Java Concurrency Essentials

Contents

Preface

Concurrency is always a challenge for developers and writing concurrent programs can be extremely hard. There is a number of things that could potentially blow up and the complexity of systems rises considerably when concurrency is introduced.

However, the ability to write robust concurrent programs is a great tool in a developer's belt and can help build sophisticated, enterprise level applications.

In this course, you will dive into the magic of concurrency. You will be introduced to the fundamentals of concurrency and concurrent code and you will learn about concepts like atomicity, synchronization and thread safety.

As you advance, the following lessons will deal with the tools you can leverage, such as the Fork/Join framework, the java.util.concurrent JDK package. To sum those up, you will learn about testing concurrent applications.

About the Author

Martin is a software engineer with more than 10 years of experience in software development. He has been involved in different positions in application development in a variety of software projects ranging from reusable software components, mobile applications over fat-client GUI projects up to larg-scale, clustered enterprise applications with real-time requirements.

After finishing his studies of computer science with a diploma, Martin worked as a Java developer and consultant for international operating insurance companies. Later on he designed and implemented web applications and fat-client applications for companies on the energy market. Currently Martin works for an international operating company in the Java EE domain and is concerned in his day-to-day work with larg-scale big data systems.

His current interests include Java EE, web applications with focus on HTML5 and performance optimizations. When time permits, he works on open source projects, some of them can be found at this github account. Martin is blogging at Martin's Developer World.

Chapter 1

Introduction to Threads and Concurrency

1.1 Basic know-how about threads

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.

To achieve a better understanding of parallel execution, we have to distinguish between processes and threads. 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. 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.

In Java, processes correspond to a running Java Virtual Machine (JVM), whereas threads live within the same JVM and can be created and stopped by the Java application dynamically during runtime. 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.

This is demonstrated in the following source code. Access to the current `Thread` is provided by the static method `currentThread()` of the JDK class `java.lang.Thread`:

```
public class MainThread {

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

As you can see from the source code of this simple application, we access the current `Thread` directly in the `main()` method and print out some of the information provided about it:

```
id=1; name=main; priority=5; state=RUNNABLE; threadGroupName=main
```

The output reveals some interesting information about each thread. Each thread has an identifier, which is unique in the JVM. The name of the threads helps to find certain threads within external applications that monitor a running JVM (e.g. a debugger or the JConsole tool). When more than one threads are executed, the priority decides which task should be executed next.

The truth about threads is that not all threads are really executed simultaneously, but the execution time on each CPU core is divided into small slices and the next time slice is assigned to the next waiting thread with the highest priority. The scheduler of the JVM decides based on the thread's priorities which thread to execute next.

Next to the priority, a thread also has a state, which can be one of the following:

- **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.

Our main thread from the example above is of course in the state RUNNABLE. State names like BLOCKED indicate here already that thread management is an advanced topic. If not handled properly, threads can block each other which in turn cause the application to hang. But we will come to this later on.

Last but not least the attribute `threadGroup` of our thread indicates that threads are managed in groups. Each thread belongs to a group of threads. The JDK class `java.lang.ThreadGroup` provides some methods to handle a whole group of `Threads`. With these methods we can for example interrupt all threads of a group or set their maximum priority.

1.2 Creating and starting threads

Now that we have taken a closer look at the properties of a thread, it is time to create and start our first thread. Basically there are two ways to create a thread in Java. The first one is to write a class that extends the JDK class `java.lang.Thread`:

```
public class MyThread extends Thread {

        public MyThread(String name) {
                super(name);
        }

        @Override
        public void run() {
                System.out.println("Executing thread "+Thread.currentThread(). ↩
                    getName());
        }

        public static void main(String[] args) throws InterruptedException {
                MyThread myThread = new MyThread("myThread");
                myThread.start();
        }
}
```

As can be seen above, the class `MyThread` extends the `Thread` class and overrides the method `run()`. The method `run()` gets executed once the virtual machine starts our Thread. As the virtual machine has to do some work in order to setup the execution environment for the thread, we cannot call this method directly to start the thread. Instead we call the method `start()` on an instance of the class `MyThread`. As this class inherits the method `stop()` from its superclass, the code behind this method tells the JVM to allocate all necessary resources for the thread and to start it. When we run the code above, we see the output "Executing thread myThread". In contrast to our introduction example, the code within the method `run()` is not executed within the "main" thread but rather in our own thread called "myThread".

The second way to create a thread is a class that implements the interface `Runnable`:

```
public class MyRunnable implements Runnable {

        public void run() {
                System.out.println("Executing thread "+Thread.currentThread(). ↩
                    getName());
        }

        public static void main(String[] args) throws InterruptedException {
                Thread myThread = new Thread(new MyRunnable(), "myRunnable");
                myThread.start();
        }
}
```

The main difference to the subclassing approach is the fact that we create an instance of `java.lang.Thread` and provide an instance of the class that implements the `Runnable` interface as an argument to the `Thread` constructor. Next to this instance we also pass the name of the `Thread`, so that we see the following output when we execute the program from command line: "Executing thread myRunnable".

Whether you should use the subclassing or the interface approach, depends to some extend on your taste. The interface is a more light-weight approach as all you have to do is the implementation of an interface. The class can still be a subclass of some other class. You can also pass your own parameters to the constructor whereas subclassing `Thread` restricts you to the available constructors that the class `Thread` brings along.

Later on in this series we will get to know about thread pools and see how to start multiple threads of the same type. Here we will again use the `Runnable` approach.

1.3 Sleeping and interrupting

Once we have started a `Thread`, it runs until the `run()` method reaches it end. In the examples above the `run()` method did nothing more than just printing out the name of the current thread. Hence the thread finished very soon.

In real world applications you will normally have to implement some kind of background processing where you let the thread run until it has for example processed all files within a directory structure, for example. Another common use case is to have a background thread that looks every n seconds if something has happened (e.g. a file has been created) and starts some kind of action. In this case you will have to wait for n seconds or milliseconds. You could implement this by using a while loop whose body gets the current milliseconds and looks when the next second has passed. Although such an implementation would work, it is a waste of CPU processing time as your thread occupies the CPU and retrieves the current time again and again.

A better approach for such use cases is calling the method `sleep()` of the class `java.lang.Thread` like in the following example:

```
public void run() {
        while(true) {
                doSomethingUseful();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

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 second (given in milliseconds) has passed.

Please note that the time passed to the `sleep()` method is only an indication for the scheduler and not an absolutely exact time frame. It may happen that the thread comes back a few nano or milliseconds earlier or later due to the scheduling that is put in

practice. Hence you should not use this method for real-time scheduling purposes. But for most use cases the achieved accuracy is sufficient.

In the code example above you may have noticed the `InterruptedException` that `sleep()` may throw. Interrupts are a very basic feature for thread interaction that can be understood as a simple interrupt message that one thread sends to another thread. The receiving thread can explicitly ask if it has been interrupted by calling the method `Thread.interrupted()` or it is implicitly interrupted while spending his time within a method like `sleep()` that throws an exception in case of an interrupt.

Let's take a closer look at interrupts with the following code example:

```
public class InterruptExample implements Runnable {

        public void run() {
                try {
                        Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                        System.out.println("["+Thread.currentThread().getName()+"]  ↩
                            Interrupted by exception!");
                }
                while(!Thread.interrupted()) {
                        // do nothing here
                }
                System.out.println("["+Thread.currentThread().getName()+"]  ↩
                    Interrupted for the second time.");
        }

        public static void main(String[] args) throws InterruptedException {
                Thread myThread = new Thread(new InterruptExample(), "myThread");
                myThread.start();

                System.out.println("["+Thread.currentThread().getName()+"] Sleeping ↩
                     in main thread for 5s...");
                Thread.sleep(5000);

                System.out.println("["+Thread.currentThread().getName()+"]  ↩
                    Interrupting myThread");
                myThread.interrupt();

                System.out.println("["+Thread.currentThread().getName()+"] Sleeping ↩
                     in main thread for 5s...");
                Thread.sleep(5000);

                System.out.println("["+Thread.currentThread().getName()+"]  ↩
                    Interrupting myThread");
                myThread.interrupt();
        }
}
```

Within the main method we start a new thread first, which would sleep for a very long time (about 290.000 years) if it would not be interrupted. To get the program finished before this time has passed by, `myThread` is interrupted by calling `interrupt()` on its instance variable in the main method. This causes an `InterruptedException` within the call of `sleep()` and is printed on the console as "Interrupted by exception!". Having logged the exception the thread does some busy waiting until the interrupted flag on the thread is set. This again is set from the main thread by calling `interrupt()` on the thread's instance variable. Overall we see the following output on the console:

```
[main] Sleeping in main thread for 5s...
[main] Interrupting myThread
[main] Sleeping in main thread for 5s...
[myThread] Interrupted by exception!
[main] Interrupting myThread
[myThread] Interrupted for the second time.
```

What is interesting in this output, are the lines 3 and 4. If we go through the code we might have expected that the string "Interrupted by exception!" is printed out before the main thread starts sleeping again with "Sleeping in main thread for 5s...". But as you can see from the output, the scheduler has executed the main thread before it started myThread again. Hence myThread prints out the reception of the exception after the main thread has started sleeping.

It is a basic observation when programming with multiple threads that logging output of threads is to some extend hard to predict as it's hard to calculate which thread gets executed next. Things get even worse when you have to cope with more threads whose pauses are not hard coded as in the examples above. In these cases the whole program gets some kind of inner dynamic that makes concurrent programming a challenging task.

1.4 Joining Threads

As we have seen in the last section we can let our thread sleep until it is woken up by another thread. Another important feature of threads that you will have to use from time to time is the ability of a thread to wait for the termination of another thread.

Let's assume you have to implement some kind of number crunching operation that can be divided into several parallel running threads. The main thread that starts the so called worker threads has to wait until all its child threads have terminated. The following code shows how this can be achieved:

```
public class JoinExample 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 JoinExample(), "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.");
        }
}
```

Within our main method we create an array of five `Threads`, which are all started one after the other. Once we have started them, we wait in the main `Thread` for their termination. The threads itself simulate some number crunching by computing one random number after the other. Once they are finished, they print out "finished". Finally the main thread acknowledges the termination of all of its child threads:

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

You will observe that the sequence of "finished" messages varies from execution to execution. If you execute the program more than once, you may see that the thread which finishes first is not always the same. But the last statement is always the main thread that waits for its children.

1.5 Synchronization

As we have seen in the last examples, 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. And often access to shared resources is exclusive, which means only one thread at a given point in time should access this resource without any other thread interfering this access.

A simple example for concurrent access of an exclusive resource would be a static variable that is incremented by more than one thread:

```
public class NotSynchronizedCounter 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[5];
                for(int i=0; i<threads.length; i++) {
                        threads[i] = new Thread(new NotSynchronizedCounter(), " ↩
                            thread-"+i);
                        threads[i].start();
                }
                for(int i=0; i<threads.length; i++) {
                        threads[i].join();
                }
        }
}
```

When we take a closer look at the output of this simple application, we see something like:

```
[thread-2] before: 8
[thread-2] after: 9
[thread-1] before: 0
[thread-1] after: 10
[thread-2] before: 9
[thread-2] after: 11
```

Here, thread-2 retrieves the current value as 8, increments it and afterwards the value is 9. This is how we would have expected it before. But what the following thread executes may wonder us. thread-1 outputs the current value as zero, increments it and afterwards the value is 10. How can this happen? When thread-1 read the value of the variable counter, it was 0. Then the context switch executed the second thread and when thread-1 had his turn again, the other threads already incremented the counter up to 9. Now he adds one and gets 10 as a result.

The solution for problems like this is the synchronized key word in Java. With synchronized you can create blocks of statements which can only be accessed by a thread, which gets the lock on the synchronized resource. Let's change the `run()` method from the last example and introduce a synchronized block for the whole class:

```
public void run() {
        while (counter < 10) {
                synchronized (SynchronizedCounter.class) {
                        System.out.println("[" + Thread.currentThread().getName() + ↩
                            "] before: " + counter);
```

```
                                counter++;
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                    "] after: " + counter);
                        }
                }
        }
```

The `synchronized(SynchronizedCounter.class)` statement works like a barrier where all threads have to stop and ask for entrance. Only the first thread that gets the lock on the resources is allowed to pass. Once it has left the synchronized block, another waiting thread can enter and so forth.

With the synchronized block around the output and increment of the counter above the output looks like the following example:

```
[thread-1] before: 11
[thread-1] after: 12
[thread-4] before: 12
[thread-4] after: 13
```

Now you will see only subsequent outputs of before and after that increment the counter variable by one. The synchronized keyword can be used in two different ways. It can either be used within a method as shown above. In this case you have to provide a resource that is locked by the current thread. This resource has to be chosen carefully because the thread barrier becomes a completely different meaning based on the scope of the variable.

If the variable is a member of the current class then all threads are synchronized regarding an instance of the class because the variable sync exists per instance of `LocalSync`:

```
        public class LocalSync {
                private int sync = 0;

                public void someMethod() {
                        synchronized (sync) {
                                // synchronized on instance level
                        }
                }
        }
```

Instead of creating a block that covers the whole body of the method, you can also add the keyword synchronized to the method signature. The code below has the same effect as the code above:

```
        public class MethodSync {
                private int sync = 0;

                public synchronized void someMethod() {
                        // synchronized on instance level
                }
        }
```

The main difference between the two approaches is the fact that the first one is finer grained, as you can make the synchronized block smaller than the method body. Please remember that synchronized blocks are executed only by one thread at a time, hence each synchronized block is a potential performance problem, as all concurrently running threads may have to wait until the current thread leaves the block. Hence we should always try to make the block as small as possible.

Most often you will have to synchronize access to some resource that only exists once per JVM. The common means to do that is to use static member variable of a class:

```
        public class StaticSync {
                private static int sync = 0;

                public void someMethod() {
                        synchronized (sync) {
                                // synchronized on ClassLoader/JVM level
                        }
```

```
                }
        }
```

The code above synchronizes all threads that run through the method `someMethod()` within the same JVM as the static variable only exists once within the same JVM. As you may know, a class is only unique within one JVM if it is loaded by the same class loader. If you load the class `StaticSync` with multiple class loaders then the static variable exists more than once. But in most day to day applications you won't have multiple class loaders that load the same class twice, hence you can assume that the static variable exists only once and therefore all threads within the same JVM have to wait at the barrier until they get the lock.

