

Java Threads and Concurrency

Efthimios Alepis



Concurrency 1/2

- Concurrency is the ability of a program to execute several computations simultaneously
- This can be achieved by distributing the computations over the available CPU cores of a machine or even over different machines within the same network
- Processes are an execution environment provided by the operating system that has its own set of private resources (e.g. memory, open files, etc.)
- Threads in contrast are processes that live within a process and share their resources (memory, open files, etc.) with the other threads of the process
- The ability to share resources between different threads makes threads more suitable for tasks where performance is a significant requirement

Concurrency 2/2

- Though it is possible to establish an inter-process communication between different processes running on the same machine or even on different machines within the same network, for performance reasons, threads are often chosen to parallelize the computation on a single machine
- Each program has at least one thread: the main thread
- This main thread is created during the start of each Java application and it is the one that calls the `main()` method of your program
- From this point on, the Java application can create new Threads and work with them

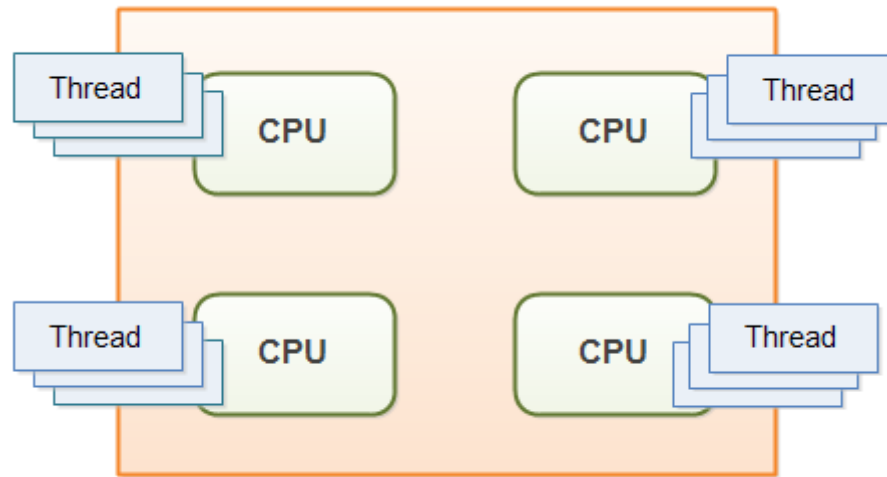
What is a thread?

- A thread is an independent path of execution through program code
- Threads can be managed independently by a scheduler, which is typically a part of the operating system
- Multiple threads can exist within the same process and share resources such as memory
- On a multiprocessor or multi-core system, threads can be executed in a true concurrent manner, with every processor or core executing a separate thread simultaneously

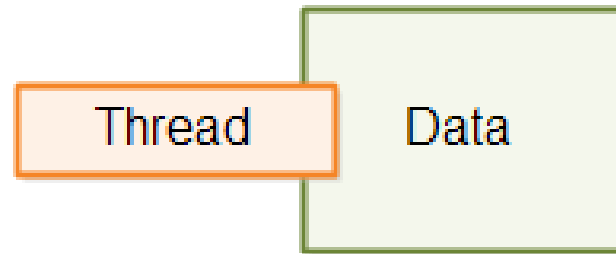
Multithreading Advantages

- Responsiveness
- Faster execution
- Lower resource consumption
- Better system utilization
- Parallelization

Multiple CPUs, Multicore CPUs

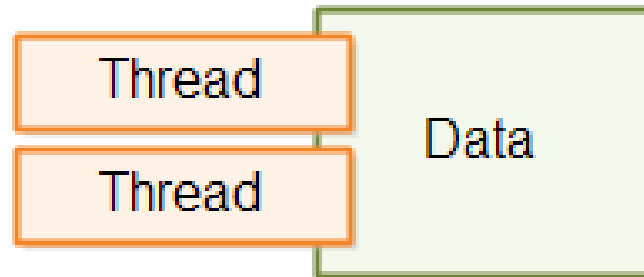


Single threaded system



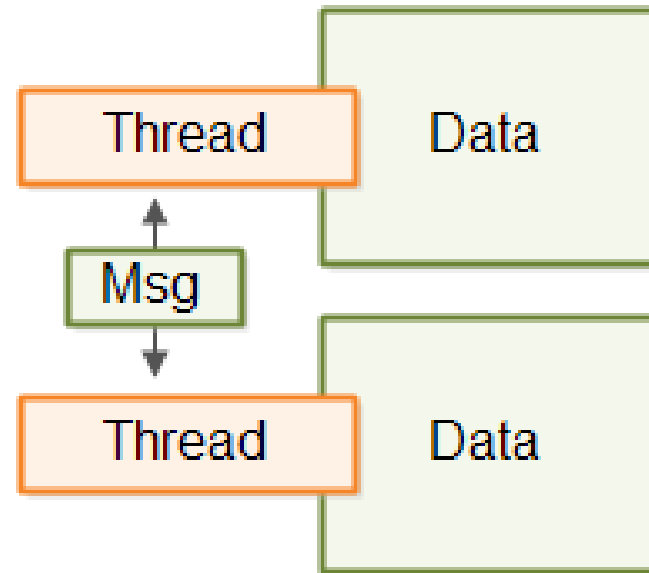
Single-threaded System

Multi threaded system



Multi-threaded System

Same threaded with messaging

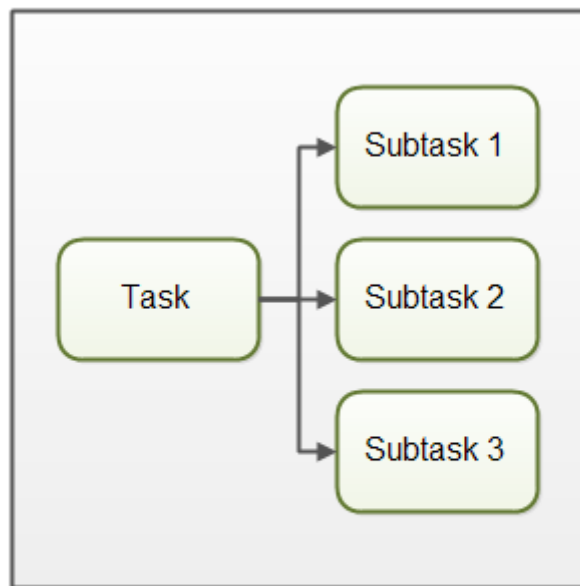


Same-threaded System

Concurrency vs. Parallelism

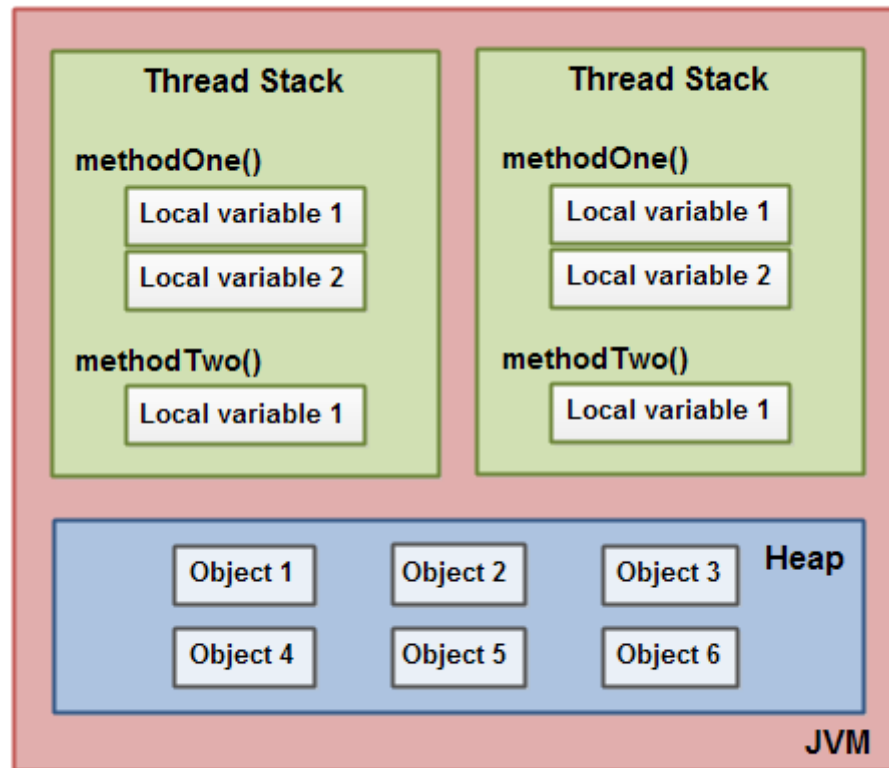


Concurrency:
Multiple tasks makes progress at the same time.

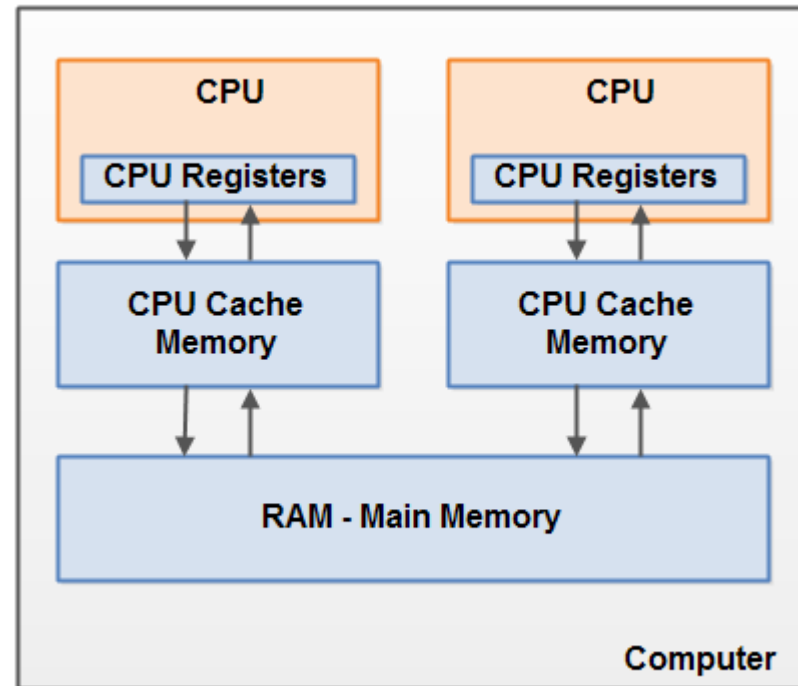


Parallelism:
Each task is broken into subtasks which can be processed in parallel.

