Java Advanced – lesson 6 – Concurrency

Wim Wiltenburg

(willem.wiltenburg@inholland.nl)

Objectives

- To have an understanding of the concurrency mechanisms in Java
- To be able to create and manage threads
- To have insight into the Executor service
- To have an understanding if synchronizing atomic operations
- To know about “Stuff that can go wrong”

Concurrency definitions

- Thread: The smallest amount of execution that can be scheduled by the Operating System.
- Task: single unit of work performed by a thread
- Process: A group of thread operating in the same **shared environment**
- Shared environment: threads in the same process share memory space and can communicate directly with eachother
- There are system threads (jvm) and user defined threads (we program)

Priority

Concurrency is possible by dividing up CPU time into slices. One thread is executing, while others wait, until it's their time to execute.

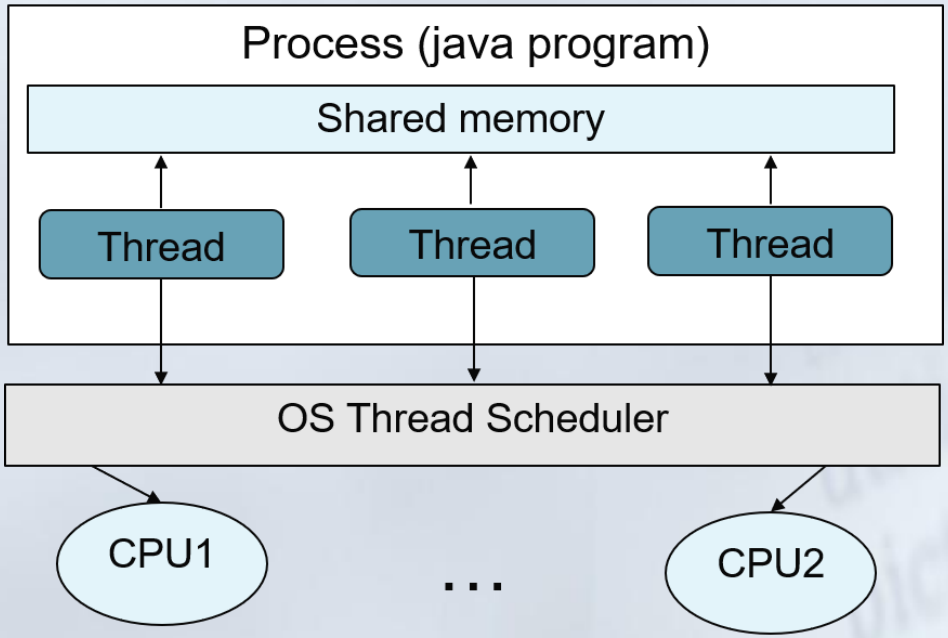
If all threads have the same priority, time would be divided equally: if 10 threads are defined with the same priority, then all threads get 100ms of every second.

We can give a higher or lower priority depending on the priority, and that priority is defined as an integer:

- Thread.MIN_PRIORITY = 1
- Thread.NORM_PRIORITY = 5
- Thread.MAX_PRIORITY = 10

Context switch

Whenever a thread gets a turn, the system does a **context switch**. So when the system switches from thread T1 to thread T2, the state (context) of T1 gets stored, the state of T2 gets loaded, and the CPU executes T2.



Creating a thread

There are two ways of creating a thread:

- Extend the Thread class and override the run() method.
- Implement the Runnable class and implement the run() method.
- Threads are started not by calling the run() method, but by calling start(). The run() method will run, but not in a separate thread
- The JVM thread always has name: main

In action


```
class ThreadClass extends Thread {
    public static void main(String[] args) {
        System.out.println("This is the main thread: " + Thread.currentThread().getName());
        ThreadClass tc = new ThreadClass();
        tc.start();
    }

    @Override
    public void run() {
        System.out.println("This thread has name: " + Thread.currentThread().getName());
    }
}
```

Result:
This is the main thread: main
This thread has name: Thread-0

```
class ThreadClass implements Runnable {
    public static void main(String[] args) {
        System.out.println("This is the main thread: " + Thread.currentThread().getName());
        ThreadClass tc = new ThreadClass();
        new Thread(tc).start(); // wrap it in a thread
    }

    @Override
    public void run() {
        System.out.println("This thread has name: " + Thread.currentThread().getName());
    }
}
```

Result:
This is the main thread: main
This thread has name: Thread-0

Runnable interface

Runnable is a functional interface with only one abstract method: run()

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

That means it can be instantiated as a lambda expression:

```
class ThreadClass {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("This thread has name: "
                + Thread.currentThread().getName()))
                .start();
    }
}
```

Why concurrency (and why not)

By dividing the work into smaller pieces that can be executed at the same time (by multiple processors) there are big advantages in time.