1.6 Atomic Access

In the previous section we saw how to synchronize access to some complex resource when many concurrent threads have to execute a certain part of code but only one thread should execute it at each point in time. We have also seen that if we do not synchronize access to common resources, operations on these resources interleave and may cause an illegal state.

The Java language provides some basic operations that are atomic and that therefore can be used to make sure that concurrent threads always see the same value:

- Read and write operations to reference variables and primitive variables (except long and double)
- Read and write operations for all variables declared as volatile

To understand this in more detail, let's assume we have a `HashMap` filled with properties that are read from a file and a bunch of threads that work with these properties. It is clear that we need some kind of synchronization here, as the process of reading the file and update the `Map` costs time and that during this time other threads are executed.

We cannot easily share one instance of this `Map` between all threads and work on this `Map` during the update process. This would lead to an inconsistent state of the `Map`, which is read by the accessing threads. With the knowledge from the last section we could of course use a synchronized block around each access (read/write) of the map to ensure that the all threads only see one state and not a partially updated `Map`. But this leads to performance problems if the concurrent threads have to read very often from the `Map`.

Cloning the `Map` for each thread within a synchronized block and letting each thread work on a separate copy would be a solution, too. But each thread would have to ask from time to time for an updated copy and the copy occupies memory, which might not be feasible in each case. But there is a more simple solution.

Since we know that write operations to a reference are atomic, we can create a new `Map` each time we read the file and update the reference that is shared between the threads in one atomic operation. In this implementation the worker threads will never read an inconsistent `Map` as the `Map` is updated with one atomic operation:

```
public class AtomicAssignment implements Runnable {
        private static final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm: ↩
            ss:SSS");
        private static Map<String, String> configuration = new HashMap<String, String>();

        public void run() {
                for (int i = 0; i < 10000; i++) {
                        Map<String, String> currConfig = configuration;
                        String value1 = currConfig.get("key-1");
                        String value2 = currConfig.get("key-2");
                        String value3 = currConfig.get("key-3");
                        if (!(value1.equals(value2) && value2.equals(value3))) {
                                throw new IllegalStateException("Values are not equal.");
                        }
                        try {
                                Thread.sleep(10);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
```

```
                }
        }

        public static void readConfig() {
                Map<String, String> newConfig = new HashMap<String, String>();
                Date now = new Date();
                newConfig.put("key-1", sdf.format(now));
                newConfig.put("key-2", sdf.format(now));
                newConfig.put("key-3", sdf.format(now));
                configuration = newConfig;
        }

        public static void main(String[] args) throws InterruptedException {
                readConfig();
                Thread configThread = new Thread(new Runnable() {
                        public void run() {
                                for (int i = 0; i < 10000; i++) {
                                        readConfig();
                                        try {
                                                Thread.sleep(10);
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                }, "configuration-thread");
                configThread.start();
                Thread[] threads = new Thread[5];
                for (int i = 0; i < threads.length; i++) {
                        threads[i] = new Thread(new AtomicAssignment(), "thread-" + i);
                        threads[i].start();
                }
                for (int i = 0; i < threads.length; i++) {
                        threads[i].join();
                }
                configThread.join();
                System.out.println("[" + Thread.currentThread().getName() + "] All threads ↩
                    have finished.");
        }
}
```

The above example is a little more complex, but not hard to understand. The `Map`, which is shared, is the configuration variable of `AtomicAssignment`. In the `main()` method we read the configuration initially one time and add three keys to the `Map` with the same value (here the current time including milliseconds). Then we start a "configuration-thread" that simulates the reading of the configuration by adding all the time the current timestamp three times to the map. The five worker threads then read the `Map` using the configuration variable and compare the three values. If they are not equal, they throw an IllegalStateException.

You can run the program for some time and you will not see any `IllegalStateException`. This is due the fact that we assign the new `Map` to the shared configuration variable in one atomic operation:

```
        configuration = newConfig;
```

We also read the value of the shared variable within one atomic step:

```
        Map<String, String> currConfig = configuration;
```

As both steps are atomic, we will always get a reference to a valid `Map` instance where all three values are equal. If you change for example the `run()` method in a way that it uses the configuration variable directly instead of copying it first to a local variable, you will see `IllegalStateExceptions` very soon because the configuration variable always points to the "current" configuration. When it has been changed by the configuration-thread, subsequent read accesses to the `Map` will already read the new values and compare them with the values from the old map.

The same is true if you work in the `readConfig()` method directly on the configuration variable instead of creating a new `Map` and assigning it in one atomic operation to the shared variable. But it may take some time, until you see the first `IllegalStateException`. And this is true for all applications that use multi-threading. Concurrency problems are not always visible at first glance, but they need some testing under heavy-load conditions in order to appear.

Chapter 2

Concurrency Fundamentals: Deadlocks and Object Monitors

2.1 Liveness

When developing an application that uses concurrency to accomplish its aims, you may come across situations where different threads may block each other. As the whole application then performs slower than expected, we would say that the application does not finish in matter of time as expected. In this section we will take a closer look at the problems that may endanger the liveness of a multi-threaded application.

2.1.1 A Deadlock

The term deadlock is well known to software developers and even most normal computer users use this term from time to time, although it is not always used in the correct sense. Strictly spoken it means that two (or more) threads are each waiting on the other thread to free a resource that it has locked, while the thread itself has locked a resource the other thread is waiting on:

```
Thread 1: locks resource A, waits for resource B

Thread 2: locks resource B, waits for resource A
```

To gain an even better understanding of the problem, let's have a look at the following source code:

```
public class Deadlock implements Runnable {
        private static final Object resource1 = new Object();
        private static final Object resource2 = new Object();
        private final Random random = new Random(System.currentTimeMillis());

        public static void main(String[] args) {
                Thread myThread1 = new Thread(new Deadlock(), "thread-1");
                Thread myThread2 = new Thread(new Deadlock(), "thread-2");
                myThread1.start();
                myThread2.start();
        }

        public void run() {
                for (int i = 0; i < 10000; i++) {
                        boolean b = random.nextBoolean();
                        if (b) {
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                    "] Trying to lock resource 1.");
                                synchronized (resource1) {
```

```
                                        System.out.println("[" + Thread.currentThread(). ↩
                                            getName() + "] Locked resource 1.");
                                        System.out.println("[" + Thread.currentThread(). ↩
                                            getName() + "] Trying to lock resource 2.");
                                        synchronized (resource2) {
                                                System.out.println("[" + Thread. ↩
                                                    currentThread().getName() + "] Locked  ↩
                                                    resource 2.");
                                        }
                                }
                        } else {
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                     "] Trying to lock resource 2.");
                                synchronized (resource2) {
                                        System.out.println("[" + Thread.currentThread(). ↩
                                            getName() + "] Locked resource 2.");
                                        System.out.println("[" + Thread.currentThread(). ↩
                                            getName() + "] Trying to lock resource 1.");
                                        synchronized (resource1) {
                                                System.out.println("[" + Thread. ↩
                                                    currentThread().getName() + "] Locked  ↩
                                                    resource 1.");
                                        }
                                }
                        }
                }
        }
}
```

As can be seen from the code above, two threads are started and try to lock the two static resources. But for a deadlock we need a different sequence for both threads, hence we utilize the Random instance to choose what resource the thread wants to lock first. If the boolean variable b is true, the resource1 is locked first and the threads then tries to get the lock for resource 2. If b is false, the thread locks resource2 first and then tries to lock resource1. This program does not have to run long until we reach the first deadlock, i.e. the program hangs forever if we would not terminate it:

```
[thread-1] Trying to lock resource 1.

[thread-1] Locked resource 1.

[thread-1] Trying to lock resource 2.

[thread-1] Locked resource 2.

[thread-2] Trying to lock resource 1.

[thread-2] Locked resource 1.

[thread-1] Trying to lock resource 2.

[thread-1] Locked resource 2.

[thread-2] Trying to lock resource 2.

[thread-1] Trying to lock resource 1.
```

In this execution thread-1 holds the lock for resource2 and waits for the lock on resource1, whereas thread-2 holds the lock for resource1 and waits for resource2.

If we would set the boolean variable b in the example code above to true, we would not experience any deadlock because the sequence in which thread-1 and thread-2 are requesting the locks is always the same. Hence one of both threads gets the lock first and then requests the second lock, which is still available because the other threads wait for the first lock.

In general the following requirements for a deadlock can be identified:

- **Mutual exclusion:** There is a resource which can be accessed only by one thread at any point in time.
- **Resource holding:** While having locked one resource, the thread tries to acquire another lock on some other exclusive resource.
- **No preemption:** There is no mechanism, which frees the resource if one threads holds the lock for a specific period of time.
- **Circular wait:** During runtime a constellation occurs in which two (or more) threads are each waiting on the other thread to free a resource that it has locked.

Although the list of requirements looks long, it is not uncommon that more advanced multi-threaded applications have deadlock problems. But you can try to avoid deadlocks if you are able to relax one of the requirements listed above:

- Mutual exclusion: This is a requirement that often cannot be relaxed, as the resource has to be used exclusively. But this must no always be the case. When using DBMS systems, a possible solution instead of using a pessimistic lock on some table row that has to be updated, one can use a technique called Optimistic Locking.
- A possible solution to circumvent resource holding while waiting for another exclusive resource is to lock all necessary resources at the beginning of the algorithm and free all resources if it is not possible to obtain all locks. This is of course not always possible, maybe the resources to lock are not known ahead or it is just as waste of resources.
- If the lock cannot be obtained immediately, a possible solution to circumvent a possible deadlock is the introduction of a timeout. The SDK class ReentrantLock for example provides the possibility to specify a timeout for locking.
- As we have seen from the example code above, a deadlock does not appear if the sequence of lock requests does not differ between the different threads. This can be easily controlled if you are able to put all the locking code into one method where all threads have to pass through.

In more advanced applications you may even consider the implementation of a deadlock detection system. Here you would have to implement some kind of thread monitoring, where each thread reports the successful acquisition of a lock and his attempt to obtain a lock. If threads and locks are modelled as a directed graph, you are able to detect when two different threads are holding resources while simultaneously requesting another blocked resource. If you would then able to force the blocking threads to release an obtained resource you are able to resolve deadlock situations automatically.

2.1.2 Starvation

The scheduler decides which of the academy.javacodegeeks.com/lesson/introduction-to-threads-and-concurrency/[threads in state RUNNABLE] it should execute next. The decision is based on the thread's priority; hence threads with lower priority gain less CPU time than threads with a higher priority. What sounds like a reasonable feature can also cause problems when abused. If most of the time threads with a high priority are executed, the threads with lower priority seem to "starve" as they get not enough time to execute their work properly. Therefore it is recommended to set the priority of a thread only if there are strong reasons for it.

A sophisticated example for thread starvation is for example the finalize() method. This feature of the Java language can be used to execute code before an object gets garbage collected. But when you take a look at the priority of the finalizer thread, you may see that it runs not with highest priority. Hence there is the possibility for thread starvation if the finalize() methods of your object spend too much time in comparison to the rest of the code.

Another problem with execution time is the problem, that it is not defined in which order threads pass a synchronized block. When many parallel threads have to pass some code that is encapsulated with a synchronized block, it may happen that certain threads have to wait longer than other threads until they can enter the block. In theory they may never enter the block.

A solution to the latter problem is the so called "fair" lock. Fair locks take the waiting time of the threads into account, when choosing the next thread to pass. An example implementation of a fair lock is provided by the Java SDK: java.util.concurrent.locks.Reentr
If the constructor with the boolean flag set to true is used, the ReentrantLock grants access to the longest-waiting thread. This guarantees a lack of starvation but introduces at the same time the problem that the thread priority is not taken into account and therefore threads with lower priority that often wait at this barrier may be executed more often. Last but not least the ReentrantLock class can of course only consider threads that are waiting for the lock, i.e. threads that have been executed often enough to reach the lock. If thread priority is too low this may not often happen and therefore threads with higher priority still pass the lock more often.

2.2 Object monitors with wait() and notify()

A common task in multi-threaded computing is to have some worker threads that are waiting for their producer to create some work for them. But as we have learned, busy waiting within a loop and checking some value is not a good option in terms of CPU time. In this use case the Thread.sleep() method is also not of much value, as we want to start our work immediately after it has been submitted.

The Java Programming Language therefore has another construct, that can be used in this scenario: wait() and notify(). The wait() method that every object inherits from the java.lang.Object class can be used to pause the current thread execution and wait until another threads wakes us up using the notify() method. In order to work correctly, the thread that calls the wait() method has to hold a lock that it has acquired before using the synchronized keyword. When calling wait() the lock is released and the threads waits until another thread that now owns the lock calls notify() on the same object instance.

In a multi-threaded application there may of course be more than one thread waiting on some object to be notified. Hence there are two different methods to wake up threads: notify() and notifyAll(). Whereas the first method only wakes up one of the waiting threads, the notifyAll() methods wakes them all up. But be aware that similar to the synchronized keyword there is no rule that specifies which thread to wake up next when calling notify(). In a simple producer and consumer example this does not matter, as we are not interested in the fact which thread exactly wakes up.