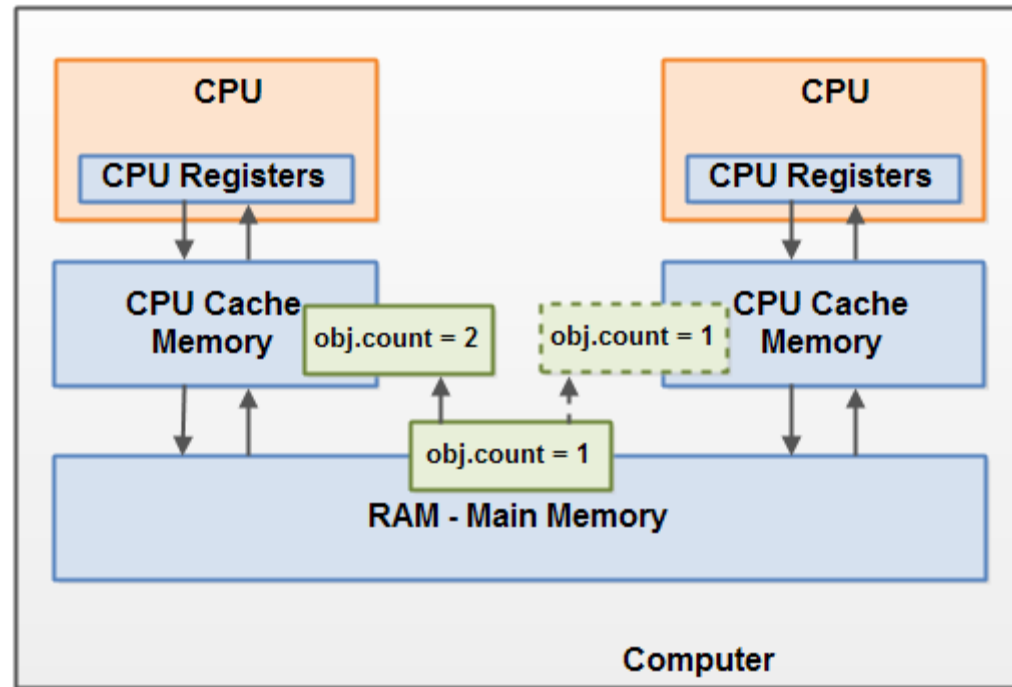
The JVM Memory Model



The Hardware Memory Model



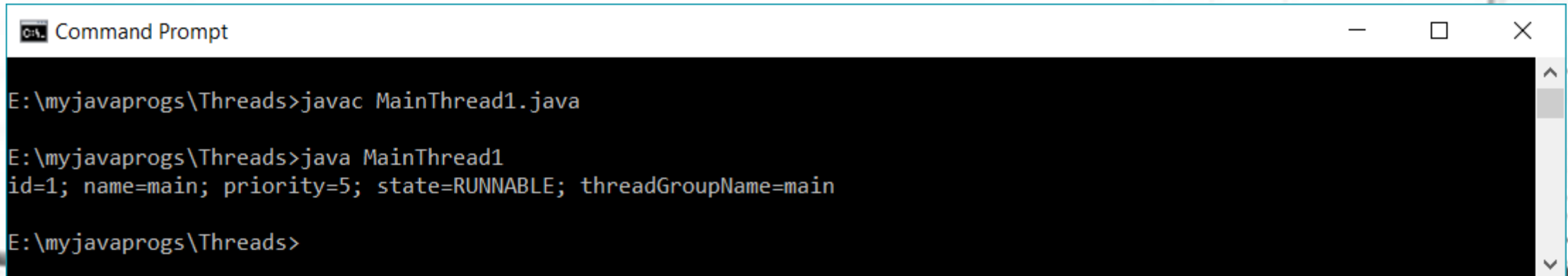
Threads running on different CPUs



Lets start with some basic thread info

```
import java.lang.*;
public class MainThread1 {

    public static void main(String[] args) {
        long id = Thread.currentThread().getId();
        String name = Thread.currentThread().getName();
        int priority = Thread.currentThread().getPriority();
        Thread.State state = Thread.currentThread().getState();
        String threadGroupName = Thread.currentThread().getThreadGroup().getName();
        System.out.println("id="+id+"; name="+name+"; priority="+priority+"; state="+state+"; threadGroupName="+
            threadGroupName);
    }
}
```



```
C:\> Command Prompt

E:\myjavaprogs\Threads>javac MainThread1.java

E:\myjavaprogs\Threads>java MainThread1
id=1; name=main; priority=5; state=RUNNABLE; threadGroupName=main

E:\myjavaprogs\Threads>
```

Thread States

- **NEW:** A thread that has not yet started is in this state
- **RUNNABLE:** A thread executing in the Java virtual machine is in this state
- **BLOCKED:** A thread that is blocked waiting for a monitor lock is in this state
- **WAITING:** A thread that is waiting indefinitely for another thread to perform a particular action is in this state
- **TIMED_WAITING:** A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state
- **TERMINATED:** A thread that has exited is in this state

Creating and starting threads

The background of the slide is a blurred, high-angle photograph of a modern building's interior. It shows a glass railing in the foreground and a window on the right side, with light streaming in from the window. The overall aesthetic is clean and architectural.

Extending the Thread class

```
class MysimpleThread extends Thread
{
    public void run ()
    {
        System.out.println ("Hallo from MysimpleThread");
    }
}
```

Using our thread

```
public class ThreadTester1
{
    public static void main (String [] args)
    {
        MysimpleThread mt = new MysimpleThread();
        mt.start ();
    }
}
```



A screenshot of a Windows command prompt window. The title bar shows the path `C:\Windows\system32\cmd.exe`. The window contains the following text:

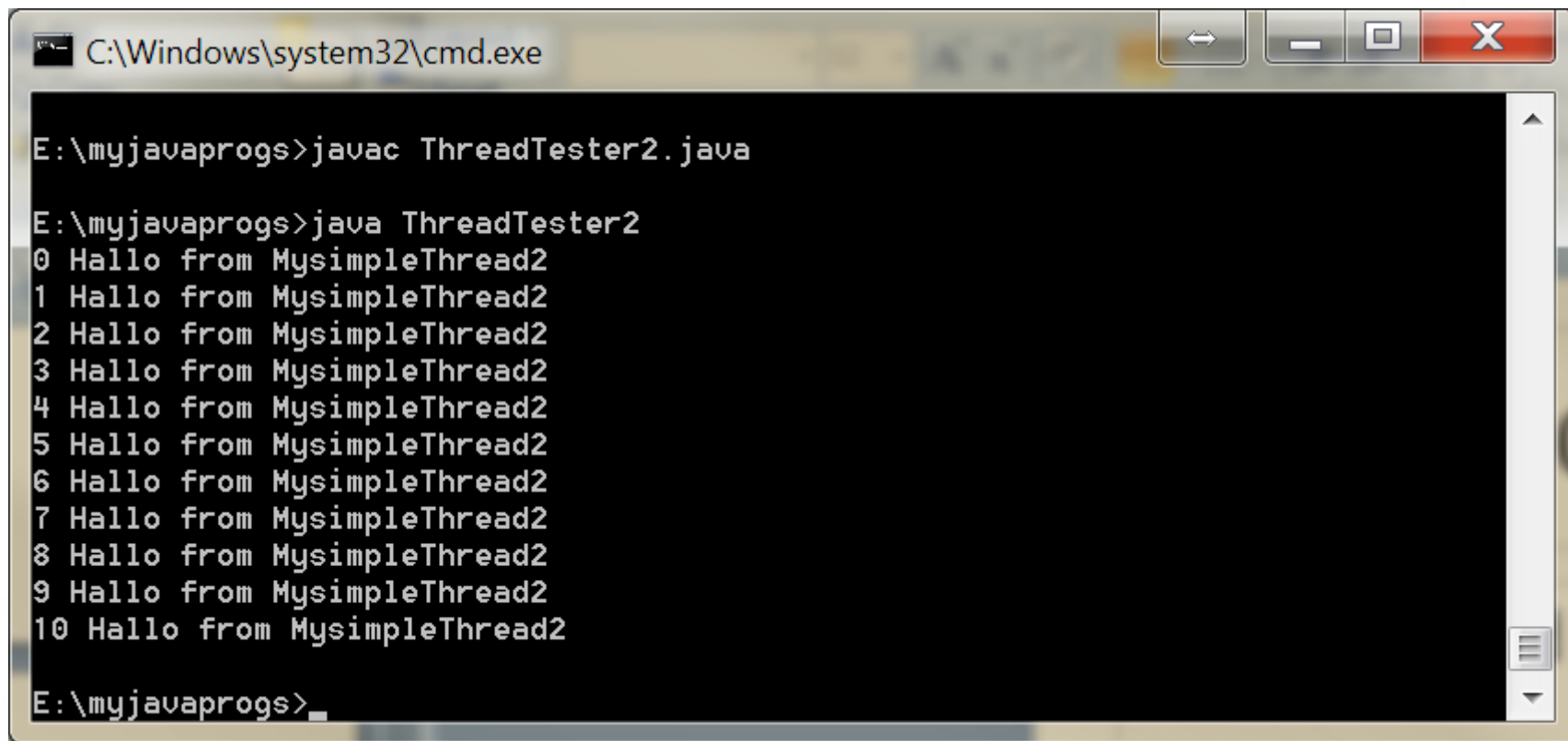
```
E:\myjavaprogs>javac ThreadTester1.java  
  
E:\myjavaprogs>java ThreadTester1  
Hallo from MysimpleThread  
  
E:\myjavaprogs>
```

Example 1/2

```
class MysimpleThread2 extends Thread
{
    public void run ()
    {
        for (int i=0;i<=10;i++)
        {
            try
            {
                Thread.sleep (1000); // Sleep for 1 second
            }
            catch (InterruptedException e)
            {
            }
            System.out.println (i+" Hallo from MysimpleThread2");
        }
    }
}
```

Example 2/2

```
public class ThreadTester2
{
    public static void main (String [] args)
    {
        MysimpleThread2 mt2 = new MysimpleThread2 ();
        mt2.start ();
    }
}
```



```
C:\Windows\system32\cmd.exe

E:\myjavaprogs>javac ThreadTester2.java

E:\myjavaprogs>java ThreadTester2
0 Hallo from MysimpleThread2
1 Hallo from MysimpleThread2
2 Hallo from MysimpleThread2
3 Hallo from MysimpleThread2
4 Hallo from MysimpleThread2
5 Hallo from MysimpleThread2
6 Hallo from MysimpleThread2
7 Hallo from MysimpleThread2
8 Hallo from MysimpleThread2
9 Hallo from MysimpleThread2
10 Hallo from MysimpleThread2

E:\myjavaprogs>
```


