEE Thread: 15 returning from run
1647383384455: INFO ---> Main.. Bees are done...Main thread exiting.
1647383384455: SHUTDOWN Shut down requested
Elapsed time: 1647383384455 - 1647383382696 = 1,759ms

The End