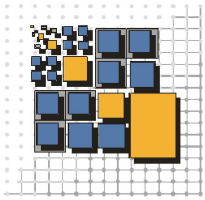

Otto-Friedrich-University Bamberg

Distributed Systems Group

Lecture Notes

Concurrency Topics

Version 1.0
(Summer term 2020)

Preface

Concurrency is a topic which is hard to grasp since the effects of multi-threaded code can not be easily deduced by a single threaded human brain :)

Programming is an easy task, but understanding written code after a few weeks or months is hard. Finding bugs is even harder. Doing this for concurrent code is - in many cases - nearly impossible. Brian Goetz, author of *Java Concurrency in Pratice*, gives good advice on how to program and deal with concurreny:

> *Sometimes abstraction and encapsulation are at odds with performance — although not nearly as often as many developers believe — but it is always a good practice first to make your code right, and then make it fast.*
>
> *It is far easier to design a class to be thread-safe than to retrofit it for thread safety later.*

These lecture notes are an introduction into concurrency programming in Java and summarize the main topics of our module DSG-PKS-B besides the message passing concepts and frameworks. Understanding the concepts and functionality of different elements in the Java API helps to understand synchronous and asynchronous mechanisms discussed in our DSG master courses and all other computer science courses. This document includes all the necessary parts for a solid understanding of concurrency programming in Java but does not include all the nit picky details presented in some great textbooks like the following ones we highly recommend to read:

- Brian Goetz with others: Java Concurrency in Practice, Addison-Wesley, 2006. ISBN 978-0321349606 (or other editions)
- Joshua Bloch: Effective Java: Third Edition, Addison-Wesley, 2018. ISBN 978-0-13-468599-1

We highly recommend you to read this document carefully and invest some time to get your hands dirty. Without programming, merely reading this document is useless.

Since concurrency is doing stuff in parallel and NOT sequentially, this document is also written for parallel use and doesn't have to be read starting with Section 1 and ending with the last page.

Sections 1 and 2 are more or less fundamentals and explain why we are teaching and have to care about concurrency.

Section 3 discusses the two basic classes for concurrency within the Java API: *Thread* and *Runnable*. All the low level aspects, Java monitors, *wait/notify*, critical sections, locking, visibility and deadlocks are discussed in Section 4, whereas we climb the high level concepts mountain in Section 5 to get rid of the error-prone low level stuff.

Where we thought it might be beneficial, we included **Rule of Thumbs**, **Examples** and **Exercises** to foster the written material. The source code of all examples and exercise skeletons can be found on GitHub: `https://github.com/johannes-manner/ConcurrencyTopics`. If you want to test your implemented code, we prepared some CodeRunner tasks in the specific VC course.

Johannes Manner
Sebastian Böhm

Bamberg, May 2020

Contents

1 Fundamentals

1.1 Call By Value

In order to understand the problems which come along with concurrency programming, understanding the difference of memory handling for objects and primitives is one cornerstone. Java organizes the memory access like C, but *References* (in C *Pointers*) are hidden in most cases. This is the case as construction (done via special constructor methods) and destruction (done by the garbage collection) are managed by the runtime environment. ***Primitives*** and ***objects*** are to some extent handled equally in Java.

As a rule of thumb:
Primitives are *Called By Value*. They are always stored inside the Stack Memory. Different methods have different Stack Spaces. So changes to the values in another Stack Space are not present in the former location.
Objects are also *Called By Value*, but many developers claim that Java uses a Call By Reference semantic. The explanation is as follows: Objects are stored inside the Heap. The Reference/Pointer is stored on the Stack. So changes to the values of the object by another method are present to all References which identify the same object. The object's Reference is passed By Value, so the heap address of the object is copied and passed as a parameter to the called method.

There is **NO** Call By Reference semantic in Java. For an extensive discussion, we recommend a stackoverflow discussion[1].

For a discussion about the differences of Stack and Heap, read Section 1.2.1.

But what is the difference in memory access paths in the following snippets Listing 1 and Listing 2?

Listing 1: Call By Value Example

```
1  public static void main(String[] args) {
2    int x = 1;
3    int y = 2;
4    System.out.println("x=" + x + ";y=" + y ); // x=1;y=2
5    modify(x,y);
6    System.out.println("x=" + x + ";y=" + y ); // x=1;y=2
7  }
8  private static void modify(int x, int y) {
9    x = 5;
10   y = 10;
11   System.out.println("x=" + x + ";y=" + y ); // x=5;y=10
12   return;
13 }
```

[1] https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

Listing 2: Another Call By Value Example with References

```
1 public static void main(String[] args) {
2   Student s = new Student();
3   System.out.println("name=" + s.getName()); // name=
4   modifyStudent(s);
5   System.out.println("name=" + s.getName()); // name=Alex
6 }
7 private static void modifyStudent(Student student) {
8   student.setName("Alex");
9 }
```

Listing 1 shows an example of a Call By Value parameter assignment with primitive values. In line 5 the method is invoked with *modify(1,2)* so the print statement in line 4 results in *x=1;y=2*. Since x and y are called By Value (they are primitives), the values of x and y are copied to new Stack variables x and y within *modify* and changed only within *modify*. Now there are two sets of x and y. One in the *main* method scope, the other in the *modify* method scope. The print in *modify* in line 11 results in *x=5;y=10*. The changes to x and y are not recognized by the *main* method. The third print result (from an execution point of view) in line 6, therefore, prints *x=1;y=2*.

The second listing shows a Call By Value example with references where an object (Student) is instantiated in line 2. We assume that a name is by default an empty string, so the print result in line 3 is *name=*. The *modifyStudent* method invoked in line 4 receives s as a parameter. Here s is passed by Value where the reference of the object s is copied. What's happening technically is that the memory address of Student s is copied and there are now two stack variables in the *main* and *modifyStudent* method scope which contain the same (!) reference to the Student s (the object is located in the heap). The reference of the object is copied but the destination (when following the reference) is the same.
Hint: The variable name s *is abbreviated due to the length of the line,* student *might be a better variable name :)*

If you want to read more about this topic, we highly recommend an article by Hussein Terek at ProgrammerGate[2].

Example:
You can find an example for primitives and objects, which is only a single concept (Call by Value) when assessing it in detail, under `de.uniba.dsg.concurrency.examples.fundamentals.CallByValueObject` and `Value`.

[2] `https://www.programmergate.com/java-pass-reference-pass-value/`

1.2 Memory Model

1.2.1 Heap and Stack

To deepen your knowledge about primitives, objects, Stack and Heap, we recommend using the python tutor[3] and investigate some programs or the examples and exercises we provide via the Virtual Campus.

A summary of the comparison of Stack and Heap memory stolen from Baeldung[4] can be found in Table 1 where important parts for our concurrency topic are highlighted in bold.

Table 1: Stack and Heap Memory Summary from Baeldung

Parameter	**Stack Memory**	**Heap Space**
Application	Stack is used in parts, **one at a time during execution of a thread**	The entire application uses Heap space during runtime
Size	Stack has size limits depending upon OS and is usually smaller than Heap	There is no size limit on Heap
Storage	**Stores only primitive variables and references to objects that are created in Heap Space**	**All newly created objects are stored here**
Order	It is accessed using Last-in First-out (LIFO) memory allocation system	This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation.
Life	**Stack memory only exists as long as the current method is running**	**Heap space exists as long as the application runs**
Efficiency	Comparatively much faster to allocate when compared to heap	Slower to allocate when compared to stack
Allocation/ Deallocation	**Stack Memory is automatically allocated and deallocated when a method is called and returned respectively**	**Heap space is allocated when new objects are created and deallocated by Garbage Collector when they are no longer referenced**

It is especially remarkable that stack memory is always thread-safe since only primitives are stored there and the stack memory only exists as long as the method is running!

[3] `http://pythontutor.com/visualize.html`: Configure the python tutor with *show all frames(Python)*, *render all objects on the heap (Python/Java)* and *draw pointers as arrows [default]*.

[4] `https://www.baeldung.com/java-stack-heap`: Complete article very detailed and a good overview. Summary is taken from section 5.

As a rule of thumb:
Scoping: Try to scope as restrictive as possible. This means that you should avoid class members (attributes) wherever the application does not require a state within an object. If an attribute, for example, is only used within a single method, you should think about deleting the class attribute and change the method signature so that the method gets this single parameter when executing.

In a later part of this document, we will discuss this scoping example of Listing 3 in more detail. For now you should assess and understand the differences in scoping. The first *CalculatorBefore* implementation has two class members. If we share the reference to a calculator of class *CalculatorBefore*, the two members are visible within the class for other methods, in our example getter and setter. If we have more than 1 actor, concurrent accesses can happen - as a first primer to our topic. Since the calculator is stored on the heap, all actors access the same *CalculatorBefore* object and can alter its state in an undefined order.

The second implementation of our code listing does not contain state within the *CalculatorAfter*. If more than one actor concurrently calls the add method of a single *CalculatorAfter* instance, they use the same instance but each invocation gets a separate address space on the stack and, therefore, the different invocations happen concurrently but are separated from each other due to the stack space.

Listing 3: Calculator as a Scoping Example

```
1  // before
2  class CalculatorBefore{
3    private double operand1;
4    private double operand2;
5
6    public Calculator(double operand1, double operand2) {
7      this.operand1 = operand1;
8      this.operand2 = operand2;
9    }
10
11   public double add(){
12     return operand1 + operand2;
13   }
14
15   // assume getter and setter
16 }
17
18 // after
19 class CalculatorAfter{
20   public double add(double operand1, double operand2){
21     return operand1 + operand2;
22   }
23 }
```

Exercise:
We created a stack/heap exercise to play a bit football. There is no source code provided for this exercise, you have to look in your VC course for the Code Runner Task *StackHeap*.

1.2.2 Java Memory Model - JSR 133

"Java Specification Requests (JSRs) are the actual descriptions of proposed and final specifications for the Java platform."[5] The JSR133 (title: Java Memory Model and Thread Specification Revision) "describes the semantics of threads, locks, volatile variables and data races. This includes what has been referred to as the Java memory model."[6]

The Java Memory model defines rules based on the Java Language Specification which have to be guaranteed by a valid JVM implementation. These so-called happens-before rules are listed in the following[7] and can be relied upon independent of the platform.

- **Program order rule** – Each action in a thread happens-before every action in that thread that comes later in the program order. (As we would assume from sequential program execution known from the main thread.)
- **Monitor lock rule** – An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock. (If your are confused now, what a monitor is, read Section 4.1.1. This specification enables the mutual exclusion of competing threads for a critical section (if you are even more confused, read Section 4.1.2).)
- **Volatile variable rule** – A write to a volatile field happens-before every subsequent read of that same field. (See Section 4.1.6 for a detailed discussion.)
- **Thread start rule** – A call to `Thread.start` on a thread happens-before every action in the started thread. (This is somehow self-explanatory.)
- **Thread termination rule** – Any action in a thread happens-before any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`. (See Section 3.3 and the Thread's methods.)
- **Interruption rule** – A thread calling `interrupt` on another thread happens-before the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`). (For interruption and how these methods interact with each other, have a look at Section 4.2.)
- **Finalizer rule** – The end of a constructor for an object happens-before the start of the finalizer for that object. (As a remark here: Calling an object's *finalize* method is deprecated since Java 9[8]. The mechanism is still valid, but other possible solutions for implementing finalization should be used.)
- **Transitivity** – If A happens-before B, and B happens-before C, then A happens-before C.

[5] Cited from the official JSR page: `https://jcp.org/en/jsr/overview`

[6] Cited from the official JSR 133 page: `https://jcp.org/en/jsr/detail?id=133`
Specification of the JSR 133 (worth but hard to read): `https://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`
An easier to read explanation to the Java Memory Model: `http://tutorials.jenkov.com/java-concurrency/java-memory-model.html`. It is probably useful, if you read the chapter 4 first, i.e. Visibility discussion (Section 4.1.6), Java Monitor (Section 4.1.1) and the basic synchronization Section 4.1.2.

[7] The happens-before enumeration is directly copied from Brian Goetz's Book "Java Concurrency in Practice". The citing page is 341. We added some comments from us in brackets.

[8] Read here for more information: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#finalize()`

Some of the rules are discussed in later sections directly or implicitly. The summary here is a first stepping stone so that you have heard some important aspects before digging deeper into the concurrency issues in Java.

1.3 Immutability

Immutability describes "the state of not changing, or being unable to be changed"[9]. For developers in different domains it is a highly important concept that an object does not change its internal state after construction. This makes concurrent programming much easier since you do not have to care about concurrent access with immutable objects. (There are some exceptions but for now we won't worry about those, for a discussion of these problems see Section 4.1.5).

For some immutability is the destination of programming[10]:

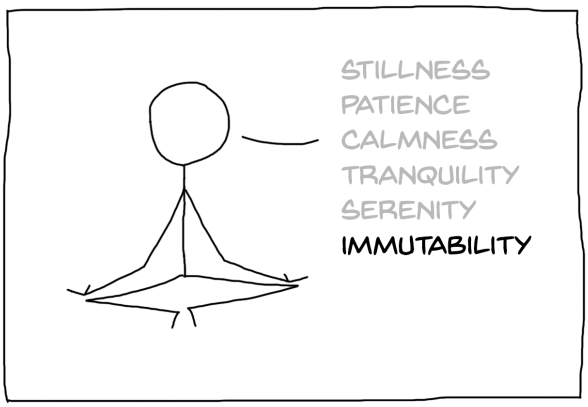

Figure 1: The final Destination for a Concurrency Programmer

As a rule of thumb:
An object is immutable if its internal state (values of the class attributes) does NOT change after construction. A class is immutable if inheritance is NOT possible.