Implementing the Runnable Interface

```
public class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hallo from runnable");  
    }  
}
```

Using it

```
public class ThreadTester3
{
    public static void main (String [] args)
    {
        Thread mt2 = new Thread(new MyRunnable());
        mt2.start ();
    }
}
```



```
C:\Windows\system32\cmd.exe

E:\myjavaprogs>javac ThreadTester3.java

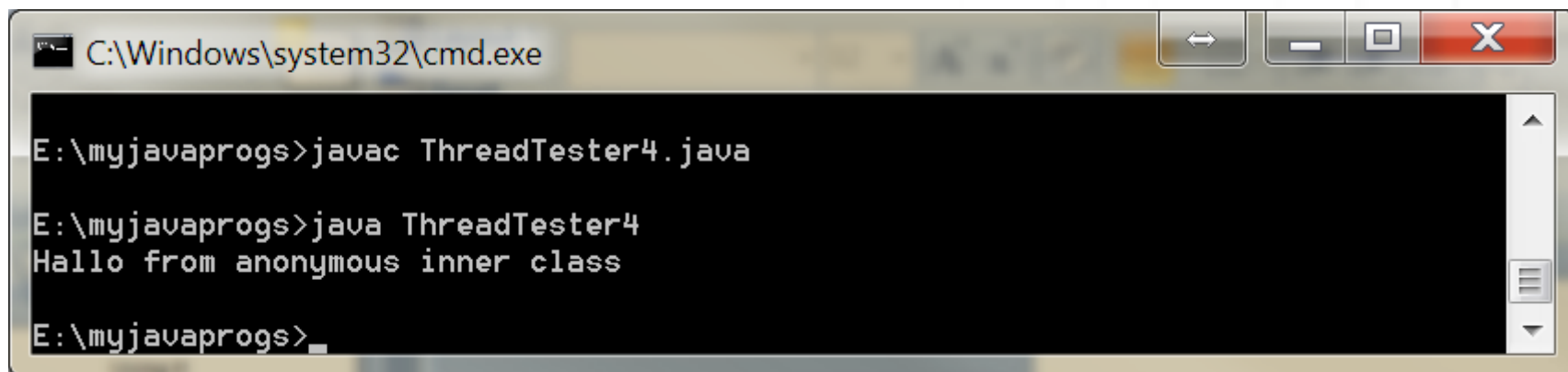
E:\myjavaprogs>java ThreadTester3
Hallo from runnable

E:\myjavaprogs>
```

Through an anonymous class

```
public class ThreadTester4
{
    public static void main (String [] args)
    {
        Thread t = new Thread() {
            public void run() {
                System.out.println("Hallo from anonymous inner class");
            }
        };
        t.start();
    }
}
```





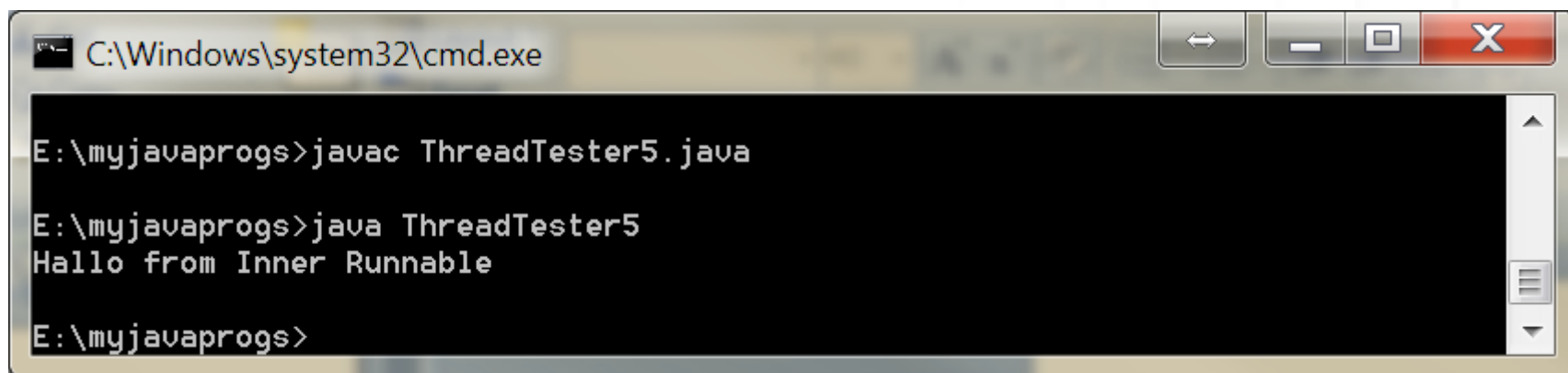
A screenshot of a Windows command prompt window. The title bar shows the path `C:\Windows\system32\cmd.exe`. The window contains the following text:

```
E:\myjavaprogs>javac ThreadTester4.java  
  
E:\myjavaprogs>java ThreadTester4  
Hallo from anonymous inner class  
  
E:\myjavaprogs>_
```

Through anonymous inner class that implements runnable interface

```
public class ThreadTester5
{
    public static void main (String [] args)
    {
        Runnable myRunnable = new Runnable() {
            public void run() {
                System.out.println("Hallo from Inner Runnable");
            }
        };
        Thread t = new Thread(myRunnable);
        t.start();
    }
}
```





A screenshot of a Windows command prompt window. The title bar shows the path `C:\Windows\system32\cmd.exe`. The window contains the following text:

```
E:\myjavaprogs>javac ThreadTester5.java  
  
E:\myjavaprogs>java ThreadTester5  
Hallo from Inner Runnable  
  
E:\myjavaprogs>
```


Pausing Thread Execution with Sleep

- Thread.sleep causes the current thread to suspend execution for a specified period
- This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system
- Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond

Make a thread sleep

- An invocation of `sleep()` puts the current Thread to sleep without consuming any processing time
- This means the current thread removes itself from the list of active threads and the scheduler doesn't schedule it for the next execution until the specified time has passed
- The time passed to the `sleep()` method is only an indication for the scheduler and not an absolutely exact time frame

Thread.sleep()

- Thread.sleep can throw an InterruptedException which is a checked exception
- All checked exceptions must either be caught and handled or else you must declare that your method can throw it
- Not declaring a checked exception that your method can throw is a compile error

Thread.sleep and InterruptedException

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
    // handle the exception...  
    // For example consider calling Thread.currentThread().interrupt(); here.  
}
```

Or declare that your method can throw an `InterruptedException` :

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

Joining Threads

- Waiting for threads to finish their work is quite useful in many cases
- Because the while loop/isAlive() method/sleep() method technique proves useful, it is packaged into some methods:
 - `join()`, `join(long millis)`, and `join(long millis, int nanos)`.

join()

- The current thread calls `join()`, via another thread's thread object reference when it wants to wait for that other thread to terminate
- The current thread calls `join(long millis)` or `join(long millis, int nanos)` when it wants to either wait for that other thread to terminate or wait until a combination of millis milliseconds and nanos nanoseconds passes

Joining Threads

```
import java.util.Random;
public class JoinExample1 implements Runnable {
    private Random rand = new Random(System.currentTimeMillis());

    public void run() {
        //simulate some CPU expensive task
        for(int i=0; i<100000000; i++) {
            rand.nextInt();
        }
        System.out.println("[ "+Thread.currentThread().getName()+" ] finished.");
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[5];
        for(int i=0; i<threads.length; i++) {
            threads[i] = new Thread(new JoinExample1(), "joinThread-"+i);
            threads[i].start();
        }
        for(int i=0; i<threads.length; i++) {
            threads[i].join();
        }
        System.out.println("[ "+Thread.currentThread().getName()+" ] All threads have finished.");
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>javac joinexample1.java
```

```
E:\myjavaprogs\Threads>java JoinExample1
```

```
[joinThread-4] finished.  
[joinThread-3] finished.  
[joinThread-2] finished.  
[joinThread-1] finished.  
[joinThread-0] finished.  
[main] All threads have finished.
```

```
E:\myjavaprogs\Threads>java JoinExample1
```

```
[joinThread-2] finished.  
[joinThread-3] finished.  
[joinThread-0] finished.  
[joinThread-4] finished.  
[joinThread-1] finished.  
[main] All threads have finished.
```

```
E:\myjavaprogs\Threads>_
```


User Threads Vs Daemon Threads

- A *user thread* performs important work for the program's user, that must finish before the application terminates
- A *daemon thread* performs “housekeeping” and other background tasks that probably do not contribute to the application's main work but are necessary for the application to continue its main work
- Unlike user threads, daemon threads do not need to finish before the application terminates

The problem of synchronization

- The exact sequence in which all running threads are executed depends next to the thread configuration like priority also on the available CPU resources and the way the scheduler chooses the next thread to execute
- Although the behavior of the scheduler is completely deterministic, it is hard to predict which threads execute in which moment at a given point in time
- This makes access to shared resources critical as it is hard to predict which thread will be the first thread that tries to access it



The Java synchronized Keyword

- Synchronized blocks in Java are marked with the synchronized keyword
- A synchronized block in Java is synchronized on some object
- All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time
- All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block

What can we “synchronize”?

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

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.

Synchronized Blocks in Instance Methods

```
public void add(int value){  
    synchronized(this){  
        this.count += value;  
    }  
}
```

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.