The following code demonstrates how the wait() and notify() mechanism can be used to let the consumer threads wait for new work that is pushed into a queue from some producer thread:

```
package a2;

import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConsumerProducer {
        private static final Queue queue = new ConcurrentLinkedQueue();
        private static final long startMillis = System.currentTimeMillis();

        public static class Consumer implements Runnable {

                public void run() {
                        while (System.currentTimeMillis() < (startMillis + 10000)) {
                                synchronized (queue) {
                                        try {
                                                queue.wait();
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                                if (!queue.isEmpty()) {
                                        Integer integer = queue.poll();
                                        System.out.println("[" + Thread.currentThread(). ↩
                                            getName() + "]: " + integer);
                                }
                        }
                }
        }

        public static class Producer implements Runnable {

                public void run() {
                        int i = 0;
                        while (System.currentTimeMillis() < (startMillis + 10000)) {
                                queue.add(i++);
                                synchronized (queue) {
                                        queue.notify();
                                }
                                try {
                                        Thread.sleep(100);
```

```
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                        synchronized (queue) {
                                queue.notifyAll();
                        }
                }

        }

        public static void main(String[] args) throws InterruptedException {
                Thread[] consumerThreads = new Thread[5];
                for (int i = 0; i < consumerThreads.length; i++) {
                        consumerThreads[i] = new Thread(new Consumer(), "consumer-" + i);
                        consumerThreads[i].start();
                }
                Thread producerThread = new Thread(new Producer(), "producer");
                producerThread.start();
                for (int i = 0; i < consumerThreads.length; i++) {
                        consumerThreads[i].join();
                }
                producerThread.join();
        }
}
```

The main() method starts five consumer and one producer thread and then waits for them to finish. The producer thread then inserts a new value into the queue and afterwards notifies all waiting threads that something has happened. The consumer threads acquire the queue lock and then fall asleep in order to be woken up later on when the queue is filled again. When the producer thread has finished its work, it notifies all consumer threads to wake up. If we won't do the last step, the consumer threads would wait forever for the next notification, as we haven't specified any timeout for the waiting. Instead we could have used the method wait(long timeout) to be woken up at least after some amount of time has passed by.

2.2.1 Nested synchronized blocks with wait() and notify()

As mentioned in the last section, calling wait() on an object monitor only frees the lock on this object monitor. Other locks which are being hold by the same thread are not freed. As this is easy to understand, it may happen in day-to-day work that the thread that calls wait() holds further locks. And if other threads are also waiting for these locks a deadlock situation can happen. Let's have a look at the following example code:

```
public class SynchronizedAndWait {
        private static final Queue queue = new ConcurrentLinkedQueue();

        public synchronized Integer getNextInt() {
                Integer retVal = null;
                while (retVal == null) {
                        synchronized (queue) {
                                try {
                                        queue.wait();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                                retVal = queue.poll();
                        }
                }
                return retVal;
        }

        public synchronized void putInt(Integer value) {
                synchronized (queue) {
```

```
                        queue.add(value);
                        queue.notify();
                }
        }

        public static void main(String[] args) throws InterruptedException {
                final SynchronizedAndWait queue = new SynchronizedAndWait();
                Thread thread1 = new Thread(new Runnable() {
                        public void run() {
                                for (int i = 0; i < 10; i++) {
                                        queue.putInt(i);
                                }
                        }
                });
                Thread thread2 = new Thread(new Runnable() {
                        public void run() {
                                for (int i = 0; i < 10; i++) {
                                        Integer nextInt = queue.getNextInt();
                                        System.out.println("Next int: " + nextInt);
                                }
                        }
                });
                thread1.start();
                thread2.start();
                thread1.join();
                thread2.join();
        }
}
```

As we have learned before, adding synchronized to the method signature is equal to creating a synchronized(this){} block. In the example above we have accidentally added the synchronized keyword to the method and afterwards synchronized on the object monitor queue in order to put the current thread into sleep while waiting for the next value from the queue. The current thread then releases the lock hold on queue but not the lock hold on this. The putInt() method notifies the sleeping thread that a new value has been added. But accidentally we have also added the keyword synchronized to this method. When now the second thread has fallen asleep, it still holds the lock. The first thread can then not enter the method putInt() as the this lock is hold by the first thread. Hence we have a deadlock situation and the program hangs. If you execute the code above, this happens right after the beginning of the program.

In everyday life the situation may not be as clear as above. The locks a thread holds may depend on runtime parameter and conditions and the synchronized block causing the problem may not be that near to the place in the code, where we put the wait() call. This makes it difficult to find such problems and if may be that these problems only appear after some time or under heavy load.

2.2.2 Conditions in synchronized blocks

Often you will have to check that some condition is fulfilled, before you execute some action on a synchronized object. When you have for example a queue you want to wait until this queue is filled. Hence you can write a method that checks if the queue is filled. If not you put the current thread asleep until it is woken up:

```
public Integer getNextInt() {
        Integer retVal = null;
        synchronized (queue) {
                try {
                        while (queue.isEmpty()) {
                                queue.wait();
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
```

```
        synchronized (queue) {
                retVal = queue.poll();
                if (retVal == null) {
                        System.err.println("retVal is null");
                        throw new IllegalStateException();
                }
        }
        return retVal;
}
```

The code above synchronizes on the queue before calling wait() and then waits within the while loop until the queue has at least one entry. The second synchronized block again uses the queue as object monitor. It polls() the queue for the value inside. For demonstration purposes an IllegalStateException is thrown when poll() returns null. This is the case when there are no values inside the queue to poll.

When you run this example, you will see that the IllegalStateException is thrown very soon. Although we have correctly synchronized on the queue monitor, the exception is thrown. The reason here is that we have two separate synchronized blocks. Imagine we have two threads that have arrived at the first synchronized block. The first thread enters the block and falls asleep because the queue is empty. The same is true for the second thread. Now when both threads wake up (by another thread calling notifyAll() on the monitor), they both see a value in the queue (the one added by the producer. Then both threads arrive at the second barrier. Here the first thread enters and polls the value from the queue. When the second thread enters, the queue is already empty. Hence it gets null as return value from the poll() call and throws the exception.

To avoid situations like the one above, you will have to execute all operations that depend on the monitor's state in the same synchronized block:

```
public Integer getNextInt() {
        Integer retVal = null;
        synchronized (queue) {
                try {
                        while (queue.isEmpty()) {
                                queue.wait();
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                retVal = queue.poll();
        }
        return retVal;
}
```

Here we execute the method poll() in the same synchronized block as the isEmpty() method. Through the synchronized block we are sure that only one thread is executing methods on this monitor at a given point in time. Hence no other thread can remove elements from the queue between the isEmpty() and the poll() call.

2.3 Designing for multi-threading

As we have seen in the last sections, implementing a multi-threading application is sometimes more complex than it might look at first glance. Therefore it is important to have a clear design in mind when starting the project.

2.3.1 Immutable object

One design rule that is considered to be very important in this context is Immutability. If you share object instances between different threads you have to pay attention that two threads do not modify the same object simultaneously. But objects that are not modifiable are easy to handle in such situations as you cannot change them. You always have to construct a new instance when you want to modify the data. The basic class java.lang.String is an example of an immutable class. Every time you want to change a string, you get a new instance:

```
String str = "abc";

String substr = str.substring(1);
```

Although object creation is an operation that does not come without costs, these costs are often overestimated. But you always have to weight out if a simple design with immutable objects outweighs to not use immutable objects with the risk of concurrency errors that are observed maybe very late in the project.

In the following you will find a set of rules to apply when you want to make a class immutable:

- All fields should be final and private.
- There should be not setter methods.
- The class itself should be declared final in order to prevent subclasses to violate the principle of immutability.
- If fields are not of a primitive type but a reference to another object:
  - There should not be a getter method that exposes the reference directly to the caller.
  - Don't change the referenced objects (or at least changing these references is not visible to clients of the object).

Instances of the following class represent a message with subject, message body and a few key/value pairs:

```
public final class ImmutableMessage {
        private final String subject;
        private final String message;
        private final Map<String,String> header;

        public ImmutableMessage(Map<String,String> header, String subject, String message)  ↩
            {
                this.header = new HashMap<String,String>(header);
                this.subject = subject;
                this.message = message;
        }

        public String getSubject() {
                return subject;
        }

        public String getMessage() {
                return message;
        }

        public String getHeader(String key) {
                return this.header.get(key);
        }

        public Map<String,String> getHeaders() {
                return Collections.unmodifiableMap(this.header);
        }
}
```

The class is immutable as all of its fields are final and private. There are no methods that would be able to modify the state of an instance after its construction. Returning the references to subject and message is safe, as String itself is an immutable class. The caller who obtains a reference for example to the message cannot modify it directly. With the Map of headers we have to pay more attention. Just returning the reference to the Map would allow the caller to change its content. Hence we have to return an unmodifiable Map acquired through a call of Collections.unmodifiableMap(). This returns a view on the Map that allows callers to read the values (which are strings again), but which does not allow modifications. An UnsupportedOperationException will be thrown when trying to modify the Map instance. In this example it is also safe to return the value for a specific key, like it is done within getHeader(String key), as the returned String is immutable again. If the Map would contain objects that are not immutable themselves, this operation would not be thread-safe.

2.3.2 API design

When designing the public methods of a class, i.e. the API of this class, you can also try to design it for multi-threaded usage. You might have methods that should not be executed when the object is within a certain state. One simple solution to overcome this situation would be to have a private flag, which indicate in which state we are and throws for example an IllegalStateException when a specific method should not be called:

```
public class Job {
        private boolean running = false;
        private final String filename;

        public Job(String filename) {
                this.filename = filename;
        }

        public synchronized void start() {
                if(running) {
                        throw new IllegalStateException("...");
                }
                ...
        }

        public synchronized List getResults() {
                if(!running) {
                        throw new IllegalStateException("...");
                }
                ...
        }
}
```

The above pattern is also often referred to as "balking pattern", as the method balks once it gets executed in the wrong state. But you could have designed the same functionality using a static factory method without checking in each method the state of the object:

```
public class Job {
        private final String filename;

        private Job(String filename) {
                this.filename = filename;
        }

        public static Job createAndStart(String filename) {
                Job job = new Job(filename);
                job.start();
                return job;
        }

        private void start() {
                ...
        }

        public synchronized List getResults() {
                ...
        }
}
```

The static factory method creates a new instance of Job using the private constructor and already calls start() on the instance. The returned reference of Job is already in the correct state to work with, hence the getResults() method only needs to be synchronized but does not have to check for the state of the object.

2.3.3 Thread local storage

We have seen so far that threads share the same memory. In terms of performance this is a good way to share data between the threads. If we would have used separate processes in order to execute code in parallel, we would have more heavy data exchange methods, like remote procedure calls or synchronization on file system or network level. But sharing memory between different threads is also difficult to handle if not synchronized properly.

Dedicated memory that is only used by our own thread and not by other threads is provided in Java through the java.lang.ThreadLocal class:

```
private static final ThreadLocal myThreadLocalInteger = new ThreadLocal();
```

The type of data that should be stored within the ThreadLocal is given by the generic template parameter T. In the example above we used just Integer, but we could have used any other data type here as well. The following code demonstrates the usage of ThreadLocal:

```
public class ThreadLocalExample implements Runnable {
        private static final ThreadLocal threadLocal = new ThreadLocal();
        private final int value;

        public ThreadLocalExample(int value) {
                this.value = value;
        }

        @Override
        public void run() {
                threadLocal.set(value);
                Integer integer = threadLocal.get();
                System.out.println("[" + Thread.currentThread().getName() + "]: " + integer ↩
                    );
        }

        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 ThreadLocalExample(i), "thread-" + i);
                        threads[i].start();
                }
                for (int i = 0; i < threads.length; i++) {
                        threads[i].join();
                }
        }
}
```

You may wonder that each thread outputs exactly the value that it gets through the constructor although the variable threadLocal is declared static. ThreadLocal's internal implementation makes sure that every time you call set() the given value is stored within a memory region that only the current thread has access to. Hence when you call afterwards get() you retrieve the value you have set before, despite the fact that in the meantime other threads may have called set().

Application servers in the Java EE world make heavy usage of the ThreadLocal feature as you have many parallel threads but each thread has for example its own transaction or security context. As you don't want to pass these objects within each method invocation, you simply store it in the thread's own memory and access it later when you need it.

Chapter 3

The java.util.concurrent Package

3.1 Introduction

The following chapter introduces the `java.util.concurrent` package. Within this package reside a bunch of interesting classes that provide necessary and helpful functionality needed to implement multi-threaded applications. After a discussion on how to use the Executor interface and its implementation, the chapter covers atomic data types and concurrent data structures. The final section throws light at the semaphores and count-down latches.

3.2 java.util.concurrent

After having read the previous articles about concurrency and multi-threading, you might have the feeling that it is not always trivial to write robust code that executes well in a multi-threaded environment. There is a proverb that illustrates this (source unknown):

- Junior programmers think concurrency is hard.
- Experienced programmers think concurrency is easy.
- Senior programmers think concurrency is hard.

Therefore a solid library of data structures and classes that provide well-tested thread-safety is of great help for anyone writing programs that make use of concurrency. Luckily the JDK provides a set of ready-to-use data structures and functionality for exactly that purpose. All these classes reside within the package java.util.concurrent.

3.2.1 Executor

The `java.util.concurrent` package defines a set of interfaces whose implementations execute tasks. The simplest one of these is the Executor interface:

```
public interface Executor {
        void execute(Runnable command);
}
```

Hence an Executor implementation takes the given Runnable instance and executes it. The interface makes no assumptions about the way of the execution, the javadoc only states "Executes the given command at some time in the future.". A simple implementation could therefore be:

```
public class MyExecutor implements Executor {

        public void execute(Runnable r) {
                (new Thread(r)).start();
        }
}
```

Along with the mere interface the JDK also ships a fully-fledged and extendable implementation named `ThreadPoolExecutor`. Under the hood the `ThreadPoolExecutor` maintains a pool of threads and dispatches the instances of `Runnable` given the `execute()` method to the pool. The arguments passed to the constructor control the behavior of the thread pool. The constructor with the most arguments is the following one:

ThreadPoolExecutor (int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

Let's go through the different arguments step by step:

- **corePoolSize**: The `ThreadPoolExecutor` has an attribute `corePoolSize` that determines how many threads it will start until new threads are only started when the queue is full.
- **maximumPoolSize**: This attribute determines how many threads are started at the maximum. You can set this to `Integer.MAX_VALUE` in order to have no upper boundary.
- **keepAliveTime**: When the `ThreadPoolExecutor` has created more than `corePoolSize` threads, a thread will be removed from the pool when it idles for the given amount of time.
- **unit**: This is just the `TimeUnit` for the `keepAliveTime`.
- **workQueue**: This queue holds the instances of Runnable given through the `execute()` method until they are actually started.
- **threadFactory**: An implementation of this interface gives you control over the creation of the threads used by the `ThreadPoolExecutor`.
- **handler**: When you specify a fixed size for the workQueue and provide a maximumPoolSize then it may happen, that the `ThreadPoolExecutor` is not able to execute your `Runnable` instance due to saturation. In this case the provided handler is called and gives you control over what should happen in this case.