Listing 4: Mutable Class Age

```
1 // mutable
2 class Age {
3   private int day, month, year;
4   // constructor
5   public Age(int day, int month, int year) {
6     // ...
7   }
8   // getter and setter
9 }
```

[9] https://dictionary.cambridge.org/de/worterbuch/englisch/immutability

[10] The Figure is copied from https://itnext.io/why-concept-of-immutability-is-so-damn-important-for-a-beginner-front-end-developer-8da85b565c8e.

Listing 5: Immutability Example

```
1  //@Immutable
2  final class ImmutableStudent {                                    //1
3
4    private final int id;                                           //2
5    // Strings are constant; their values cannot be changed
6    private final String name;                                      //2
7    private final Age age;                                          //2
8
9    public ImmutableStudent(int id, String name, Age age) {
10     this.name = name;
11     this.id = id;
12     this.age = new Age(age.getDay(),age.getMonth(), age.getYear()); //5(a)
13   }
14
15   // no setters                                                    //3
16   // 'normal' getters for id and name (final)
17   public Age getAge() {
18     return new Age(age.getDay(),age.getMonth(), age.getYear());   //5(b)
19   }
20   public ImmutableStudent hasMarried(String newName) {            //4
21     return new ImmutableStudent(this.id, newName, this.age);      //4
22   }
23 }
```

The following list gives you instructions on how to make a class immutable. Learn these steps by heart and if you are interested, the following link[11] explains some stuff in more detail:

1. Declare the class as final to avoid inheritance from the class since there are constellations where this could compromise immutability!
2. Declare all attributes as final! Final means that reassigning some value to a variable which has already been instantiated is not possible any more (Nice language feature!).
3. No setters! This is also not possible when doing step 2 right.
4. If you implement methods which change the state of the object, the return value of this method MUST be a new instance since the old instance remains unchanged! The returned instance then contains the changed values as in our example.
5. If the class contains a **mutable** object as a class member, the following two steps are necessary:
   (a) During construction: Copy the mutable object (in our case *age*) so that the reference/pointer to that object is encapsulated only within the current object. The reason for this copying is that the caller of the constructor also has a reference to the *age* object and can alter its state. **HINT: Make a deep copy of the mutable object! If the mutable object contains other objects, you have to copy these objects as well!! (So make a recursive deep copy :))**

[11] https://dzone.com/articles/how-to-create-an-immutable-class-in-java

(b) Copy the mutable object whenever you share the object to the outer object world. Only share a copy of the object as presented here for the getter. **HINT: Keep your internal state safe from outside accesses!**

Exercise:
Change two already implemented classes *Student* and *Group* to make them immutable. Don't add any other public methods to the source code (private methods are ok). You find the sample under `de.uniba.dsg.concurrency.exercises.immutability` and a corresponding *Code Runner Task (Code Runner is a automatic code assessment tool which provides an integration for Virtual Campus)* in your VC course.

Example:
`de.uniba.dsg.concurrency.examples.fundamentals.Immutability` contains the source code of an immutability example which is quite similar to your exercise!

2 Why Threading?

2.1 Situation

It's all about performance!

To name a few examples: Multi-Core processors, asynchronous event and data processing, background tasks etc. The reasons for concurrency - running more than 1 application, task etc. concurrently - are manyfold.

2.1.1 Models for Concurrent Programming

In general, there are two models for concurrent programming. The first one is shared memory, discussed in more detail in Section 4 and the second one is message passing (or distributed memory).

As a primer, in a shared memory scenario, two or more threads share the same memory space. In a message passing situation two or more processes exchange messages and use different memory spaces.

But what are processes and threads?

2.1.2 Processes vs. Threads

"Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process."[12]

Figure 2 shows the difference of the two concepts and highlights the message passing in an inter-process communication and the shared memory paradigm between threads. There are also situations where more processes and within the single process more threads are active than CPU cores are available. Scheduling on a process level and context switches on a thread level make this possible.

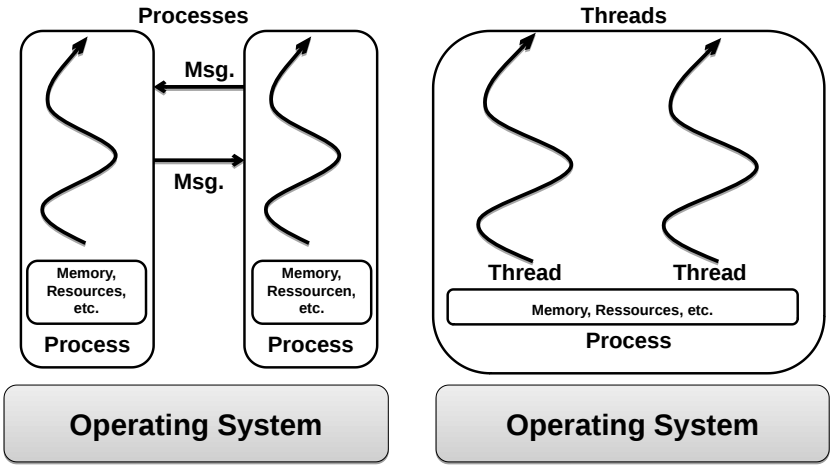

Figure 2: Difference between Processes and Threads

2.2 Problem

Especially in shared memory scenarios the value of a computation is dependent on the execution order of the program. Think about an increment of a variable *counter++*. Assume that more than one actor concurrently increments this counter.

You may probably say now: 'That's no problem. *counter++* is a single expression and therefore an atomic command'

Sadly not!

counter++ is a compound action, composed out of three atomic operations: read the value into the register, add 1 to the actual value, write the result back.

Figure 3 depicts the increments of two actors A and B. A reads the value first and sees for example 5. B subsequentially reads the value and also reads 5. A adds 1 to the counter, B adds 1 to the counter, A writes 6 to the counter, B writes also 6 to the counter. So a single update is lost (LOST UPDATE problem).

[12] `https://docs.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads?redirectedfrom=MSDN`

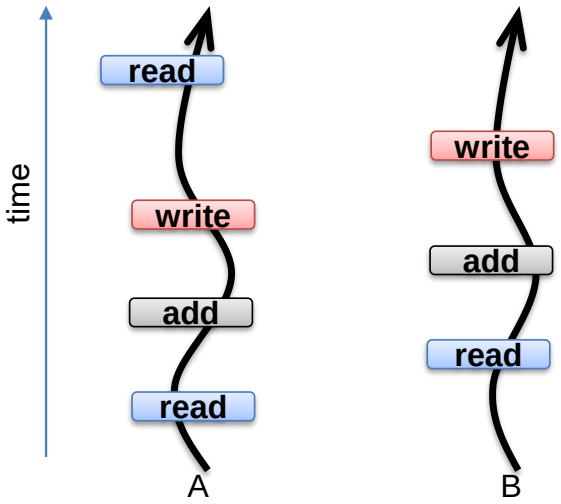

Figure 3: Two Actors A and B increment a shared Variable

2.3 Solution

As ascertained in the previous section, the problem of compound actions is that different actors can interleave and the result is corrupted to some extent. So the obvious solution for this problem is to define some set of actions as a ***Critical Section***, where only a single actor is allowed to enter this section and perform the given set of actions as if there was only a single action from a user point of view.

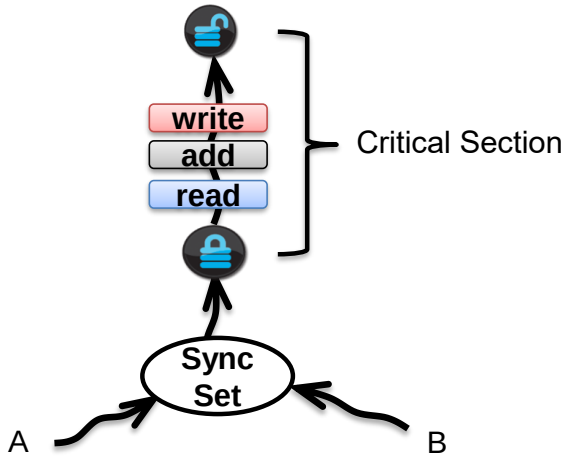

Figure 4: Two Actors A and B increment a shared Variable. Critical Section enables Atomicity from a User's Perspective.

This concept is presented in Figure 4 where the two Actors A and B want to enter this *critical section* of read, add and write. A and B first compete for the mutually exclusive right to enter the *critical section*. In our metaphor the *critical section* is a room. They enter a *SyncSet*[13], where a single Actor is randomly picked and gets the key for the room to *lock* the door after she enters the room. Then the picked actor reads, adds and writes and after finishing the critical stuff, she unlocks the door, leaves the room and the next

[13]SyncSet is introduced by us for a later usage in the Java domain. There exists no SyncSet in Java, but we need it for our metaphor later on.

actor is randomly picked (in our case the remaining Actor), gets the key, locks the door, does her stuff and leaves by unlocking the room.

So for this critical operation consisting of these 3 atomic steps, interleaving is not possible anymore since the critical section is protected by the lock.

3 Thread and Runnable

Runnable (interface, introduced with Java 1.0) and Thread (class, introduced with Java 1.0)[14] are the two main elements of the concurrency API when Java started in the 90s.

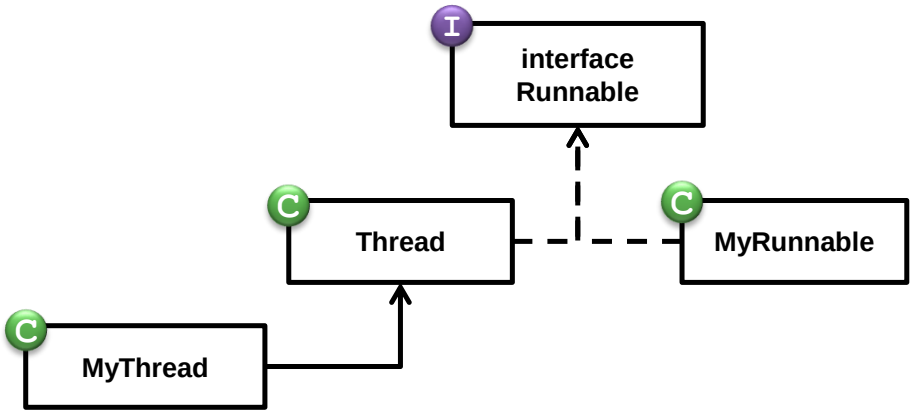

Figure 5: Runnable and Thread Class Diagram

Runnable as an interface only specifies a single method *void run()*. When implementing this interface in a class *MyRunnable* a user has to add his business logic and can extend (inherit from) other classes. An instance of this class can then be executed by a runtime environment of a programmer's choice.

As Figure 5 shows, class Thread implements Runnable and a user has to add his business logic also to the *run* method when extending a Thread with a class *MyThread*. Furthermore, some lifecycle methods are also implemented by the Thread class and therefore the user does not have to implement these methods (so he can run concurrent code quite easily). HINT: Since Java does not allow multiple extends, inheritance is no longer possible.

The code in Figure 6 produces the same output for both sides. On the left hand side, *MyRunnable* implements the *Runnable* interface. Here the thread (execution environment) and the runnable (logic) are separated. On the right hand side, *MyThread* extends *Thread* class and implements/overrides the *run* method.

[14] see the documentation for Runnable: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Runnable.html`
See the documentation for Thread: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html`

Runnable

```java
public class MyRunnable implements Runnable {
    private String name;
    private int someInt;
    public MyRunnable(String name, int someInt) {
        this.name = name;
        this.someInt = someInt;
    }
    public void run() {
        System.out.println("Thread name is "
        + name + " " + someInt);
    }
}
```

```java
public class RunnableMain {
  public static void main(String[] args) {
    MyRunnable r = new MyRunnable("My runnable", 1);
    Thread myThread = new Thread(r);
    myThread.start();
  }
}
```

Thread

```java
public class MyThread extends Thread {
    private int someInt;
    public MyThread(String name, int someInt) {
        super(name);      // attribute in superclass
        this.someInt = someInt;
    }
    public void run() {
        System.out.println("Thread name is "
        + this.getName() + " " + someInt);
    }
}
```

```java
public class ThreadMain {
  public static void main(String[] args) {
    Thread t = new MyThread("My first thread", 1);
    t.start();
  }
}
```

Figure 6: Runnable and Thread Code Comparison

As a rule of thumb:
When possible, implement a Runnable since separation of concerns is enforced. The logic (*run* method) is separated from the execution environment (*Thread*). Implementing runnables is also the *normal* way when using different APIs (see section 5.3.

3.1 Thread Methods

The most important methods of class Thread are as follows (for further information look at the JavaDocs):

- *setName(String name): void* – Sets the thread's name.
- *getName(): String* – Gets the thread's name.
- *getId(): int* – Gets a unique ID, but IDs may be reused.
- *getState(): Thread.State* – Gets the current thread state.
- *interrupt(): void* – Interrupts the thread and sets the interrupted flag to true.
- *isInterrupted(): boolean* – Checks the interrupted flag.
- *interrupted(): boolean* – Checks the interrupted flag and resets it to false.
- *start(): void* – Starts the thread and executes it (invokes the *run()* method) concurrently to the main thread and all other running threads and daemon threads.
- *join(): void* – Blocking method. Waits until the thread is finished with its computation - means when the *run()* method exits.

As shown in the excerpt of Thread's methods, there is a method which returns the state of a thread. Once a thread is terminated, there is no possibility to restart it for a second execution.