```
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);  
    }  
}
```


Synchronized Blocks in Static Methods

```
public static void log2(String msg1, String msg2){  
    synchronized(MyClass.class){  
        log.writeln(msg1);  
        log.writeln(msg2);  
    }  
}
```

Examples



```
public class NotSynchronizedCounter1 implements Runnable {
    private static int counter = 0;
    public void run() {
        while(counter < 10) {
            System.out.println("[ "+Thread.currentThread().getName()+" ] before: "+counter);
            counter++;
            System.out.println("[ "+Thread.currentThread().getName()+" ] after: "+counter);
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[3];
        for(int i=0; i<threads.length; i++) {
            threads[i] = new Thread(new NotSynchronizedCounter1(), "thread-"+i);
            threads[i].start();
        }
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>javac notsynchronizedcounter1.java
```

```
E:\myjavaprogs\Threads>java NotSynchronizedCounter1
```

```
[thread-0] before: 0  
[thread-0] after: 1  
[thread-0] before: 1  
[thread-2] before: 0  
[thread-1] before: 0  
[thread-2] after: 3  
[thread-0] after: 2  
[thread-2] before: 4  
[thread-1] after: 4  
[thread-2] after: 5  
[thread-0] before: 4  
[thread-2] before: 5  
[thread-1] before: 5  
[thread-2] after: 7  
[thread-0] after: 6  
[thread-2] before: 8  
[thread-1] after: 8  
[thread-2] after: 9  
[thread-0] before: 8  
[thread-2] before: 9  
[thread-1] before: 9  
[thread-2] after: 11  
[thread-0] after: 10  
[thread-1] after: 12
```

```
public class SynchronizedCounter1 implements Runnable {
    private static int counter = 0;
    public void run() {
        while(counter < 10) {
            synchronized (SynchronizedCounter1.class) {
                System.out.println("[ "+Thread.currentThread().getName()+" ] before: "+counter);
                counter++;
                System.out.println("[ "+Thread.currentThread().getName()+" ] after: "+counter);
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[3];
        for(int i=0; i<threads.length; i++) {
            threads[i] = new Thread(new SynchronizedCounter1(), "thread-"+i);
            threads[i].start();
        }
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>java SynchronizedCounter1
```

```
[thread-0] before: 0
```

```
[thread-0] after: 1
```

```
[thread-0] before: 1
```

```
[thread-0] after: 2
```

```
[thread-0] before: 2
```

```
[thread-0] after: 3
```

```
[thread-0] before: 3
```

```
[thread-0] after: 4
```

```
[thread-0] before: 4
```

```
[thread-0] after: 5
```

```
[thread-0] before: 5
```

```
[thread-0] after: 6
```

```
[thread-0] before: 6
```

```
[thread-0] after: 7
```

```
[thread-0] before: 7
```

```
[thread-0] after: 8
```

```
[thread-0] before: 8
```

```
[thread-0] after: 9
```

```
[thread-0] before: 9
```

```
[thread-0] after: 10
```

```
[thread-2] before: 10
```

```
[thread-2] after: 11
```

```
[thread-1] before: 11
```

```
[thread-1] after: 12
```

```
E:\myjavaprogs\Threads>
```

```
public class SynchronizedCounter2 implements Runnable {
    private static int counter = 0;
    public void run() {
        synchronized (SynchronizedCounter2.class) {
            while(counter < 10) {
                System.out.println("[ "+Thread.currentThread().getName()+" ] before: "+counter);
                counter++;
                System.out.println("[ "+Thread.currentThread().getName()+" ] after: "+counter);
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[3];
        for(int i=0; i<threads.length; i++) {
            threads[i] = new Thread(new SynchronizedCounter2(), "thread-"+i);
            threads[i].start();
        }
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>java SynchronizedCounter2
```

```
[thread-0] before: 0
```

```
[thread-0] after: 1
```

```
[thread-0] before: 1
```

```
[thread-0] after: 2
```

```
[thread-0] before: 2
```

```
[thread-0] after: 3
```

```
[thread-0] before: 3
```

```
[thread-0] after: 4
```

```
[thread-0] before: 4
```

```
[thread-0] after: 5
```

```
[thread-0] before: 5
```

```
[thread-0] after: 6
```

```
[thread-0] before: 6
```

```
[thread-0] after: 7
```

```
[thread-0] before: 7
```

```
[thread-0] after: 8
```

```
[thread-0] before: 8
```

```
[thread-0] after: 9
```

```
[thread-0] before: 9
```

```
[thread-0] after: 10
```

```
E:\myjavaprogs\Threads>_
```


Synchronized access to a resource that exists once per JVM

```
public class StaticSync {  
    private static Integer sync = 0;  
  
    public void someMethod() {  
        synchronized (sync) {  
            // synchronized on ClassLoader/JVM level  
        }  
    }  
}
```

Using a static member variable of a class

Java “volatile” Visibility

- The Java volatile keyword is used to mark a Java variable as "being stored in main memory"
- Every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and every write to a volatile variable will be written to main memory, and not just to the CPU cache
- The Java volatile keyword guarantees visibility of changes to variables across threads
- *Note: Reading from and writing to main memory is more expensive than accessing the CPU cache*
- *Note: In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons*

Example

```
public class SharedObject {  
    public volatile int counter = 0;  
}
```

volatile since Java 5

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable

Thread A:

```
sharedObject.nonVolatile = 123;  
sharedObject.counter     = sharedObject.counter + 1;
```

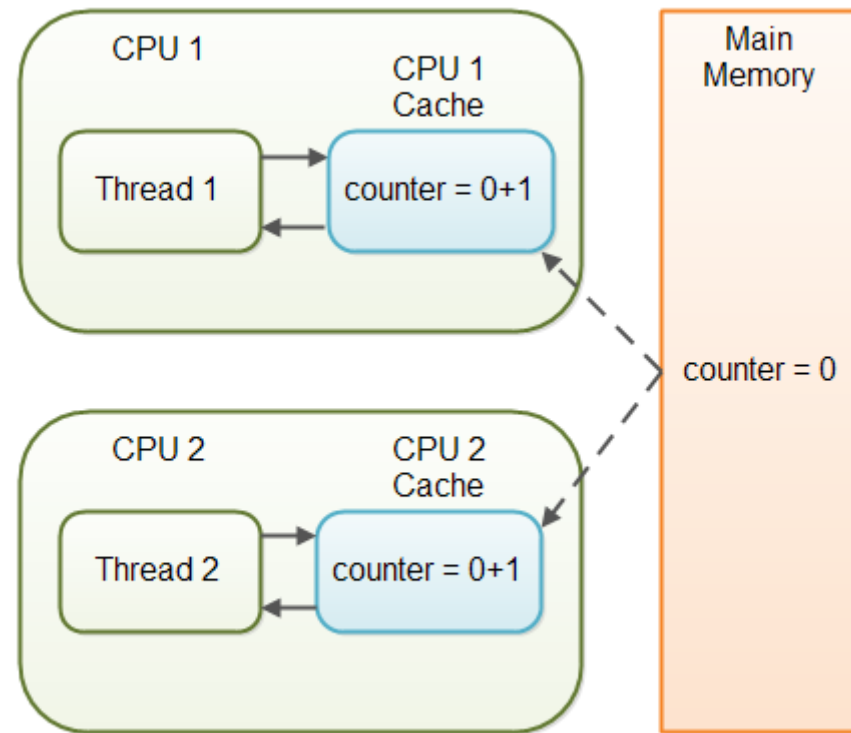
Thread B:

```
int counter      = sharedObject.counter;  
int nonVolatile = sharedObject.nonVolatile;
```

Is “volatile” enough?

- Not always, of course...!
- If a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct visibility
- The short time gap in between the reading of the volatile variable and the writing of its new value, creates a race condition where multiple threads might read the same value of the volatile variable, generate a new value for the variable, and when writing the value back to main memory - overwrite each other's values

Even with volatile



Race Conditions and Critical Sections

- A race condition is a special condition that may occur inside a critical section
- A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section
- When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition
- The term race condition stems from the metaphor that the threads are racing through the critical section

Preventing Race Conditions

- To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction
- That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section
- Thread synchronization can be achieved using (1) synchronized blocks of Java code, (2) locks, or (3) atomic variables like `java.util.concurrent.atomic.AtomicInteger`.

Discuss about the differences!

```
public class TwoSums {  
  
    private int sum1 = 0;  
    private int sum2 = 0;  
  
    public void add(int val1, int val2){  
        synchronized(this){  
            this.sum1 += val1;  
            this.sum2 += val2;  
        }  
    }  
}
```

VS

```
public class TwoSums {  
  
    private int sum1 = 0;  
    private int sum2 = 0;  
  
    private Integer sum1Lock = new Integer(1);  
    private Integer sum2Lock = new Integer(2);  
  
    public void add(int val1, int val2){  
        synchronized(this.sum1Lock){  
            this.sum1 += val1;  
        }  
        synchronized(this.sum2Lock){  
            this.sum2 += val2;  
        }  
    }  
}
```

ThreadLocal: A class that helps

- The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread
- Even if two threads are executing the same code, and the code has a reference to a ThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables

Example

```
//ThreadLocal demos  
  
private ThreadLocal myThreadLocal = new ThreadLocal();  
  
//Set a value to be stored  
myThreadLocal.set("A thread local value");  
  
//Read a value  
String threadLocalValue = (String) myThreadLocal.get();
```

The get() method returns an Object and the set() method takes an Object as parameter.

Generic Version (to avoid casting...)

```
//Generic ThreadLocal  
  
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();  
  
//Set a value to be stored  
myThreadLocal.set("A thread local value");  
  
//Read a value  
String threadLocalValue = myThreadLocal.get();
```

```
public class ThreadLocalExample {
    public static class MyRunnable implements Runnable {
        private ThreadLocal<Integer> threadLocal =
            new ThreadLocal<Integer>();
        @Override
        public void run() {
            threadLocal.set( (int) (Math.random() * 100D) );
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
            }
            System.out.println(threadLocal.get());
        }
    }
    public static void main(String[] args) {
        MyRunnable sharedRunnableInstance = new MyRunnable();
        Thread thread1 = new Thread(sharedRunnableInstance);
        Thread thread2 = new Thread(sharedRunnableInstance);
        thread1.start();
        thread2.start();
        try {
            thread1.join(); //wait for thread 1 to terminate
            thread2.join(); //wait for thread 2 to terminate
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>javac ThreadLocalExample.java
```

```
E:\myjavaprogs\Threads>java ThreadLocalExample
```

```
66
```

```
97
```

```
E:\myjavaprogs\Threads>
```

Thread Safety



- Code that is safe to call by multiple threads simultaneously is called thread safe
 - If a piece of code is thread safe, then it contains no race conditions
 - Local variables are stored in each thread's own stack
 - All local primitive variables are thread safe
 - If an object created locally never escapes the method it was created in, it is thread safe
- Object member variables (fields) 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 member variables, the method is not thread safe

Local Objects vs Member Objects

```
public void someMethod(){  
    LocalObject localObject = new LocalObject();  
  
    localObject.callMethod();  
    method2(localObject);  
}
```

VS

```
public class NotThreadSafe{  
    StringBuilder builder = new StringBuilder();  
  
    public add(String text){  
        this.builder.append(text);  
    }  
}
```


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*

Some important notes!

- Race conditions occur only if multiple threads are accessing the same resource, and one or more of the threads writes to the resource
- If multiple threads only read the same resource race conditions do not occur
- If we make sure that the objects shared between threads are never updated by any of the threads, e.g. by making the shared objects immutable, then these objects will be thread safe

Example: Value through constructor and only getter method