As there are a lot of parameters to adjust, let's examine some code that uses them:

```
public class ThreadPoolExecutorExample implements Runnable {
        private static AtomicInteger counter = new AtomicInteger();
        private final int taskId;

        public int getTaskId() {
                return taskId;
        }

        public ThreadPoolExecutorExample(int taskId) {
                this.taskId = taskId;
        }

        public void run() {
                try {
                        Thread.sleep(5000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

        public static void main(String[] args) {
                BlockingQueue<Runnable> queue = new LinkedBlockingQueue<Runnable>(10);
                ThreadFactory threadFactory = new ThreadFactory() {
```

```
                        public Thread newThread(Runnable r) {
                                int currentCount = counter.getAndIncrement();
                                System.out.println("Creating new thread: " + currentCount);
                                return new Thread(r, "mythread" + currentCount);
                        }
                };
                RejectedExecutionHandler rejectedHandler = new RejectedExecutionHandler() {
                        public void rejectedExecution(Runnable r, ThreadPoolExecutor  ↩
                            executor) {
                                if (r instanceof ThreadPoolExecutorExample) {
                                        ThreadPoolExecutorExample example = ( ↩
                                            ThreadPoolExecutorExample) r;
                                        System.out.println("Rejecting task with id " +  ↩
                                            example.getTaskId());
                                }
                        }
                };
                ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 1, TimeUnit. ↩
                    SECONDS, queue, threadFactory, rejectedHandler);
                for (int i = 0; i < 100; i++) {
                        executor.execute(new ThreadPoolExecutorExample(i));
                }
                executor.shutdown();
        }
}
```

Our `run()` implementation only falls asleep for 5 seconds, but this is not the main focus of this code. The `ThreadPoolExecutor` starts with 5 core threads and allows the pool to grow up to 10 threads at the maximum. For demonstration purposes we allow an unused thread only to idle for about 1 second. The queue implementation here is a `LinkedBlockingQueue` with a capacity of 10 `Runnable` instances. We also implement a simple `ThreadFactory` in order to track the thread creation. The same is true for the `RejectedExecutionHandler`.

The loop in the `main()` method now issues 100 `Runnable` instance to the pool within a short amount of time. The output of the sample shows that we have to create 10 threads (up the maximum) to handle all pending `Runnables`:

```
Creating new thread: 0
...
Creating new thread: 9
Rejecting task with id 20
...
Rejecting task with id 99
```

But it also shows that all tasks with taskId greater than 19 are forwarded to the `RejectedExecutionHandler`. This is due to the fact that our `Runnable` implementation sleeps for 5 seconds. After the first 10 threads have been started the queue can only hold another 10 `Runnable` instances. All further instances then have to be rejected.

Finally the `shutdown()` method lets the `ThreadPoolExecutor` reject all further tasks and waits until the already issued tasks have been executed. You can replace the call of `shutdown()` with a call of `shutdownNow()`. The latter tries to interrupt all running threads and shuts down the thread pool without waiting for all threads to finish. In the example above you would see ten InterruptedException exceptions as our ten sleeping threads are woken up immediately.

3.2.2 ExecutorService

The `Executor` interface is very simple, it only forces the underlying implementation to implement the `execute()` method. The `ExecutorService` goes on step further as it extends the `Executor` interface and adds a series of utility methods (e.g. to add a complete collection of tasks), methods to shut down the thread pool as well as the ability to query the implementation for the result of the execution of one task. We have seen that the `Runnable` interface only defines a `run()` method is void as return value. Hence the introduction of a new interface named `Callable` was necessary that defines similar to `Runnable` also only one method, but this methods returns a value:

```
V call();
```

But how does the JDK handle the fact that a task returns a value but is submitted to a thread pool for execution?

The submitter of the task cannot know ahead when the task gets executed and how long the executions lasts. Letting the current thread wait for the result is obviously no solution. The work to check if the result is already available with the feature to block or to wait a certain amount of time is implemented in another class: `java.util.concurrent.Future<V>`. This class has only a few methods to check whether the task is done, to cancel the task as well as to retrieve its result.

Last but not least we have another interface which extends the `Executor` interface as well as the `ExecutorService` interface by some methods to schedule a task at a given point in time. The name of the interface is `ScheduledExecutorService` and it provides basically a `schedule()` method that takes an argument how long to wait until the task gets executed:

```
schedule(Callable<V> callable, long delay, TimeUnit unit);
schedule(Runnable command, long delay, TimeUnit unit);
```

Like for the `ExecutorService` the `schedule()` method is available in two variants: One for the `Runnable` interface and one for tasks that return a value using the `Callable` interface. The `ScheduledExecutorService` also provides a method to execute tasks periodically:

```
scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);
```

Next to the initial delay we can specify the period at which the task should run.

The last example has already shown how to create a `ThreadPoolExecutor`. The implementation for the `ScheduledExecutorService` is named `ScheduledThreadPoolExecutor` and has to be handled very similar to the `ThreadPoolExecutor` used above. But often the full control over all features of an ExecutorService is not necessary. Just imagine a simple test client that should invoke some server methods using a simple `ThreadPool`.

The creators of the JDK have therefore created a simple factory class named `Executors` (please mind the trailing s). This class provides a handful of static methods to create a read-to-use `ThreadPoolExecutor`. All of this together lets us implement a simple thread pool that executes a bunch of tasks that compute some number (the number crunching operation is here for demonstration purposes substituted by a simple `Thread.sleep()`):

```
public class ExecutorsExample implements Callable<Integer> {
        private static Random random = new Random(System.currentTimeMillis());

        public Integer call() throws Exception {
                Thread.sleep(1000);
                return random.nextInt(100);
        }

        public static void main(String[] args) throws InterruptedException,  ↩
            ExecutionException {
                ExecutorService executorService = Executors.newFixedThreadPool(5);
                Future<Integer>[] futures = new Future[5];
                for (int i = 0; i < futures.length; i++) {
                        futures[i] = executorService.submit(new ExecutorsExample());
                }
                for (int i = 0; i < futures.length; i++) {
                        Integer retVal = futures[i].get();
                        System.out.println(retVal);
                }
                executorService.shutdown();
        }
}
```

The creation of the `ExecutorService` is a one-liner. To execute some tasks we just need a for-loop that creates a few new instances of ExecutorsExample and stores the returned Future in an array. After we have submitted the tasks to the service, we just wait for the result. The method `get()` of `Future` is blocking, i.e. the current threads sleeps until the result is available. An overridden version of this method takes a timeout specification in order to let the waiting thread proceed if the task does not finish within the defined time period.

3.2.3 Concurrent collections

The Java collections framework encompasses a wide range of data structures that every Java programmers uses in his day to day work. This collection is extended by the data structures within the `java.util.concurrent` package. These implementations provided thread-safe collections to be used within a multi-threaded environment.

Many Java programmers even use thread-safe data structures from time to time even without knowing that. The "old" classes `Hashtable` and `Vector` are examples for such classes. Being part of the JDK since version 1.0, these basic data structures were designed with thread-safety in mind. Although the thread-safety here only means that all methods are synchronized on instance level. The following code is taken from Oracle's JDK implementation:

```
public synchronized void clear() {
        Entry tab[] = table;
        modCount++;
        for (int index = tab.length; --index >= 0; )
                tab[index] = null;
        count = 0;
}
```

This is crucial difference to the "newer" collection classes like `HashMap` or `ArrayList` (both available since JDK 1.2), which themselves are not thread-safe. But there is a convenient way to retrieve a thread-safe instance of such a "newer" collection class:

```
HashMap<Long,String> map = new HashMap<Long, String>();
Map<Long, String> synchronizedMap = Collections.synchronizedMap(map);
```

As we see in the code above, the `Collections` class lets us create at runtime a synchronized version of a formerly unsynchronized collections class.

As we have learned before, adding the keyword synchronized to a method results in the effect that at each point in time only one thread executes a method of the object under investigation. This is of course the easiest way to make a simple collection class thread-safe. More advanced techniques encompass special algorithms that are designed for concurrent access. These algorithms are implemented in the collection classes of the `java.util.concurrent` package.

An example for such a class is `ConcurrentHashMap`:

```
ConcurrentHashMap<Long,String> map = new ConcurrentHashMap<Long,String>();
map.put(key, value);
String value2 = map.get(key);
```

The code above looks nearly the same as for a normal `HashMap`, but the underlying implementation is completely different. Instead of using only one lock for the whole table the `ConcurrentHashMap` subdivides the whole table into many small partitions. Each partition has its own lock. Hence write operations to this map from different threads, assuming they are writing at different partitions of the table, do not compete and can use their own lock.

The implementation also introduces the idea of a commit of write operations to reduce the waiting time for read operations. This slightly changes the semantics of the read operation as it will return the result of the latest write operation that has finished. This means that the number of entries may not be the same directly before and after executing the read method, like it would be when using a synchronized method, but for concurrent applications this is not always important. The same is true for the iterator implementation of the `ConcurrentHashMap`.

To get a better feeling for the different performance of `Hashtable`, synchronized `HashMap` and `ConcurrentHashMap`, let's implement a simple performance test. The following code starts a few threads and lets each thread retrieve a value from the map at a random position and afterwards updates a value at another random position:

```
public class MapComparison implements Runnable {
        private static Map<Integer, String> map;
        private Random random = new Random(System.currentTimeMillis());

        public static void main(String[] args) throws InterruptedException {
                runPerfTest(new Hashtable<Integer, String>());
                runPerfTest(Collections.synchronizedMap(new HashMap<Integer,String>()));
                runPerfTest(new ConcurrentHashMap<Integer, String>());
```

```
                runPerfTest(new ConcurrentSkipListMap<Integer, String>());
        }

        private static void runPerfTest(Map<Integer, String> map) throws  ↩
            InterruptedException {
                MapComparison.map = map;
                fillMap(map);
                ExecutorService executorService = Executors.newFixedThreadPool(10);
                long startMillis = System.currentTimeMillis();
                for (int i = 0; i < 10; i++) {
                        executorService.execute(new MapComparison());
                }
                executorService.shutdown();
                executorService.awaitTermination(1, TimeUnit.MINUTES);
                System.out.println(map.getClass().getSimpleName() + " took " + (System. ↩
                    currentTimeMillis() - startMillis) + " ms");
        }

        private static void fillMap(Map<Integer, String> map) {
                for (int i = 0; i < 100; i++) {
                        map.put(i, String.valueOf(i));
                }
        }

        public void run() {
                for (int i = 0; i < 100000; i++) {
                        int randomInt = random.nextInt(100);
                        map.get(randomInt);
                        randomInt = random.nextInt(100);
                        map.put(randomInt, String.valueOf(randomInt));
                }
        }
}
```

The output of this program is the following:

```
Hashtable took 436 ms
SynchronizedMap took 433 ms
ConcurrentHashMap took 75 ms
ConcurrentSkipListMap took 89 ms
```

As we have expected, the `Hashtable` and the synchronized `HashMap` implementations are far behind the ones from the concurrent package. This example also introduces a skip list implementation of the HashMap, where the linked items within one bucket form a skip list, meaning the list is sorted and there are different levels of linking the items within the list. The highest level pointer points directly to some item in the middle of the list. If this item is already greater than the current item, the iterator has to take the next lower level of linkage to skip fewer elements than on the highest level. A detailed description of skip lists can be found here. The interesting point about skip list is that all read access takes about log(n) time, even if all items are stored within the same bucket.

3.2.4 Atomic Variables

When having multiple threads sharing a single variable, we have the task to synchronize access to this variable. The reason for this is the fact, that even a simple instruction like i++ is not atomic. It basically consists of the following bytecode instructions:

```
iload_1
iinc 1, 1
istore_1
```

Without knowing too much about the Java bytecode, one sees that the current value of the local variable 1 is pushed onto the operand stack, that it is incremented by the constant value 1 and afterwards popped from the stack and stored in the local variable

number one. This means we need three atomic operations in order to increment a local variable by one. In a multi-threading environment this also means that the scheduler can stop the execution of the current thread between each of these instructions and start a new thread, which in turn then can work on the same variable.

To cope with situations like this you can of course synchronize the access to this specific variable:

```
synchronized(i) {
        i++;
}
```

But this also means the current thread has to acquire the lock on i which needs some internal synchronization and computation within the JVM. This approach is also called pessimistic locking as we assume that it is highly probable that another thread currently holds the lock we want to acquire. A different approach called optimistic locking, assumes that there are not so many threads competing for the resource and hence we just try to update the value and see if this has worked. One implementation of this approach is the compare-and-swap (CAS) method. This operation is implemented on many modern CPUs as atomic operation. It compares the content of a given memory location with a given value (the "expected value") and updates it to a new value if the current value equals to the expected value. In pseudo code this looks like:

```
int currentValue = getValueAtMemoryPosition(pos);
if(currentValue == expectedValue) {
        setValueAtMemoryPosition(pos, newValue);
}
```

The CAS operation implements the code above as one atomic operation. Therefore it can be used to see if the value of some variable has still the value the current thread holds and updates it to the incremented value in this case. As the usage of the CAS operation needs hardware support, the JDK provides special classes to support these operations. They all reside within the package java.util.concurrent.atomic.

One representative of these classes is java.util.concurrent.atomic.AtomicInteger. The CAS operation discussed above is implemented by the method

```
        boolean compareAndSet(int expect, int update)
```

The boolean return value indicates if the update operation was successful or not. On the basis of this functionality further operation like an atomic increment operation can be implemented (here taken from Oracle' JDK implementation):

```
    public final int getAndIncrement() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }
```

Now we are able to increment an integer variable by different threads without using pessimistic locks:

```
public class AtomicIntegerExample implements Runnable {
        private static final AtomicInteger atomicInteger = new AtomicInteger();

        public static void main(String[] args) {
                ExecutorService executorService = Executors.newFixedThreadPool(5);
                for (int i = 0; i < 5; i++) {
                        executorService.execute(new AtomicIntegerExample());
                }
                executorService.shutdown();
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        int newValue = atomicInteger.getAndIncrement();
                        if (newValue == 42) {
```

```
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                    "]: " + newValue);
                        }
                }
        }
}
```

The code above starts five threads and lets each of them increment the `AtomicInteger` variable. The lucky thread that gets the answer 42 prints this to the console. When executing this example code in repetition, the output will only be created by exactly one thread.

