Table of Contents

1 Introduction 3

2 Multithreading Benefits 4

3 Multithreading Costs 6

4 Concurrency Models 7

5 Concurrency vs. Parallelism 15

6 The ways of define a thread 16

6.1 Extending Thread class 16

6.2 Implementing Runnable Interface 17

7 Race Conditions and Critical Sections 19

7.1 Critical Sections 19

8 Thread Safety and Shared Resources 21

9 Thread Safety and Immutability 23

10 Java Memory Model 25

11 Java Volatile Keyword 25

12 Thread Signaling 25

13 Getting and Setting Name of Thread 25

14 Thread Priorities 26

15 Methods to prevent thread execution 27

15.1 Yield() 27

15.2 join() 27

15.3 sleep() 27

16 Synchronization 28

17 Inter-thread communication 29

18 Deadlock 30

19 Deamon Thread 31

20 Starvation and Fairness 32

21 Nested Monitor Lockout 32

22 Slipped Conditions 32

23 Locks in Java 32

24 Read / Write Locks in Java 32

25 Reentrance Lockout 32

26 Semaphores 32

27 Blocking Queues 32

28 Thread Pools 32

29 Compare and Swap 32

30 Anatomy of a Synchronizer 32

31 Non-blocking Algorithms 32

32 Multithreading enhancements 32

32.1 Thread Local 32

1 Introduction

Executing several tasks simultaneously is the concept of multitasking, there are 2 types of multitasking

1. Process based
2. Thread based

Process based:

Executing several tasks simultaneously, where each task is separate independent program (process) is called process based multi task.

Ex- while typing java programs in editor, we can listen audio songs same machine at the same time we can download a file internet. All these tasks will be simultaneously and independent of each other, hence it is process based multi-tasking.

This is best suitable at operating system level.

Thread based:

Executing several tasks simultaneously, where each task is separate independent part of the same program is called thread based multi task, and each independent is part is called is thread.

This is best suitable at programmatic level to do multiple tasks.

Context switching is less, shares same memory space.

Whether is process based or thread based, the main object of multi-tasking is to reduce response time of the system and improve performance.

The main important application areas of multi-threading are -

To develope multi-media graphics

To develop animations

Video games

To develope Web server/Application Server, etc

When compared with old languages, developing multithreaded applications in java is very easy because java provide inbuilt support for multithreading, with rich API (Thread, Runnable, ThreadGroup).

2 Multithreading Benefits

The reason multithreading is still used in spite of its challenges is that multithreading can have several benefits. Some of these benefits are:

Better resource utilization.

Simpler program design in some situations.

More responsive programs.

Better resource utilization

Imagine an application that reads and processes files from the local file system. Lets say that reading af file from disk takes 5 seconds and processing it takes 2 seconds. Processing two files then takes

5 seconds reading file A

2 seconds processing file A

5 seconds reading file B

2 seconds processing file B

---

14 seconds total

When reading the file from disk most of the CPU time is spent waiting for the disk to read the data. The CPU is pretty much idle during that time. It could be doing something else. By changing the order of the operations, the CPU could be better utilized. Look at this ordering:

5 seconds reading file A

5 seconds reading file B + 2 seconds processing file A

2 seconds processing file B

---

12 seconds total

Simpler Program Design

If you were to program the above ordering of reading and processing by hand in a singlethreaded application, you would have to keep track of both the read and processing state of each file. Instead you can start two threads that each just reads and processes a single file. Each of these threads will be blocked while waiting for the disk to read its file. While waiting, other threads can use the CPU to process the parts of the file they have already read. The result is, that the disk is kept busy at all times,

reading from various files into memory. This results in a better utilization of both the disk and the CPU. It is also easier to program, since each thread only has to keep track of a single file.

More responsive programs

Another common goal for turning a singlethreaded application into a multithreaded application is to achieve a more responsive application. Imagine a server application that listens on some port for incoming requests. when a request is received, it handles the request and then goes back to listening. The server loop is sketched below:

```
while(server is active){
  listen for request
  process request
}
```

If the request takes a long time to process, no new clients can send requests to the server for that duration. Only while the server is listening can requests be received.

An alternate design would be for the listening thread to pass the request to a worker thread, and return to listening immediatedly. The worker thread will process the request and send a reply to the client. This design is sketched below:

```
while(server is active){
    listen for request
    hand request to worker thread
}
```

This way the server thread will be back at listening sooner. Thus more clients can send requests to the server. The server has become more responsive.