3.2 Thread States

After the thread is instantiated with *new Thread()*, it is in the status *new*. After invoking start() on the thread instance, the runtime environment starts the thread concurrently (status *runnable*) to already running threads[15]. When the thread wants to acquire a lock and waits for the monitor (due to another thread holding the monitor's lock), the thread is in state *blocked* and resumes when the lock is granted (so it can acquire the lock and access the critical section). With *wait()* and *join()*, the executing thread is in a monitor's *Wait Set* and can only be awakened by *notify(), notifyAll()* or *interrupt()*. Method *notify()* selects a random thread within the monitor's wait set, whereas *notifyAll()* awakes all threads which are competing with each other for the monitor's lock (remember all besides a single thread, are then in the state blocked :). Timed waiting is the same as waiting besides the millisecond period x.

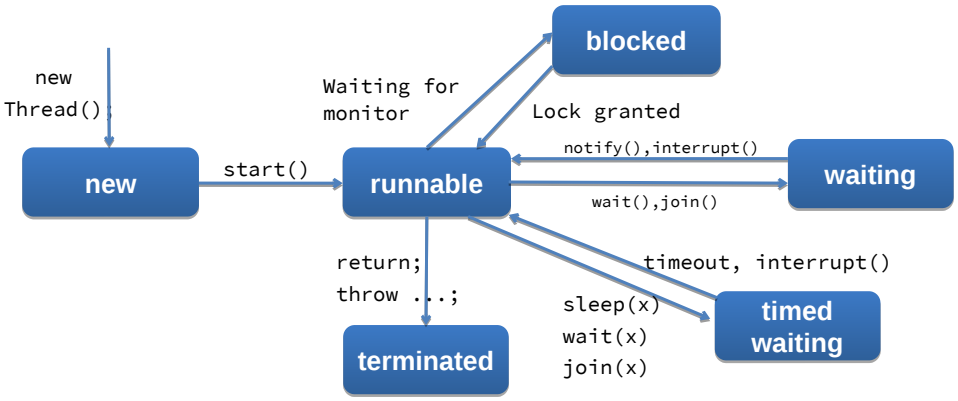

Figure 7: Thread States and their Transitions

3.3 A Running Example

This running example shows the most important methods and how they interact with each other. We will try to emphasize some common pitfalls, which are often neglected when doing thread programming.

As *rule of thumb*, we are separating the logic (Runnable) and the execution environment (Thread) from each other. We further know from the previous section that *start* starts the thread instance concurrently to the main thread and executes the *run* method of the passed Runnable. Listing 6 shows how to create and start threads:

[15]Main thread is always running until the program terminates. Some frameworks do also use daemon threads, e.g. the garbage collector (see Java Concurrency in Practice, page 165 for a description), which run also concurrently but do NOT hinder the JVM to terminate properly.

Listing 6: Create and start Threads

```
1  public static void main(String[] args) {
2
3    Runnable r = new Runnable() {
4      @Override
5      public void run() {
6        try {
7          Thread.sleep(10000);
8          printIt(") waited for a long time and terminate now!");
9        } catch (InterruptedException e) {
10         printIt(") was interrupted and terminate now!");
11       }
12     }
13   };
14
15   // thread object creation,
16   // like every other object by calling the constructor
17   Thread a = new Thread(r, "A");
18   Thread b = new Thread(r, "B");
19
20   // starting the threads
21   a.start();
22   b.start();
23
24   // exit main thread
25   printIt(") am the master and terminate now!");
26 }
27
28 private static void printIt(String s) {
29   System.out.println("I (" + Thread.currentThread().getName() + s);
30 }
```

Console output:
I (main) am the master and terminate now!
I (B) waited for a long time and terminate now!
I (A) waited for a long time and terminate now!

As we passed names to the threads, the output here should be no surprise. Or does the output surprise you? When reading the code snippet and thinking about it, we might say, we start the threads in 21 and 22 but why is the print in 25 faster then the prints of our threads of line 8? And why is the print of thread B prior to the one of thread A?

As we already know (and when you execute the snippets of our example project, you will see it), the threads are executed concurrently. So the ordering is not deterministic, since the scheduling of different threads on different CPUs happens without any possibility to intervene[16].

So how do we get the correct ordering so that the print in line 25 is present after the prints within the threads?

[16] Besides the stuff we discuss in the next Section.

A first suggestion would be to call *run* instead of *start* in line 21 (*a.run()*) and 22 (*b.run()*). The console output is the following:

Console output:
I (main) waited for a long time and terminate now!
I (main) waited for a long time and terminate now!
I (main) am the master and terminate now!

Surprisingly all print statements are executed by the main thread. Remember the dualism of class *Thread* here.

As a rule of thumb:
An instance of class *Thread* is like any other object in Java. When I call a method, in this case *run* directly, it is executed in the current thread (which is per default the main thread, since there is no other). The specialty of *Thread* is, that the JVM starts a concurrent thread when calling start and executes within this new thread the business logic specified in *run*.

What you also might have recognized is the longer execution time. Since it is executed sequentially, the JVM needs a bit more than 20 seconds (in each thread we wait for 10 seconds) to terminate. When using *start()* it only took a bit more than 10 seconds, but the ordering was corrupted.

So how to solve this, waiting for the termination of some threads, before doing some important stuff, but having the performance boost of concurrent executions?

In the previous chapter we introduced the most important Thread methods, *join* being one of them. The API documentation of *Thread#join()* says: "Waits for this thread to die."[17] So after starting the thread, we call *join* to wait for the specific thread to die. The adjusted code sample looks like the following, we omitted some code compared to the previous sample:

[17] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#join()`

Listing 7: Join Threads

```
1  public static void main(String[] args) {
2
3    Runnable r = ...;
4
5    // thread object creation,
6    // like every other object by calling the constructor
7    Thread a = new Thread(r, "A");
8    Thread b = new Thread(r, "B");
9
10   // starting the threads
11   a.start();
12   b.start();
13
14   // joining threads
15   a.join();
16   b.join();
17
18   // exit main thread
19   printIt(") am the master and terminate now!");
20 }
```

Example:
Under `de.uniba.dsg.concurrency.examples.threads.Threading` you will find the example code.

Exercise:
As an exercise, try also to implement some runnables and threads and play a bit around with thread methods and states.

For sake of simplicity, we do not discuss *join*'s checked *InterruptedException* here. You can find a detailed explanation about interruptions in Section 4.2.

When reading this single sentence from the API doc, two questions arise. Who waits? And who dies?

As we already know, all the code we see in Listing 7 is executed by the main thread (think about the run example we did previously and the corresponding console output)[18]. So, in line 15 the main thread calls *a.join()*, therefore the main thread has to wait here until thread *a* terminates (dies). When thread *a* is terminated, the main thread executes the join of *b* in line 16. **It is very important, that the two joins are executed in sequence and the main thread can only proceed when both threads die (first A, then B).**

So in the end the two prints of our threads A and B are executed before the print in line 19, where the main thread terminates and also the JVM. The order of A and B is not preserved since both threads are started at the same time and they both sleep for 10 seconds and awake at the same time. So both orderings, A before B or vice versa, are possible.

[18] The stuff specified within the anonymous implementation of Runnable in line 3 is executed concurrently to the main thread! We hope that this is already clear and the remark here is only to foster your knowledge.

4 Shared Memory

4.1 Low Level Concepts

4.1.1 Java Monitor

Each object in Java is associated with a monitor (the type of monitor similar to the ones discussed in operating system courses). For our understanding it is sufficient to think of a monitor in Java as an entity which consists of an *inherent lock object* and a *Wait Set*. The Wait Set contains threads which are waiting upon acquiring the monitor's lock (see also Section 3.2 for a discussion about different thread states).

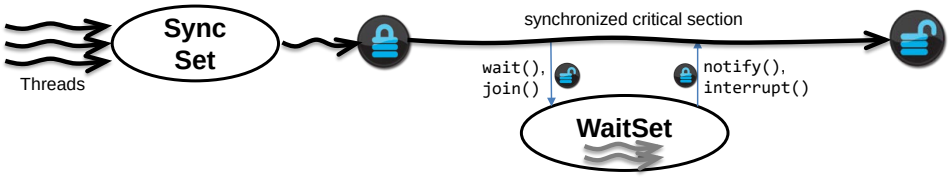

Figure 8: Java Monitor with SyncSet, Lock Object and WaitSet

When threads compete for the lock, they enter a *SyncSet*[19] and one of the competing threads can acquire the lock while the others are blocked as depicted and explained in Section 3.2.

4.1.2 Lock - *synchronized* and Wait Set

We will now discuss the *inherent lock object* part of a Java monitor. The *synchronized* keyword helps us to create a critical section in Java. Listing 8 shows the syntax for using *synchronized*. The implications are discussed after the Listing.

Listing 8: Usage of synchronized Keyword

```
1  class DoSomeWork {
2    private final Object mutex = new Object(); // our lock object
3
4    // . . . attributes, constructor etc.
5
6    public void doIt(){
7      synchronized(mutex){
8        // critical section
9      }
10   }
11 }
```

The lock object is specified in brackets where the mutual (sequential) access is enforced. For *DoSomeWork* we create a private object *mutex* which does all the monitor stuff for us. As a first remark here: the lock object *mutex* is not accessible from outside the class.

[19]The concept of a SyncSet was introduced by us. There is no such thing as a SyncSet in Java but we need it for our metaphor later on.

Therefore, the synchronization policy is encapsulated within the class (this is an important decision, we will see in Section 4.1.2 why).

In our example, all threads, which have a reference to the same instance of *DoSomeWork* and concurrently invoking *doIt* compete for the lock *mutex* to gain access to the critical section. But only executing a single function without coordination is not sufficient in most cases and also not really exciting! We now know how the *lock object part* of a Java monitor works. Let's look at the Wait Set.

As a rule of thumb:
Guarded Wait - wait/notify
In multithreading scenarios, when many actors want to achieve a specific goal, it is usual to check a condition before doing some work (Guarded Wait) since there are often scenarios where a thread has to pass the control over to another thread which is blocked and reacquire the lock later again.

So the first question is, what is the guard in our example?

The guard is a condition at the beginning of a critical section. Here a thread checks whether it is its turn or not. If this is not the case, the thread releases the lock and *wait*s until another thread or framework informs it that it can wake up. When awoken, the thread reacquires the lock and before entering the critical section stuff, it has to check the condition again! It might be the case that the situation has changed but its condition is still not true. Then this thread has to wait again, wake up again and check the condition until it evaluates to true.

Now we add the concept of *Guarded Wait* to our previous Listing 8 in Listing 9.

Listing 9: Calling *wait()*

```
1  class DoSomeWork {
2    private final Object mutex = new Object(); // our lock object
3
4    // . . . attributes, constructor etc.
5
6    public void doIt(){
7      synchronized(mutex){
8        while(condition == false){
9          mutex.wait();
10       }
11
12       // critical section stuff
13     }
14   }
15 }
```

There is a second *wait* method within the Java API for getting a thread to wait and enter the wait set of a lock object. This method includes a time interval, how long the thread is waiting before waking up again. This method can be beneficial in cases when the situation changes but the waiting thread does not recognize since it is not informed by another thread in the system. **So keep in mind, that calling *wait* with a timer is also a possible solution to awake a thread after some time and checking the condition again.**

Since *wait* is related to the Java monitor, the thread enters the *wait set* if the condition is false. *Wait* MUST be called on the lock object. Because of that the lock object knows which threads are waiting for it. Otherwise you would get an *IllegalMonitorStateException*.

Wait API Doc: "Causes the current thread to wait until it is awakened, typically by being **notified** or **interrupted**, or until a certain amount of real time has elapsed."[20]

Notify API Doc: "The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object."[21,22]

As a rule of thumb:
Code Style: Always use *wait* and *notify/notifyAll* within the same *synchronized* code block. The reason for this rule of thumb is the readability due to proximity of the two related methods.

As stated in our example, the thread is in the wait set and we consider the two possibilities for the awakening process[23]. **In both cases the waiting thread has to reacquire the lock, otherwise it can't resume**.

The two options to wake up a thread are the following:

1. *Notify()* - as the name suggest - notifies an **arbitrary** thread within the wait set. This awoken thread reaquires the lock, checks the guard condition and - if applicable - enters the critical section. But what does an implementation look like and where should I call *notify*? Listing 10 provides some answers.

Listing 10: Calling *notify()*

```
1  class DoSomeWork {
2    private final Object mutex = new Object(); // our lock object
3    // . . . attributes, constructor etc.
4
5    public void doIt(){
6      synchronized(mutex){
7        while(condition == false){
8          mutex.wait();
9        }
10       // critical section stuff
11
12       // leave critical section, notify another thread
13       mutex.notify();
14     }
15   }
16 }
```

[20] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#wait(long,int)

[21] Same holds true for the interrupt we discuss in the second item.

[22] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#notify()

[23] There is a third possibility as the documentation says which are *spurious wakeups*, but these wake ups are handled by implementing *Guarded Wait* right. So if you are interested read it up on your own.

Assume we have two threads, the first enters the synchronized block/critical section and evaluates the condition, which is false, and has to wait. The second thread subsequentially checks the condition, evaluates it to true, enters the critical section and afterwards calls *notify* on the lock object. Only the java monitor (lock object) knows which threads are waiting. The consequence is that an arbitrary thread within the wait set (we know only the first thread is there), is awoken and aquires the lock if the condition evaluates to true. Given that this is the case, it does the critical section stuff, notifies another thread (hint: no one is there, but who cares), returns from the method and the JVM exits (program terminates).

NotifyAll, as the name implies, wakes up all threads within the monitor's wait set. The awoken threads then compete for the lock in order to enter the critical section[24].