Next to `AtomicInteger` the JDK also offers classes for atomic operations on long values, integer and long arrays and references.

3.2.5 Semaphore

Semaphores are used to control access to a shared resource. In contrast to simple synchronized blocks a semaphore has an internal counter that is increased each time a thread acquires a lock and decreased each time a thread releases a lock it obtained before. The increasing and decreasing operations are of course synchronized, hence a semaphore can be used to control how many threads pass simultaneously through a critical section. The two basic operations of a thread are:

```
void acquire();
void release();
```

The constructor takes next to the number of concurrently locks a fairness parameter. The fairness parameter decides if new threads, which try to acquire a lock, are set at the beginning or at the end of the list of waiting threads. Putting the new thread at the end of the threads guarantees that all threads will acquire the lock after some time and hence no thread starves.

```
Semaphore(int permits, boolean fair)
```

To illustrate the described behavior, let's setup a simple thread pool with five threads but control through a semaphore that at each point in time not more than three of them are running:

```
public class SemaphoreExample implements Runnable {
        private static final Semaphore semaphore = new Semaphore(3, true);
        private static final AtomicInteger counter = new AtomicInteger();
        private static final long endMillis = System.currentTimeMillis() + 10000;

        public static void main(String[] args) {
                ExecutorService executorService = Executors.newFixedThreadPool(5);
                for (int i = 0; i < 5; i++) {
                        executorService.execute(new SemaphoreExample());
                }
                executorService.shutdown();
        }

        public void run() {
                while(System.currentTimeMillis() < endMillis) {
                        try {
                                semaphore.acquire();
                        } catch (InterruptedException e) {
                                System.out.println("["+Thread.currentThread().getName()+"]  ↩
                                    Interrupted in acquire().");
                        }
                        int counterValue = counter.incrementAndGet();
                        System.out.println("["+Thread.currentThread().getName()+"]  ↩
                            semaphore acquired: "+counterValue);
                        if(counterValue > 3) {
                                throw new IllegalStateException("More than three threads  ↩
                                    acquired the lock.");
```

```
                        }
                        counter.decrementAndGet();
                        semaphore.release();
                }
        }
}
```

The Semaphore is constructed by passing 3 as the number of concurrent permits. When trying to acquire a lock, the blocked thread may experience an `InterruptedException` that has to be caught. Alternatively one could also call the utility method `acquireUninterruptibly()` to circumvent the try-catch construct.

To ensure that we have not more than three concurrent threads within the critical section, we use an `AtomicInteger` that gets incremented each time a process enters the section and decremented before it leaves the section. When the counter has a value higher than four, an `IllegalStateException` is thrown. Finally we `release()` the semaphore and let another waiting thread enter the critical section.

3.2.6 CountDownLatch

The `CountDownLatch` class is another helpful class for thread synchronization from the JDK. Similar to the `Semaphore` class it provides a counter, but the counter of the `CountDownLatch` can only be decreased until it reaches zero. Once the counter has reached zero all threads waiting on the `CountDownLatch` can proceed. Such functionality is often necessary when all threads of a pool have to synchronize at some point in order to proceed. A simple example would be an application that has to gather data from different sources before being able to store a new data set to the database.

The following code demonstrates how five threads sleep for a random amount of time. Each thread which wakes up counts down the latch and then awaits the latch to become zero. Finally all threads output that they have finished.

```
public class CountDownLatchExample implements Runnable {
        private static final int NUMBER_OF_THREADS = 5;
        private static final CountDownLatch latch = new CountDownLatch(NUMBER_OF_THREADS);
        private static Random random = new Random(System.currentTimeMillis());

        public static void main(String[] args) {
                ExecutorService executorService = Executors.newFixedThreadPool( ↩
                    NUMBER_OF_THREADS);
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        executorService.execute(new CountDownLatchExample());
                }
                executorService.shutdown();
        }

        public void run() {
                try {
                        int randomSleepTime = random.nextInt(20000);
                        System.out.println("[" + Thread.currentThread().getName() + "]  ↩
                            Sleeping for " + randomSleepTime);
                        Thread.sleep(randomSleepTime);
                        latch.countDown();
                        System.out.println("[" + Thread.currentThread().getName() + "]  ↩
                            Waiting for latch.");
                        latch.await();
                        System.out.println("[" + Thread.currentThread().getName() + "]  ↩
                            Finished.");
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

When running this example you will see that the output "Waiting for latch." comes at different points in time but that the "Finished." message of each thread is printed immediately one after the other.

3.2.7 CyclicBarrier

In contrast to the `CountDownLatch`, the `CyclicBarrier` class implements a counter that can be reset after being counted down to zero. All threads have to call its method `await()` until the internal counter is set to zero. The waiting threads are then woken up and can proceed. Internally the counter is then reset to its original value and the whole procedure can start again:

```
public class CyclicBarrierExample implements Runnable {
        private static final int NUMBER_OF_THREADS = 5;
        private static AtomicInteger counter = new AtomicInteger();
        private static Random random = new Random(System.currentTimeMillis());
        private static final CyclicBarrier barrier = new CyclicBarrier(5, new Runnable() {
                public void run() {
                        counter.incrementAndGet();
                }
        });

        public static void main(String[] args) {
                ExecutorService executorService = Executors.newFixedThreadPool( ↩
                    NUMBER_OF_THREADS);
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        executorService.execute(new CyclicBarrierExample());
                }
                executorService.shutdown();
        }

        public void run() {
                try {
                        while(counter.get() < 3) {
                                int randomSleepTime = random.nextInt(10000);
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                    "] Sleeping for " + randomSleepTime);
                                Thread.sleep(randomSleepTime);
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                    "] Waiting for barrier.");
                                barrier.await();
                                System.out.println("[" + Thread.currentThread().getName() + ↩
                                    "] Finished.");
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

The example above is very similar to the `CountDownLatch`, but in contrast to the previous example, I have added a while loop to the `run()` method. This `run()` implementations lets each thread continue the sleeping and `await()` procedure until the counter is three. Also note the anonymous Runnable() implementation provided to the constructor of `CyclicBarrier`. Its `run()` method gets executed each time the barrier is tripped. Here we increase the counter that is checked by the concurrent threads.

3.3 Download the source code

You can download the source code of this lesson: concurrency-4.zip

Chapter 4

Performance, Scalability and Liveness

4.1 Introduction

This article discusses the topic of performance for multi-threaded applications. After defining the terms performance and scalability, we take a closer look at Amdahl's Law. Further in the course we see how we can reduce lock contention by applying different techniques, which are demonstrated with code examples.

4.2 Performance

Threads can be used to improve the performance of applications. The reason behind this may be that we have multiple processors or CPU cores available. Each CPU core can work on its own task, hence dividing on big task into a series of smaller tasks that run independently of each other, can improve the total runtime of the application. An example for such a performance improvement could be an application that resizes images that lie within a folder structure on the hard disc. A single threaded approach would just iterate over all files and scale each image after the other. If we have a CPU with more than one core, the resizing process would only utilize one of the available cores. A multi-threaded approach could for example let a producer thread scan the file system and add all found files to a queue, which is processed by a bunch of worker threads. When we have as many worker threads as we have CPU cores, we make sure that each CPU core has something to do until all images are processed.

Another example where multi-threading can improve the overall performance of an application are use cases with a lot of I/O waiting time. Let's assume we want to write an application that mirrors a complete website in form of HTML files to our hard disc. Starting with one page, the application has to follow all links that point to the same domain (or URL part). As the time between issuing the request to the remote web server up to the moment until all data has been received may be long, we can distribute the work over a few threads. One or more threads can parse the received HTML pages and put the found links into a queue, while other threads issue the requests to the web server and then wait for the answer. In this case we use the waiting time for newly requested pages with the parsing of the already received ones. In contrast to the previous example, this application may even gain performance if we add more threads than we have CPU cores.

These two examples show that performance means to get more work done in a shorter time frame. This is of course the classical understanding of the term performance. But the usage of threads can also improve the responsiveness of our applications. Imagine as simple GUI application with an input form and a "Process" button. When the user presses the button, the application has to render the pressed button (the button should lock like being pressed down and coming up again when releasing the mouse) and the actual processing of the input data has to be done. If this processing takes longer, a single threaded application could not react to further user input, i.e. we need an additional thread that processes events from the operating system like mouse clicks or mouse pointer movements.

Scalability means the ability of a program to improve the performance by adding further resources to it. Imagine we would have to resize a huge amount of images. As the number of CPU cores of our current machine is limited, adding more threads does not improve the performance. The performance may even degrade as the scheduler has to manage more threads and thread creation and shutdown also costs CPU power.

4.2.1 Amdahl's Law

The last section has shown that in some cases the addition of new resources can improve the overall performance of our application. In order to able to compute how much performance our application may gain when we add further resources, we need to identify the parts of the program that have to run serialized/synchronized and the parts of the program that can run in parallel. If we denote the fraction of the program that has to run synchronized with B (e.g. the number of lines that are executed synchronized) and if we denote the number of available processors with n, then Amdahl's Law lets us compute an upper limit for the speedup our application may be able to achieve:

$$\frac{1}{B + \frac{1}{n}(1 - B)}$$

Figure 4.1: screenshot

If we let n approach infinity, the term (1-B)/n converges against zero. Hence we can neglect this term and the upper limit for the speedup converges against 1/B, where B is the fraction of the program runtime before the optimization that is spent within non-parallelizable code. If B is for example 0.5, meaning that half of the program cannot be parallelized, the reciprocal value of 0.5 is 2; hence even if we add an unlimited number of processor to our application, we would only gain a speedup of about two. Now let's assume we can rewrite the code such that only 0.25 of the program runtime is spent in synchronized blocks. Now the reciprocal value of 0.25 is 4, meaning we have built an application that would run with a large number of processors about four times faster than with only one processor.

The other way round, we can also use Amdahl's Law to compute the fraction of the program runtime that has to be executed synchronized to achieve a given speedup. If we want to achieve a speedup of about 100, the reciprocal value is 0.01, meaning we should only spent about 1 percent of the runtime within synchronized code.

To summarize the findings from Amdahl's Law, we can conclude that the maximum speed up a program can get through the usage of additional processor is limited by the reciprocal value of the fraction of time the program spends in synchronized code parts. Although it is not always easy to compute this fraction in practice, even not if you think about large business applications, the law gives us a hint that we have to think about synchronization very carefully and that we have to keep the parts of the program runtime small, that have to be serialized.

4.2.2 Performance impacts of threads

The writings of this article up to this point indicate that adding further threads to an application can improve the performance and responsiveness. But on the other hand, this does not come for free. Threads always have some performance impact themselves.

The first performance impact is the creation of the thread itself. This takes some time as the JVM has to acquire the resources for the thread from the underlying operating system and prepare the data structures in the scheduler, which decides which thread to execute next.

If you use as many threads as you have processor cores, then each thread can run on its own processor and might not be interrupted much often. In practice the operating system may require of course its own computations while your application is running; hence even in this case threads are interrupted and have to wait until the operating system lets them run again. The situation gets even worse, when you have to use more threads than you have CPU cores. In this case the scheduler can interrupt your thread in order to let another thread execute its code. In such a situation the current state of the running thread has to be saved, the state of the scheduled thread that is supposed to run next has to be restored. Beyond that the scheduler itself has to perform some updates on its internal data structures which again use CPU power. All together this means that each context switch from one thread to the other costs CPU power and therefore introduces performance degeneration in comparison to a single threaded solution.

Another cost of having multiple threads is the need to synchronize access to shared data structures. Next to the usage of the keyword synchronized we can also use volatile to share data between multiple threads. If more than one thread competes for the shared data structured we have contention. The JVM has then to decide which thread to execute next. If this is not the current thread, costs for a context switch are introduced. The current thread then has to wait until it can acquire the lock. The

JVM can decide itself how to implement this waiting. When the expected time until the lock can be acquired is small, then spin-waiting, i.e. trying to acquire the lock again and again, might be more efficient compared to the necessary context switch when suspending the thread and letting another thread occupy the CPU. Bringing the waiting thread back to execution entails another context switch and adds additional costs to the lock contention.

Therefore it is reasonable to reduce the number of context switches that are necessary due to lock contention. The following section describes two approaches how to reduce this contention.

4.2.3 Lock contention

As we have seen in the previous section, two or more thread competing for one lock introduce additional clock cycles as the contention may force the scheduler to either let one thread spin-waiting for the lock or let another thread occupy the processor with the cost of two context switches. In some cases lock contention can be reduced by applying one of the following techniques:

- The scope of the lock is reduced.
- The number of times a certain lock is acquired is reduced.
- Using hardware supported optimistic locking operations instead of synchronization.
- Avoid synchronization where possible
- Avoid object pooling

4.2.3.1 2.3.1 Scope reduction

The first technique can be applied when the lock is hold longer than necessary. Often this can be achieved by moving one or more lines out of the synchronized block in order to reduce the time the current thread holds the lock. The less number of lines of code to execute the earlier the current thread can leave the synchronized block and therewith let other threads acquire the lock. This is also aligned with Amdahl's Law, as we reduce the fraction of the runtime that is spend within synchronized blocks.