But there are drawbacks:

- Increased complexity
- Access to shared memory can give ambiguous results
- Deadlocks, livelocks and starvation of resources
- There is no guarantee about order of execution

What is the result of this


```
class Scratch {

    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
            for (int j = 0; j < 5; j++) {
                    System.out.println(Thread.currentThread().getName() + ", j = " + j);
                }}).start();

            new Thread(() -> System.out.println(Thread.currentThread().getName() + ": Hello world")).start();
        }
    }
}
```

We create two threads. One counts from 0 to 5, and the other one just prints “Hello World”.

This gets repeated three times.

Results

We can in no way predict the outcome:

```
Thread-0, j = 0
Thread-0, j = 1
Thread-0, j = 2
Thread-0, j = 3
Thread-0, j = 4
Thread-1: Hello world
Thread-4, j = 0
Thread-4, j = 1
Thread-4, j = 2
Thread-4, j = 3
Thread-4, j = 4
Thread-3: Hello world
Thread-2, j = 0
Thread-2, j = 1
Thread-2, j = 2
Thread-2, j = 3
Thread-2, j = 4
Thread-5: Hello world
```

```
Thread-0, j = 0
Thread-0, j = 1
Thread-0, j = 2
Thread-0, j = 3
Thread-0, j = 4
Thread-1: Hello world
Thread-2, j = 0
Thread-2, j = 1
Thread-2, j = 2
Thread-2, j = 3
Thread-2, j = 4
Thread-4, j = 0
Thread-4, j = 1
Thread-4, j = 2
Thread-4, j = 3
Thread-4, j = 4
Thread-3: Hello world
Thread-5: Hello world
```

```
Thread-0, j = 0
Thread-1: Hello world
Thread-0, j = 1
Thread-0, j = 2
Thread-0, j = 3
Thread-0, j = 4
Thread-2, j = 0
Thread-4, j = 0
Thread-4, j = 1
Thread-4, j = 2
Thread-4, j = 3
Thread-4, j = 4
Thread-3: Hello world
Thread-5: Hello world
Thread-2, j = 1
Thread-2, j = 2
Thread-2, j = 3
Thread-2, j = 4
```

Executor Service

- The Executor Service can create and manage threads. To submit a thread to the service, we obtain an instance of it, and then a variety of methods are at our disposal. The thread will be executed at **some point in the future** (nothing is guaranteed)
- The service can create single thread, or a pool of threads, depending on our needs
- A service is created by the Factory class Executors

Excutors factory class

A factory class is a class, that can create objects, like a chocolate factory produces chocolate. The Executors class has many methods. The most important are:

- newSingleThreadExecutor: returns an executor service that manager only a single thread. This would be like having no concurrency, except that it runs apart from main

```
service = Executors.newSingleThreadExecutor()
```

