CONCURRENCY

CONTENT

- Threads
- Creating threads
- Synchronizing
- Concurrent collections
- Parallel streams
- Concurrent process management
- Threading problems

THREADS

- Path of execution, usually multiple tasks, a thread can execute one task at a time
- Process consist of at least one thread, but usually more
- Threads can be executed at the same time

- Example: counting all the votes after an election
  - Single-threaded process: If Maaike would do all the counting, it would take a really long time
  - multi-threaded process: If in every municipality (around 355 in the Netherlands) someone will do the counting for their municipality and this is all summed by Maaike, it goes a lot faster. The process is not finished until the last municipality has been counted. Each municipality represents a thread, all the threads represent a process

THREAD TYPES

- System thread: created by JVM, e.g. garbage collector
- User-defined thread: created by developer to complete tasks/processes
- Single-threaded process: process with only one user-defined thread

THREAD CONCURRENCY

- Concurrency: executing threads at the same time
- More threads at the same time than CPUs available is common, this is done with a thread scheduler
- Thread scheduler: OS uses this to decide which thread should be using the CPU
- Context switch: switching thread, storing the state and continuing with the next one. The thread that was switched from will be dealt with later.
- Thread priority: threads have an integer value that specifies priority. High priority threads will interrupt or be dealt with before low priority threads.

RUNNABLE

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

The interface is used to specify the work that a thread should do. Class implements Runnable and then specifies what should be done in the overwritten method run().

CREATING A THREAD

- Create instance of java.lang.Thread
- Define tasks that need to be done by thread:
  - Option A: Provide a runnable object or implementation to the Thread constructor
  - Option B: Make a class extend Thread and override run() method
- Start the thread with Thread.start() method

- Question: What is general best practice: option A or option B? Why?

EXAMPLE A: PROVIDE A RUNNABLE OBJECT OR ANONYMOUS IMPLEMENTATION TO THE THREAD CONSTRUCTOR

```
public class RunnableExample implements Runnable {
        @Override
        public void run(){
                //some tasks
        }

        public static void main(String[] args) {
                (new Thread(new RunnableExample())).start();
        }
}
```

EXAMPLE B: MAKE A CLASS EXTEND THREAD AND OVERRIDE RUN() METHOD

```
public class ThreadExample extends Thread {
        @Override
        public void run(){
                //some tasks
        }

        public static void main(String[] args) {
                (new ThreadExample()).start();
        }
}
```

EXERCISE: CREATING THREADS

- Create two threads
- Create two different classes that define the work that needs to be done:
  - RunnableClass (using Runnable interface)
  - SubThread (using Thread class)
- Make the class print the thread id
- Start the threads

WHAT DOES THIS PRINT TO THE CONSOLE?

```
public class RunnableExample implements Runnable {
        @Override
        public void run(){
                System.out.println("Hello");
        }

        public static void main(String[] args) {
                (new Thread(new RunnableExample())).start();
                System.out.println("world");
        }
}
```

QUESTION: HOW MANY (USER-DEFINED) THREADS?

```
public class ThreadExample extends Thread {
        @Override
        public void run(){
                //some tasks
        }

        public static void main(String[] args) {
                (new ThreadExample()).start();
        }
}
```

WHAT DOES THIS PRINT TO THE CONSOLE?

```
public class RunnableExample implements Runnable {
        @Override
        public void run(){
                System.out.println("Hello");
        }

        public static void main(String[] args) {
                (new Thread(new RunnableExample())).run();
                System.out.println("world");
        }
}
```

POLLING

- Polling is checking data / state at fixed intervals
- Interval can be set by calling the sleep method on the thread that is polling, this can be done by Thread.sleep(*nrOfMilliseconds*)
- Sleep might throw the checked InterruptedException, so this need to be caught or added to the method declaration in which sleep is called

EXERCISE: SLEEP

- Use sleep to create an InterruptedException

- Why does this happen?
- And how to use sleep?