2. The second option to release a thread from the wait set is to interrupt it directly via the thread's *interrupt()* method or via some frameworks (for further discussion see Section 5.3). This causes an *InterruptedException* which we haven't discussed so far intentionally as it is a highly difficult problem/exception to handle it right. Section 4.2 discusses these problems in detail. When the thread is interrupted, it has to compete for the lock (same policy as before). If the lock is acquired the control flow resumes at line 14 (catch block) in Listing 11.

Listing 11: Adding InterruptedException and complete Example

```
1  class DoSomeWork {
2    private final Object mutex = new Object(); // our lock object
3
4    // . . . attributes, constructor etc.
5
6    public void doIt(){
7
8      // do other stuff, which needs no synchronization
9
10     synchronized(mutex){
11       while(condition == false){
12         try {
13           mutex.wait();
14         } catch ( InterruptedException e ){
15           // handle it
16         }
17       }
18       // critical section stuff
19
20       // leave critical section, notify another thread
21       mutex.notify();
22     }
23
24     // do other stuff, which needs no synchronization
25   }
26 }
```

[24] So in our metaphor, they enter the SyncSet and competing there for the lock.

As a rule of thumb:
Using a dedicated lock object gives a developer the opportunity to decide within a method which parts need synchronization (*critical section*) and which do not (*do other stuff*).

As a rule of thumb:
Wait(), **Notify()** and **NotifyAll()** are *monitor methods*. The *monitor methods* are included in the Class Object. Since each class inherits from Object, every object can be used as a lock. When you call one of these three methods, you have to do it by invoking them on the *lock object*.

Synchronized within the Method Signature We introduced the synchronized keyword with an explicit lock object to separate different concepts from each other. Our example from Listing 8- 11 implemented a class *DoSomeWork*. When we create an instance of *DoSomeWork*, a single member is responsible for all the mutex stuff which is protected from outside class access via the *private* modifier and can't be changed because of the *final* keyword[25]. But there is also the possibility to include *synchronized* keyword in the method signature (HINT: this is more or less the default for Java APIs or code you see implemented by others. Looks easier, but in the end synchronization policy is harder to enforce - you will see in a page why :).

So we rewrite our class *DoSomeWork* and synchronize the *doIt* method by including the keyword in the method signature in Listing 12:

Listing 12: Synchronized Methods

```
1 class DoSomeWork {
2   // . . . attributes, constructor etc.
3   public synchronized void doIt(){
4     // critical section
5   }
6 }
```

Synchronized included in the method signature.
But where is the lock object?

As already said and explained, each object in Java has an associated monitor. And as you might know, you can self-reference an object within its implementation via *this*.

When *synchronized* is included in the method signature, implicitly the self-reference of the object is used as the lock. If this dualism is already in your mind, you will not be surprised that the following snippet does the same as the previous, except for a single difference.

[25]This is also an important restriction! If objects alter their identity during locking, the synchronization policy is corrupted, see Section 4.1.5 for a discussion about final objects which change their identity.

Listing 13: Synchronized Blocks within Methods

```
class DoSomeWork {
  // . . . attributes, constructor etc.
  public void doIt(){
    synchronized(this){
      // critical section
    }
  }
}
```

The only, tiny difference is, that we can do some unsynchronized work before or after the *synchronized* block.

Another difference to our *final Object mutex* approach is that we expose the synchronization policy to everyone using an instance of *DoSomeWork*. That's somehow dangerous, but for many APIs necessary to enable **outside class synchronization**. To shed some light into this, we adjust our *DoSomeWork* class with another *synchronized* method *doItAgain*. But before we proceed, another rule of thumb.

As a rule of thumb:
Decide which encapsulation strategy you use for the synchronization of a class. As we have seen, there is a huge difference in a private, encapsulated lock object and using the self-reference for synchronization, which means exposing the synchronization policy to everyone using the instance. Despite for really good arguments, choose the first strategy.

We use the notation of Listing 12 for *doIt* and the notation of Listing 14 for *doItAgain*. (But, they are both the same ;) despite the simple difference)

Listing 14: Synchronized Methods - Mixture

```
class DoSomeWork {
  // . . . attributes, constructor etc.

  public synchronized void doIt(){
    // critical section
  }

  public void doItAgain(){
    synchronized(this){
      // critical section
    }
  }
}
```

Outside class synchronization Assuming we have two threads and a runnable implementation as shown in the following Listing 15:

Listing 15: Two Threads doing something

```
1  public static void main(String[] args) {
2    DoSomeWork work = new DoSomeWork();
3
4    Runnable r = new Runnable() {
5      @Override
6      public void run() {
7        work.doItAgain();
8        work.doIt();
9      }
10   };
11
12   Thread one = new Thread(r, "1");
13   Thread two = new Thread(r, "2");
14
15   // start threads
16   one.start();  two.start();
17
18   // join threads
19   one.join();   two.join();
20 }
```

Due to the implementation we provided for the two methods, the console output is in the following order. Reason for this order is a context switch after *doItAgain*.

Console output:
Executing thread: 1 Do it again
Executing thread: 2 Do it again
Executing thread: 1 Do It
Executing thread: 2 Do It
Executing thread: main DONE

We can see that the two threads interleave. At the beginning thread 1 acquires the lock, thread 2 is blocked. Thread 1 executes *doItAgain*, when leaving the synchronized block (now the lock is free again), thread 2 can enter the critical section in *doItAgain*. Same procedure, thread 2 exits the synchronized block/releases the lock and another context switch occurred[26]. Thread 1 enters the critical section of *doIt*, since no other thread currently holds the lock. After executing the method, it releases the lock, return from the run method and terminates. Thread 2 then executes *doIt* and also terminates.

Since we have a really urgent business demand, we want to have a compound action of *doIt* and *doItAgain* without interleaving. Because **locks are reentrant**, we can synchronize the two methods outside of the class. As another hint here: Most APIs offer this method style synchronization and a developer can then build *compound actions* in his/her code.

[26] We simulated these context switches via *Thread#sleep* invocations within the implementations.

The lock object in our *DoSomeWork* class is implicitly the *this* reference. It is important to synchronize with the identical lock, otherwise the synchronization does not work properly. So we can use the *work* object in line 2 as a lock object for doing our compound action. Lines 12 to 20 of Listing 15 are skipped for sake of simplicity.

Listing 16: Enabling outside Class Synchronization - Compound Action

```
1  public static void main(String[] args) {
2    DoSomeWork work = new DoSomeWork();
3
4    Runnable r = new Runnable() {
5      @Override
6      public void run() {
7        // outside class synchronization - compound action
8        synchronized (work) {
9          work.doItAgain();
10         work.doIt();
11       }
12     }
13   };
14 }
```

And the output is as expected:

Console output:
Executing thread: 1 Do it again
Executing thread: 1 Do It
Executing thread: 2 Do it again
Executing thread: 2 Do It
Executing thread: main DONE

Example:
We also included an example in our Java example project.
`de.uniba.dsg.concurrency.examples.lowlevel.DoSomeWork` contains the code. We used another lock object (*Object mutex*) within the *main* method to enable the compound action and explain the problems there.
As a primer: when using another lock object, other threads which only execute *doIt* or *doItAgain* can interleave between the two methods calls of our compound action, since they compete for the this lock every time independently. If we use *work* here, then there is a competition of the main thread which wants to execute the compound action and subsequentially reenters the lock twice. Any other thread which wants to execute one of the two methods is blocked.

Exercise:
Investigate the aforementioned example and also play around with the lock objects and understand the problems and differences and discuss these problems with your fellow students (when you can explain it - you got it :).

As a rule of thumb:
When not implementing an API which you share via Maven Central or another package repository, use the encapsulated private lock object strategy. Otherwise a developer/user which is normally not aware of all the nit-picky details will use your stuff in a wrong way and corrupt your internal state.

As a rule of thumb:
All accesses (reading and writing) to private fields (internal state) MUST be synchronized by an identical instance! (must be the same(identical!) instance!)

4.1.3 *Static* Synchronization

So far, we discussed how to secure the internal state of objects from concurrent accesses, but what about static members of a class?

As we already explained, objects are stored on the heap. Static members are also stored on the heap (primitives and objects - otherwise they wouldn't be accessible from all threads concurrently), but they are not stored *normally*, they are included in a meta space of the JVM.

Each class also has an implicit (so to say static instance), which *identifies* the class and can be used as lock object (same guarantees as discussed in Section 4.1.2).

Listing 17 shows how static fields are synchronized to avoid concurrency issues:

Listing 17: Static Member Synchronization

```
1  class CounterClass {
2
3    private static int x;
4
5    public int incrementAndGetX() {
6      synchronized(CounterClass.class) {
7        x++;
8        return x;
9      }
10   }
11 }
```

A good explanation with all the possibilities to synchronize a static member can be found at StackOverflow[27].

4.1.4 A Running Example

For our running example, where we explain the more theoretical considerations of the previous subsection in more detail, we assume that the monitor object is an instance *ball* of a implemented Class *Ball*. Since playing ball against a wall is somehow boring, we (A)

[27] https://stackoverflow.com/questions/2120248/how-to-synchronize-a-static-variable-among-threads-running-different-instances-o

want to play with our friends B,C and D. Assuming that thread A[28] acquires the lock of *ball*, thread B,C and D are *blocked* now. For now, acquiring the lock and getting into the critical section works with the *synchronized* keyword where the lock object (in our case ball) is specified within brackets.

So the current code looks like the following:

Listing 18: Running Synchronization Example

```
1  class Ball {
2    // ....
3  }
4
5  class Player extends Thread{
6    private final Ball ball;  // lock object
7    private int noOfTurns=2;  // exits when turns == 0
8
9    public Player(Ball ball, String name){
10     super(name);
11     this.ball = ball;
12   }
13
14   public void run() {
15     this.play();
16   }
17
18   public void play(){
19     synchronized (ball) {
20       // critical section
21     }
22   }
23
24   // other methods, like itsMyTurn . . .
25 }
```

So thread A is now in the critical section and checks the condition (business logic) to determine whether it's its turn.

Thread	**A**	**B**	**C**	**D**
State	runnable	blocked	blocked	blocked

Assuming that the condition we check is only true, when it's thread D's turn.

So thread A checks the condition, it is false and thread A says: "ok, let's wait until someone notifies me and I will then check, if it is my turn". So that everybody knows in the Java world that we are waiting for the *ball*, we call *wait()* on our lock object *ball*. Thread A is then in the wait set of ball and one of the other threads can grant the lock and switch the state from blocked to running. We assume that thread C (a random thread is picked from the SyncSet (where the threads are competing for the lock, see thread states in Section 3.2)) now gets control and can acquire the lock.

[28] What's meant here is the player A which is executed by a thread. For simplicity we assume that player A is executed by thread A, ...

Listing 19: Running Synchronization Example

```
class Ball {
  // ....
}

class Player extends Thread{

  // omitted . . .

  public void play(){
    synchronized (ball) {
      // critical section
      while(!this.itsMyTurn()) {  // check the condition
        try {
          ball.wait();
        } catch (InterruptedException e) {
          // we do not care for the moment
        }
      }

      makeMyTurn();
      // end critical section
    }
  }

// other methods, like itsMyTurn . . .
}
```

The actual thread state situation is now the following:

Thread	**A**	**B**	**C**	**D**
State	waiting	blocked	running	blocked

As we already know it's thread D's turn, so thread C checks the condition, evaluates that condition is false and also calls *wait* on the lock object *ball*.

Thread	**A**	**B**	**C**	**D**
State	waiting	blocked	waiting	blocked

So the next player getting control is thread B. Same "problem". So finally thread D grants the lock and executes its turn, the play method return and also the run method returns So thread D is now in the state *terminated* (Leaves the game).

Thread	**A**	**B**	**C**	**D**
State	waiting	waiting	waiting	terminated

Coool :) Thread D does its turn finally, BUT what about A, B and C. They are all waiting that someone will call them, that things have changed and it's another thread's turn. (Also the JVM does not terminate, which is another urgent problem...)

As we already know from the previous subsection, we forgot to notify another thread, when we finished our execution. So the easiest way is to call *notify()* when we did our turn.

For the sake of simplicity we only show the play method:

Listing 20: Awake one Thread via *notify()*

```
1  public void play(){
2    synchronized (ball) {
3      // critical section
4      while(!this.itsMyTurn()) {  // check the condition
5        try {
6          ball.wait();
7        } catch (InterruptedException e) {
8          // we do not care for the moment
9        }
10     }
11
12     makeMyTurn();
13     // end critical section
14     ball.notify();
15   }
16 }
```

So thread D, after doing its turn, notifies an **arbitrary** thread. We assume after D, thread B's condition becomes true. The arbitrary scheduling selects thread A. A can acquire the lock and currently the situation looks like the following.

Thread	**A**	**B**	**C**	**D**
State	running	waiting	waiting	terminated

Now A checks its condition, it is sadly false and calls *wait* again. So the overall status of our program is like the following:

Thread	**A**	**B**	**C**	**D**
State	waiting	waiting	waiting	terminated

Damn! Another scenario where the program is stuck. So how to solve this issue.

There is another method *notifyAll()*, which awakes all threads from a monitor's wait set. So we change our method and call *notifyAll()* instead of *notify()*.

Listing 21: Awake all Threads via *notifyAll()*

```
1  public void play(){
2    synchronized (ball) {
3      // critical section
4      while(!this.itsMyTurn()) {  // check the condition
5        try {
6          ball.wait();
7        } catch (InterruptedException e) {
8          // we do not care for the moment
9        }
10     }
11
12     makeMyTurn();
13     // end critical section
14     ball.notifyAll();
15   }
16 }
```

All threads included in the wait set awaken and compete to enter the critical section. The scenario know is as follows (all threads in the wait set change their state from *waiting* to *blocked*):

Thread	**A**	**B**	**C**	**D**
State	blocked	blocked	blocked	terminated

Assuming, as already said, thread A wins the race, recognizes that it is not its turn and waits again. Now B and C are blocked and lock is granted to B. So the current thread state is as follows:

Thread	**A**	**B**	**C**	**D**
State	waiting	running	blocked	terminated

B executes the critical section stuff and terminates. But before termination, it awakes all waiting threads (in our case A). So the implementation is fair/unfair here (as you will), so that A and C are blocked again and competing for the lock.