3 Multithreading Costs

More complex design

Though some parts of a multithreaded applications is simpler than a singlethreaded application, other parts are more complex. Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.

Context Switching Overhead

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.

Context switching isn't cheap. You don't want to switch between threads more than necessary.

Increased Resource Consumption

A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread. Try creating a program that creates 100 threads that does nothing but wait, and see how much memory the application takes when running.

4 Concurrency Models

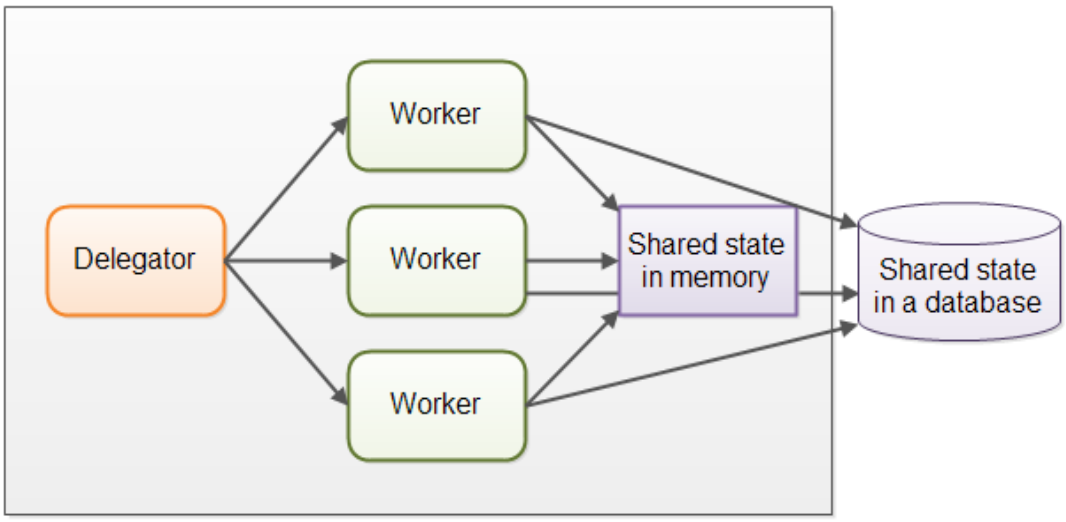
Concurrent systems can be implemented using different concurrency models. A concurrency model specifies how threads in the the system collaborate to complete the jobs they are are given. Different concurrency models split the jobs in different ways, and the threads may communicate and collaborate in different ways.

Concurrency Models and Distributed System Similarities

The concurrency models described in this text are similar to different architectures used in distributed systems. In a concurrent system different threads communicate with each other. In a distributed system different processes communicate with each other (possibly on different computers). Threads and processes are quite similar to each other in nature. That is why the different concurrency models often look similar to different distributed system architectures.

Parallel Workers

The first concurrency model is what I call the parallel worker model. Incoming jobs are assigned to different workers. Here is a diagram illustrating the parallel worker concurrency model:The first concurrency model is what I call the parallel worker model. Incoming jobs are assigned to different workers. Here is a diagram illustrating the parallel worker concurrency model:



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 (although that is changing). **Many of the concurrency utilities in the java.util.concurrent Java package are designed for use with this model. You can also see traces of this model in the design of the Java Enterprise Edition application servers.**

Parallel Workers Advantages

The advantage of the parallel worker concurrency model is that it is easy to understand. To increase the parallelization of the application you just add more workers.

For instance, if you were implementing a web crawler, you could crawl a certain amount of pages with different numbers of workers and see which number gives the shortest total crawl time (meaning the highest performance). Since web crawling is an IO intensive job you will probably end up with a few threads per CPU / core in your computer. One thread per CPU would be too little, since it would be idle a lot of the time while waiting for data to download.

Parallel Workers Disadvantages

The parallel worker concurrency model has some disadvantages lurking under the simple surface, though. I will explain the most obvious disadvantages in the following sections.

Shared State Can Get Complex

In reality the parallel worker concurrency model is a bit more complex than illustrated above. The shared workers often need access to some kind of shared data, either in memory or in a shared database. The following diagram shows how this complicates the parallel worker concurrency model:

The parallel worker concurrency model with shared state illustrated

Some of this shared state is in communication mechanisms like job queues. But some of this shared state is business data, data caches, connection pools to the database etc.