EXECUTORSERVICE

- Interface that is part of the Concurrency API
- Creates and manages threads for the developer

- Steps:
  - Get an instance of ExecutorService
  - Execute tasks using the ExecutorService
  - Close ExecutorService

EXAMPLE: WHAT DOES THIS PRINT?

```
public static void main(String[] args) {
        ExecutorService executorService = null;
        try {
                executorService = Executors.newSingleThreadExecutor();
                System.out.print("Hello ");
                exectorService.execute(()-> System.out.print(" world"));
                exectorService.execute(()-> System.out.print("!"));
        } finally {
                if(executorService != null) {
                        executorService.shutdown();
                }
        }
}
```

.SHUTDOWN()

- Necessary, because it creates a non-daemon thread, and the application would not terminate if this thread won't be killed with .shutdown().
- ExecutorService will finish tasks that are submitted to it before shutting down, but it won't accept new tasks after .shutdown() has been called
- Submitting tasks to an ExecutorService after .shutdown has been called will result in a RejectedExecutionException
- Cannot be closed with try-with-resources, because it doesn't implement AutoCloseable

.SHUTDOWNNOW()

- Tries to stop running tasks and doesn't start tasks that have yet been submitted, then it shuts down the executor service
- Returns a list of tasks that were submitted but have never started
- No garantees on whether it succeeds to stop running tasks

SUBMITTING TASKS FOR EXECUTION USING THE EXECUTORSERVICE

Method	Additional info
void execute(Runnable r)	
Future<?> submit(Runnable r)	Returns Future of the task
<T> Future<T> submit(Callable<T> c)	Returns a Future representing the pending result
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> c) throws InterruptedException	Executes tasks synchronously, returns all the tasks in the order they were in the input parameter
<T> T invokeAny(Collection<? extends Callable<T>> c) throws InterruptedException, ExecutionException	Executes tasks synchronously, returns the result of one finished tasks and cancels unfinished tasks

FUTURE OBJECT

- java.util.concurrent.Future<V> object
- Returned by methods executing tasks
- Can be used to see the result of the executed tasks

- Methods on Future object can be called:
  - isDone() >> true if task is completed (completed may be cancelled, done, exception)
  - isCancelled() >> true if it was cancelled
  - cancel() >> tries to cancel the task
  - V get() >> gets result, and waits forever it is not available (also one available that takes a specified amount of time and throws a TimeoutException if it's not done after waiting the specified amount of time)

EXERCISE: POLLING WITH EXECUTORSERVICE AND FUTURE OBJECT

- Create a class that contains the dishes to do
- Write an ExecutorService with a single thread
- Use this ExecutorService to perform doing the dishes using Future object
- Use polling to check whether the dishes are done

CALLABLE

- Interface java.util.concurrent.Callable
- Like Runnable, but the method is call()
- call() returns a value
- call() can throw checked exceptions
- Usually preferred over Runnable

- Exercise:

Write a class with an ExecutorService that submits using a Callable that concats two Strings

Catch the result in a Future

Print the result

EXERCISE: CALLABLE

- Rewrite the previous assignment with a callable

SCHEDULEDEXECUTORSERVICE

- Subinterface of ExecutorService

- Steps:
  - Get an instance of ScheduledExecutorService (newSingleThreadScheduledExecutor())
  - Schedule and execute tasks using the ScheduledExecutorService
  - Close ScheduledExecutorService

- Methods:
  - schedule(Callable<V>/Runnable c, long delay, TimeUnit tu)
  - scheduleAtFixedRate(Runnable r, long initialDelay, long fixedRate, TimeUnit tu)
  - scheduleAtFixedDelay(Runnable r, long initialDelay, long fixedDelay, TimeUnit tu)

CONCURRENCY AND POOLS

- Thread pool: group of reusable threads that will perform tasks that need to be executed
- Single thread vs thread pool: single thread can only do one task at a time and then needs to wait until thread becomes available again. Thread pool can do as many tasks at the same time as it has threads. After that tasks will be queued and pciked up when a thread becomes available.

CONCURRENCY AND POOLS

Returns	Method	Description
ExecutorService	newSingleThreadExecutor()	
ScheduledExecutorService	newSingleThreadScheduledExecutor()	
ExecutorService	newCachedThreadPool()	Creates pool that will add threads as needed and reuses available ones
ExecutorService	newFixedThreadPool(int nrOfThreads)	Creates a pool with a fixed number of threads that can be reused
ScheduledExecutorService	newScheduledThreadPool(int nrOfThreads)	Creates a thread pool of a certain size that schedules tasks after a delay or to execute after a certain period

EXERCISE: LET'S COUNT

- Create a class that holds a counter and a method that increments this counter
- Make multiple threads call this method and increment the counter and print the value of the counter

SYNCHRONIZING DATA ACCESS

- Problem arises when two threads read the same data at the same time, and then update it and overwrite each others' results
- java.util.concurrency.atomic contains classes that give access to primitives and references in a concurrency safe way
- E.g. incrementing: atomic increment considers the read and write one single operation that cannot be interrupted by other processes

ATOMIC CLASSES

Atomic class	
AtomicBoolean	
AtomicInteger	
AtomicIntegerArray	
AtomicLong	
AtomicLongArray	
AtomicReference	Generic object reference that can be updated atomically
AtomicReferenceArray	

METHODS ON ATOMIC CLASSES

- Get()
- Set()
- getAndSet()

For numeric classes:

- incrementAndGet() (++value)
- getAndIncrement() (value++)
- decrementAndGet() (--value)
- getAndDecrement() (value--)

EXERCISE: LET'S COUNT

- Replace the type of the counter with an atomic type.
- What happens?
- What about the order?