To better understand this technique, take a look at the following source code:

```
public class ReduceLockDuration implements Runnable {
        private static final int NUMBER_OF_THREADS = 5;
        private static final Map<String, Integer> map = new HashMap<String, Integer>();

        public void run() {
                for (int i = 0; i < 10000; i++) {
                        synchronized (map) {
                                UUID randomUUID = UUID.randomUUID();
                                Integer value = Integer.valueOf(42);
                                String key = randomUUID.toString();
                                map.put(key, value);
                        }
                        Thread.yield();
                }
        }

        public static void main(String[] args) throws InterruptedException {
                Thread[] threads = new Thread[NUMBER_OF_THREADS];
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        threads[i] = new Thread(new ReduceLockDuration());
                }
                long startMillis = System.currentTimeMillis();
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        threads[i].start();
                }
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        threads[i].join();
```

```
                }
                System.out.println((System.currentTimeMillis()-startMillis)+"ms");
        }
}
```

In this sample application, we let five threads compete for the access of the shared Map. To let only one thread at a time access the Map, the code that accesses the Map and adds a new key/value pair is put into a synchronized block. When we take a closer look at the block, we see that the computation of the key as well as the conversion of the primitive integer 42 into an Integer object must not be synchronized. They belong conceptually to the code that accesses the Map, but they are local to the current thread and the instances are not modified by other threads. Hence we can move them out of the synchronized block:

```
        public void run() {
                for (int i = 0; i < 10000; i++) {
                        UUID randomUUID = UUID.randomUUID();
                        Integer value = Integer.valueOf(42);
                        String key = randomUUID.toString();
                        synchronized (map) {
                                map.put(key, value);
                        }
                        Thread.yield();
                }
        }
```

The reduction of the synchronized block has an effect on the runtime that can be measured. On my machine the runtime of the whole application is reduced from 420ms to 370ms for the version with the minimized synchronized block. This makes a total of 11% less runtime just by moving three lines of code out of the synchronized block. The statement `Thread.yield()` is introduced to provoke more context switches, as this method call tells the JVM that the current thread is willing to give the processor to another waiting thread. This again provokes more lock contention as otherwise one thread may run too long on the processor without any competing thread.

4.2.3.2 2.3.2 Lock splitting

Another technique to reduce lock contention is to split one lock into a number of smaller scoped locks. This technique can be applied if you have one lock for guarding different aspects of your application. Assume we want to collect some statistical data about our application and implement a simple counter class that holds for each aspect a primitive counter variable. As our application is multi-threaded, we have to synchronize access to these variables, as they are accessed from different concurrent threads. The easiest way to accomplish this is to use the synchronized keyword within the method signature for each method of Counter:

```
        public static class CounterOneLock implements Counter {
                private long customerCount = 0;
                private long shippingCount = 0;

                public synchronized void incrementCustomer() {
                        customerCount++;
                }

                public synchronized void incrementShipping() {
                        shippingCount++;
                }

                public synchronized long getCustomerCount() {
                        return customerCount;
                }

                public synchronized long getShippingCount() {
                        return shippingCount;
                }
        }
```

This approach also means that each increment of a counter locks the whole instance of Counter. Other threads that want to increment a different variable have to wait until this single lock is released. More efficient in this case is to use separate locks for each counter like in the next example:

```
        public static class CounterSeparateLock implements Counter {
                private static final Object customerLock = new Object();
                private static final Object shippingLock = new Object();
                private long customerCount = 0;
                private long shippingCount = 0;

                public void incrementCustomer() {
                        synchronized (customerLock) {
                                customerCount++;
                        }
                }

                public void incrementShipping() {
                        synchronized (shippingLock) {
                                shippingCount++;
                        }
                }

                public long getCustomerCount() {
                        synchronized (customerLock) {
                                return customerCount;
                        }
                }

                public long getShippingCount() {
                        synchronized (shippingLock) {
                                return shippingCount;
                        }
                }
        }
```

This implementation introduces two separate synchronization objects, one for each counter. Hence a thread trying to increase the number of customers in our system only has to compete with other threads that also increment the number of customers but it has not to compete with threads trying to increment the number of shipping.

By using the following class we can easily measure the impact of this lock splitting:

```
public class LockSplitting implements Runnable {
        private static final int NUMBER_OF_THREADS = 5;
        private Counter counter;

        public interface Counter {
                void incrementCustomer();

                void incrementShipping();

                long getCustomerCount();

                long getShippingCount();
        }

        public static class CounterOneLock implements Counter { ...     }

        public static class CounterSeparateLock implements Counter { ... }

        public LockSplitting(Counter counter) {
                this.counter = counter;
        }
```

```
        public void run() {
                for (int i = 0; i < 100000; i++) {
                        if (ThreadLocalRandom.current().nextBoolean()) {
                                counter.incrementCustomer();
                        } else {
                                counter.incrementShipping();
                        }
                }
        }

        public static void main(String[] args) throws InterruptedException {
                Thread[] threads = new Thread[NUMBER_OF_THREADS];
                Counter counter = new CounterOneLock();
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        threads[i] = new Thread(new LockSplitting(counter));
                }
                long startMillis = System.currentTimeMillis();
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        threads[i].start();
                }
                for (int i = 0; i < NUMBER_OF_THREADS; i++) {
                        threads[i].join();
                }
                System.out.println((System.currentTimeMillis() - startMillis) + "ms");
        }
}
```

On my machine the implementation with one single lock takes about 56ms in average, whereas the variant with two separate locks takes about 38ms. This is a reduction of about 32 percent.

Another possible improvement is to separate locks even more by distinguishing between read and write locks. The `Counter` class for example provides methods for reading and writing the counter value. While reading the current value can be done by more than one thread in parallel, all write operations have to be serialized. The `java.util.concurrent` package provides a ready to use implementation of such a `ReadWriteLock`.

The `ReentrantReadWriteLock` implementation manages two separate locks. One for read accesses and one for write accesses. Both the read and the write lock offer methods for locking and unlocking. The write lock is only acquired, if there is no read lock. The read lock can be acquired by more than on reader thread, as long as the write lock is not acquired. For demonstration purposes the following shows an implementation of the counter class using a `ReadWriteLock`:

```
        public static class CounterReadWriteLock implements Counter {
                private final ReentrantReadWriteLock customerLock = new  ↩
                    ReentrantReadWriteLock();
                private final Lock customerWriteLock = customerLock.writeLock();
                private final Lock customerReadLock = customerLock.readLock();
                private final ReentrantReadWriteLock shippingLock = new  ↩
                    ReentrantReadWriteLock();
                private final Lock shippingWriteLock = shippingLock.writeLock();
                private final Lock shippingReadLock = shippingLock.readLock();
                private long customerCount = 0;
                private long shippingCount = 0;

                public void incrementCustomer() {
                        customerWriteLock.lock();
                        customerCount++;
                        customerWriteLock.unlock();
                }

                public void incrementShipping() {
                        shippingWriteLock.lock();
                        shippingCount++;
```

```
                shippingWriteLock.unlock();
        }

        public long getCustomerCount() {
                customerReadLock.lock();
                long count = customerCount;
                customerReadLock.unlock();
                return count;
        }

        public long getShippingCount() {
                shippingReadLock.lock();
                long count = shippingCount;
                shippingReadLock.unlock();
                return count;
        }
}
```

All read accesses are guarded by an acquisition of the read lock, while all write accesses are guarded by the corresponding write lock. In case the application uses much more read than write accesses, this kind of implementation can even gain more performance improvement than the previous one as all the reading threads can access the getter method in parallel.

4.2.3.3 2.3.3 Lock striping

The previous example has shown how to split one single lock into two separate locks. This allows the competing threads to acquire only the lock that protects the data structures they want to manipulate. On the other hand this technique also increases complexity and the risk of dead locks, if not implemented properly.

Lock striping on the other hand is a technique similar to lock splitting. Instead of splitting one lock that guards different code parts or aspects, we use different locks for different values. The class `ConcurrentHashMap` from JDK's `java.util.concurrent` package uses this technique to improve the performance of applications that heavily rely on `HashMap`s. In contrast to a synchronized version of `java.util.HashMap`, `ConcurrentHashMap` uses 16 different locks. Each lock guards only 1/16 of the available hash buckets. This allows different threads that want to insert data into different sections of the available hash buckets to do this concurrently, as their operation is guarded by different locks. On the other hand it also introduces the problem to acquire more than one lock for specific operations. If you want to copy for example the whole Map, all 16 locks have to be acquired.

4.2.3.4 2.3.4 Atomic operations

Another way to reduce lock contention is to use so called atomic operations. This principle is explained and evaluated in more detail in one of the following articles. The `java.util.concurrent` package offers support for atomic operations for some primitive data types. Atomic operations are implemented using the so called compare-and-swap (CAS) operation provided by the processor. The CAS instruction only updates the value of a certain register, if the current value equals the provided value. Only in this case the old value is replaced by the new value.

This principle can be used to increment a variable in an optimistic way. If we assume our thread knows the current value, then it can try to increment it by using the CAS operation. If it turns out, that another thread has meanwhile incremented the value and our value is no longer the current one, we request the current value and try with it again. This can be done until we successfully increment the counter. The advantage of this implementation, although we may need some spinning, is that we don't need any kind of synchronization.

The following implementation of the `Counter` class uses the atomic variable approach and does not use any synchronized block:

```
public static class CounterAtomic implements Counter {
        private AtomicLong customerCount = new AtomicLong();
        private AtomicLong shippingCount = new AtomicLong();

        public void incrementCustomer() {
```

```
                customerCount.incrementAndGet();
        }

        public void incrementShipping() {
                shippingCount.incrementAndGet();
        }

        public long getCustomerCount() {
                return customerCount.get();
        }

        public long getShippingCount() {
                return shippingCount.get();
        }
}
```

Compared to the `CounterSeparateLock` class the total average runtime decreases from 39ms to 16ms. This is a reduction in runtime of about 58 percent.

4.2.3.5 2.3.5 Avoid hotspots

A typical implementation of a list will manage internally a counter that holds the number of items within the list. This counter is updated every time a new item is added to the list or removed from the list. If used within a single-threaded application, this optimization is reasonable, as the `size()` operation on the list will return the previously computed value directly. If the list does not hold the number of items in the list, the `size()` operation would have to iterate over all items in order to calculate it.

What is a common optimization in many data structures can become a problem in multi-threaded applications. Assume we want to share an instance of this list with a bunch of threads that insert and remove items from the list and query its size. The counter variable is now also a shared resource and all access to its value has to be synchronized. The counter has become a hotspot within the implementation.

The following code snippet demonstrates this problem:

```
public static class CarRepositoryWithCounter implements CarRepository {
        private Map<String, Car> cars = new HashMap<String, Car>();
        private Map<String, Car> trucks = new HashMap<String, Car>();
        private Object carCountSync = new Object();
        private int carCount = 0;

        public void addCar(Car car) {
                if (car.getLicencePlate().startsWith("C")) {
                        synchronized (cars) {
                                Car foundCar = cars.get(car.getLicencePlate());
                                if (foundCar == null) {
                                        cars.put(car.getLicencePlate(), car);
                                        synchronized (carCountSync) {
                                                carCount++;
                                        }
                                }
                        }
                } else {
                        synchronized (trucks) {
                                Car foundCar = trucks.get(car.getLicencePlate());
                                if (foundCar == null) {
                                        trucks.put(car.getLicencePlate(), car);
                                        synchronized (carCountSync) {
                                                carCount++;
                                        }
                                }
                        }
                }
```

```
                }

                public int getCarCount() {
                        synchronized (carCountSync) {
                                return carCount;
                        }
                }
        }
```

The `CarRepository` implementation holds two lists: One for cars and one for trucks. It also provides a method that returns the number of cars and trucks that are currently in both lists. As an optimization it increments the internal counter each time a new car is added to one of the two lists. This operation has to be synchronized with the dedicated `carCountSync` instance. The same synchronization is used when the count value is returned.

In order to get rid of this additional synchronization, the `CarRepository` could have also been implemented by omitting the additional counter and computing the number of total cars each time the value is queried by calling `getCarCount()`:

```
        public static class CarRepositoryWithoutCounter implements CarRepository {
                private Map<String, Car> cars = new HashMap<String, Car>();
                private Map<String, Car> trucks = new HashMap<String, Car>();

                public void addCar(Car car) {
                        if (car.getLicencePlate().startsWith("C")) {
                                synchronized (cars) {
                                        Car foundCar = cars.get(car.getLicencePlate());
                                        if (foundCar == null) {
                                                cars.put(car.getLicencePlate(), car);
                                        }
                                }
                        } else {
                                synchronized (trucks) {
                                        Car foundCar = trucks.get(car.getLicencePlate());
                                        if (foundCar == null) {
                                                trucks.put(car.getLicencePlate(), car);
                                        }
                                }
                        }
                }

                public int getCarCount() {
                        synchronized (cars) {
                                synchronized (trucks) {
                                        return cars.size() + trucks.size();
                                }
                        }
                }
        }
```

Now we need to synchronized with the cars and trucks list in the `getCarCount()` method and compute the size, but the additional synchronization during the addition of new cars can be left out.

4.2.3.6 2.3.6 Avoid object pooling

In the first versions of the Java VM object creation using the new operator was still an expensive operation. This led many programmers to the common pattern of object pooling. Instead of creating certain objects again and again, they constructed a pool of these objects and each time an instance was needed, one was taken from the pool. After having used the object, it was put back into the pool and could be used by another thread.

What makes sense at first glance can be a problem when used in multi-threaded applications. Now the object pool is shared between all threads and access to the objects within the pool has to be synchronized. This additional synchronization overhead

can now be bigger than the costs of the object creation itself. This is even true when you consider the additional costs of the garbage collector for collecting the newly created object instances.

As with all performance optimization this example shows once again, that each possible improvement should be measured carefully before being applied. Optimizations that seem to make sense at first glance can turn out to be even performance bottlenecks when not implemented correctly.

Chapter 5

Fork/Join Framework

5.1 Introduction

This article gives an introduction into the Fork/Join Framework that is part of the JDK since version 1.7. It describes the basic features of the frameworks and provides some examples in order to provide some practical experience.

5.2 Fork/Join

The base class of the Fork/Join Framework is `java.util.concurrent.ForkJoinPool`. This class implements the two interfaces `Executor` and `ExecutorService` and subclasses the `AbstractExecutorService`. Hence the `ForkJoinPool` is basically a thread pool that takes special kinds of tasks, namely the `ForkJoinTask`. This class implements the already known interface `Future` and therewith methods like `get()`, `cancel()` and `isDone()`. Beyond that this class also provides two methods that gave the whole framework its name: `fork()` and `join()`.

While a call of `fork()` will start an asynchronous execution of the task, a call of `join()` will wait until the task has finished and retrieve its result. Hence we can split a given task into multiple smaller tasks, fork each task and finally wait for all tasks to finish. This makes the implementation of complex problems easier.