Another two or three iterations, all threads reach the terminated state and the JVM exits.

Example:
`de.uniba.dsg.concurrency.examples.lowlevel.Play` contains the sources of our running example.

As a rule of thumb:
If you are not sure, if *notify()* is sufficient and your program will terminate in all conceivable situations, use *notifyAll()* instead. It is only a question of a micro-performance optimization when using *notify*.

4.1.5 Another Example - Using Wrapper/final Objects as Locks

To understand locking and identity of lock objects a bit better, we have the following example code snippet, where we use an Integer *value*, which is incremented concurrently, as a lock object. Since the wrapper class *Integer* is an object and not a primitive type like its little brother *int*, it has all the necessary methods, we need for synchronization. The following code snippet - with the knowledge we gained so far - is syntactically correct, but there is a tiny problem with the lock object:

Listing 22: A simple Counter - so where is the Problem?

```
1  class Counter {
2    private Integer value;
3    public Counter(Integer value) {
4      this.value = value;
5    }
6
7    public void increment(){
8      synchronized(value) {
9        value++;
10     }
11   }
12 }
```

You might ask, where is the problem? From the prior chapter and the discussion until now there is no error when looking at the *Counter* in Listing 22. As a hint here: Integer is final. So all instances of Integer do not change their value. If the value of an Integer is changed, the reference to the next Integer is set on the stack's variable.

So let's do an example. Assume we have three threads which use all the same instance of counter and concurrently call increment on it. As our Figure 9 shows, the first two threads want to do an increment, so the lock object *Integer value* has the value 5 and the hash code (*hashCode* method in Java) of 5[29]. So A and B synchronize on the Integer object 5. A gets the lock and executes the critical section (incrementing the value to 6), leaves the critical section. B resumes from his blocking state and changes its state to running and acquires the critical section (the blocking state was - when waiting for object - 5), so it acquires the lock for object 5 and also enters the critical section and increments *value*.

So our lock object changes its identity (the reference which is stored in the stack variable value). B doesn't recognize this state change since it is blocked on the prior object (5). So the update of the first thread A is lost.

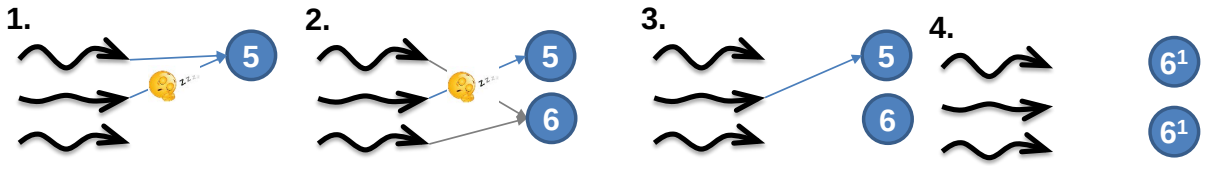

Figure 9: Using a Wrapper or other final Objects as Lock Object

[29]For Integer, hashCode and actual value is identical.

As a rule of thumb:
As we know from the Immutability Section 1.3, immutable objects alter their identity if a change happens. Immutable objects can only be lock objects, if and only if they do not change their state (think about *final* keyword) during program execution, otherwise the problem, we seen in this section, happens.

4.1.6 Visibility

We stated in our synchronization example that different threads have access to the same, identical object. But how can this happen, if we have a multi-core computer and the threads are scheduled on different CPUs. You may answer that's fair enough, they are on the same system and therefore have access to the same main memory (RAM). We would agree, but that is not the complete truth.

There is some sort of optimization implemented we have to discuss here and which makes this access to consistent data in all situations somehow fragile. Let's look at a simple example in Listing 23[30]:

Listing 23: A simple Visibility Class - so where is the Problem?

```
1 class Visibility {
2   public boolean ready = false;
3   public int number = 0;
4 }
```

Problem So when we are thinking about two threads, which concurrently read and write values (we are aware that writing is not an atomic action, see Section 2.2) they can see different values of the variables due to **caching** and **reordering**.

Before explaining the caching and reordering problem, we have to introduce the *volatile* keyword which is part of our solution. We also add an additional boolean member to our *Visibility* class of Listing 23 in Listing 24.

Listing 24: Visibility Class with a *volatile* Member

```
1 class Visibility {
2   public volatile boolean breakLoop;
3   public boolean ready = false;
4   public int number = 0;
5 }
```

Volatile "Volatile fields are special fields which are used for communicating state between threads. **Each read of a volatile will see the last write to that volatile by any thread**; in effect, they are designated by the programmer as fields for which **it is never acceptable to see a "stale" value** as a result of caching or reordering. The compiler and runtime are **prohibited from allocating them in registers**. They must also ensure that after they

[30]The public fields *ready* and *number* are here for sake of simplicity. We highly recommend to use *private* as a field modifier. The example is taken and adapted from Brian Goetz's "NoVisibility.java", page 34 in his book "Java Concurrency in Practice".

are written, they are flushed out of the cache to main memory, so they can immediately become visible to other threads. Similarly, before a volatile field is read, the cache must be invalidated so that the value in main memory, not the local processor cache, is the one seen. There are also additional restrictions on reordering accesses to volatile variables."[31]

Caching So we now know that volatile guarantees to see the most recent consistent value of all fields over all threads of the program. You may say - ok that's not a new information, I already know this, where is the real problem?

Each CPU has caches nearby, where it looks for attributes when executing a command. So some attributes are cached for a better performance and not read and written from the main memory every time a command with this attribute is executed[32]. The MESI protocol helds all CPU caches (normally L1 and L2) in sync. So where is the problem? CPU registers are the problem which are not held in sync by the MESI protocol. But *volatile* prohibits the allocation in CPU registers. So first problem solved :)

Reordering "There are a number of cases in which accesses to program variables (object instance fields, class static fields, and array elements) **may appear to execute in a different order than was specified by the program.** The compiler is free to take liberties with the ordering of instructions in the name of optimization. Processors may execute instructions out of order under certain circumstances. Data may be moved between registers, processor caches, and main memory in different order than specified by the program.

For example, if a thread writes to field a and then to field b, and the value of b does not depend on the value of a, then the compiler is free to reorder these operations, and the cache is free to flush b to main memory before a. There are a number of potential sources of reordering, such as the compiler, the JIT, and the cache.

The compiler, runtime, and hardware are supposed to conspire the illusion of as-if-serial semantics, which means that in a single-threaded program, the program should not be able to observe the effects of reorderings. However, reorderings can come into play in incorrectly synchronized multithreaded programs, where one thread is able to observe the effects of other threads, and may be able to detect that variable accesses become visible to other threads in a different order than executed or specified in the program.

Most of the time, one thread doesn't care what the other is doing. But when it does, that's what synchronization is used for."[33,34]

[31] Since this is a good explanation for volatile, we copied it from the FAQ of the JSR133: `https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`

[32] This is also a good reference to read about caching and how the MESI protocol works for a few examples...
`https://software.rajivprab.com/2018/04/29/myths-programmers-believe-about-cpu-caches/`

[33] Since this is a good explanation for reordering, we copied it from the FAQ of the JSR133: `https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`,

[34] Good explanation and figures can also be found on following sites: `http://igoro.com/archive/volatile-keyword-in-c-memory-model-explained/` and `http://tutorials.jenkov.com/java-concurrency/java-memory-model.html`, but be aware that these sources sometimes do not distinguish between registers and caches and therefore introduce some inaccuracy.

Volatile gives us the guarantee from the previous section, that all volatile fields are consistent and not affected by caching. All instructions in a thread before the access to a volatile field, are therefore also visible to any other thread. Instructions after the access to a volatile field can be reordered and also executed before the access to a volatile field.

Let's make a running example to understand the implications: Caching and Reordering.

A running example We use our example of Listing 24 and add two threads to this example, which concurrently change values of our shared object *Vibility* in Listing 25.

In our example we start the *reader* first, so it is up and running. Subsequentially, we start the *changer* which changes the values of *breakLoop* and the other two attributes. Since *breakLoop* is volatile, it is guaranteed that the *reader* sees the change of *breakLoop* made by the *changer*. The other instructions might be reordered or the changes are committed at least to L1 and therefore reader can read the changes.

For 10,000 iterations, a possible result (dump of my console) could be[35]:

key	**no of occurences**
true - 42	9799
false - 0	70
true - 0	131

Theoretically, when reordering and caching happens at the same time, also *false - 42* is possible. It is not only theoretically possible, you can also see this in practice [36].

You may say now, ok that's not a proof, that visibility is the problem here, maybe the reader read the values before the changer changed it. You are right, we agree completely that this example is not a proof. But there is one possibility to have the clarity that visibility (and therefore caching within the CPU register is a problem, assumed the MESI protocol is implemented correctly - which it is for mature processors like Intel and AMD). **And this clarity is easily gained when removing volatile before *breakLoop***. When executing the sample with volatile in line 3 (having the guarantee that each thread sees the most recent value und therefore the change of the changer thread), the program executes in 2.8 seconds on our machine. When removing volatile in line 3, the program hangs (print the summary here after each iteration and you can also determine which iteration hangs). The only option is to exit the JVM by hand.

Example:
`de.uniba.dsg.concurrency.examples.lowlevel.Visibility` contains the source code of this running example. Try to understand the different steps and try it on your machine.

[35] You can play around by using the *Visibility* class of our example.

[36] If you are interested in this, you have to deal with JCStress (`https://openjdk.java.net/projects/code-tools/jcstress/`) and we can provide some tips and sources, but that's really for those who want to understand how the last bit in the JVM is accessed ;)

Listing 25: Visibility Example with two Threads

```
1  class Visibility {
2
3    public volatile boolean breakLoop;
4    public boolean ready = false;
5    public int number = 0;
6
7    public Thread createChanger() {
8      Runnable changer = new Runnable() {
9        @Override
10       public void run() {
11         breakLoop = true;    //1
12         ready = true;        //2
13         number = 42;         //3
14       }
15     };
16
17     Thread t = new Thread(changer, "Changer");
18     t.start();
19     return t;
20   }
21
22   public Thread createReader() {
23     Runnable reader = new Runnable() {
24       @Override
25       public void run() {
26         while(!breakLoop){
27         // spin waiting
28         }
29
30         boolean tempReady = ready;
31         int tempNumber = number;
32         String key = "" + ready + " - " + number;
33         System.out.println(key);
34       }
35     };
36
37     Thread s = new Thread(reader, "Reader");
38     s.start();
39     return s;
40   }
41
42   public static void main(String[] args) throws InterruptedException {
43     Visibility v = new Visibility();
44     Thread reader = v.createReader();
45     Thread changer = v.createChanger();
46
47     changer.join();
48     reader.join();
49 }
```

The explanation for this possible output is quite easy. Since *breakLoop* in the reader is used frequently, the CPU caches it in its register or the changer does not write it directly to L1.

Now we are looking at the ordering and how volatile does help here. Therefore we change line 11 and line 12, so the green commented statements 1 and 2 in Listing 25. We already explained that all writes to non-volatile variables before a volatile variable access are seen by any other thread. Reordering is prohibited here. Therefore our console output, when running the example again, should not include some *false -0* results.

On our machine the following output was generated when using 10,000 iterations again:

key	**no of occurences**
true - 42	9782
true - 0	218

And finally changing statement of line 11 and 13, so the commented statement 1 and 3, only a single result remain (*true – 42*).

key	**no of occurences**
true - 42	10000

Summary and Comparison of volatile and synchronized ***Volatile*** guarantees us that attributes are not cached, volatile statements are not reordered and every thread sees the most recent value of an attribute. It does NOT guarantee atomicity when thinking about an increment on a counter. Use cases for volatile are completion, interruption and status flags, where a concurrent read and write does not lead to problems since for example the flag is accessed via a loop and a read of stale data is not problematic.

Synchronized on the other hand gives us the same guarantee as volatile, BUT enables critical sections. So the set of statements within a sychronized block is for all threads in the system like an atomic action.

4.1.7 Deadlock

A deadlock is a reciprocal waiting condition, where A waits for a resource B has acquired and B waits for a resource A has acquired. Deadlocks are also possible with more than two members when thinking about circular waiting conditions.

As a developer, we have to ensure deadlock free programs. But how do we do that? Let's have a look at the following example and the problems which might occur and are often easily solvable.

We have two nice and friendly bar keepers, who have two different preparation styles for gin and tonic. One gin tonic recipe is with ice and the other without. All bar keepers believe, that for a gin and tonic without ice you need to put in gin first and then tonic. For a gin and tonic with ice it is the other way around.

Sounds like a possible deadlock scenario, doesn't it?

Two guests concurrently order gin and tonics from two nice bar keepers. There is only a single bottle of each gin and tonic within the bar. So the first guest - let's call him Max - orders a gin and tonic without ice. Sebastian, one of the bar keepers, takes the gin and also wants to get hold of the tonic, but the bottle isn't there.

WHY? - A case for Sherlock?

At the same time as Max made his order, Sofia, another guest, ordered a gin and tonic with ice from the other bar keeper called Leonie. So Leonie picked up the bottle of tonic first, but can't find the gin bottle since Sebastian holds it in his hand. As Sebastian has the gin and Leonie has the tonic, neither of the bar keepers is able to finish their drink. Because both bar keepers do not talk to each other during work, Max and Sofia don't get a gin tonic and leave the bar thirsty.