SYNCHRONIZED BLOCKS

- Solves the order problem of the previous exercise
- Increases data integrity
- Allows only one thread at a time in the block
- Monitor/lock for synchronized access

```
synchronized(objectInstanceOrClassToBeSynchronized) {
//task for one thread at a time
}
```

EXERCISE: LET'S COUNT

- Improve the counter using the synchronized keyword
- How long does it take to count to 100.000? And to 1.000.000?

EXERCISE: SYNCHRONIZED BLOCKS

Improve the previous exercise by adding synchronized to ensure the right order

Bonus question: do you still need the atomic class?

SYNCHRONIZED METHODS

- public synchronized void doSomething(){} >> synchronizes the instance for the method is being called upon
- public static synchronized void doSomething(){} >> synchronizes the class object for the method that is being called
- Synchronized slows down the application, since the threads need to wait for each other to enter the synchronized code

CONCURRENT COLLECTIONS

- Java Collection Framework has concurrent alternatives for the most popular collection types
- Concurrent collections are safe for multi-threading
- All of this could be achieved with synchronized keyword, but concurrent collections are optimized for performance and avoid mistakes

CONCURRENTMODIFICATIONEXCEPTION

```
List<String> ls = new ArrayList<>();
ls.add("hoi");
ls.add("hey");

for (String s: ls) {
    ls.add(s);
}
```

NO CONCURRENTMODIFICATIONEXCEPTION

```
List<String> ls = new CopyOnWriteArrayList<>();
ls.add("hoi");
ls.add("hey");

for (String s: ls) {
    ls.add(s);
}
```

CONCURRENT COLLECTION CLASSES

Class	Java Collection Framework Interface
ConcurrentHashMap	ConcurrentMap
ConcurrentLinkedDeque	Deque
ConcurrentLinkedQueue	Queue
ConcurrentSkipListMap	ConcurrentMap, SortedMap, NavigableMap
ConcurrentSkipListSet	SortedSet, NavigableSet
CopyOnWriteArrayList	List
CopyOnWriteArraySet	Set
LinkedBlockingDeque	BlockingQueue, BlockingDeque
LinkedBlockingQueue	BlockingQueue

Concurrent classes

- Create a customer class. This class should have two types of groceries lists.
  - Create a concurrent set for the groceries you need to do this week. Add some groceries to the list and loop through the list printing all the groceries.
  - Create a concurrent map for the groceries. The key should be the String of the name of the item and the value should be the number of items you need.
- Get the groceries on the list multithreaded (with two “persons”, in two threads)
- Create a main app with a concurrent queue for the line at the checkouts in the supermarket. Write some code that helps the next customer in line with the groceries. There are multiple checkouts. Write some context to test this process.

- What conclusion can you draw about using concurrent classes compared to using non-concurrent classes?

BLOCKINGQUEUE AND BLOCKINGDEQUE

- Inherits all the methods from Queue and Deque respectively.
- Have in addition methods that take two extra argument, a long and one of type Timeunit; the methods wait the long amount of time unit to complete the action.

- E.g.: poll(long timeout, TimeUnit unit)

SKIPLIST COLLECTIONS

- Sorted concurrent collections
- ConcurrenSkipListSet > concurrent version of TreeSet
- ConcurrentSkipListMap > concurrent version of TreeMap
- Assign to interface references
  - SortedSet<String> set = new ConcurrentSkipListSet<>();

COPYONWRITE COLLECTIONS

- Every time an element is added, removed or changed (reference, not the content of the element), all the elements are copied to an underlying structure
- An iterator will look at the unchanged original structure and can iterate while the collection changes
- Can use a lot of memory when many writes to the collection are being performed. Particularly useful when there are many reads but not as many writes

Exercise: concurrent and non-concurrent class

- Create a concurrent List and add your favorite foods to this list
- Loop through this list and in the loop, copy every item on the list and add it to the list as well
- Count the items in the list and print this in the loop
- Count the items in the list and print this outside the loop

- What would happen if you do this with a non-concurrent class?