As soon as shared state sneaks into the parallel worker concurrency model it starts getting complicated. The threads need to access the shared data in a way that makes sure that changes by one thread are visible to the others (pushed to main memory and not just stuck in the CPU cache of the CPU executing the thread). Threads need to avoid race conditions, deadlock and many other shared state concurrency problems.

Additionally, part of the parallelization is lost when threads are waiting for each other when accessing the shared data structures. Many concurrent data structures are blocking, meaning one or a limited set of threads can access them at any given time. This may lead to contention on these shared data structures. High contention will essentially lead to a degree of serialization of execution of the part of the code that access the shared data structures.

Modern non-blocking concurrency algorithms may decrease contention and increase performance, but non-blocking algorithms are hard to implement.

Persistent data structures are another alternative. A persistent data structure always preserves the previous version of itself when modified. Thus, if multiple threads point to the same persistent data structure and one thread modifies it, the modifying thread gets a reference to the new structure. All other threads keep a reference to the old structure which is still unchanged and thus consistent. The Scala programming contains several persistent data structures.

While persistent data structures are an elegant solution to concurrent modification of shared data structures, persistent data structures tend not to perform that well.

For instance, a persistent list will add all new elements to the head of the list, and return a reference to the newly added element (which then point to the rest of the list). All other threads still keep a reference to the previously first element in the list, and to these threads the list appear unchanged. They cannot see the newly added element.

Such a persistent list is implemented as a linked list. Unfortunately linked lists don't perform very well on modern hardware. Each element in the list is a separate object, and these objects can be spread out all over the computer's memory. Modern CPUs are much faster at accessing data sequentially, so on modern hardware you will get a lot higher performance out of a list implemented on top of an array. An array stores data sequentially. The CPU caches can load bigger chunks of the array into the cache at a time, and have the CPU access the data directly in the CPU cache once loaded. This is not really possible with a linked list where elements are scattered all over the RAM.

Stateless Workers

Shared state can be modified by other threads in the system. Therefore workers must re-read the state every time it needs it, to make sure it is working on the latest copy. This is true no matter whether the shared state is kept in memory or in an external database. A worker that does not keep state internally (but re-reads it every time it is needed) is called stateless .

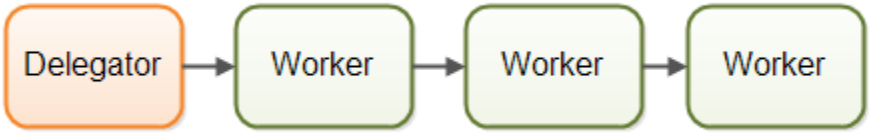
Re-reading data every time you need it can get slow. Especially if the state is stored in an external database.

Job Ordering is Nondeterministic

Another disadvantage of the parallel worker model is that the job execution order is nondeterministic. There is no way to guarantee which jobs are executed first or last. Job A may be given to a worker before job B, yet job B may be executed before job A.
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. It also makes it harder (if not impossible) to guarantee that one jobs happens before another.

Assembly Line

The second concurrency model is what I call the assembly line concurrency model. I chose that name just to fit with the "parallel worker" metaphor from earlier. Other developers use other names (e.g. reactive systems, or event driven systems) depending on the platform / community. Here is a diagram illustrating the assembly line concurrency model:



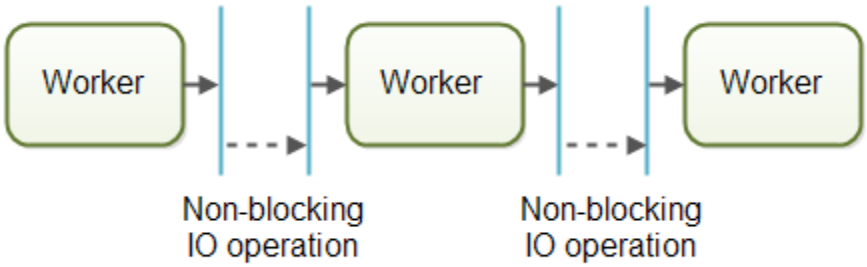
The assembly line concurrency model.

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. Non-blocking IO means that when a worker starts an IO operation (e.g. reading a file or data from a network connection) the worker does not wait for the IO call to finish. IO operations are slow, so waiting for IO operations to complete is a waste of CPU time. The CPU could be doing something else in the meanwhile. When the IO operation finishes, the result of the IO operation (e.g. data read or status of data written) is passed on to another worker.