- newFixedThreadPool(int nThreads): return an executor service that manages a defined number of threads. It can be a large number, but it's good practice to limit to the number of available processor cores:

```
service = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```

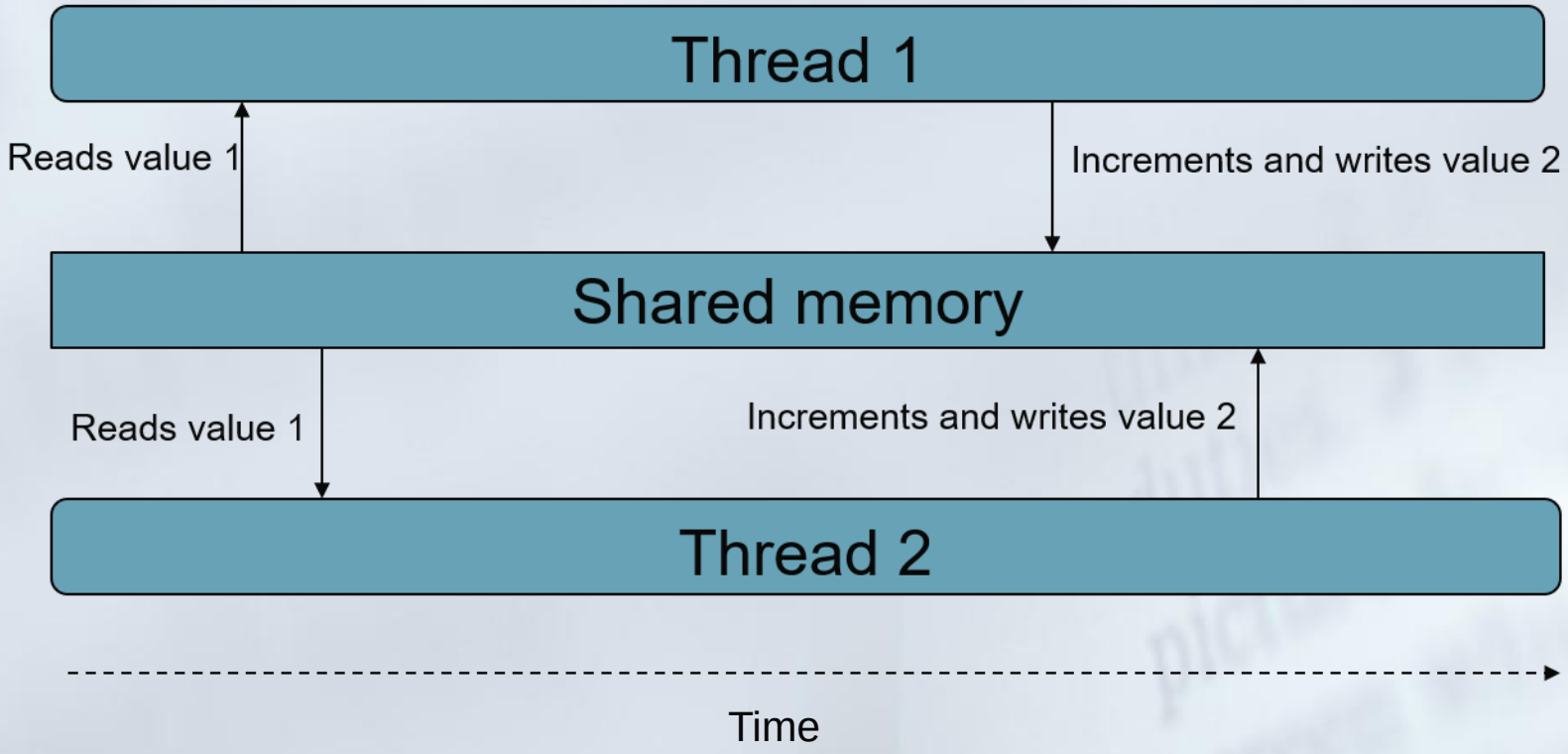
Running the single thread


```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Scratch {
     private int counter = 0;

    private void incrementAndPrint() {
        counter++;
        System.out.println(Thread.currentThread().getName() + ", " + counter);
    }

    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newSingleThreadExecutor();
            Scratch scratch = new Scratch();
            for (int i = 0; i < 10; i++) {
                service.submit(scratch::incrementAndPrint);
            }
        } finally {
            if (service != null) {
                service.shutdown();
            }
        }
    }
}
```

Result:
pool-1-thread-1, 1
pool-1-thread-1, 2
pool-1-thread-1, 3
pool-1-thread-1, 4
pool-1-thread-1, 5
pool-1-thread-1, 6
pool-1-thread-1, 7
pool-1-thread-1, 8
pool-1-thread-1, 9
pool-1-thread-1, 10

Race condition

When multiple threads access the same data, clashes can occur, when one thread processes data, that has already chaged value in another thread



Same code, different executor


```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Scratch {
     private int counter = 0;

    private void incrementAndPrint() {
        counter++;
        System.out.println(Thread.currentThread().getName() + ", " + counter);
    }

    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
            Scratch scratch = new Scratch();
            for (int i = 0; i < 10; i++) {
                service.submit(scratch::incrementAndPrint);
            }
        } finally {
            if (service != null) {
                service.shutdown();
            }
        }
    }
}
```

