# Threads 3

- Asynchronous code in Java with Futures and Callables and Executor framework


- Litterature: The Well-Grounded Java developer

# Learning goals until now

- Threads
  - What is concurrency?
  - What is: a thread, race-conditions, deadlocking, producer/consumer, observables?
- Network
  - What is the TCP/IP stack?
  - What is: port, socket, IP, TCP, UDP, HTTP and HTTPS?
- Virtualisation
  - What is virtualisation?
  - What is: load balancing, reverse proxy, Nginx?

# Learning goals for today

- Understand and use modern Java Concurrency features
  - Immutability
  - Thread pools
    - Executors and fork-join
  - Futures and Callables
  - Completable Futures

# Agenda

- Recap on Threads
- Concurrent design concepts
- Immutability
- Threads vs Tasks
- Callables
- Futures
- Completable Futures

# Spørgsmål:

- The three different ways to create/start threads
- The purpose of the join command (on a Thread instance)
- When and why we will use Threads in our programs?
- Explain about the Race Condition Problem and ways to solve it in Java
- Explain how Threads can help us in making responsive User Interfaces
- Explain about deadlocks, how to detect them and ways to solve the Deadlock Problem

# Synchronized

- Locks on what?
  - Only objects—not primitives—can be locked.
  - Locking an array of objects doesn't lock the individual objects.
  - A static synchronized method locks the Class object, because there's no instance object to lock
  - Unsynchronized methods don't care about any locks. They progress while synchronized methods are running
  - Java's locks are reentrant: Threads can take the same lock many times.

# Concurrent design concepts

- Safety (also known as concurrent type safety)
  - Thread-safety: Consistent state
- Liveness
  - Temporary failures
    - Locking, waiting, starvation
  - Permanent failures
    - Deadlocks, unrecoverable failures
- Performance
- Reusability

# Immutability

- Immutability = unchangeable != mutable
  - Same as `final` keyword in Java

- Shared memory leads to race-conditions and starvation. And possibly deadlocking

- Solution: Don't change!
- Problem: But how do get the data?

# Runnable

- Interface to normal Threads as well as executors
- Have one abstract method *run()* that takes no arguments and return void:

```java
public class MyTask implements Runnable{
    @Override
    public void run() {
        // Your method here
    }
}

Runnable r = () -> System.out.println("hi");
```

# Callable

- Have one abstract method *call()* that takes no arguments
  - Generic type: Object of <T> where T can be any Class
- Below is an example with T as String:

```java
import java.util.concurrent.Callable;
public class MyTask implements Callable<String> {
    @Override
    public String call() throws Exception {
        // Your method that returns a String here
        return null;
    }
}

MyTask t = () -> "Hello!";
```

# Threads vs tasks

- Threads are "heavy" to start
  - Takes time and memory

- Better to divide your work into *tasks*
  - Let reusable threads run them.

- Tasks is both Runnable, and Callable in Java that are called through executors

| i7 2.4 GHz cpu | Thread | Task |
|---|---|---|
| Create | 1030 ns | 77 ns |

# Futures

- Threads are asynchronous and non-deterministic, so we do not know when we will get the result back
- How do we extract the value from a `Callable`?
- Meet Future<T>
  - Parameterised over the type T

```java
Future<String> fut = executor.submit(
    new Callable<String>() {
        public String call() {
            // do something
        }
    }
);
```

# Working with Futures

- Future represent the work that at some point will be executed
- There is several ways to get the result from a Future:

-
```
//Get the result when it's ready.
//Blocks the thread until then.
future.get();
//If you want to escape from the blocking method
//no later than a given time, you can set that.
future.get(10, TimeUnit.MINUTES);
//Or you can ask if the task is done.
//Returns a boolean.
future.isDone();
```

# ExecutorService

- ExecutorService executor = Executors.newXXXThreadPool();

  - NewCachedThreadPool()
    - Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

  - newFixedThreadPool(int nThreads)
    - Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

  - NewSingleThreadExecutor()
    - Creates an Executor that uses a single worker thread operating off an unbounded queue.

# ExecutorService

- newScheduledThreadPool()
  - Executes periodically (eg. For database clean up)

- Only Java 8: WorkStealingPool()
  - Creates a thread pool that maintains enough threads to support the given parallelism level, and may use multiple queues to reduce contention.

# Completable Futures

- A Future that can be completed

- `java.util.concurrent.CompletableFuture<T>`

- `thenApply(Function<T, R> fn)`

- `thenApplyAsync(Function<T, R> fn,`
  `                Exectutor e)`

# Concurrent design concepts

- Safety (also known as concurrent type safety)
  - Thread-safety: Consistent state
- Liveness
  - Temporary failures
    - Locking, waiting, starvation
  - Permanent failures
    - Deadlocks, unrecoverable failures
- Performance
- Reusability