With non-blocking IO, the IO operations determine the boundary between workers. A worker does as much as it can until it has to start an IO operation. Then it gives up control over the job. When the IO operation finishes, the next worker in the assembly line continues working on the job, until that too has to start an IO operation etc.

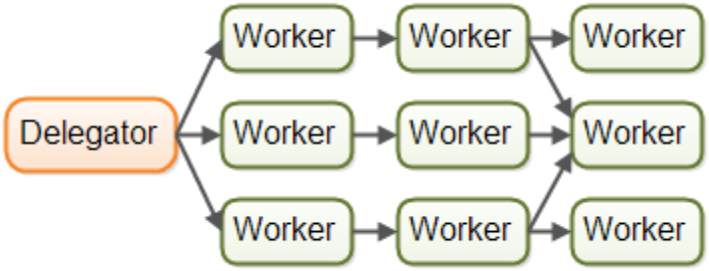


The assembly line concurrency model with non-blocking IO operations marking the boundaries between worker responsibility.

In reality, the jobs may not flow along a single assembly line. Since most systems can perform more than one job, jobs flows from worker to worker depending on the job that needs to be done. In reality there could be multiple different virtual assembly lines going on at the same time. This is how job flow through assembly line system might look in reality:

The assembly line concurrency model with multiple assembly lines.

Jobs may even be forwarded to more than one worker for concurrent processing. For instance, a job may be forwarded to both a job executor and a job logger. This diagram illustrates how all three assembly lines finish off by forwarding their jobs to the same worker (the last worker in the middle assembly line):



Reactive, Event Driven Systems

Systems using an assembly line concurrency model are also sometimes called reactive systems, or event driven systems. The system's workers react to events occurring in the system, either received from the outside world or emitted by other workers. Examples of events could be an incoming HTTP request, or that a certain file finished loading into memory etc.

At the time of writing, there are a number of interesting reactive / event driven platforms available, and more will come in the future. Some of the more popular ones seems to be:

Vert.x

Akka

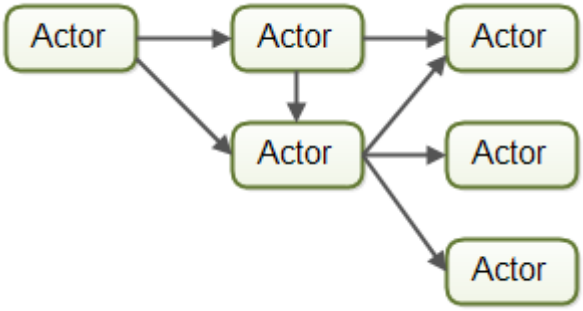
Node.JS (JavaScript)

Personally I find Vert.x to be quite interesting (especially for a Java / JVM dinosaur like me).

Actors vs. Channels

Actors and channels are two similar examples of assembly line (or reactive / event driven) models.

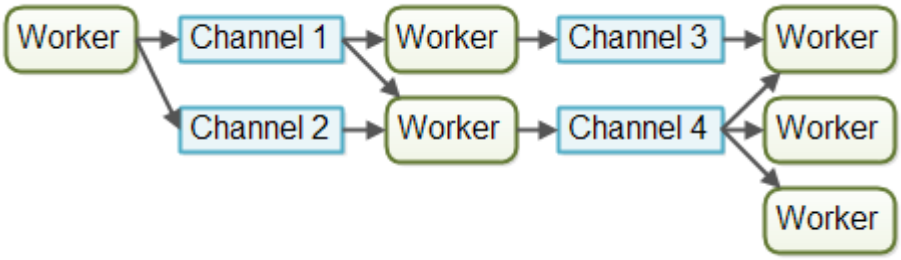
In the actor model each worker is called an actor. Actors can send messages directly to each other. Messages are sent and processed asynchronously. Actors can be used to implement one or more job processing assembly lines, as described earlier. Here is a diagram illustrating the actor model:



The assembly line concurrency model implemented using actors.

In the channel model, workers do not communicate directly with each other. Instead they publish their messages (events) on different channels. Other workers can then listen for messages on these channels without the sender knowing who is listening. Here is a diagram illustrating the channel model:

The assembly line concurrency model implemented using channels.



At the time of writing, the channel model seems more flexible to me. A worker does not need to know about what workers will process the job later in the assembly line. It just needs to know what channel to forward the job to (or send the message to etc.). Listeners on channels can subscribe and unsubscribe without affecting the workers writing to the channels. This allows for a some what looser coupling between workers.

Assembly Line Advantages

The assembly line concurrency model has several advantages compared to the parallel worker model. I will cover the biggest advantages in the following sections.