Listing 26: A Deadlock which you don't wanna experience

```
1  class GinTonic {
2
3    private final Object gin = new Object();
4    private final Object tonic = new Object();
5
6    public void prepare() {
7      synchronized (gin) {
8        synchronized (tonic) {
9          mixIt();
10       }
11     }
12   }
13   //Warning: DON'T do this
14   public void prepareWithIce() {
15     synchronized (tonic) {
16       synchronized (gin) {
17         mixItWithIce();
18       }
19     }
20   }
21 }
```

As in this sad real world example (:)), having two resources which are reciprocally acquired and not freed after some time, the whole system ends in a deadlock.

Since there is no check whether a lock has already been acquired, a developer can only ensure deadlock free programs by checking the call paths of her solution. So our example in Listing 26 is highly deadlock prone since both methods can be invoked concurrently.

If there really is a need for two (or more) distinct lock objects like in our example, make sure that locks are always acquired in the same order as shown in 27[37]. That way, locks can not be distributed among threads.

[37] You might agree that you need really good reasons for such a design!

Listing 27: Always use the same order for acquiring locks

```
1  class GinTonic {
2
3    private final Object gin = new Object();
4    private final Object tonic = new Object();
5
6    public void prepare() {
7      synchronized (gin) {
8        synchronized (tonic) {
9          mixIt();
10       }
11     }
12   }
13
14   public void prepareWithIce() {
15     synchronized (gin) {
16       synchronized (tonic) {
17         mixItWithIce();
18       }
19     }
20   }
21 }
```

4.2 Interruption

4.2.1 Non-Blocking Operations

So far, we managed the life cycle of a thread with the methods *start()* to start the execution of a thread and *join()* to wait for a thread's termination. As shown in the previous Listings, we implemented the *run()* method with several statements which are concurrently executed. Each thread will execute the statements in the *run()* method until its termination.

Usually that is not a problem if the threads execute computations with a low complexity. If we have a short total runtime (e.g. only a few seconds) and all calculations are performed without any external dependencies, there is no need for an abnormal termination. But what if the user realizes that the input data to a long-running operation was incorrect? Or if we need an interactive application which properly reacts to the user's intent to terminate it early? If we look back to our running example, there are no complex statements in the *run()* method. However, we need a mechanism for an abnormal termination of a *Player* because we want a responsive application which properly reacts to the user's intention to terminate the game while it has't finished yet. Up to now, we only have the possibility to stop the execution by shutting down the JVM, regardless of the results.

Fortunately, it is possible to interrupt a thread during its execution. For this, the class Thread offers the life cylce methods *interrupt()*, *isInterrupted()*, and *interrupted()*. ***Thread interruption* is a collaborative mechanism.** It allows a thread A to disturb another thread B by sending an interruption signal. Each thread owns the so-called *interrupted flag*. It is a boolean member of a thread object and is initially false. We can derive the current value of the flag with the boolean method *isInterrupted()*. If thread A wants

to interrupt thread B, it can call the *interrupt()* method on thread B's object instance. As a result, the interrupted flag of thread B changes from *false* to *true*. For now, that's enough. Let's have a look at a very simple example to get an understanding of the basic aspects (Listing 28).

Listing 28: Simple Interruption Example

```
1  public static void main(String[] args) throws InterruptedException {
2
3    Runnable r = new Runnable() {
4      @Override
5      public void run() {
6        while (true) {
7          // the run method does nothing
8        }
9      }
10   };
11   Thread t1 = new Thread(r);
12
13   // start t1 and check for the interrupted flag
14   t1.start();
15   System.out.println("Interrupted?: " + t1.isInterrupted());
16   // Interrupted?: false
17
18   // interrupt t1 and check for the interrupted flag
19   t1.interrupt();
20   System.out.println("Interrupted?: " + t1.isInterrupted());
21   // Interrupted?: true
22
23   // is a still running?
24   System.out.println("IsAlive?: " + t1.isAlive());   // IsAlive?: true
25
26   // wait for the termination of Thread t1
27   t1.join();
28 }
```

As in the previous listings, we implemented the thread's logic by using the interface Runnable (Line 3 – 10). The implementation is just a simple while loop with no other statements (Line 6 – 8). After the creation of the thread object, we start the thread in line 14 and check for the *interrupted flag* by using the method *isInterrupted()*. The method returns the current state of the *interrupted flag* which evaluates to *false*.

Currently, there are two threads running: The *main* thread and thread *t1*. The *main* thread invokes *interrupt()* in line 19 on the thread (object) *t1*. After that, we check for the *interrupted flag* again. Finally, the *main* thread waits for the termination of *t1* in line 27.

The interrupt signal sent by the *main* thread changed *t1*'s interrupted-flag successfully to *true*. Nevertheless, thread *t1* is still alive. An additional side-effect is that the program/JVM does not terminate anymore. The *main* thread waits forever in the wait set of *t1* for the termination of *t1*. Thus, this implementation will never terminate.

How can we change that? We need to check for the state of the *interrupted flag* at regular intervals, for example by using a *while loop*. Hence, the solution could be:

Listing 29: Interuption-responsive Thread

```
1  Runnable r = new Runnable() {
2    @Override
3    public void run() {
4      // Thread will terminate if the interrupted-flag is true
5      while (!Thread.currentThread().isInterrupted()) {
6        // Busy waiting
7      }
8      // After leaving while-loop,
9      // thread terminates as there are no further statements in run
10   }
11 };
```

In Listing 29 the thread is continuously evaluating the value of its *interrupted flag* with the previously introduced method *isInterrupted()*. The static method *Thread.currentThread()* returns a reference to the currently executing thread on the JVM. Because *isInterrupted()* initially returns *false*, the thread continues the execution as long as there is no change. If the *interrupted flag* changes to *true* because of an *interrupt* from another thread, the condition of the *while-loop* does not hold anymore. The loop terminates and as a consequence, the executing thread of the Runnable as well. Of course you don't have to use *while-loops* but can also use *if-statements* at several points in the thread's logic to achieve responsiveness.

As a rule of thumb:
The interruption signal from thread A to thread B does not necessarily mean the immediate termination of thread B. It is just a way of politely asking to try an early termination. The behaviour of the interrupted thread depends on its implementation, meaning the statements in the *run()* method. A thread must support its interruption by providing an implementation that considers the current state of the *interrupted flag*.

The method *interrupted()* sounds similar to *interrupt()* but works differently. Firstly, the method *interrupt()* is an instance method and can be called on an instance of the class Thread. The method *interrupted()* works in a static context like *isInterrupted()* in the above sample and does not need an instance. But there is an important difference. The method *interrupted()* returns the current state of the *interrupted flag* and **resets the flag to false subsequently** while *isInterrupted()* **only reads the value** of the flag without changing it.

Listing 30 is quite similar to the previous listing. However, we replaced the Runnable's implementation by iterating via a *for-loop*. Within the *for-loop*, there might be some complex operations with a long runtime (ommited in line 7). To achieve responsiveness, we add a check for the *interrupted flag* by using the method *interrupted()* inside the thread's logic.

Listing 30: Thread Interruption with *interrupted()*

```
1  public static void main(String[] args) throws InterruptedException {
2
3    Runnable r = new Runnable() {
4      @Override
5      public void run() {
6        for (int i = 0; i < 1_000_000; i++) {
7          // some complex operations
8          if (Thread.interrupted()) {
9            // there was an interrupt
10           System.out.println("Interrupted after " + i + "!");
11           // Interrupted after 1332 iterations!
12           return;
13         }
14       }
15     }
16   };
17   Thread t2 = new Thread(r);
18
19   // start t2 and check for the interrupted flag
20   t2.start();
21   System.out.println("Interrupted: " + t2.isInterrupted());
22   // Interrupted?: false
23
24   // interrupt t2 and check for the interrupted flag
25   t2.interrupt();
26   System.out.println("Interrupted: " + t2.isInterrupted());
27   // Interrupted?: false
28
29   t2.join();  // wait for termination
30 }
```

Again, the flag is initially *false* (line 21). In line 25, the *main* thread signals *t2* the interrupt. The *interrupted flag* of *t2* is now *true*. As a consequence, *t2* enters the *if-statement* in line 8–12 and prints the current iteration (line 10), because the condition holds in line 8. Finally, it *returns* (line 12) and terminates. However, the second check for the interrupted flag also returns *false* in line 26. The reason for that is due to the *interrupted()* method. It **returns the *interrupted flag*** of the thread **and clears the *flag* immediately!** In the end, *t2* was stopped after 1332 iterations. If we execute this program multiple times, we would always get different results. Can you explain why? What is the range of iterations we can expect? Discuss the questions with your fellow students.

In this chapter we introduced basic mechanism of interruptions. We presented two simple ways to make threads responsive for interruptions from outside. Certainly, the method *Thread.interrupted()* should be used wisely. Often, this method can be replaced with the approach from Listing 30 to achieve the same behavior. In general, there are multiple ways to make a multi-threaded application responsive to unexpected interactions. Usually, there is a master thread that is in charge of managing several worker threads. If a user wants to cancel a complex calculation, the master thread receives the interrupt and propagates it to the worker threads. This enables a graceful shutdown of all child threads and the entire application as well.

4.2.2 Blocking Operations

The former listings only covered threads with a simple logic. No computation had dependencies on data structures, other threads or resources like databases. Usually, those operations are defined as non-blocking operations because they are just dependent on available resources like CPU, memory, and I/O troughput. If we think about a table tennis example, there are a few situations where threads are forced to enter the wait set of the class *Ball* (e.g. it is not the thread's turn).

Now the question is: What happens if the thread is currently suspended (in the wait set) and an interrupt occurs due to the user's choice to cancel the game? How can suspended threads be resumed other than with the methods *notify()* and *notifyAll()*? What are the necessary mechanisms to achieve a responsive application where multiple threads potentially wait for an indefinite amount of time to access a shared resource?

Here we enter the field of so-called blocking-methods which are responsive for interrupts while threads are waiting for the lock or other resources. Take Listing 21 as an example. A player waits for acquiring the lock on the ball. If something goes wrong and the lock is never released, the player will wait forever. And there is no way to achieve a graceful shutdown besides a cruel shutdown of the JVM. Therefore, *InterruptedExceptions* exist for this reason and allow us to overcome this issue. Consider our table tennis example again:

Listing 31: Awake all Threads via *notifyAll()*

```
1  public void play(){
2    synchronized (ball) {
3      // critical section
4      while(!this.itsMyTurn()) {  // check the condition
5        try {
6          ball.wait();
7        } catch (InterruptedException e) {
8          // now we care - preserve the state for the further execution
9          Thread.currentThread.interrupt();
10         return;
11       }
12     }
13
14     makeMyTurn();
15     // end critical section
16     ball.notifyAll();
17   }
18 }
```

Let's assume our players are waiting for their turns in line 6 and the threads are waiting as long as there is no invocation of *notify()* or *notifyAll()* by another thread. Now, the user wants to cancel the game, therefore, he or she sends an interrupt. Fortunately, the blocking method *wait()* enables our threads being responsive. As you might have noticed, this method throws the checked *InteruptedException*. So far we had just swallowed that exception. But now what happens if we get an interrupt during our execution? Basically, the former mechanism remains the same. The interrupted flag changes to *true*. If the thread is currently executing or in a blocking state as e.g. caused by *wait()*, the following occurs: **(1)** As soon as the thread is able to aquire the lock, it unblocks and an

InterruptedException is thrown. And **(2)** the *interrupted flag* is reset to *false*.

Then, the thread enters the catch block. So far, nothing would happen and the thread would continue its execution. **Don't do this! Do never swallow InterruptedExceptions!**[38] It is always good practice to preserve the interruption status e.g. by interrupting ourselves in line 9. This is especially the case, when further blocking operations are in the control flow and may hinder a program from termination. In our example, we also need a termination policy. As before, we can leave the execution with *return* in line 10.

If we include the check for the *interrupted flag* in the surrounding while loop in line 4, we could omit the return. In this simple example, we do not need to undertake any further actions. Usually, a catch block is used to enable the Thread to clean its state after the cancellation (e.g. remove large data structures from temporary data or close database connections).

As a rule of thumb:
Take care of the responsiveness of threads to interruptions. If there are no regular checks for an interrupt (e.g. with *isInterrupted()*/*interrupted()*) or there are no blocking methods that lead to an *InterruptedException* with a subsequent termination policy, a thread ignores the interrupt and will not stop its execution. Never swallow an *InterruptedException*, rather implement an appropriate termination policy.

Object.wait() is only one of the methods that throw an *InterruptedException*. The same applies to other low-level-blocking methods like *Thread.join()*, or *Thread.sleep()*, as mentioned in Section 3.3. Later on, we will get familiar with high-level data structures like *BlockingQueues* with further methods that might cause *InterruptedExceptions*.

Lastly, we want to give you a short overview of the differences between blocking and non-blocking methods:

Table 2: Non-Blocking and Blocking Methods Comparison

Parameter	**Blocking Methods**	**Non-Blocking Methods**
Termination	dependent on external events (operations of other threads, e.g. Releasing of locks, I/O, Timeout)	dependent on available resources
Estimated time for termination	hard to estimate, reduces the responsiveness of an application inherently	quite good (dependent on the implementation); method tries to terminate as soon as possible
Examples	Low-level blocking methods like *Thread.join()*, *Object.wait()*, *Thread.sleep()*; database queries waiting for incoming requests (e.g. on a TCP server)	long-running and complex computations (e.g. loops) *without* any external conditions

This detailed understanding of blocking and non-blocking methods helps to implement an appropriate termination policy and as a result, responsive, multi-threaded applications. Interruptions is a difficult topic and might be hard to grasp. So, give it a try and put it into practice.