In computer science this approach is also known as divide-and-conquer approach. Every time a problem is too complex to solve at once, it is divided into multiple smaller and easier to solve problems. This can be written in pseudo code like that:

```
if(problem.getSize() > THRESHOLD) {
        SmallerProblem smallerProblem1 = new SmallerProblem();
        smallerProblem1.fork();
        SmallerProblem smallerProblem2 = new SmallerProblem();
        smallerProblem2.fork();
        return problem.solve(smallerProblem1.join(), smallerProblem1.join());
} else {
        return problem.solve();
}
```

First we check if the current size of the problem is bigger than a given threshold. If this is the case, we divide the problem into smaller problems, `fork()` each new task and then wait for the results by calling `join()`. As `join()` returns the results for each subtask, we have to find the best solution of the smaller problems and return this as our best solution. These steps are repeated until the given threshold is too low and the problem so small that we can compute its solution directly without further division.

5.2.1 RecursiveTask

To grasp this procedure a little bit better, we implement an algorithm that finds the smallest number within an array of integer values. This problem is not one you would solve in your day to day work using a `ForkJoinPool`, but the following implemen-

tation shows the basic principles very clearly. In the `main()` method we setup an integer array with random values and create a new `ForkJoinPool`.

The first parameter passed to its constructor is an indicator for the level of desired parallelism. Here we query the `Runtime` for the number of available CPU cores. Then we call the `invoke()` method and pass an instance of `FindMin`. `FindMin` extends the class `RecursiveTask`, which itself is a subclass of the `ForkJoinTask` mentioned before. The class `ForkJoinTask` has actually two subclasses: one is designed for tasks that return a value (`RecursiveTask`) and one that is designed for tasks without return value (`RecursiveAction`). The superclass forces us to implement `compute()`. Here we take a look at the given slice of the integer array and decide whether the current problem is too big to be solved immediately or not.

When finding the smallest number within an array, the smallest problem size to be solved directly is to compare two elements with each other and return the smallest value of these. If we have currently more than two elements, we divide the array into two parts and find again the smallest number within these two parts. This is done by creating two new instances of `FindMin`.

The constructor is fed with the array and the start and end index. Then we start the execution of these two task asynchronously by calling `fork()`. This call submits the two tasks in the queue of the thread pool. The thread pool implements a strategy called work-stealing, i.e. if all other threads have enough to do, the current threads steals its work from one of the other tasks. This makes sure that tasks get executed as fast as possible.

```
public class FindMin extends RecursiveTask<Integer> {
        private static final long serialVersionUID = 1L;
        private int[] numbers;
        private int startIndex;
        private int endIndex;

        public FindMin(int[] numbers, int startIndex, int endIndex) {
                this.numbers = numbers;
                this.startIndex = startIndex;
                this.endIndex = endIndex;
        }

        @Override
        protected Integer compute() {
                int sliceLength = (endIndex - startIndex) + 1;
                if (sliceLength > 2) {
                        FindMin lowerFindMin = new FindMin(numbers, startIndex, startIndex  ↩
                            + (sliceLength / 2) - 1);
                        lowerFindMin.fork();
                        FindMin upperFindMin = new FindMin(numbers, startIndex + ( ↩
                            sliceLength / 2), endIndex);
                        upperFindMin.fork();
                        return Math.min(lowerFindMin.join(), upperFindMin.join());
                } else {
                        return Math.min(numbers[startIndex], numbers[endIndex]);
                }
        }

        public static void main(String[] args) {
                int[] numbers = new int[100];
                Random random = new Random(System.currentTimeMillis());
                for (int i = 0; i < numbers.length; i++) {
                        numbers[i] = random.nextInt(100);
                }
                ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime(). ↩
                    availableProcessors());
                Integer min = pool.invoke(new FindMin(numbers, 0, numbers.length - 1));
                System.out.println(min);
        }
}
```

5.2.2 RecursiveAction

As mentioned above next to `RecursiveTask` we also have the class `RecursiveAction`. In contrast to `RecursiveTask` it does not have to return a value, hence it can be used for asynchronous computations that can be directly performed on a given data structure. Such an example is the computation of a grayscale image out of a colored image. All we have to do is to iterate over each pixel of the image and compute the grayscale value out of the RGB value using the following formula:

```
gray = 0.2126 * red + 0.7152 * green + 0.0722 * blue
```

The floating point numbers represent how much the specific color contributes to our human perception of gray. As the highest value is used for green, we can conclude that a grayscale image is computed to nearly 3/4 just of the green part. So the basic implementation would look like this, assuming that image is our object representing the actual pixel data and the methods `setRGB()` and `getRGB()` are used to retrieve the actual RGB value:

```
for (int row = 0; row < height; row++) {
        for (int column = 0; column < bufferedImage.getWidth(); column++) {
                int grayscale = computeGrayscale(image.getRGB(column, row));
                image.setRGB(column, row, grayscale);
        }
}
```

The above implementation works fine on a single CPU machine. But if we have more than one CPU available, we might want to distribute this work to the available cores. So instead of iterating in two nested for loops over all pixels, we can use a `ForkJoinPool` and submit a new task for each row (or column) of the image. Once one row has been converted to grayscale, the current thread can work on another row.

This principle is implemented in the following example:

```
public class GrayscaleImageAction extends RecursiveAction {
        private static final long serialVersionUID = 1L;
        private int row;
        private BufferedImage bufferedImage;

        public GrayscaleImageAction(int row, BufferedImage bufferedImage) {
                this.row = row;
                this.bufferedImage = bufferedImage;
        }

        @Override
        protected void compute() {
                for (int column = 0; column < bufferedImage.getWidth(); column++) {
                        int rgb = bufferedImage.getRGB(column, row);
                        int r = (rgb >> 16) & 0xFF;
                        int g = (rgb >> 8) & 0xFF;
                        int b = (rgb & 0xFF);
                        int gray = (int) (0.2126 * (float) r + 0.7152 * (float) g + 0.0722 ↩
                            * (float) b);
                        gray = (gray << 16) + (gray << 8) + gray;
                        bufferedImage.setRGB(column, row, gray);
                }
        }

        public static void main(String[] args) throws IOException {
                ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime(). ↩
                    availableProcessors());
                BufferedImage bufferedImage = ImageIO.read(new File(args[0]));
                for (int row = 0; row < bufferedImage.getHeight(); row++) {
                        GrayscaleImageAction action = new GrayscaleImageAction(row, ↩
                            bufferedImage);
                        pool.execute(action);
                }
                pool.shutdown();
```

```
                ImageIO.write(bufferedImage, "jpg", new File(args[1]));
        }
}
```

Within our main() method we read the image using Java's `ImageIO` class. The returned instance of `BufferedImage` has all the methods we need. We can query the number of rows and columns and retrieve and set the RGB value for each pixel. So all we do is to iterate over all rows and submit a new `GrayscaleImageAction` to our `ForkJoinPool`. The latter has received a hint about the available processors as a parameter to his constructor.

The `ForkJoinPool` now starts the tasks asynchronously by invoking their `compute()` method. In this method we iterate over each row and update the corresponding RGB values by its grayscale value. After having submitted all tasks to the pool we wait in the main thread for the shutdown of the whole pool and then write the updated `BufferedImage` back to disc by using the `ImageIO.write()` method.

Astonishingly we need only a few lines of code more than we would have needed without using the available processors. This again shows how much work we can save by using the available resources of the `java.util.concurrent` package.

The `ForkJoinPool` offers three different methods for submitting a task:

- `execute(ForkJoinTask)`: This methods executes the given task asynchronously. It has no return value.
- `invoke(ForkJoinTask)`: This methods awaits the tasks return value.
- `submit(ForkJoinTask)`: This methods executes the given task asynchronously. It returns a reference to the task itself. Hence the task reference can be used to query the result (as it implements the interface `Future`).

With this knowledge it is clear why we have submitted the `GrayscaleImageAction` above using the method execute(). If we would have used `invoke()` instead, the main thread would have waited for the task to finish and we would not have utilized the available level of parallelism.

We find the same differences when we take a closer look at the ForkJoinTask-API:

- `ForkJoinTask.fork()`: The `ForkJoinTask` is executed asynchronously. It has no return value.
- `ForkJoinTask.invoke()`: Immediately executes the `ForkJoinTask` and returns the result after completion.

5.2.3 ForkJoinPool and ExecutorService

Now that we know the `ExecutorService` and the `ForkJoinPool`, you may ask yourself why we should use the `ForkJoinPool` and not the `ExecutorService`. The difference between both is not that big. Both have `execute()` and `submit()` methods and take either instances of some common interface like `Runnable`, `Callable`, `RecursiveAction` or `RecursiveTask`.

To understand the difference a little bit better, let's try to implement the `FindMin` class from above using an `ExecutorService`:

```
public class FindMinTask implements Callable<Integer> {
        private int[] numbers;
        private int startIndex;
        private int endIndex;
        private ExecutorService executorService;

        public FindMinTask(ExecutorService executorService, int[] numbers, int startIndex,  ↩
            int endIndex) {
                this.executorService = executorService;
                this.numbers = numbers;
                this.startIndex = startIndex;
                this.endIndex = endIndex;
        }

        public Integer call() throws Exception {
```

```
                int sliceLength = (endIndex - startIndex) + 1;
                if (sliceLength > 2) {
                        FindMinTask lowerFindMin = new FindMinTask(executorService, numbers ↩
                            , startIndex, startIndex + (sliceLength / 2) - 1);
                        Future<Integer> futureLowerFindMin = executorService.submit( ↩
                            lowerFindMin);
                        FindMinTask upperFindMin = new FindMinTask(executorService, numbers ↩
                            , startIndex + (sliceLength / 2), endIndex);
                        Future<Integer> futureUpperFindMin = executorService.submit( ↩
                            upperFindMin);
                        return Math.min(futureLowerFindMin.get(), futureUpperFindMin.get()) ↩
                            ;
                } else {
                        return Math.min(numbers[startIndex], numbers[endIndex]);
                }
        }

        public static void main(String[] args) throws InterruptedException,  ↩
            ExecutionException {
                int[] numbers = new int[100];
                Random random = new Random(System.currentTimeMillis());
                for (int i = 0; i < numbers.length; i++) {
                        numbers[i] = random.nextInt(100);
                }
                ExecutorService executorService = Executors.newFixedThreadPool(64);
                Future<Integer> futureResult = executorService.submit(new FindMinTask( ↩
                    executorService, numbers, 0, numbers.length-1));
                System.out.println(futureResult.get());
                executorService.shutdown();
        }
}
```

The code looks very similar, expect the fact that we `submit()` our tasks to the `ExecutorService` and then use the returned instance of `Future` to `wait()` for the result. The main difference between both implementations can be found at the point where the thread pool is constructed. In the example above, we create a fixed thread pool with 64(!) threads. Why did I choose such a big number? The reason here is, that calling `get()` for each returned `Future` block the current thread until the result is available. If we would only provide as many threads to the pool as we have CPUs available, the program would run out of resources and hang indefinitely.

The `ForkJoinPool` implements the already mentioned work-stealing strategy, i.e. every time a running thread has to wait for some result; the thread removes the current task from the work queue and executes some other task ready to run. This way the current thread is not blocked and can be used to execute other tasks. Once the result for the originally suspended task has been computed the task gets executed again and the join() method returns the result. This is an important difference to the normal `ExecutorService` where you would have to block the current thread while waiting for a result.

Chapter 6

Testing Concurrent Applications

6.1 Introduction

This article provides an introduction into the testing of multi-threaded applications. We implement a simple blocking queue and test its blocking behavior as well as its behavior and performance under stress test conditions. Finally we shed some light on available frameworks for unit testing of multi-threaded classes.

6.2 SimpleBlockingQueue

In this section we are going to implement a very basic and simple blocking `Queue`. This queue should do nothing more than hold the elements we have put into it and give them back when calling `get()`. The `get()` should block until a new element is available.

It is clear that the `java.util.concurrent` package already provides such functionality and that there is no need to implement this again, but for demonstration purposes we do it here in order to show how to test such a class.

As backing data structure for our queue we choose a standard `LinkedList` from the `java.util` package. This list is not synchronized and call its `get()` method does not block. Hence we have to synchronized access to the list and we have to add the blocking functionality. The latter can be implemented with a simple `while()` loop that calls the `wait()` method on the list, when the queue is empty. If the queue is not empty, it returns the first element:

```
public class SimpleBlockingQueue<T> {
        private List<T> queue = new LinkedList<T>();

        public int getSize() {
                synchronized(queue) {
                        return queue.size();
                }
        }

        public void put(T obj) {
                synchronized(queue) {
                        queue.add(obj);
                        queue.notify();
                }
        }

        public T get() throws InterruptedException  {
                while(true) {
                        synchronized(queue) {
                                if(queue.isEmpty()) {
                                        queue.wait();
```

```
                                } else {
                                        return queue.remove(0);
                                }
                        }
                }
        }
}
```

6.2.1 Test blocking operations

Although this implementation is very simple, it is not that easy to test all the functionality, especially the blocking feature. When we just call `get()` on an empty queue, the current thread is blocked until another threads inserts a new item into the queue. This means that we need at least two different threads in our unit test. While the one thread blocks, the other thread waits for some specific time. If during this time the other thread does not execute further code, we can assume that the blocking feature is working. One way to check that the blocking thread is not executing any further code is the addition of some boolean flags that are set, when either an exception was thrown or the line after the `get()` call was executed:

```
        private static class BlockingThread extends Thread {
                private SimpleBlockingQueue queue;
                private boolean wasInterrupted = false;
                private boolean reachedAfterGet = false;
                private boolean throwableThrown;

                public BlockingThread(SimpleBlockingQueue queue) {
                        this.queue = queue;
                }

                public void run() {
                        try {
                                try {
                                        queue.get();
                                } catch (InterruptedException e) {
                                        wasInterrupted = true;
                                }
                                reachedAfterGet = true;
                        } catch (Throwable t) {
                                throwableThrown = true;
                        }
                }

                public boolean isWasInterrupted() {
                        return wasInterrupted;
                }

                public boolean isReachedAfterGet() {
                        return reachedAfterGet;
                }

                public boolean isThrowableThrown() {
                        return throwableThrown;
                }
        }
```

The flag `wasInterrupted` indicates whether the blocking thread was interrupted, the flag `reachedAfterGet` shows that the line after the get has been executed and finally the `throwableThrown` would tell us that any kind of `Throwable` was thrown. With the getter methods for these flags we can now write a unit test, that first creates an empty queue, starts our `BlockingThread`, waits for some time and then inserts a new element into the queue.

```
        @Test
        public void testPutOnEmptyQueueBlocks() throws InterruptedException {
```

```
            final SimpleBlockingQueue queue = new SimpleBlockingQueue();
            BlockingThread blockingThread = new BlockingThread(queue);
            blockingThread.start();
            Thread.sleep(5000);
            assertThat(blockingThread.isReachedAfterGet(), is(false));
            assertThat(blockingThread.isWasInterrupted(), is(false));
            assertThat(blockingThread.isThrowableThrown(), is(false));
            queue.put(new Object());
            Thread.sleep(1000);
            assertThat(blockingThread.isReachedAfterGet(), is(true));
            assertThat(blockingThread.isWasInterrupted(), is(false));
            assertThat(blockingThread.isThrowableThrown(), is(false));
            blockingThread.join();
    }
```

Before the insertion all flags should be false. If this is the case, we put a new element into the queue and check that the flag `reachedAfterGet` is set to true. All other flags should still be false. Finally we can `join()` the `blockingThread`.