No Shared State

The fact that workers share no state with other workers means that they can be implemented without having to think about all the concurrency problems that may arise from concurrent access to shared state. This makes it much easier to implement workers. You implement a worker as if it was the only thread performing that work - essentially a singlethreaded implementation.

Stateful Workers

Since workers know that no other threads modify their data, the workers can be stateful. By stateful I mean that they can keep the data they need to operate in memory, only writing changes back the eventual external storage systems. A stateful worker can therefore often be faster than a stateless worker.

Better Hardware Conformity

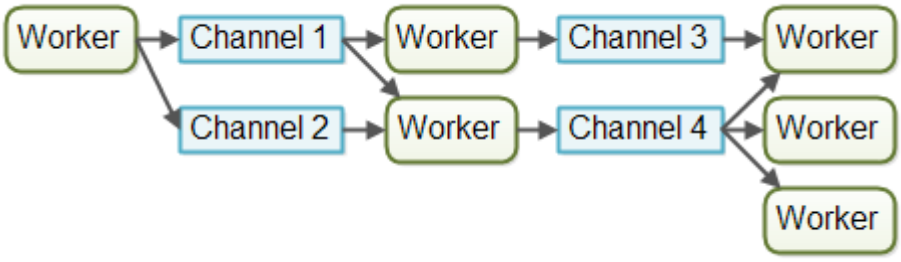
Singlethreaded code has the advantage that it often conforms better with how the underlying hardware works. First of all, you can usually create more optimized data structures and algorithms when you can assume the code is executed in single threaded mode.

Second, singlethreaded stateful workers can cache data in memory as mentioned above. When data is cached in memory there is also a higher probability that this data is also cached in the CPU cache of the CPU executing the thread. This makes accessing cached data even faster.

I refer to it as hardware conformity when code is written in a way that naturally benefits from how the underlying hardware works. Some developers call this mechanical sympathy. I prefer the term hardware conformity because computers have very few mechanical parts, and the word "sympathy" in this context is used as a metaphor for "matching better" which I believe the word "conform" conveys reasonably well. Anyways, this is nitpicking. Use whatever term you prefer.

Job Ordering is Possible

It is possible to implement a concurrent system according to the assembly line concurrency model in a way that guarantees job ordering. Job ordering makes it much easier to reason about the state of a system at any given point in time. Furthermore, you could write all incoming jobs to a log. This log could then be used to rebuild the state of the system from scratch in case any part of the system fails. The jobs are written to the log in a certain order, and this order becomes the guaranteed job order. Here is how such a design could look:



The assembly line concurrency model with a job logger.

Implementing a guaranteed job order is not necessarily easy, but it is often possible. If you can, it greatly simplifies tasks like backup, restoring data, replicating data etc. as this can all be done via the log file(s).

Assembly Line Disadvantages

The main disadvantage of the assembly line concurrency model is that the execution of a job is often spread out over multiple workers, and thus over multiple classes in your project. Thus it becomes harder to see exactly what code is being executed for a given job.

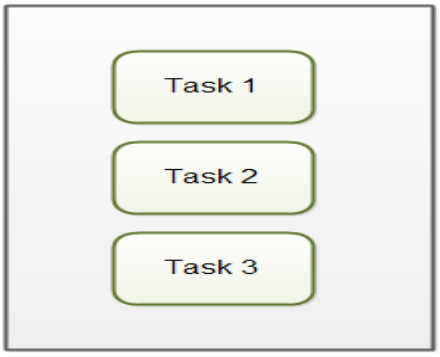
It may also be harder to write the code. Worker code is sometimes written as callback handlers. Having code with many nested callback handlers may result in what some developer call callback hell. Callback hell simply means that it gets hard to track what the code is really doing across all the callbacks, as well as making sure that each callback has access to the data it needs.

With the parallel worker concurrency model this tends to be easier. You can open the worke r code and read the code executed pretty much from start to finish. Of course parallel worker code may also be spread over many different classes, but the execution sequence is often easier to read from the code.