[38] There are situations where swallowing an exception is the right choice but you should really think about it and document your decision.

4.2.3 Strategies to handle an InterruptedException

Listing 30 already showed a way to handle an *InterruptedException*. After the interruption, we recovered the state of the *interruption flag* and exited the invoked method with *return*. Of course, there are several other appropriate strategies. Which one you should choose depends on the software architecture, the kind of applications and the use case. The two most frequent ways are direct propagation and preserving the state.

Direct propagation This strategy could be considered to be the easiest one. After the interruption has occurred, the corresponding methods within the runnable implementation throw the *InterruptedException* to the next level in the call stack until the *run* method is reached. Let's have a look at Listing 32:

Listing 32: Interruption - Direct Propagation

```
1  class MyPropagationRunnable implements Runnable {
2
3    @Override
4    public void run() {
5      try {
6        heavyComputation();
7      } catch (InterruptedException e) {
8        // print to System.err for logging
9      }
10   }
11
12   public int heavyComputation() throws InterruptedException {
13     for (int i = 0; i < 100_000; i++) {
14       takeANap();
15     }
16     return 42;
17   }
18
19   private void takeANap() throws InterruptedException {
20     Thread.sleep(100);
21   }
22 }
```

Since *run* is not able to throw checked exceptions, a developer has to think about a good termination - the best way would be to log the error and do some clean up if necessary. Listing 32 shows such a scenario. The method *makeASleep* only sleeps and is, therefore, blocking the execution. The *InterruptedException* is thrown to the caller if the current thread is interrupted. The caller is the method *heavyComputation*. Within this method, the *InterruptedException* can't be handled properly. Because of that the *InterruptedException* is thrown up the call stack. Finally, the *InterruptedException* is caught within *run* and the thread can terminate.

Example:
`de.uniba.dsg.concurrency.examples.interruption.DirectPropagation` contains the source code of the above example and some additional explanation. So have a look at it.

Preserve the state Only small modifications are necessary to preserve the state. We have to guarantee that the state of the interrupted-flag is preserved regardless of what has happened. Again, we have a look at the Runnable which is only sleeping and doesn't do any computations (Listing 33).

This time, we extended the runnable by a *BlockingQueue* member (see Section 5.2[39]). Furthermore, we added *cleanUp* to do the clean up after the heavy computation by taking an element from a cleanUpQueue.

Listing 33: Interruption - Perserving the State

```
1  class MyPreserveStatusRunnable implements Runnable {
2
3    private final BlockingQueue<String> cleanUpQueue;
4
5    @Override
6    public void run() {
7      heavyComputation();
8      cleanUp();
9    }
10
11   private void cleanUp() {
12     try {
13       cleanUpQueue.take();
14     } catch (InterruptedException e) {
15       Thread.currentThread().interrupt();
16     }
17   }
18
19   public int heavyComputation() {
20     for (int i = 0; i < 10; i++) {
21       takeANap(1000);
22     }
23     return 42;
24   }
25
26   private void takeANap(int millis) {
27     try {
28       Thread.sleep(1000);
29     } catch (InterruptedException e) {
30       Thread.currentThread().interrupt();
31     }
32   }
33 }
```

Both methods in *run* contain some blocking methods (*sleep* and *take*). When one of the two methods throws the *InterruptedException*, the *interrupted flag* is set to *false*. So within the corresponding catch block, the interrupted-flags has to be reset to true again.

If our thread is interrupted during e.g. the third iteration in line 21 (and therefore 28), *Thread.sleep()* in line 28 throws an *InterruptedException*. Therefore, the *interrupted flag* is *false* (the exception resets the *flag*). As we want to preserve the state, however, we set it

[39] We included the *BlockingQueue* here since we want to show another blocking method.

to *true* again in line 30. During all further iterations in line 21 the *Thread.sleep* in line 28 immediately throws *InterruptedException* since the status is *true*. This behaviour is the same for *take* in line 13. As the last iteration in 21 (makeASleep) resets the status to *true*, *take* in line 13 is not executed since the method directly throws the exception.

HINT: The control flow is somehow corrupted since an exception is thrown, but *heavyComputation* returns normally (have this in mind when designing your interruption logic).

Example:
`de.uniba.dsg.concurrency.examples.interruption.PreserveStatus` contains the source code of the above example and some additional explanation. So have a look at it.

As a rule of thumb:
There are many ways to design cooperative interruption mechanism to provide a responsive application. Depending on the use case and application, the interruption mechanism can be used to implement an appropriate termination mechanism. Sometimes it is helpful to swallow interruptions (e.g. with non-cancelable tasks like reading from a queue or sending a message). But then you should at least preserve the interrupted-flag to give the thread owner the possibility to decide if the result of the irregularly terminated computation is eligible for further usage.
Also a mixture of the two concepts presented here is possible, but be aware of the consequences.

Example:
`de.uniba.dsg.concurrency.examples.interruption.PlayInterrupted` contains a mixture of both strategies. It uses the preservation when waiting for the lock shown in Listing 31 and uses the interruption for termination purposes.

4.3 Thread Safety

Thread Safety is an important concept, but there are many definitions around which are only partly correct from our point of view. Therefore, we define two aspects, which make a class thread safe[40].

[40]For further information and as a source for our definition, a good article of Brian Goetz in online here: (`https://www.ibm.com/developerworks/java/library/j-jtp09263/index.html`). Note that point two of our definition is only mentioned indirectly by Brian's article.

Our definition of Thread Safety

- A class is thread-safe, if all method accesses, in particular concurrent accesses, lead to a consistent view (READ) and state (WRITE) of the object's internal state.
- The class must secure access of mutable data from unsecured memory spaces, i.e. input objects (constructor or set methods etc.) must be copied (deep copy). The reference of internal values is not allowed to cross the object's border (so only the object itself should have a reference (and the only reference) to internal state objects). If there is a need to share internal state objects, the internal state object must be copied (deep copy) and the copied object will be returned by the corresponding method.

Thread Safety includes a view on the internal data of an object, but also the references to internal objects. If I want to share objects or use input data, I have to do a deep copy of that data. (Think about the heap and stack memory model.)

Hint: for deep copying stuff, look at the immutability example in Section 1.3.

Hint: The difference of immutable and thread-safe classes is that an immutable class must be declared as final, whereas other classes can inherit from thread-safe classes. The second and even more important difference is that internal state changes do not lead to a new object for thread-safe classes. This is the case because the synchronized methods and blocks secure the state changes and lead to a consistent view on the internal state without synchronization issues, like lost updates and race conditions.

Thread Safety Documentation JavaDocs has no keyword for documenting the thread safety status of a class. *Synchronized* and other keywords are implementation specific and not part of the generated JavaDoc. Due to this limitation, Brian Goetz published a few thread safety annotations[41] in combination with his book on concurrency programming[42]. We combine the thread safety annotations and the different level of thread safety, included in "Effective Java, Third Edition page 330", in the following enumeration:

1. **Immutable** – As we already know from Section 1.3, immutable objects don't need any type of synchronization since their state is safe when modifying it concurrently. Examples are: Integer, String . . .
   The corresponding thread safety annotation is `@Immutable`
2. **Unconditionally thread-safe** – Classes which encapsulate their thread safety policy, e.g. using a *private final Object mutex;*.
   Examples are: Random, ConcurrentHashMap . . .
   The corresponding thread safety annotation is `@ThreadSafe`
3. **Conditionally thread-safe** – As the name already implies, the class is only thread-safe under some conditions which a developer using this class has to think about. All state changing methods are synchronized, but their combination is not! So external

[41] These annotations do not have any implementation semantics. They are only for documentation purpose.

[42] JavaDocs of the thread safety annotations can be found here: `http://jcip.net.s3-website-us-east-1.amazonaws.com/annotations/doc/index.html`

synchronization is needed to enable some compound actions as we already have seen in the corresponding paragraph in Section 4.1.2.
Examples are: iterators of *Collection.synchronizedXXX* implementations.
For example, *SynchronizedList* uses its self-reference as the mutex lock object and therefore offers the developer the possibility to implement compound actions. However it also introduces concurrency bugs when a developer is not aware of this.
("It is imperative that the user manually synchronize on the returned list when traversing it via Iterator, Spliterator or Stream. [...] Failure to follow this advice may result in non-deterministic behavior."[43])
The corresponding thread safety annotation is `@NotThreadSafe`[44].

Example:
We included a sample which shows compound actions and the iterator problem mentioned here.
`de.uniba.dsg.concurrency.examples.highlevel.SynchronizedListSample` contains the code. The implementation does not show an iterator but includes a similar operation, where we want to check if my list contains some elements and then remove the first element. It shows the necessity for compound actions by stating the problem and the solution. For sake of simplicity the source code from the problem was copied and fixed.

4. **Not thread-safe** – Class is not thread-safe at all. A developer has to care about all the synchronization stuff and has to enable his synchronization policy when using the class.
   Examples are: ArrayList, HashMap . . .
   The corresponding thread safety annotation is `@NotThreadSafe`

In addition to these three thread safety annotations, Brian Goetz also introduced a `@GuardedBy` annotation[45]. The value of this annotation is the lock object, for self-referencing lock object *this* is used. Listing 34 concludes this chapter about thread safety and their documentation with an example.

Listing 34: Documentation of Thread Safety and Attribute's Synchronization

```
1 @ThreadSafe
2 public class MyClass {
3   @GuardedBy(value = "this")
4   private int value;
5
6   public synchronized int useValue() {  value++;  }
7 }
```

[43] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html#synchronizedList(java.util.List)`

[44] This is a strict definition of thread-safety, but in some cases, concurrency errors can happen, when thinking about the check of a thread A if the list does contain a single element and it does. The next statement of A is to remove the element, but in the meantime between checking the size and removing the item, thread B removes the item. Therefore, thread A tries to remove it, but there is no item any more, which results in a *IndexOutOfBoundsException*.
See *SynchronizedListSample.java* in our example project code.

[45] This annotation is somehow special since it documents also which lock object and construct's synchronize internal state etc. For a detailed explanation and a sample have a look at `http://jcip.net.s3-website-us-east-1.amazonaws.com/annotations/doc/net/jcip/annotations/GuardedBy.html`.

Exercise:
`de.uniba.dsg.concurrency.exercises.documentation` contains some examples, where you can think about what thread safety annotation is needed for the classes. Also use the *@GuardedBy* annotations. We included another package `documentation.solution` where you can find the solution for the tasks.

5 High Level Concepts

As you have seen in the Sections 3 and 4, concurrency programming when using low level constructs introduced with Java 1.0 is quite hard, and doing it right is even harder. Therefore, we present a selection of high level API classes in this section and link to the most comparable low level implementation in the previous sections.

5.1 *Semaphor*

"Conceptually, a semaphore maintains a set of permits. Each *acquire()* blocks if necessary until a permit is available, and then takes it. Each *release()* adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly" [46].

When used right, a semaphore enables mutual exclusion and therefore critical sections like in Section 4.1.2. With a single permit, the semaphore acts like a lock. All acquiring threads have to wait until some thread releases the semaphore.

After acquiring the semaphore, you have to take care, that the semaphore is released properly, therefore we suggest the following (save) release:

Listing 35: Acquire and savely release a Semaphore

```
1  public void doCriticalStuff() {
2    semaphore.acquire();
3    try {
4      // CRITICAL SECTION
5      // only one thread active in here guaranteed by semaphore
6      // if semaphore impl correct ;)
7    } finally {
8      semaphore.release();
9    }
10 }
```

As documented, the semaphore contains a set of permits. This is useful for resources with a bounded capacity like database connections, where only a fixed size of clients can access the resource.

[46] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Semaphore.html`

Exercise:
Implement an *UnfairBooleanSemaphore*, which can be used as shown in the previous Listing 35 on your own. Use only low level constructs for that (*wait/notify/synchronized*).
`de.uniba.dsg.concurrency.exercises.semaphore` contains the template source code. You do NOT have to care about fairness (so that the first thread waits to acquire the lock has to be the next one which can acquire the lock. This is quite hard to support).There is an additional Code Runner Task where you can check your implementation.

5.2 Producer-Consumer Paradigm

Producer Consumer Paradigm is one of the most important communication paradigms in computer science. The benefit of such an architecture is the decoupling of producer and consumer through the queue. Figure 10 shows the two communication partners and the queue.

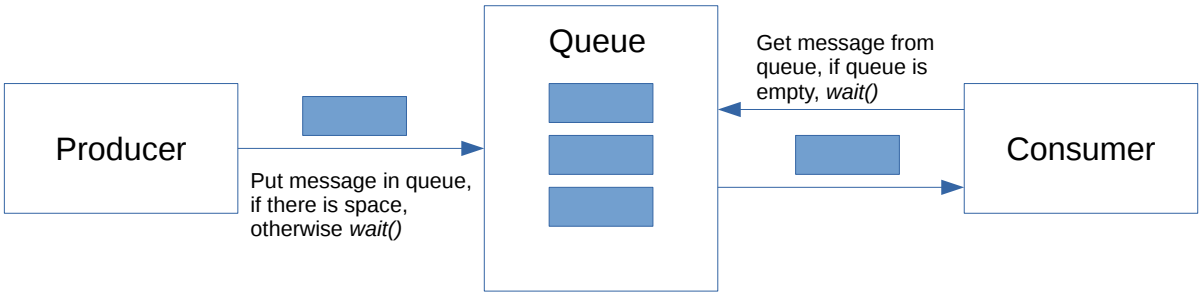

Figure 10: Producer Consumer Communication Paradigm

A problem are the two waiting conditions, for a producer when the queue is full and for a consumer when the queue is empty. Without the *wait/notify* mechanism (see Section 4.1.2) a producer has to spin-wait (consuming CPU and other resources without doing something useful despite producing heat) until an item in the queue is available.