```
pool-1-thread-1, 2
pool-1-thread-2, 2
pool-1-thread-3, 3
pool-1-thread-2, 4
pool-1-thread-2, 5
pool-1-thread-2, 7
pool-1-thread-2, 8
pool-1-thread-3, 6
pool-1-thread-1, 9
pool-1-thread-4, 10
```

Atomic operations

In order to avoid race conditions, it's necessary to make sure that the data is read and changed, before another thread gets it. In other words we need to lock the data.

That can be arranged by synchronizing the operation, making the operation "atomic".

Atom: from Greek *atomos* 'indivisible', based on *a-* 'not' + *temnein* 'to cut'.

synchronized

To make a block of code atomic we use the "synchronized" keyword. The synchronized code needs something to lock with, and to release when it's done. That can be anything, as long as it's final.

Example:

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

private void incrementAndPrint() {
    synchronized (lock) {
        System.out.println(Thread.currentThread().getName() + ", " + (counter++));
    }
}
```

Here it's an object of type Object, but it can also be a String, the current class (plus .class) or anything…

Synchronized methods

Another way of synchronizing is using the synchronized keyword on the method itself:

```
private synchronized void incrementAndPrint() {
    {
        System.out.println(Thread.currentThread().getName() + ", " + (counter++));
    }
}
```

Stuff that can go wrong: Deadlock

The code


```
class Scratch {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {

        new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lock1) {
                    System.out.println(Thread.currentThread().getName() + ", I have lock 1");
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                    }
                    System.out.println(Thread.currentThread().getName() + ", waiting for lock 2");
                    synchronized (lock2) {
                        System.out.println(Thread.currentThread().getName() + ", holding lock 1 & 2");
                    }
                }
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lock2) {
                    System.out.println(Thread.currentThread().getName() + ", I have lock 2");
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException ie) {
                    }
                    System.out.println(Thread.currentThread().getName() + ", waiting for lock 1");
                    synchronized (lock1) {
                        System.out.println(Thread.currentThread().getName() + ", holding lock 1 & 2");
                    }
                }
            }
        }).start();
    }
}
```

Stuff that can go wrong: Deadlock

When two threads are blocked forever, because they are waiting for eachother to release a lock, we speak of Deadlock

Solution:Change the order of the lock:

```
Thread-0, I have lock 1
Thread-1, I have lock 2
Thread-0, waiting for lock 2
Thread-1, waiting for lock 1
```

```
synchronized (lock1) {
    System.out.println(Thread.currentThread().getName() + ", I have lock 1");
    try {
        Thread.sleep(10);
    } catch (InterruptedException ie) {
    }
    System.out.println(Thread.currentThread().getName() + ", waiting for lock 2");
    synchronized (lock2) {
        System.out.println(Thread.currentThread().getName() + ", holding lock 1 & 2");
    }
}
```

```
Thread-0, I have lock 1
Thread-0, waiting for lock 2
Thread-0, holding lock 1 & 2
Thread-1, I have lock 1
Thread-1, waiting for lock 2
Thread-1, holding lock 1 & 2
```

Other stuff that can go wrong

Starvation: When a thread cannot get access to a shared resource, because other threads are taking priority (greedy threads), we speak of starvation

Livelock: When two threads are busy responding to eachother, but not making progress, we speak of livelock. These threads are not blocked.

Threadsafe collections

An ArrayList is not thread safe:

```
class Scratch {
    static List<String> list = new ArrayList<>();

    public static void main(String[] args) {

        new Thread(new Runnable() {
            @Override
            public void run() {
                list.add("3");
               list.add("4");
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                for (String s: list) {
                    list.remove(s);
                }

            }
        }).start();
        System.out.println(list);
    }
}
```

```
Exception in thread "Thread-1" java.util.ConcurrentModificationException
                    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
                    at java.util.ArrayList$Itr.next(ArrayList.java:859)
                    at Scratch$2.run(scratch_5.java:21)
                    at java.lang.Thread.run(Thread.java:748)
```

Use:

```
static List<String> list = Collections.synchronizedList(new ArrayList<>());
```

Threadsafe collections:

Vector – compares to ArrayList

Hashtable – old version of HashMap

ConcurrentHashMap – same as HashMap, but threadsafe. *Always use this, even though it's not needed.* The overhead is so small, that it always pays

CopyOnWriteArrayList

CopyOnWriteSet

} Out of scope, but when writing to the collection, it creates a copy, so the data is always safe

Home work: none