5 Concurrency vs. Parallelism

Concurrency

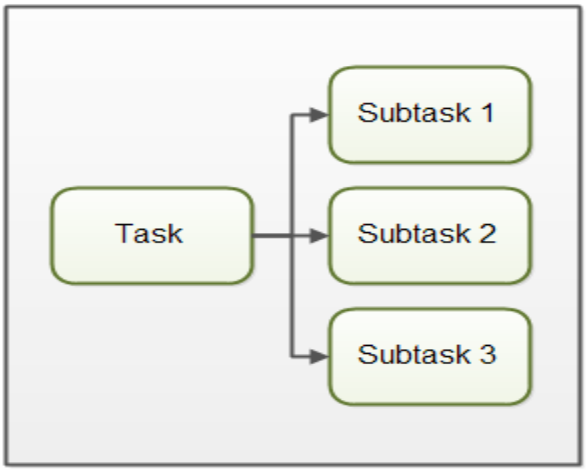
Concurrency means that an application is making progress on more than one task at the same time (concurrently). Well, if the computer only has one CPU the application may not make progress on more than one task at exactly the same time, but more than one task is being processed at a time inside the application. It does not completely finish one task before it begins the next.



Concurrency:
Multiple tasks makes progress at the same time.

Parallelism

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time.



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

6 The ways of define a thread

There are two ways to specify what code the thread should execute.

6.1 Extending Thread class

The first is to create a subclass of Thread and override the run() method. The second method is to pass an object that implements Runnable (java.lang.Runnable to the Thread constructor. Both methods are covered below.

Thread Subclass

The first way to specify what code a thread is to run, is to create a subclass of Thread and override the run() method. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

```
public class MyThread extends Thread {

  public void run(){

    System.out.println("MyThread running");

  }

}
```

To create and start the above thread you can do like this:

```
MyThread myThread = new MyThread();

myTread.start();
```

The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU. When the run() method executes it will print out the text "MyThread running".

You can also create an anonymous subclass of Thread like this:

```
Thread thread = new Thread(){

  public void run(){

    System.out.println("Thread Running");

  }

}

thread.start();
```

This example will print out the text "Thread running" once the run() method is executed by the new thread.

6.2 Implementing Runnable Interface

Runnable Interface Implementation

The second way to specify what code a thread should run is by creating a class that implements java.lang.Runnable. The Runnable object can be executed by a Thread.

Here is a Java Runnable example:

```
public class MyRunnable implements Runnable {

 public void run(){

   System.out.println("MyRunnable running");

 }

}
```

To have the run() method executed by a thread, pass an instance of MyRunnable to a Thread in its constructor. Here is how that is done:

```
Thread thread = new Thread(new MyRunnable());

thread.start();
```

When the thread is started it will call the run() method of the MyRunnable instance instead of executing it's own run() method. The above example would print out the text "MyRunnable running".

You can also create an anonymous implementation of Runnable, like this:

```
Runnable myRunnable = new Runnable(){

 public void run(){

    System.out.println("Runnable running");

  }

}

Thread thread = new Thread(myRunnable);

thread.start();
```

Subclass or Runnable?

There are no rules about which of the two methods that is the best. Both methods works. Personally though, I prefer implementing Runnable, and handing an instance of the implementation to a Thread instance. When having the Runnable's executed by a thread pool it is easy to queue up the Run nable instances until a thread from the pool is idle. This is a little harder to do with Thread subclasses.

Thread Names

When you create a Java thread you can give it a name. The name can help you distinguish different threads from each other. For instance, if multiple threads write to System.out it can be handy to see which thread wrote the text. Here is an example:

```
Thread thread = new Thread("New Thread") {
    public void run(){
      System.out.println("run by: " + getName());
    }
  };
thread.start();
System.out.println(thread.getName());

MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable, "New Thread");
thread.start();
System.out.println(thread.getName());
```

Thread.currentThread()

The Thread.currentThread() method returns a reference to the Thread instance executing currentThread() . This way you can get access to the Java Thread object representing the thread executing a given block of code. Here is an example of how to use Thread.currentThread() :

```
Thread thread = Thread.currentThread();
```

Once you have a reference to the Thread object, you can call methods on it. For instance, you can get the name of the thread currently executing the code like this:

```
String threadName = Thread.currentThread().getName();
```

Java Thread Example

Here is a small example. First it prints out the name of the thread executing the main() method. This thread is assigned by the JVM. Then it starts up 10 threads and give them all a number as name ("" + i). Each thread then prints its name out, and then stops executing.

```
public class ThreadExample {
  public static void main(String[] args){
    System.out.println(Thread.currentThread().getName());
    for(int i=0; i<10; i++){
      new Thread("" + i){
        public void run(){
```

```
            System.out.println("Thread: " + getName() + " running");
          }
        }.start();
      }
    }
}
```

7 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, and that the result of that race impacts the result of executing the critical section.