SYNCHRONIZED COLLECTIONS

- In the concurrency API methods exist to obtain a synchronized version of a non-concurrent collection
- E.g.:
- List<String> list = Collections.synchronizedList(someListName);

SYNCHRONIZED COLLECTIONS METHODS

MethodName
synchronizedCollection(Collection<T> c)
synchronizedList(List<T> list)
synchronizedMap(Map<K, V> m)
synchronizedNavigableMap<NavigableMap<K,V> m)
synchronizedNavigableSet<NavigableSet<T> s)
synchronizedSet(Set<T> set)
synchronizedSortedMap<SortedMap<K,V> m)
synchronizedSortedSet<SortedSet<T> s)

PARALLEL STREAMS

- Streams API have built-in concurrency
- They make it possible to not only work with serial streams, but also with parallel streams
- Parallel stream uses multiple threads to process concurrently
- Increases performance of application
- Might lead to unexpected results

HOW TO CREATE PARALLEL STREAMS?

- Call .parallel() on a stream
- Or create a stream with .parallelStream() (instead of .stream())

EXERCISE: STREAM TERMINAL OPERATIONS CAN PARALLELSTREAM IMPROVE ANY OF THESE?

Create an infinite stream of numbers and a finite stream of Strings and test each by using the following functions to do:

allMatch() >> write to find if all have the third letter of the string being an F / are higher than 6

anyMatch() >> find any number between 77 and 100 and any string starting with a T

noneMatch() >> see if none match a string longer than 25, see if none match 5

collect() >> collect all the string in a stringbuilder and sum all the even integers

count() >> count the stream and print the result

findAny() >> see what happens for empty and when there are values

findFirst() >> idem

forEach() >> see what happens for printing on the finite and infinite stream

min() >> find the shortest String

max()>> find the longest String

reduce() >> reduce the string stream in a long string consisting of the first letters of all string

PARALLELSTREAM EXERCISE (REWRITE THE OLD ONE)

- Create a stream from a list with cats
- Remove the cats with long hair (or short, whatever you prefer)
- Remove duplicates
- Sort the cats on first name, reversed alphabetical order
- Create a list of these cats

- Create a stream of sequential odd numbers starting at 1 (1,3,5,etc)
- Only have 100 odd numbers in the stream
- Remove all the non-prime numbers
- Remove all numbers above 100
- Count the prime numbers under a 100

MANAGING CONCURRENT PROCESSES

- CyclicBarrier
- Fork/Join framework
- RecursiveTask

CYCLICBARRIER

- Makes it possible to pause between to sets of tasks
- Used when there is a fixed number of threads that must wait for eachother
- The moment during execution when the threads must wait is called the barrier
- It is cyclic, because it can be re-used when the threads are done waiting

- Exercise: use CyclicBarrier to make Java replace the window frames in the house, do the sanding and paint them with a fixed thread pool of 4

FORKJOINPOOL

- Recursion to do task > task calls itself to do the task
  - Base case – non-recursive method, else the process would never be terminated
  - Recursive case – calls itself once or more to solve the problem

How to use:

1. Create ForkJoinTask (recursive process definition)
2. Create ForkJoinPool (one line of code)
3. Start ForkJoinTask (one line of code)

RECURSIVEACTION AND RECURSIVETASK

- Both implement ForkJoinTask interface
- Extend one of these to create a ForkJoinTask
- Implement compute() method
- RecursiveAction: compute returns void
- RecursiveTask: compute returns generic object

EXERCISE

- Create a ForkJoinTask for these two cases:
  - Counting all the votes in the Dutch elections
  - Picking up all the trash after a festival

- Did you use RecursiveAction or RecursiveTask? And why?

THREADING PROBLEMS

- Liveness: ability of application to execute in an acceptable amount of time

- Deadlock: two or more threads block each other forever
- Starvation: one thread doesn't get access to a shared resource or locked block of code
- Livelock: special starvation, both alive trying to complete the task but fail and restart the process. Common as a result of the effort to overcome a deadlock.
- Race conditions: two tasks completed at the same time that should be sequential (e.g. two users signing up with same name)

THREADING PROBLEMS

- Prepare a slightly more elaborated explanation of the threading problem in pairs
- Attempt to write code to showcase the threading problem
- You'll have 25 minutes to prepare + max 5 minutes to present your threading problem