**<http://tutorials.jenkov.com/java-concurrency/thread-pools.html>**

**Race Conditions and Critical Sections**

The situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

Race conditions can be avoided by proper thread synchronization in critical sections.

**Thread Safety and Shared Resources**

Code that is safe to call by multiple threads simultaneously is called thread safe.

**Q) What resources Java threads share when executing.**

## Local Variables

Local variables are **stored** in **each thread's own stack**. That means that ***local variables are never shared between threads***. That also means that all local primitive variables are thread safe. Here is an example of a thread safe local primitive variable:

public void someMethod(){

long threadSafeInt = 0;

threadSafeInt++;

}

## Local Object References

The reference itself is not shared. The object referenced however, is not stored in each thread’s local stack. All objects are stored in the shared heap.

If an object created locally never escapes the method it was created in, it is thread safe.

In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads. Here is an example of a thread safe local object:

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value");

}

## Object Members

Object members are stored on the heap along with the object. if two threads call a method on the same object instance and this method updates object members, the method is not thread safe. Here is an example of a method that is not thread safe:

public class NotThreadSafe{

StringBuilder builder = new StringBuilder();

public add(String text){

this.builder.append(text);

}

}

## The Thread Control Escape Rule

If a resource is created, used and disposed within

the control of the same thread,

and never escapes the control of this thread,

the use of that resource is thread safe.

Even if the use of an object is thread safe, if that object points to a shared resource like a file or database, your application as a whole may not be thread safe.

## Q) The Java synchronized Keyword

The synchronized keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

## Synchronized Instance Methods

public **synchronized** void add(int value){

this.count += value;

}

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance**. One thread per instance.**

## Synchronized Static Methods

public **static synchronized** void add(int value){

count += value;

}

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.

**One thread per class** **regardless of which static synchronized method it calls**.

## Synchronized Blocks in Instance Methods

public void add(int value){

**synchronized(this){**

this.count += value;

**}**

}

In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

public class MyClass {

public **synchronized** void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public void log2(String msg1, String msg2){

**synchronized(this){**

log.writeln(msg1);

log.writeln(msg2);

**}**

}

}

Thus **only a single thread** can **execute** **inside either of the two synchronized blocks** in this example.

Had the second synchronized block been synchronized on a different object than this, then one thread at a time had been able to execute inside each method.

## Synchronized Blocks in Static Methods

public class MyClass {

public static synchronized void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public static void log2(String msg1, String msg2){

synchronized(MyClass.class){

log.writeln(msg1);

log.writeln(msg2);