This may all sound a bit complicated, so I will elaborate more on race conditions and critical sections in the following sections.

7.1 Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a critical section Java code example that may fail if executed by multiple threads simultaneously:

```
public class Counter {
  protected long count = 0;
  public void add(long value){
     this.count = this.count + value;
  }
}
```

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the Java virtual machine. Rather it is executed as a set of smaller instructions, similar to this:

Read this.count from memory into register.
Add value to register.
Write register to memory.

The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

Race Conditions in Critical Sections

The code in the add() method in the example earlier contains a critical section. When multiple threads execute this critical section, race conditions occur.

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.

Race conditions can be avoided by proper thread synchronization in critical sections. Thread synchronization can be achieved using a synchronized block of Java code. Thread synchronization can also be achieved using other synchronization constructs like locks or atomic variables like java.util.concurrent.atomic.AtomicInteger.

Critical Section Throughput

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section.

```
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;
    }
  }
}
```

8 Thread Safety and Shared Resources

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. Race condition only occur when multiple threads update shared resources. Therefore it is important to know 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

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads'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");
}
```

The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to the localObject reference. Therefore the use of the LocalObject here is thread safe.

In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe.

The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way that allows access to it from other threads.

Object Member Variables

Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, 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);
   }
}
```

If two threads call the add() method simultaneously on the same NotThreadSafe instance then it leads to race conditions. For instance:

```
NotThreadSafe sharedInstance = new NotThreadSafe();
new Thread(new MyRunnable(sharedInstance)).start();
new Thread(new MyRunnable(sharedInstance)).start();
public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;
  public MyRunnable(NotThreadSafe instance){
        this.instance = instance;
  }
  public void run(){
        this.instance.add("some text");
  }
}
```

Notice how the two MyRunnable instances share the same NotThreadSafe instance. Therefore, when they call the add() method on the NotThreadSafe instance it leads to race condition.

However, if two threads call the add() method simultaneously on different instances then it does not lead to race condition. Here is the example from before, but slightly modified:

```
new Thread(new MyRunnable(new NotThreadSafe())).start();
new Thread(new MyRunnable(new NotThreadSafe())).start();
```

Now the two threads have each their own instance of NotThreadSafe so their calls to the add method doesn't interfere with each other. The code does not have race condition anymore. So, even if an object is not thread safe it can still be used in a way that doesn't lead to race condition.

9 Thread Safety and Immutability

Race conditions occur only if multiple threads are accessing the same resource, and one or more of the threads write to the resource. If multiple threads read the same resource race conditions do not occur.

We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe. Here is an example:

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

Notice how the value for the ImmutableValue instance is passed in the constructor. Notice also how there is no setter method. Once an ImmutableValue instance is created you cannot change its value. It is immutable. You can read it however, using the getValue() method.

If you need to perform operations on the ImmutableValue instance you can do so by returning a new instance with the value resulting from the operation. Here is an example of an add operation:

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

Notice how the add() method returns a new ImmutableValue instance with the result of the add operation, rather than adding the value to itself.

The Reference is not Thread Safe!

It is important to remember, that even if an object is immutable and thereby thread safe, the reference to this object may not be thread safe. Look at this example:

```
public class Calculator{
  private ImmutableValue currentValue = null;
  public ImmutableValue getValue(){
        return currentValue;
  }
  public void setValue(ImmutableValue newValue){
        this.currentValue = newValue;
  }
  public void add(int newValue){
        this.currentValue = this.currentValue.add(newValue);
  }
}
```

The Calculator class holds a reference to an ImmutableValue instance. Notice how it is possible to change that reference through both the setValue() and add() methods. Therefore, even if the Calculator class uses an immutable object internally, it is not itself immutable, and therefore not thread safe. In other words: The ImmutableValue class is thread safe, but the use of it is not. This is something to keep in mind when trying to achieve thread safety through immutability.

To make the Calculator class thread safe you could have declared the getValue(), setValue(), and add() methods synchronized. That would have done the trick.

10 Java Memory Model

11 Java Volatile Keyword

12 Thread Signaling

13 Getting and Setting Name of Thread

14 Thread Priorities

15 Methods to prevent thread execution

15.1 Yield()

15.2 join()

15.3 sleep()

16 Synchronization

Race conditions occur only if multiple threads are accessing the same resource, and one or more of the threads write to the resource. If multiple threads read the same resource race conditions do not occur.