```
public class ImmutableValue{  
    private int value = 0;  
  
    public ImmutableValue(int value){  
        this.value = value;  
    }  
  
    public int getValue(){  
        return this.value;  
    }  
}
```

Locks



- Additionally to the monitors (through synchronized keyword), Java has support of the mutual exclusion locks
- A lock is a more flexible and sophisticated thread synchronization mechanism than the standard synchronized block
- Locks are available through `java.util.concurrent.locks` package
- Any lock must be explicitly released by calling the `unlock()` method

Lock API

- **void lock()** – acquire the lock if it's available; if the lock is not available a thread gets blocked until the lock is released
- **void lockInterruptibly()** – this is similar to the lock(), but it allows the blocked thread to be interrupted and resume the execution through a thrown `java.lang.InterruptedExecution`
- **boolean tryLock()** – this is a non-blocking version of lock() method; it attempts to acquire the lock immediately, return true if locking succeeds
- **boolean tryLock(long timeout, TimeUnit timeUnit)** – this is similar to tryLock(), except it waits up the given timeout before giving up trying to acquire the Lock
- **void unlock()** – unlocks the Lock instance

Code Snippets

```
private final ReentrantLock lock = new ReentrantLock();

public void performAction() {
    lock.lock();

    try {
        // Some implementation here
    } finally {
        lock.unlock();
    }
}

public void performActionWithTimeout() throws InterruptedException {
    if( lock.tryLock( 1, TimeUnit.SECONDS ) ) {
        try {
            // Some implementation here
        } finally {
            lock.unlock();
        }
    }
}
```

Differences between Lock and Synchronized block

- A synchronized block is fully contained within a method – We can have Lock API's lock() and unlock() operation in separate methods
- A synchronized block does not support the fairness: any thread can acquire the lock once released, no preference can be specified - We can achieve fairness within the Lock APIs by specifying the fairness property: It makes sure that longest waiting thread is given access to lock
- A thread gets blocked if it can't get an access to the synchronized block - The Lock API provides the tryLock() method: the thread acquires lock only if it's available and not held by any other thread. This reduces blocking time of thread waiting for the lock
- A thread which is in "waiting" state to acquire the access to synchronized block, can't be interrupted - The Lock API provides a method, lockInterruptibly(), which can be used to interrupt the thread when it is waiting for the lock

Thread Signaling

- The purpose of thread signaling is to enable threads to send signals to each other
- Additionally, thread signaling enables threads to wait for signals from other threads
- In order to achieve thread signaling there is a number of proposed techniques/solutions



Signaling via Shared Objects

- A simple way for threads to send signals to each other is by setting the signal values in some shared object variable
- Thread A may set a Boolean member variable to true from inside a synchronized block, and thread B may try to read the member variable, also inside a synchronized block
- Thread A and B must have a reference to a shared instance for the signaling to work
- If threads A and B have references to different instances, they will not detect each others signals

Pros & Cons of this solution

- Pros:
 - Easy to understand
 - Easy to implement
 - It works for short waiting periods
- Cons:
 - “Busy Wait”(One of the threads is using the CPU, while waiting...): Busy waiting is not a very efficient utilization of the CPU in the computer running the waiting thread, except if the average waiting time is very small



Java built-in signaling mechanisms

- Class `java.lang.Object` defines three methods, `wait()`, `notify()`, and `notifyAll()`
- A thread that calls `wait()` on any object becomes inactive until another thread calls `notify()` on that object
- The `notifyAll()` method will wake all threads waiting on a given object
- In order to invoke either `wait()`, `notify()`, or `notifyAll()`, on any object, the calling thread must first obtain the lock on that object
- In other words, the calling thread must call these methods from inside a synchronized block

General code structure

```
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();
        }
    }
}
```

Time for examples

without and with `wait()-notify()`



```
import java.util.Random;
```

```
public class BadWaitNotifyDemo {  
    private static String message;  
    public static void main (String[] args) {  
        Random random = new Random();  
        int r = random.nextInt(5);  
        Thread thread1 = new Thread(() -> {  
            int counter=0;  
            while (message == null) {  
                counter++;  
                System.out.println(Thread.currentThread().getName()+" : waiting..." +counter);  
            }  
            System.out.println(Thread.currentThread().getName()+" : Received:" +message);  
            System.out.println(Thread.currentThread().getName()+" : Ok, finished!..");  
        });  
        thread1.setName("thread1");  
        Thread thread2 = new Thread(() -> {  
            try {  
                Thread.sleep(r*1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            message = "A message from thread2";  
        });  
        thread1.start();  
        thread2.start();  
    }  
}
```

Command Prompt

```
thread1: waiting...12669
thread1: waiting...12670
thread1: waiting...12671
thread1: waiting...12672
thread1: waiting...12673
thread1: waiting...12674
thread1: waiting...12675
thread1: waiting...12676
thread1: waiting...12677
thread1: waiting...12678
thread1: waiting...12679
thread1: waiting...12680
thread1: waiting...12681
thread1: waiting...12682
thread1: waiting...12683
thread1: waiting...12684
thread1: waiting...12685
thread1: waiting...12686
thread1: waiting...12687
thread1: waiting...12688
thread1: waiting...12689
thread1: waiting...12690
thread1: waiting...12691
thread1: waiting...12692
thread1: waiting...12693
thread1: waiting...12694
thread1: Received:A message from thread2
thread1: Ok, finished!..
```

```
E:\myjavaprogs\Threads>
```



```
import java.util.Random;
```

```
public class GoodWaitNotifyDemo {  
    private static String message;  
    public static void main (String[] args) {  
        Object mylock = new Object();  
        Random random = new Random();  
        int r = random.nextInt(5);  
        Thread thread1 = new Thread(() -> {  
            synchronized (mylock){  
                while (message == null) {  
                    try {  
                        mylock.wait();  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
                System.out.println(Thread.currentThread().getName()+": Received:"+message);  
                System.out.println(Thread.currentThread().getName()+": Ok, finished!..");  
            }  
        });  
        thread1.setName("thread1");  
        Thread thread2 = new Thread(() -> {  
            synchronized (mylock){  
                try {  
                    Thread.sleep(r*1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                message = "A message from thread2";  
                mylock.notify();  
            }  
        });  
        thread1.start();  
        thread2.start();  
    }  
}
```


Command Prompt

```
thread1: waiting...12675
thread1: waiting...12676
thread1: waiting...12677
thread1: waiting...12678
thread1: waiting...12679
thread1: waiting...12680
thread1: waiting...12681
thread1: waiting...12682
thread1: waiting...12683
thread1: waiting...12684
thread1: waiting...12685
thread1: waiting...12686
thread1: waiting...12687
thread1: waiting...12688
thread1: waiting...12689
thread1: waiting...12690
thread1: waiting...12691
thread1: waiting...12692
thread1: waiting...12693
thread1: waiting...12694
thread1: Received:A message from thread2
thread1: Ok, finished!..
```

```
E:\myjavaprogs\Threads>javac GoodWaitNotifyDemo.java
```

```
E:\myjavaprogs\Threads>java GoodWaitNotifyDemo
thread1: Received:A message from thread2
thread1: Ok, finished!..
```

```
E:\myjavaprogs\Threads>_
```

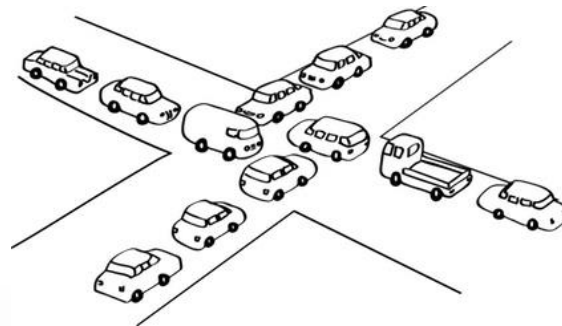
Exercise!

- What do you think will happen if in the previous example the one thread calls notify (once) before the other thread is put in a “waiting” condition?
- Make some tests and try to see if there is a problem
- If you find a problem, suggest/write some code (and an example) of how this could be solved



Deadlocks 1/2

- Lock management is really hard and full of pitfalls
- The most infamous of them is deadlock: a situation in which two or more competing threads are waiting for each other to proceed and thus neither ever does so
- Deadlocks usually occur when more than one locks or monitor locks are involved
- Sometimes JVM is able to detect the deadlocks in the running applications and warn the developers



Deadlocks 2/2

- A deadlock occurs when two or more threads are blocked waiting to obtain locks that some of the other threads involved are holding
- Deadlocks can occur when multiple threads need the same locks, at the same time, but obtain them in different order
- E.g. if thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises. Thread 1 can never get B, and thread 2 can never get A

```
Thread 1  locks A, waits for B  
Thread 2  locks B, waits for A
```



```
private final ReentrantLock lock1 = new ReentrantLock();
private final ReentrantLock lock2 = new ReentrantLock();

public void performAction() {
    lock1.lock();
    try {
        // Some implementation here
        try {
            lock2.lock();
            // Some implementation here
        } finally {
            lock2.unlock();
        }
        // Some implementation here
    } finally {
        lock1.unlock();
    }
}

public void performAnotherAction() {
    lock2.lock();
    try {
        // Some implementation here
        try {
            lock1.lock();
            // Some implementation here
        } finally {
            lock1.unlock();
        }
        // Some implementation here
    } finally {
        lock2.unlock();
    }
}
```

More Complicated Deadlocks

```
Thread 1 locks A, waits for B  
Thread 2 locks B, waits for C  
Thread 3 locks C, waits for D  
Thread 4 locks D, waits for A
```

Or even Database Deadlocks