6.2.2 Testing for correctness

The previous test has just shown how to test a blocking operation. Even more interesting is the real multi-threading test case, where we have more than one thread adding elements to the queue and a bunch of worker threads that retrieve these values from the queue. Basically this means to create a few producer threads that insert new elements into the queue and to setup a bunch of worker threads that call `get()`.

But how do we know that the worker threads have got exactly the same elements from the queue, which the producer threads have inserted before? One possible solution would be to have a second queue where we add and remove the elements based on some unique id (e.g. a UUID). But as we are in a multi-threaded environment, we also have to synchronize access to this second queue and the creation of some unique ID also enforces some kind of synchronization.

A better solution would be some mathematical means that can be implemented without any additional synchronization. The easiest means is here to use integer values as queue elements, which are retrieved from a thread local random generator. The special class `ThreadLocalRandom` manages a random generator for each thread; hence we don't have any synchronization overhead. While the producer threads compute the sum over the elements inserted by them, the worker threads also compute their local sum. At the end sum over all producer threads is compared to the sum of all consumer threads. If it is the same, we can assume that with high probability we have retrieved all tasks that have been inserted before.

The following unit test implements these ideas by submitting the consumer and producer threads as tasks to a fixed thread pool:

```
    @Test
    public void testParallelInsertionAndConsumption() throws InterruptedException,  ↩
        ExecutionException {
            final SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer ↩
                >();
            ExecutorService threadPool = Executors.newFixedThreadPool(NUM_THREADS);
            final CountDownLatch latch = new CountDownLatch(NUM_THREADS);
            List<Future<Integer>> futuresPut = new ArrayList<Future<Integer>>();
            for (int i = 0; i < 3; i++) {
                    Future<Integer> submit = threadPool.submit(new Callable<Integer>()  ↩
                        {
                            public Integer call() {
                                    int sum = 0;
                                    for (int i = 0; i < 1000; i++) {
                                            int nextInt = ThreadLocalRandom.current(). ↩
                                                nextInt(100);
                                            queue.put(nextInt);
                                            sum += nextInt;
                                    }
                                    latch.countDown();
                                    return sum;
                            }
```

```
                        });
                        futuresPut.add(submit);
                }
                List<Future<Integer>> futuresGet = new ArrayList<Future<Integer>>();
                for (int i = 0; i < 3; i++) {
                        Future<Integer> submit = threadPool.submit(new Callable<Integer>()  ↩
                            {
                                public Integer call() {
                                        int count = 0;
                                        try {
                                                for (int i = 0; i < 1000; i++) {
                                                        Integer got = queue.get();
                                                        count += got;
                                                }
                                        } catch (InterruptedException e) {

                                        }
                                        latch.countDown();
                                        return count;
                                }
                        });
                        futuresGet.add(submit);
                }
                latch.await();
                int sumPut = 0;
                for (Future<Integer> future : futuresPut) {
                        sumPut += future.get();
                }
                int sumGet = 0;
                for (Future<Integer> future : futuresGet) {
                        sumGet += future.get();
                }
                assertThat(sumPut, is(sumGet));
        }, is(sumGet));
        }
```

We use a `CountDownLatch` in order to wait until all threads have finished. Finally we can compute the sum over all submitted and retrieved integers and assert that both are equal.

The order in which the different threads are executed is hard to predict. It depends on many dynamic factors like interrupts handled by the operating system and how the scheduler chooses the next thread to execute. In order to achieve more context switches, one can call the method `Thread.yield()`. This gives the scheduler a hint that the current thread is willing to yield the CPU in favor of another thread. As the javadoc states, this is only a hint, i.e. the JVM can completely ignore this hint and execute the current thread further. But for testing purposes one may use this method to introduce more context switches and therefore provoke race conditions, etc.

6.2.3 Testing performance

Another aspect next to the correct behavior of a class is its performance. In many real world applications performance is an important requirement and therefore has to be tested.

We can utilize an `ExecutorService` to setup a varying number of threads. Each thread is supposed to insert an element into our queue and take it afterwards from it. Within an outer loop we increment the number of threads in order to see how the number of threads impacts the throughput.

```
        @Test
        public void testPerformance() throws InterruptedException {
                for (int numThreads = 1; numThreads < THREADS_MAX; numThreads++) {
                        long startMillis = System.currentTimeMillis();
                        final SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue< ↩
                            Integer>();
```

```
                        ExecutorService threadPool = Executors.newFixedThreadPool( ↩
                            numThreads);
                        for (int i = 0; i < numThreads; i++) {
                                threadPool.submit(new Runnable() {
                                        public void run() {
                                                for (long i = 0; i < ITERATIONS; i++) {
                                                        int nextInt = ThreadLocalRandom. ↩
                                                            current().nextInt(100);
                                                        try {
                                                                queue.put(nextInt);
                                                                nextInt = queue.get();
                                                        } catch (InterruptedException e) {
                                                                e.printStackTrace();
                                                        }
                                                }
                                        }
                                });
                        }
                        threadPool.shutdown();
                        threadPool.awaitTermination(5, TimeUnit.MINUTES);
                        long totalMillis = System.currentTimeMillis() - startMillis;
                        double throughput = (double)(numThreads * ITERATIONS * 2) / (double ↩
                            ) totalMillis;
                        System.out.println(String.format("%s with %d threads: %dms ( ↩
                            throughput: %.1f ops/s)", LinkedBlockingQueue.class. ↩
                            getSimpleName(), numThreads, totalMillis, throughput));
                }
        }
```

To get a feeling on how our simple queue implementation performs, we can compare it to an implementation from the JDK. A candidate for this is the `LinkedBlockingQueue`. Its two methods `put()` and `take()` work similar to our implementation, expect the circumstance that `LinkedBlockingQueue` is optionally bounded and therefore has to keep track of the number of inserted elements and lets the current thread sleep if the queue is full. This functionality needs additional bookkeeping and checking on insert operations. On the other hand the JDK implementation does not use synchronized blocks and has been implemented with tedious performance measurements.

When we implement the same test case as above using the `LinkedBlockingQueue`, we get the following output for both test cases:

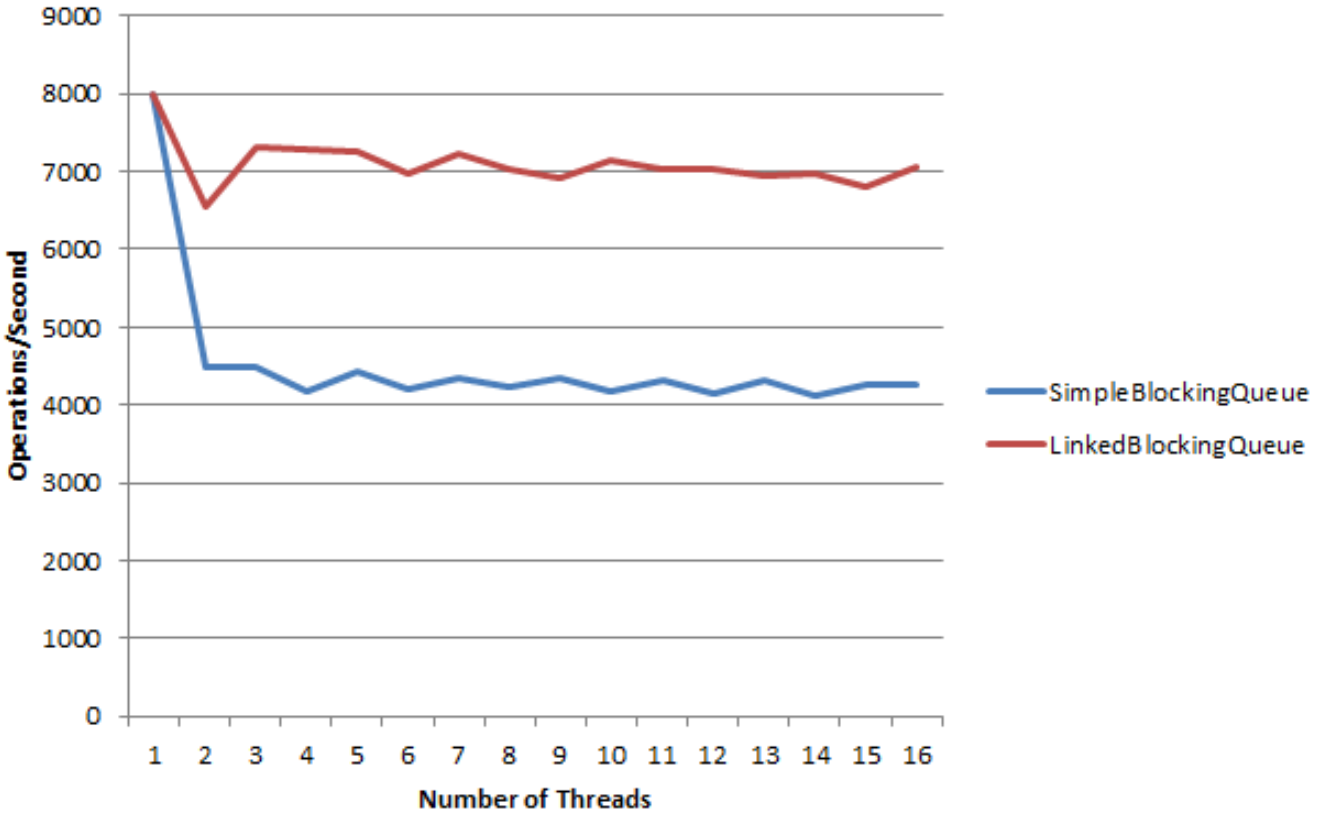

Figure 6.1: screenshot

The figure clearly shows that the throughput rate, i.e. the number of operations performed per time unit, is nearly twice as good for the `LinkedBlockingQueue` implementation. But it also shows that with only one thread both implementations make about the same number of operations per second. Adding more threads improves the throughput, although the curve converges very soon against its saturation value. Adding more threads does not improve the performance of the application any more.

6.3 Testing frameworks

Instead of writing your own framework for implementing multi-threaded test cases for your application, you can have look at the available test framework. This section puts a light on two of them: JMock and Grobo Utils.

6.3.1 JMock

For stress testing purposes the mocking framework JMock provides the class Blitzer. This class implements functionality similar to what we have done in section "Testing for correctness" as it internally set ups a `ThreadPool` to which tasks are submitted that execute some specific action. You provide the number of tasks/actions to perform as well as the number of threads to the constructor:

```
Blitzer blitzer = new Blitzer(25000, 6);
```

This instance has a method `blitz()` to which you just provide an implementation of the `Runnable` interface:

```
        @Test
        public void stressTest() throws InterruptedException {
                final SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer ↩
                    >();
```

```
                blitzer.blitz(new Runnable() {
                        public void run() {
                                try {
                                        queue.put(42);
                                        queue.get();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                });
                assertThat(queue.getSize(), is(0));
        }
```

The `Blitzer` class therefore makes the implementation of stress tests even simpler than it is with an `ExecutorService`.

6.3.2 Grobo Utils

Grobo Utils is a framework providing support for testing multi-threaded applications. The ideas behind the framework are described in article.

Similar to the previous example we have the class `MultiThreadedTestRunner` that internally constructs a thread pool and executes a given number of Runnable implementations as separate threads. The `Runnable` instances have to implement a special interface called `TestRunnable`. It is worth mentioning that its only method `runTest()` throws an exception. This way, exceptions thrown within the threads have an influence on the test result. This is not the case when we use a normal `ExecutorService`. Here the tasks have to implement `Runnable` and its only method `run()` does not throw any exception. Exceptions thrown within these tasks are swallowed and don't break the test.

After having constructed the `MultiThreadedTestRunner` we can call its `runTestRunnables()` method and provide the number of milliseconds it should wait until the test should fail. Finally the `assertThat()` call verifies that the queue is empty again, as all test threads have removed the previously added element.

```
public class SimpleBlockingQueueGroboUtilTest {

        private static class MyTestRunnable extends TestRunnable {
                private SimpleBlockingQueue<Integer> queue;

                public MyTestRunnable(SimpleBlockingQueue<Integer> queue) {
                        this.queue = queue;
                }

                @Override
                public void runTest() throws Throwable {
                        for (int i = 0; i < 1000000; i++) {
                                this.queue.put(42);
                                this.queue.get();
                        }
                }
        }

        @Test
        public void stressTest() throws Throwable {
                SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer>();
                TestRunnable[] testRunnables = new TestRunnable[6];
                for (int i = 0; i < testRunnables.length; i++) {
                        testRunnables[i] = new MyTestRunnable(queue);
                }
                MultiThreadedTestRunner mttr = new MultiThreadedTestRunner(testRunnables);
                mttr.runTestRunnables(2 * 60 * 1000);
                assertThat(queue.getSize(), is(0));
        }
}
```