We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe. Here is an example:

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

Notice how the value for the ImmutableValue instance is passed in the constructor. Notice also how there is no setter method. Once an ImmutableValue instance is created you cannot change its value. It is immutable. You can read it however, using the getValue() method.

If you need to perform operations on the ImmutableValue instance you can do so by returning a new instance with the value resulting from the operation. Here is an example of an add operation:

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

Notice how the add() method returns a new ImmutableValue instance with the result of the add operation, rather than adding the value to itself.

The Reference is not Thread Safe!

It is important to remember, that even if an object is immutable and thereby thread safe, the reference to this object may not be thread safe. Look at this example:

```
public class Calculator{
  private ImmutableValue currentValue = null;
  public ImmutableValue getValue(){
        return currentValue;
  }
  public void setValue(ImmutableValue newValue){
        this.currentValue = newValue;
  }
  public void add(int newValue){
        this.currentValue = this.currentValue.add(newValue);
  }
}
```

The Calculator class holds a reference to an ImmutableValue instance. Notice how it is possible to change that reference through both the setValue() and add() methods. Therefore, even if the Calculator class uses an immutable object internally, it is not itself immutable, and therefore not thread safe. In other words: The ImmutableValue class is thread safe, but the use of it is not. This is something to keep in mind when trying to achieve thread safety through immutability.

To make the Calculator class thread safe you could have declared the getValue(), setValue(), and add() methods synchronized. That would have done the trick.

17 Inter-thread communication

18 Deadlock

19 Deamon Thread

20 Starvation and Fairness

21 Nested Monitor Lockout

22 Slipped Conditions

23 Locks in Java

24 Read / Write Locks in Java

25 Reentrance Lockout

26 Semaphores

27 Blocking Queues

28 Thread Pools

29 Compare and Swap

30 Anatomy of a Synchronizer

31 Non-blocking Algorithms

32 Multithreading enhancements

32.1 Thread Local

The ThreadLocal class in Java enables you to create variables that can only be read and written by the **same thread**. Thus, 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**.

Creating a ThreadLocal

Here is a code example that shows how to create a ThreadLocal variable:

```
private ThreadLocal myThreadLocal = new ThreadLocal();
```

As you can see, you instantiate a new ThreadLocal object. This only needs to be done once per thread. Even if different threads execute the same code which accesses a ThreadLococal , each thread will see only its own ThreadLocal instance. Even if two different threads set different values on the same ThreadLocal object, they cannot see each other's values.

Accessing a ThreadLocal

Once a ThreadLocal has been created you can set the value to be stored in it like this:

```
myThreadLocal.set("A thread local value");
```

You read the value stored in a ThreadLocal like this:

```
String threadLocalValue = (String) myThreadLocal.get();
```

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

Generic ThreadLocal

You can create a generic ThreadLocal so that you do not have to typecast the value returned by get(). Here is a generic ThreadLocal example:

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
```

Now you can only store strings in the ThreadLocal instance. Additionally, you do not need to typecast the value obtained from the ThreadLocal:

```
myThreadLocal.set("Hello ThreadLocal");
String threadLocalValue = myThreadLocal.get();
```

Initial ThreadLocal Value

Since values set on a ThreadLocal object only are visible to the thread who set the value, no thread can set an initial value on a ThreadLocal using set() which is visible to all threads. Instead you can specify an initial value for a ThreadLocal object by subclassing ThreadLocal and overriding the initialValue() method. Here is how that looks:

```
private ThreadLocal myThreadLocal = new ThreadLocal<String>() {
  @Override protected String initialValue() {
    return "This is the initial value";
  }
```

```
        };
```

Now all threads will see the same initial value when calling get() before having called set() .

Full ThreadLocal Example

Here is a fully runnable Java ThreadLocal example:

```
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(2000);
         } 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();
      thread1.join(); //wait for thread 1 to terminate
      thread2.join(); //wait for thread 2 to terminate
   }
}
```

This example creates a single MyRunnable instance which is passed to two different threads. Both threads execute the run() method, and thus sets different values on the ThreadLocal instance. If the access to the set() call had been synchronized, and it had not been a ThreadLocal object, the second thread would have overridden the value set by the first thread.

However, since it is a ThreadLocal object then the two threads cannot see each other's values. Thus, they set and get different values.

InheritableThreadLocal

The InheritableThreadLocal class is a subclass of ThreadLocal. Instead of each thread having its own value inside a ThreadLocal, the InheritableThreadLocal grants access to values to a thread and all child threads created by that thread.