```
Transaction 1, request 1, locks record 1 for update  
Transaction 2, request 1, locks record 2 for update  
Transaction 1, request 2, tries to lock record 2 for update.  
Transaction 2, request 2, tries to lock record 1 for update.
```

Deadlock Prevention

- Some basic techniques:
 - ✓ Lock Ordering
 - ✓ Lock Timeout
 - ✓ Deadlock Detection

Lock Ordering

- Deadlock occurs when multiple threads need the same locks but obtain them in different order
- If we are able to make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur
- Lock ordering is a simple yet effective deadlock prevention mechanism
- However, it can only be used if you know about all locks needed ahead of taking any of the locks, which is not always possible

Example

Thread 1:

lock A
lock B

Thread 2:

wait for A
lock C (when A locked)

Thread 3:

wait for A
wait for B
wait for C

Lock Timeout

- Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for a specific amount of time before giving up
- If a thread does not succeed in taking all necessary locks within the given timeout, it will (1) backup, (2) free all locks taken, (3) wait for a random amount of time and then (4) retry
- The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking
- An issue to keep in mind is, that just because a lock times out it does not necessarily mean that the threads had deadlocked

Example

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

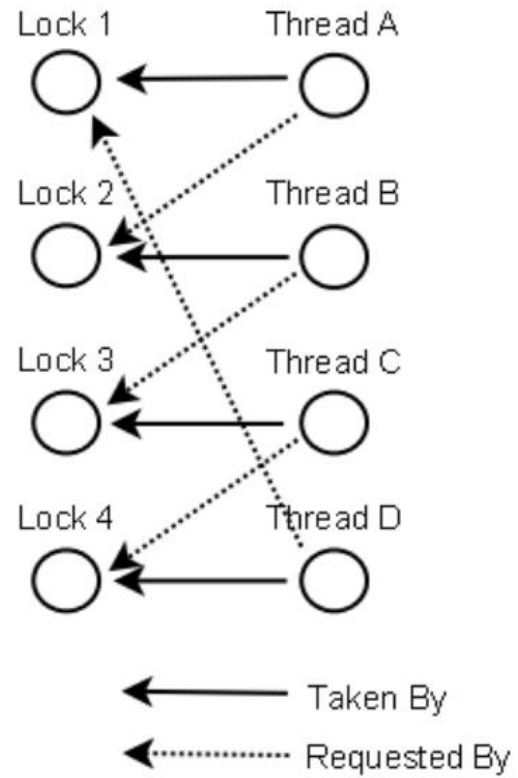
Thread 2 backs up and releases B as well

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

Deadlock Detection

- Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible
- Every time a thread takes a lock it is noted in a data structure (e.g. map, graph, etc.) of threads and locks
- Additionally, whenever a thread requests a lock this is also noted in this data structure
- When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks
- In case of a deadlock detection one possible action is to release all locks, backup, wait a random amount of time and then retry

Example in a graphical illustration



Executors and Thread Pools

- Java standard library provides extremely useful abstractions in the form of executors and thread pools targeted to simplify threads management
- In its simplest implementation, thread pool creates and maintains a list of threads, ready to be used right away
- Applications, instead of spawning new threads every time, just borrow as many threads as needed from the pool
- Once a borrowed thread finishes its job, it is returned back to the pool, and becomes available to pick up its next task

Thread Pools

- Thread Pools are useful when you need to limit the number of threads running in your application at the same time
- There is a performance overhead associated with starting a new thread, and each thread is also allocated some memory for its stack etc.
- Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool
- Internally the tasks are inserted into a Blocking Queue which the threads in the pool are dequeuing from
- As soon as the pool has any idle threads the task is assigned to one of them and executed
- Java 5 comes with built in thread pools in the `java.util.concurrent` package

Blocking Queues

- A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full
- A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue
- A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely
- Java 5 comes with blocking queue implementations in the `java.util.concurrent` package

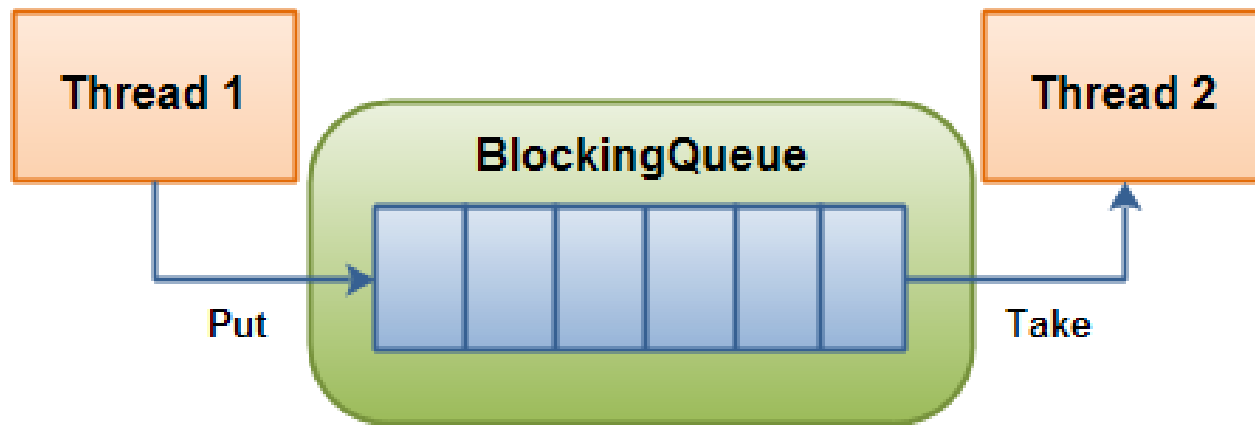
BlockingQueue interface methods

	Throws Exception	Special Value	Blocks	Times Out
Insert	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)
Remove	remove(o)	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()		

BlockingQueue implementations

- The `java.util.concurrent` package has the following implementations of the `BlockingQueue` interface (in Java 6):
 - `ArrayBlockingQueue`
 - `DelayQueue`
 - `LinkedBlockingQueue`
 - `PriorityBlockingQueue`
 - `SynchronousQueue`

Blocking Queue illustrated



Full example

Using a Producer, a Consumer and an ArrayBlockingQueue



```
class Producer implements Runnable{
    protected BlockingQueue<String> queue = null;
    public Producer(BlockingQueue<String> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            queue.put("item 1");
            System.out.println("Producer put item 1");
            Thread.sleep(2000);
            queue.put("item 2");
            System.out.println("Producer put item 2");
            Thread.sleep(2000);
            queue.put("item 3");
            System.out.println("Producer put item 3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
class Consumer implements Runnable{
    protected BlockingQueue<String> queue = null;
    public Consumer(BlockingQueue<String> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            System.out.println("Consumer Processed: "+queue.take());
            System.out.println("Consumer Processed: "+queue.take());
            System.out.println("Consumer Processed: "+queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueExample {
    public static void main(String[] args) throws Exception {
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);
        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}
```


Command Prompt

```
E:\myjavaprogs\Threads>javac BlockingQueueExample.java -Xlint
```

```
E:\myjavaprogs\Threads>java BlockingQueueExample
```

```
Producer put item 1
```

```
Consumer Processed: item 1
```

```
Producer put item 2
```

```
Consumer Processed: item 2
```

```
Producer put item 3
```

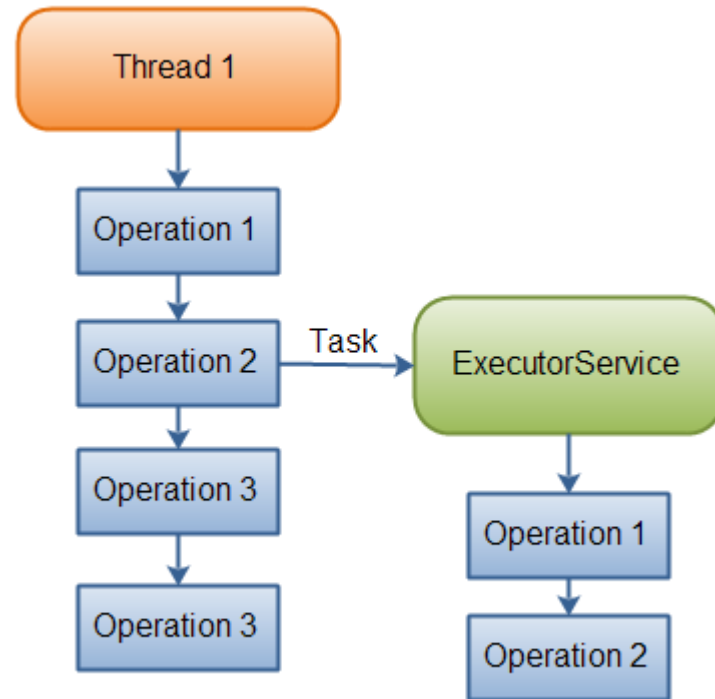
```
Consumer Processed: item 3
```

```
E:\myjavaprogs\Threads>_
```

ExecutorService

- The `java.util.concurrent.ExecutorService` interface represents an asynchronous execution mechanism which is capable of executing tasks in the background
- An `ExecutorService` is thus very similar to a thread pool
- An `ExecutorService` can be created using the `newFixedThreadPool()` factory method
- Since `ExecutorService` is an interface, you need to its implementations in order to make any use of it:
 - `ThreadPoolExecutor`
 - `ScheduledThreadPoolExecutor`
- Alternatively you can use the `Executors` factory class to create `ExecutorService` instances
- When you are done using the `ExecutorService` you should shut it down, so the threads do not keep running

ExecutorService illustrated



Basic ExecutorService instantiation

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();  
ExecutorService executorService2 = Executors.newFixedThreadPool(10);  
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

Basic ExecutorService implementation

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

ExecutorService basic tasks

- `execute(Runnable)`
- `submit(Runnable)`
- `submit(Callable)`
- `invokeAny(...)`
- `invokeAll(...)`

execute(Runnable)

- The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

submit(Runnable)

- The submit(Runnable) method also takes a Runnable implementation, but returns a **Future** object. This Future object can be used to check if the Runnable is finished executing

```
Future future = executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

future.get(); //returns null if the task has finished correctly.
```


submit(Callable)

- The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

invokeAny(...)

- The `invokeAny()` method takes a collection of `Callable` objects, or subinterfaces of `Callable`
- Invoking this method does not return a `Future`, but returns the result of one of the `Callable` objects
- You have no guarantee about which of the `Callable`'s results you get
- If one of the tasks complete (or throws an exception), the rest of the `Callable`'s are cancelled

invokeAll(...)

- The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter
- The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable
- A task might finish due to an exception, so it may not have "succeeded"
- There is no way on a "Future" to tell the difference

Some examples

Using `invokeAny()`



```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceDemo {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<Callable<String>>();
        callables.add(new Callable<String>() {
            public String call() throws Exception {
                return "Task 1";
            }
        });
        callables.add(new Callable<String>() {
            public String call() throws Exception {
                return "Task 2";
            }
        });
        callables.add(new Callable<String>() {
            public String call() throws Exception {
                return "Task 3";
            }
        });
        String result = executorService.invokeAny(callables);
        System.out.println("result = " + result);
        executorService.shutdown();
    }
}
```

```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceDemo2 {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<>();
        callables.add(() -> {return "Task 1";});
        callables.add(() -> {return "Task 2";});
        callables.add(() -> {return "Task 3";});
        String result = executorService.invokeAny(callables);
        System.out.println("result = " + result);
        executorService.shutdown();
    }
}
```




```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceDemo3 {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<>();
        callables.add(() -> "Task 1");
        callables.add(() -> "Task 2");
        callables.add(() -> "Task 3");
        String result = executorService.invokeAny(callables);
        System.out.println("result = " + result);
        executorService.shutdown();
    }
}
```



Command Prompt