Therefore, the Java API has some BlockingQueues (e.g. *java.util.concurrent.ArrayBlockingQueue*[47]), which work with *wait/notify* in their implementations.

The two most important methods are *put(E entry)* and *E take()* and are self-explanatory. For an orderly shutdown the producer has to send a termination message, whereas the consumer has to react on this specific message and terminates.

[47] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ArrayBlockingQueue.html`

Exercise:
Implement a simple blocking queue, where the queue has for sake of simplicity only a single field. Implement the interface *BlockingQueue and your class SimpleBlockingQueue* as generic as possible. Only use low level constructs like *wait/notify* and *synchronized*.
`de.uniba.dsg.concurrency.exercises.semaphore.queue` contains the source code. There is an additional Code Runner Task where you can check your implementation.

5.3 Manged Runtime Frameworks

Motivation Why do we bother us with another framework/API mess? Why not using the low level stuff we discussed in shared memory (Section 4)?

The second question already contains the motivation. Since the low level stuff is hard to grasp and a lot of nitpicky errors can be made, we introduced some high level concepts for locking (semaphore) and communication (producer - consumer). But what about thread management. Do we really need to start a thread for each runnable or is there an easier solution?

If we do not use a management facility for threads, we face urgent performance problems. Most important is the overhead for creating and terminating threads. Since each thread consumes memory and CPU cycles, there is a stability limit for the number of threads on a system.

So what about reusing threads?

5.3.1 Executor Service and Thread Pools

An *Executor* is "An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanism of how each task will be run, including details of thread use, scheduling, etc. An executor is normally used instead of explicitly creating threads."[48] It is another layer of abstraction for executing tasks.

Listing 36: Executor Interface

```
1 package java.util.concurrent;
2
3 public interface Executor {
4   void execute(Runnable var1);
5 }
```

An *ExecutorService* implements the *Executor* interface and adds life cycle methods for e.g. termination, thread pool handling, queuing of the tasks etc. With an executor service, you do NOT have to care about basic thread management at all :)

[48] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executor.html`

Listing 37: Executor Service Interface

```
1  package java.util.concurrent;
2
3  import java.util.Collection;
4  import java.util.List;
5
6  public interface ExecutorService extends Executor {
7
8    // life cycle methods
9
10   void shutdown();
11
12   List<Runnable> shutdownNow();
13
14   boolean isShutdown();
15
16   boolean isTerminated();
17
18   boolean awaitTermination(long timeout, TimeUnit unit) throws
19     InterruptedException;
20
21   // further methods discussed later on
22 }
```

Listing 37[49] shows all life cycle methods of an executor service.

- **void shutdown()** – Shuts the executor service down. Executing tasks (already executing tasks and submitted ones) are unaffected, but the executor service does not accept new submitted tasks. Important to note here is that this method does not block until all executing tasks are finished, therefore use *awaitTermination(timeout, unit)*.
- **List<Runnable> shutdownNow()** – Tries to terminate all running tasks and returns a list of already submitted but not yet started tasks. Important to note here is that this method does not block until all executing tasks are finished, therefore use *awaitTermination*. As *shutdown*, it does NOT accept newly submitted tasks. The **"termination" is implemented via an interruption of all running tasks**, since this is a cooperative mechanism, it is NOT a guarantee that the executor service terminates. If the implemented runnables are not sensible for interruption, nothing will happen (also read Section 4.2 for information about interruptions).
- **boolean isShutdown()** – checks, if *shutdown()* or *shutdownNow()* was called. The executor service may not be terminated.
- **boolean isTerminated()** – checks, if the executor service is already terminated.
- **boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException** – Blocking method, see Section 4.2.2. Waits for a specific amount of time to give the tasks a chance to terminate properly.

[49] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html`. See also the Java doc for further method infos. There you can also find a good shutdown policy for executor services. Adjust the time unit and timeout setting!

HINT: Without calling a shutdown method, the executor service does NOT terminate and therefore also the JVM does NOT.

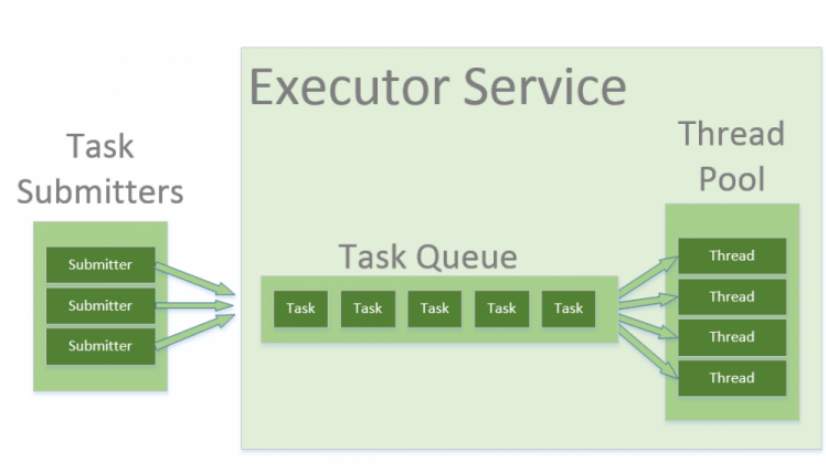

Figure 11: Executor Service Illustration

Figure 11[50] depicts how an executor service is working and summarizes the aforementioned points (life cycle management, reusing of threads within the tread pool ...). What's especially interesting is the task queue (Producer-Consumer, Section 5.2), which decouples submitters and the executor service and enable some optimized algorithms we will see later on.

So the final question is, how to instantiate an executor service and which types are available for usage?

Java offers factory methods for creating already implemented executor services (and normally this set of executor services is sufficient for most use cases). These methods are included in the *Executors* class[51]. The following enumeration contains the most important ones and adds a short explanation. An instantiation can be done via the following command. Return type is for all methods an instance of an executor service:

Listing 38: Create an Executor Service via Factory Methods

```
1 ExecutorService exec = Executors.newCachedThreadPool();
```

- **newFixedThreadPool(int poolSize)** – Pool with a maximum number of *poolSize* threads. The executor service starts threads on demand until it reaches this limit. Once started, the threads are kept for later usage.
- **newCachedThreadPool()** – Number of threads is dependent on the incoming number of tasks. The executor service can dynamically scale up and down threads. (Often the best choice)
- **newSingleThreadExecutor()** – Executor with a single thread. Tasks are executed in strict order (means the order in which they arrived at the executor service). (It's in most cases better to use a single threaded executor service then starting a thread by hand.)

[50] Figure copied from: `https://www.baeldung.com/thread-pool-java-and-guava`

[51] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html`

- **newScheduledThreadPool(int poolSize)** – Thread pool with a fixed number of threads which enables submitting scheduled tasks.
  Caution: If all threads are busy and you execute another task, this task has to wait for execution and is therefore not executed at the slot you intended!

- **newWorkStealingPool()** – Uses a ForkJoin pool, for further information see the API Docu, to execute the tasks. Good choice for executing asynchronous tasks.[52]

Exercise:
`de.uniba.dsg.concurrency.exercises.executor` package contains the source code for the next exercise. Implement the class *SummationRunnable*, description is included in the file. Also implement *ExecutorComparison*, where you compare the above mentioned executor services (despite the scheduled one, which makes no sense here). Also take care that the program/JVM exits properly (think about a reasonable termination policy).

5.3.2 Futures and Callables

Now we have a managed thread execution environment where we can submit runnables. You may say - that's fine, we do not have to care about any nitpicky thread stuff, but what about return values of long running tasks? Executing tasks without getting the result back is only half of the story.

To enable this, *Callables* are introduced which have the following interface, Listing 39 presents[53]:

Listing 39: Callable Interface

```
1 package java.util.concurrent;
2
3 @FunctionalInterface
4 public interface Callable<V> {
5   V call() throws Exception;
6 }
```

Like Runnables, Callables can be executed concurrently, but how do we get the return value? Therefore another class is introduced, which gives us a handle to the return value - *Futures*.

"A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a

[52] Work stealing algorithm uses a double ended queue. If a thread does all its stuff it can steal from another's thread queue and execute the stolen task.

[53] If you are not aware, what a FunctionalInterface is, we recommend to read `https://www.baeldung.com/java-8-functional-interfaces` for a first overview how functional programming in Java works

usable result, you can declare types of the form Future<?> and return null as a result of the underlying task."[54]

When using an executor service, we can submit our callable and get a future back.

```
Future<MyResult> future = executor.submit(myCallable);
```

Important methods of Future<V> are as follows:

- *boolean isDone()* – "Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation – in all of these cases, this method will return true." (Java Doc)
- *V get() throws InterruptedException, ExecutionException* – "Waits if necessary for the computation to complete, and then retrieves its result." (JavaDoc) So get is a blocking method. It also throws an unchecked *CancellationException*, if the task was canceled.
- *boolean cancel(boolean mayInterruptIfRunning)* – "Attempts to cancel execution of this task. This attempt will fail if the task has already completed, has already been cancelled, or could not be cancelled for some other reason. If successful, and this task has not started when cancel is called, this task should never run. If the task has already started, then the *mayInterruptIfRunning* parameter determines whether the thread executing this task should be interrupted in an attempt to stop the task." (JavaDoc)

Now we know what a future is and how to implement them, but how can we execute the Future? Listing 40 completes the API method overview of Listing 37 and shows how to invoke or submit Runnables and Callables. For a detailed method overview look at the API docu.

Listing 40: Executor Service Interface completed

```
public interface ExecutorService extends Executor {
  // life cycle methods . . .

  // methods to execute and submit runnables and callables

  Future<?> submit(Runnable r);

  Future<T> submit(Callable<T> c);

  List<Future<T>> invokeAll(Collection<? extends Callable<T>> c)
        throws InterruptedException;

  T invokeAny(Collection<? extends Callable<T>> c)
        throws InterruptedException, ExecutionException;

  // execute method from the Executor interface
  void execute(Runnable r);   // already discussed
}
```

[54] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Future.html

- *Future<?> submit(Runnable r)* – "Submits a Runnable task for execution and returns a Future representing that task."[55] Beneficial when you want to wait on a certain point for the completion of a specific runnable via the Futures#get() method.
- *Future<T> submit(Callable<T> c)* – "Submits a value-returning task for execution and returns a Future representing the pending results of the task." That's what a callable promised. submit (for runnables and callables) do NOT block!
- *List <Future <T> > invokeAll(Collection<? extends Callable <T> > c) throws InterruptedException* – "Executes the given tasks, returning a list of Futures holding their status and results when all complete." An important point here is, that differently to submit and execute, invokeAll blocks (!) until all tasks completed their executions.
- *Future<T> submit(T invokeAny(Collection<? extends Callable <T> > c) throws InterruptedException, ExecutionException* – "Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do." This method also blocks until the first computations successfully terminates.[56]

Exercise:
`de.uniba.dsg.concurrency.exercises.futures` of the future task. The descriptions are included in the corresponding files. You have to implement the *call()* method of the class *RandomNumberCallable*. Compute a random number via (*new Random(2000).nextInt()*), let your current thread sleep that amount of random millis and return the random value to the caller. If the thread gets interrupted during its sleep, terminate immediately to be as responsive as possible.

5.3.3 A Running Example

The following listing summarizes this section.

Example:
A more extensive example is included in our example project. `de.uniba.dsg.concurrency.examples.highlevel.ExecutorSample` contains the source code. There are two executor service implementations used (one is uncommented). Try to understand, why a value for the fixed one under 11 is problematic? Also question yourself why the cached service is that fast!

[55] All quotes are copied from the API documentation, see `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html`.

[56] For *invokeAll* and *invokeAny*, there also exist some methods, where you specify a time out how long the thread is waiting for all or one result. See the API docu for further infos.

Listing 41: ExecutorService - Runnable - Callable and Future

```
1  public static void main(String[] args){
2
3    // chached thread pool is
4    ExecutorService executor = Executors.newCachedThreadPool();
5
6    // callable and two runnables
7    Future<Integer> handle = executor.submit(new ShareResultCallable());
8    executor.execute(new DoSomethingRunnable());
9    executor.execute(new DoSomethingCriticalRunnable());
10
11   try {
12     System.out.println("Callable return value: " + f.get());
13   } catch (InterruptedException e) {
14     // restore the flag, if a succeeding method also blocks
15     Thread.currentThread().interrupt();
16   } catch (ExecutionException e) {
17     // log exception to error stream
18     exceptionList.add(e);
19   }
20
21   // copied from the API Docu
22   shutdownAndAwaitTermination(executor);
23 }
```

5.4 Other Useful API Stuff

5.4.1 Atomic Classes Primitive Datatypes

For simple use cases like counters etc., Java offers in its *java.util.concurrent.atomic* package some wrapper for primitive datatypes (int -> AtomicInteger). See the package docu [57] for further information.

5.4.2 Thread Safe Data Structures

WARNING: We talk about thread safe data structures now, but also think about the underlying data. See for an extensive discussion Sections 1.2.1 and 4.3.

External Synchronization needed for some classes As already explained in Section 4.3 for conditionally thread safe classes, some operations, e.g. the iterators, need external synchronization. Wrapper implementation for non thread safe datastructures (lists, maps, sets etc.) can be found in the *Collections* class[58].

[57] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

[58] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html

Internally synchronized data structures Since errors happen easily due to the *conditionally* thread safety by the former introduced set of classes, Java 1.5 added a lot of *Concurrent* implementations for maps, sets etc. Have a look in the package summary for more details (also the explanation at the beginning is worth to read) [59].

[59] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html`