}

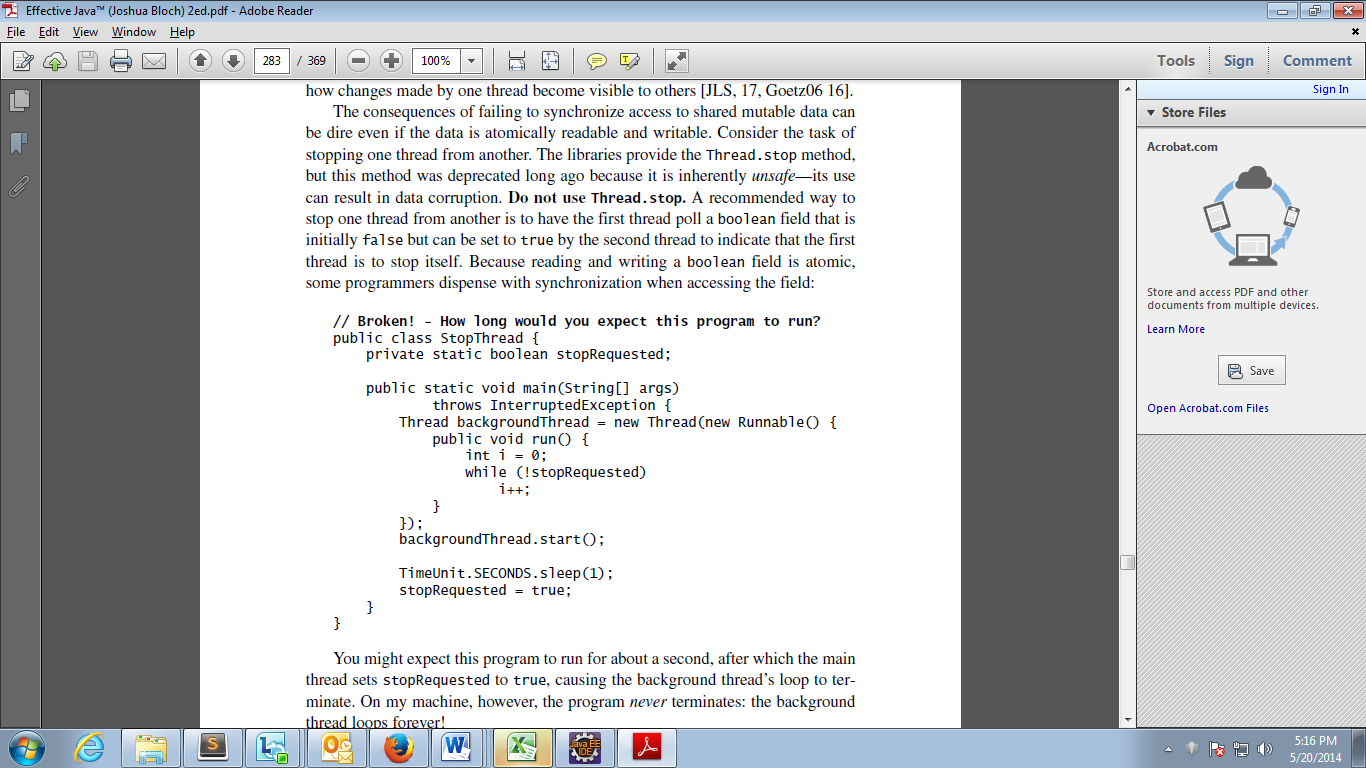
}

}

Only one thread can execute inside any of these two methods at the same time.

**Q) What does synchronize method does ?**

Ans) Not only does synchronization prevent a thread from observing an object in an inconsistent state, but it ensures that each thread entering a synchronized method or block sees the effects

of all previous modifications that were guarded by the same lock .**Synchronization is required for reliable communication between threads as well as for mutual exclusion.**

The above code runs forever because The problem is that in the absence of synchronization, there is no guarantee as to when, if ever, the background thread will see the change in the value of stop-

Requested that was made by the main thread.



The actions of the synchronized methods in **StopThread** would be atomic even without synchronization. In other words, the synchronization on these methods is used *solely* for its communication effects, not for mutual exclusion.

cost of synchronizing on each iteration of the loop is small, there is a correct

alternative that is less verbose and whose performance is likely to be better. The

locking in the second version of StopThread can be omitted if stopRequested is

declared volatile.

// Cooperative thread termination with a volatile field

public class StopThread {

private static volatile boolean stopRequested;

public static void main(String[] args) throws InterruptedException {

Thread backgroundThread = new Thread(new Runnable() {

public void run() {

int i = 0;

while (!stopRequested)

i++;

}

});

backgroundThread.start();

TimeUnit.SECONDS.sleep(1);

stopRequested = true;

}

}

**levels of thread safety**. It is not exhaustive but covers the common cases:

• **immutable**—Instances of this class appear constant. No external synchronization is necessary. Examples include String, Long, and BigInteger (Item 15).

• **unconditionally thread-safe**—Instances of this class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without the need for any external synchronization. Examples include Random and ConcurrentHashMap.

• **conditionally thread-safe**—Like unconditionally thread-safe, except that some methods require external synchronization for safe concurrent use. Examples include the collections returned by the Collections.synchronized wrappers, whose iterators require external synchronization.