```
E:\myjavaprogs\Threads>java ExecutorServiceDemo  
result = Task 1
```

```
E:\myjavaprogs\Threads>java ExecutorServiceDemo2  
result = Task 3
```

```
E:\myjavaprogs\Threads>java ExecutorServiceDemo3  
result = Task 3
```

```
E:\myjavaprogs\Threads>_
```


Example using invokeAll()

- In this example we have a list of tasks that we need them all to be executed
- We will create a list of them, then submit them to the ExecutorService
- Finally, we will wait for all the results
- P.S. we will also calculate how much time the whole operation took to finish



```
public class MyStopWatch {
    private long startTime;
    public void start() {
        startTime = System.nanoTime();
    }
    public long stop() {
        return System.nanoTime() - startTime;
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
public class ExecutorServiceDemoAll {
    static List<Callable<String>> callables = new ArrayList<>();
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        callables.clear();
        addCallableTask("Efthimios Alepis");
        addCallableTask("Good morning to everyone!");
        addCallableTask("Unipi");
        addCallableTask("Hello students of Informatics!!");
        List<Future<String>> futures = null;
        MyStopWatch stopWatch = new MyStopWatch();
        stopWatch.start();
        futures = executorService.invokeAll(callables);
        for(Future<String> future : futures){
            System.out.println("Result = " + future.get());
        }
        long duration = stopWatch.stop();
        System.out.println("Tasks took "+String.valueOf(duration)+" to complete");
        executorService.shutdown();
    }
    private static void addCallableTask(String s){
        callables.add(() -> {
            StringBuffer sb = new StringBuffer();
            return (sb.append("Length of string \"" + s + "\" is ").
                append(s.length())).toString();
        });
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>javac MyStopWatch.java
```

```
E:\myjavaprogs\Threads>javac ExecutorServiceDemoAll.java
```

```
E:\myjavaprogs\Threads>java ExecutorServiceDemoAll
```

```
Result = Length of string "Efthimios Alepis" is 16
```

```
Result = Length of string "Good morning to everyone!" is 25
```

```
Result = Length of string "Unipi" is 5
```

```
Result = Length of string "Hello students of Informatics!!" is 31
```

```
Tasks took 2480770 to complete
```

```
E:\myjavaprogs\Threads>
```

Semaphores



- The `java.util.concurrent.Semaphore` class is a counting semaphore
- The class has two main methods:
 - `acquire()`
 - `release()`
- The counting semaphore is initialized with a given number of "permits", as a simple counter
- For each call to `acquire()` a permit is taken by the calling thread
- For each call to `release()` a permit is returned to the semaphore
- At most N threads can pass the `acquire()` method without any `release()` calls, where N is the number of permits the semaphore was initialized with

Guarding Critical Sections

```
Semaphore semaphore = new Semaphore(1);  
  
//critical section  
semaphore.acquire();  
  
...  
  
semaphore.release();
```

Semaphore fairness

- No guarantees are made about fairness of the threads acquiring permits from the Semaphore => there is no guarantee that the first thread to call `acquire()` is also the first thread to obtain a permit
- If you want to enforce fairness, the Semaphore class has a constructor that takes a boolean telling if the semaphore should enforce fairness
- e.g. `Semaphore semaphore = new Semaphore(1, true);`
- Enforcing fairness comes at a performance / concurrency penalty, so don't enable it unless you need it

ForkJoinPool



- The fork/join framework was presented in Java 7
- It provides tools to help speed up parallel processing by attempting to use all available processor cores
- This is accomplished through a divide and conquer approach
- The framework first “forks”, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously
- After that, the “join” part begins, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed

Instantiation and Tasks

- In Java 8, the most convenient way to get access to the instance of the ForkJoinPool is to use its static method `commonPool()`
- `ForkJoinTask<V>` is the base type for tasks executed inside `ForkJoinPool`
- In practice, one of its two subclasses should be extended:
 - the `RecursiveAction` for void tasks
 - and the `RecursiveTask<V>` for tasks that return a value
 - ✓ Both classes have an abstract method `compute()` in which the task's logic is defined
- `ForkJoinPool`'s `invokeAll()` method is perhaps the most convenient way to submit a sequence of `ForkJoinTasks` to the `ForkJoinPool`:
 - It takes tasks as parameters (two tasks, `var args`, or a collection), forks them and returns a collection of `Future` objects in the order in which they were produced

Example

- In this example we are going to create a simple task:
 - We will uppercase a given string, that might be quite long
 - We will define a threshold which will indicate if the initial string will be divided in two substrings
 - Recursively the substrings may be divided again until the threshold is met
 - The result will be printed on the screen
 - On purpose, we don't mind about the sequence of the returned results, but only whether the job is done, divided in its parts!



```
class CustomRecursiveAction extends RecursiveAction {
    private String workload = "";
    private static final int THRESHOLD = 5;
    public CustomRecursiveAction(String workload) {
        this.workload = workload;
    }
    @Override
    protected void compute() {
        if (workload.length() > THRESHOLD) {
            ForkJoinTask.invokeAll(createSubtasks());
        } else {
            processing(workload);
        }
    }
    private List<CustomRecursiveAction> createSubtasks() {
        List<CustomRecursiveAction> subtasks = new ArrayList<>();
        String partOne = workload.substring(0, workload.length() / 2);
        String partTwo = workload.substring(workload.length() / 2, workload.length());
        subtasks.add(new CustomRecursiveAction(partOne));
        subtasks.add(new CustomRecursiveAction(partTwo));
        return subtasks;
    }
    private void processing(String work) {
        String result = work.toUpperCase();
        System.out.println("This result - (" + result + ") - was processed by "
            + Thread.currentThread().getName());
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveAction;

public class ForkJoinPoolDemo {
    public static void main(String[] args){
        ForkJoinPool commonPool = ForkJoinPool.commonPool();
        CustomRecursiveAction recursiveAction = new CustomRecursiveAction("Hello Students of Unipi. " +
            "How was your day today?");
        commonPool.invoke(recursiveAction);
    }
}
```

Command Prompt

```
E:\myjavaprogs\Threads>javac ForkJoinPoolDemo.java
```

```
E:\myjavaprogs\Threads>java ForkJoinPoolDemo
```

```
This result - (HEL) - was processed by ForkJoinPool.commonPool-worker-1  
This result - (LO ) - was processed by ForkJoinPool.commonPool-worker-7  
This result - (UNI) - was processed by ForkJoinPool.commonPool-worker-6  
This result - (STU) - was processed by ForkJoinPool.commonPool-worker-5  
This result - ( HO) - was processed by ForkJoinPool.commonPool-worker-2  
This result - (TS ) - was processed by ForkJoinPool.commonPool-worker-3  
This result - (R D) - was processed by ForkJoinPool.commonPool-worker-4  
This result - (OF ) - was processed by ForkJoinPool.commonPool-worker-3  
This result - (W W) - was processed by ForkJoinPool.commonPool-worker-2  
This result - (DEN) - was processed by ForkJoinPool.commonPool-worker-1  
This result - (PI.) - was processed by ForkJoinPool.commonPool-worker-7  
This result - (YOU) - was processed by ForkJoinPool.commonPool-worker-5  
This result - (AS ) - was processed by ForkJoinPool.commonPool-worker-2  
This result - (AY ) - was processed by ForkJoinPool.commonPool-worker-4  
This result - (AY?) - was processed by ForkJoinPool.commonPool-worker-3  
This result - (TOD) - was processed by ForkJoinPool.commonPool-worker-7
```

```
E:\myjavaprogs\Threads>
```

Optimization in synchronized blocks



BONUS

- Question: Synchronize an entire method call, or only the thread-safe subset of that method?
- Of course it depends on the actual requirements, but in the situation where the operations are identical, does it make any difference?
- To answer it is helpful to know that when the Java compiler converts this source code to byte code, it handles synchronized methods and synchronized blocks very differently
- It is also helpful to know how to look at the Java byte code in general and use it whenever you want!


```
1 package com.unipi.talepis;
2
3 public class TestSync {
4     private int counter;
5     public synchronized int getCounterV1 () {
6         return counter;
7     }
8     public int getCounterV2 () {
9         synchronized (this) {
10            return counter;
11        }
12    }
13    public static void main(String[] args) {
14        TestSync testSync = new TestSync();
15        testSync.counter = 15;
16        System.out.println(testSync.getCounterV1());
17        System.out.println(testSync.getCounterV2());
18    }
19 }
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
15
15
Process finished with exit code 0
```

Steps to follow (to view java bytecode)

- Build (or Run) your project
- Then choose from IntelliJ menu: View -> Show ByteCode
- Or use javap (the Java Class Disassembler)


```

1 package com.unipi.talepis;
2
3 public class TestSync {
4     private int counter;
5     public synchronized int
6         return counter;
7 }
8 public int getCounterV2
9     synchronized (this)
10        return counter;
11 }
12 }
13 public static void main
14     TestSync testSync =
15     testSync.counter =
16     System.out.println(
17     System.out.println(
18 }
19 }
    
```

TestSync > getCounterV2()

Run Main

```

"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
15
15
Process finished with exit code 0
    
```

Compilation completed successfully in 1s 203ms (moments ago)

```

LINENUMBER 9 L4
ALOAD 0
DUP
ASTORE 1
MONITORENTER
L0
LINENUMBER 10 L0
ALOAD 0
GETFIELD com/unipi/talepis/TestSync.counter : I
ALOAD 1
MONITOREXIT
L1
IRETURN
L2
LINENUMBER 11 L2
FRAME FULL [com/unipi/talepis/TestSync java/lang/Object] [java
ASTORE 2
ALOAD 1
MONITOREXIT
L3
ALOAD 2
ATHROW
L5
LOCALVARIABLE this Lcom/unipi/talepis/TestSync; L4 L5 0
MAXSTACK = 2
MAXLOCALS = 3

// access flags 0x9
public static main([Ljava/lang/String;)V
L0
LINENUMBER 14 L0
NEW com/unipi/talepis/TestSync
DUP
INVOKESPECIAL com/unipi/talepis/TestSync.<init> ()V
ASTORE 1
L1
LINENUMBER 15 L1
ALOAD 1
BIPUSH 15
PUTFIELD com/unipi/talepis/TestSync.counter : I
L2
LINENUMBER 16 L2
GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
ALOAD 1
INVOKEVIRTUAL com/unipi/talepis/TestSync.getCounterV1 ()I
INVOKEVIRTUAL java/io/PrintStream.println (I)V
    
```

IDE interface showing a debugger window with a stack trace and various control buttons (play, stop, step, etc.). The stack trace area is currently empty.

getCounterV1()



```
0: aload_0  
1: getfield  
4: ireturn
```

#2

```
// Field counter:I
```

3 lines of bytecode...

getCounterV2()



14 lines of bytecode...

```
0: aload_0
1: dup
2: astore_1
3: monitorenter
4: aload_0
5: getfield    #2          // Field counter:I
8: aload_1
9: monitorexit
10: ireturn
11: astore_2
12: aload_1
13: monitorexit
14: aload_2
15: athrow
```

Using javap

```
Command Prompt
E:\myjavaprogs\Threads>javap -c TestSync.class
Compiled from "TestSync.java"
public class TestSync {
  public TestSync();
    Code:
      0: aload_0
      1: invokespecial #1          // Method java/lang/Object."<init>":()V
      4: return

  public synchronized int getCounterV1();
    Code:
      0: aload_0
      1: getfield      #2          // Field counter:I
      4: ireturn

  public int getCounterV2();
    Code:
      0: aload_0
      1: dup
      2: astore_1
      3: monitorenter
      4: aload_0
      5: getfield      #2          // Field counter:I
      8: aload_1
      9: monitorexit
     10: ireturn
     11: astore_2
     12: aload_1
     13: monitorexit
     14: aload_2
     15: athrow
```

Atomic Variables in Java

- The AtomicReference class provides an object reference variable which can be read and written atomically
- By atomic is meant that multiple threads attempting to change the same AtomicReference will not end up in an inconsistent state
- You can get the reference stored in an AtomicReference using the get() method
- You can set the reference stored in an AtomicReference instance using its set() method
- The compareAndSet() method compares the reference stored in the AtomicReference instance with an expected reference, and if they two references are the same then a new reference is set on the variable

AtomicInteger

- The AtomicInteger class provides you with a int variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet()
- The AtomicInteger class is located in the `java.util.concurrent.atomic` package

AtomicInteger basics

- Creating an AtomicInteger:
 - `AtomicInteger atomicInteger = new AtomicInteger();`
- Getting the AtomicInteger Value:
 - `int theValue = atomicInteger.get();`
- Setting the AtomicInteger Value:
 - `atomicInteger.set(234);`
- Compare and Set the AtomicInteger Value:
 - `int expectedValue = 123;`
`int newValue = 234;`
`atomicInteger.compareAndSet(expectedValue, newValue);`

Some more interesting methods of AtomicInteger

- `addAndGet()`
- `getAndAdd()`
- `getAndIncrement()`
- `incrementAndGet()`

What do you think these methods do?

Class	Description
AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
AtomicStampedReference<V>	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.
LongAccumulator	One or more variables that together maintain a running long value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero long sum.

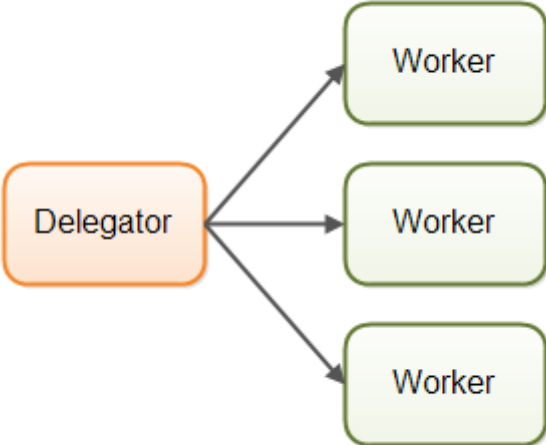
Concurrency Models

- A concurrency model specifies how threads in the system collaborate to complete the jobs they are given
- Different concurrency models split the jobs in different ways, and the threads may communicate and collaborate in different ways
- Because concurrency models are similar to distributed system architectures, they can often borrow ideas from each other

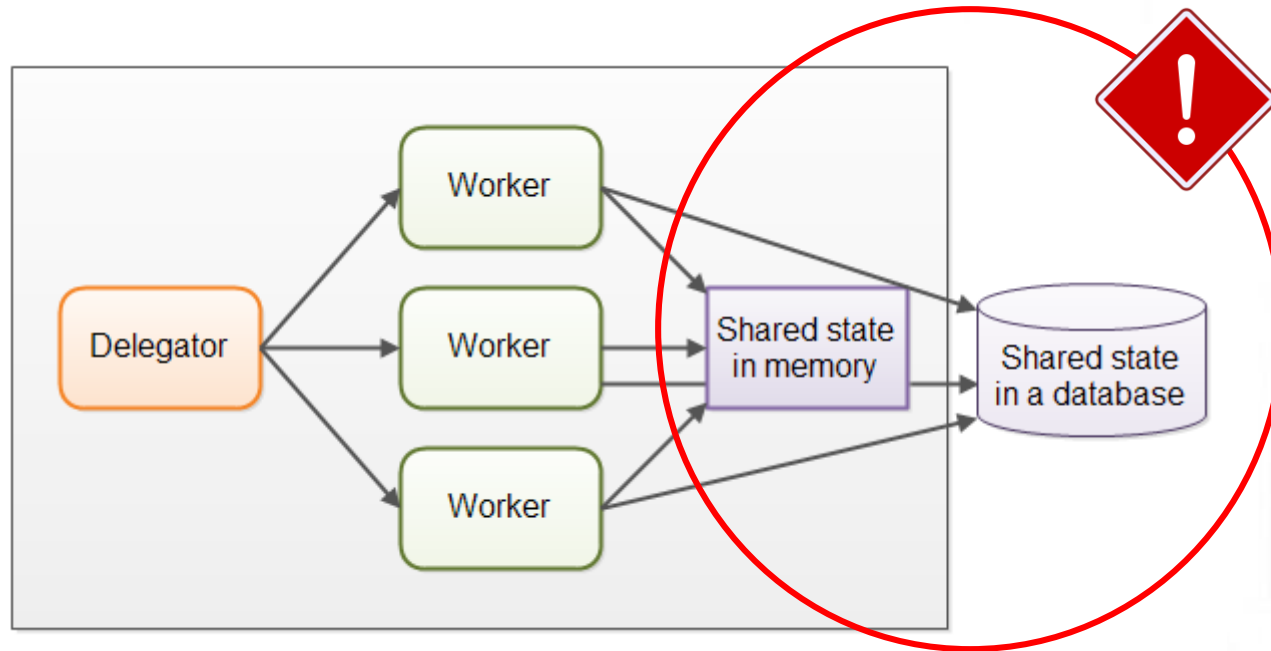
Concurrency model: Parallel Workers

- In the parallel worker concurrency model a delegator distributes the incoming jobs to different workers
- Each worker completes the full job
- The workers work in parallel, running in different threads, and possibly on different CPUs
- If the parallel worker model was implemented in a car factory, each car would be produced by one worker. The worker would get the specification of the car to build, and would build everything from start to end.
- The parallel worker concurrency model is the most commonly used concurrency model in Java applications

Parallel Workers



Parallel Workers' disadvantages



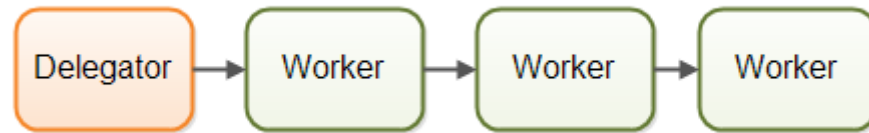
Notes on parallel workers

- A worker that does not keep state internally (but re-reads it every time it is needed) is called *stateless*
- In the parallel worker model, the job execution order is nondeterministic: There is no way to guarantee which jobs are executed first or last
- The nondeterministic nature of the parallel worker model makes it hard to reason about the state of the system at any given point in time

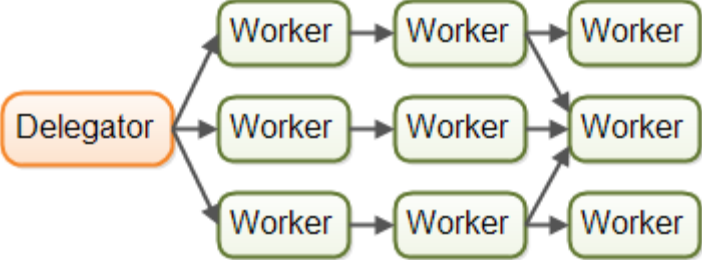
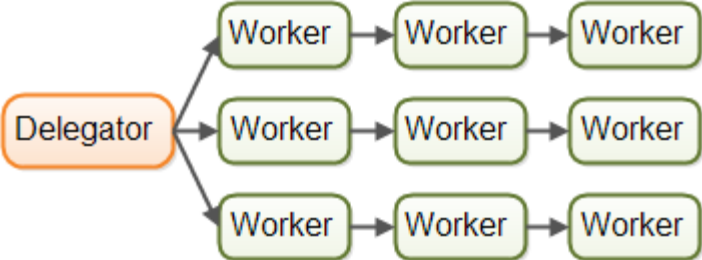
Concurrency model: Assembly Line

- The workers are organized like workers at an assembly line in a factory
- Each worker only performs a part of the full job
- When that part is finished the worker forwards the job to the next worker
- Each worker is running in its own thread, and shares no state with other workers
- This is also sometimes referred to as a shared nothing concurrency model
- Systems using the assembly line concurrency model are usually designed to use non-blocking IO

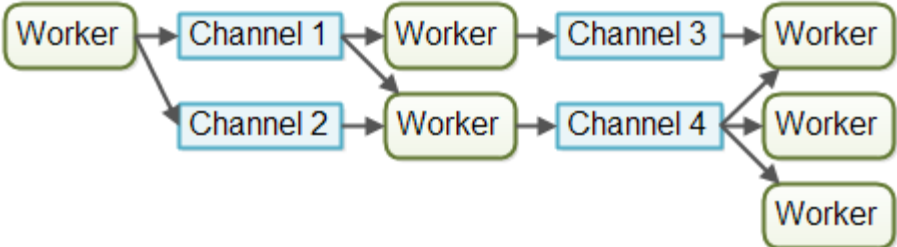
Assembly Line



Assembly Line variations



Channel Model variation



Assembly Line Advantages

- No Shared State
- Stateful Workers
- Better Hardware Conformity
- Job Ordering is Possible

Concurrency Model: Functional Parallelism

- The basic idea of functional parallelism is that you implement your program using function calls
- Functions can be seen as "agents" or "actors" that send messages to each other, just like in the assembly line concurrency model
- When one function calls another, that is similar to sending a message
- All parameters passed to the function are copied, so no entity outside the receiving function can manipulate the data
- This copying is essential to avoiding race conditions on the shared data
- This makes the function execution similar to an atomic operation
- Each function call can be executed independently of any other function call

Recommended Tools



Recommended Book (Advanced concurrency only)