• **not thread-safe**—Instances of this class are mutable. To use them concurrently, clients must surround each method invocation (or invocation sequence) with external synchronization of the clients’ choosing. Examples include the general-purpose collection implementations, such as ArrayList and HashMap.

• **thread-hostile—**This class is not safe for concurrent use even if all method invocations

are surrounded by external synchronization. Thread hostility usually results from modifying static data without synchronization. No one writes a thread-hostile class on purpose; such classes result from the failure to consider concurrency. Luckily, there are very few thread-hostile classes or methods in the Java libraries. The System.runFinalizersOnExit method is thread-hostile

and has been deprecated.

## Thread Signaling ?

1. **Signaling via Shared Objects.**

public class MySignal{

protected boolean hasDataToProcess = false;

public synchronized boolean hasDataToProcess(){

return this.hasDataToProcess;

}

public synchronized void setHasDataToProcess(boolean hasData){

this.hasDataToProcess = hasData;

}

}

**Busy wait.**

protected MySignal sharedSignal = ...

...

while(!sharedSignal.hasDataToProcess()){

//do nothing... busy waiting

}

1. **wait(), notify() and notifyAll()**

The class java.lang.Object defines three methods, wait(), notify(), and notifyAll(), to facilitate this.

A thread that calls wait() on any object becomes inactive until another thread calls notify() on that object.

In order to call either wait() or notify the calling **thread must first obtain the lock on that object**.

public class MonitorObject{

}

public class MyWaitNotify{

MonitorObject myMonitorObject = new MonitorObject();

public void doWait(){

synchronized(myMonitorObject){

try{

**myMonitorObject.wait();**

} catch(InterruptedException e){...}

}

}

public void doNotify(){

synchronized(myMonitorObject){

**myMonitorObject.notify();**

}

}

}

A thread cannot call wait(), notify() or notifyAll() without holding the lock on the object the method is called on. If it does, an IllegalMonitorStateException is thrown.

Note : Once a thread calls wait() it releases the lock it holds on the monitor object. This allows other threads to call wait() or notify() too, since these methods must be called from inside a synchronized block.

Once a thread is awakened it cannot exit the wait() call until the thread calling notify() has left its synchronized block.

The awakened thread must re-obtain the lock on the monitor object before it can exit the wait() call.

1. **Missed Signals**

If a thread calls notify() before the thread to signal has called wait(), the signal will be missed by the waiting thread.

This may or may not be a problem, but in some cases this may result in the waiting thread waiting forever, never waking up, because the signal to wake up was missed.

To avoid losing signals they should be stored inside the signal class.

public class MyWaitNotify2{

MonitorObject myMonitorObject = new MonitorObject();

**boolean wasSignalled = false;**

public void doWait(){

synchronized(myMonitorObject){

if(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

**\\\\**

## Spurious Wakeups

For inexplicable reasons it is possible for threads to wake up even if notify() and notifyAll() has not been called. This is known as spurious wakeups. Wakeups without any reason. This could cause serious problems in your application.

To guard against spurious wakeups the signal member variable is checked inside a while loop instead of inside an if-statement . Such a while loop is also called a **spin lock.**

public class MyWaitNotify3{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

**while(!wasSignalled){**

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify(); } }

}

## Deadlock Prevention

1. [Lock Ordering](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#ordering)
2. [Lock Timeout](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#timeout)
3. [Deadlock Detection](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#detection)

## 1 Lock Ordering

It can only be used if you know about all locks needed ahead of taking any of the locks.

## Lock Timeout

put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up.

If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry.

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

A problem with the lock timeout mechanism is that it is not possible to set a timeout for entering a synchronized block in Java. You have to create a custom lock class.

## [Deadlock Detection](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#detection)

It is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.

Prepare a wait-for graph.

**Q) Java concurency? How to create a thread pool?**

Creating a running thread in Java is an expensive operation, and an operating system may limit the number of threads provided to a running application at any one time.