

Modul - Fortgeschrittene Programmierkonzepte

Bachelor Informatik

11 - Futures

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Before we start ...

The plan for the 'X-Mas Lecture' 18.12.2019

Robocode Tournament, Glühwein and Pizza!

Plan

8:00 -- Introduction into Robocode (R0.03)

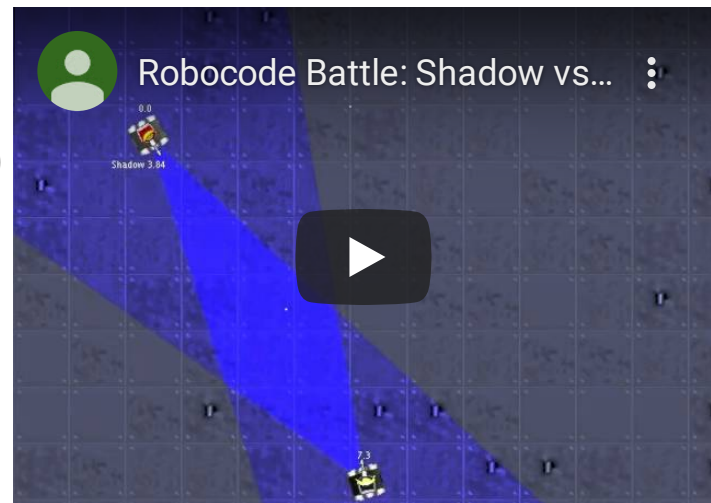
09:45 -- Start Coding and Testing (S1.31)

11:00 -- Tournament starts

16:00 -- last round!

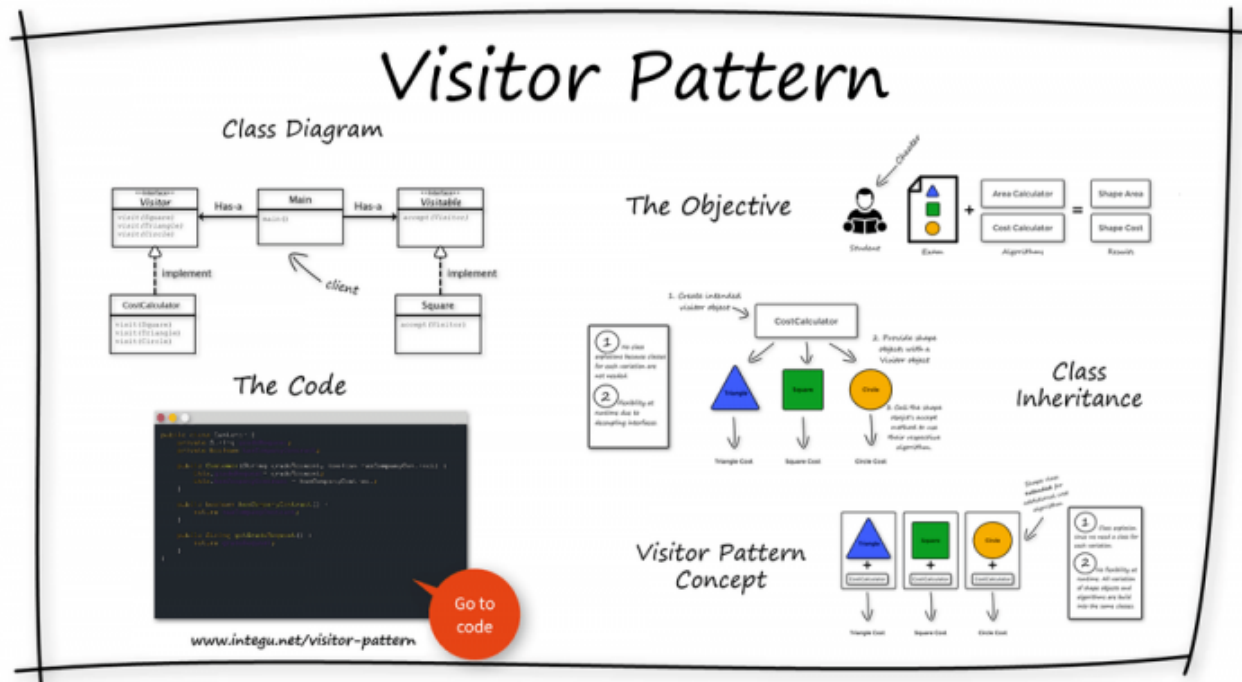
16:30 -- And the winner is? Celebration

17:00 -- End of show



Today in 'Übung 3'

- Refactoring and Visitor Pattern



taken from <https://www.integu.net/visitor-pattern/>



Agenda for today

What is on the menu for today?

- Threads (again)
- Futures, Callable, Executors
- Chaining



Parallel Processing

Working Asynchronously

Last week, we [used threads to do stuff in parallel](#). In modern application design, these threads are used to take workload off the main thread so that the actual application (e.g. a GUI or microservice) remains responsive to external input.

Using threads in Java is fairly simple:

1. Extend `Thread` and overwrite `run()`, or create a new thread with reference to a `Runnable`
2. Call `start()` on the thread instance to signal that the thread is ready to run.
3. Optionally use `join()` on the thread instance to wait for its completion.



Parallel Processing

Working Asynchronously (Code!)

```
Thread t = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello from my custom thread!");  
    }  
});  
  
t.start();  
System.out.println("Hello from the application main thread!")  
  
System.out.println("Waiting for thread to complete...");  
t.join();  
  
System.out.println("All done.");
```

Parallel Processing

Working Asynchronously

From the main application's point of view, the code in `run()` is run *asynchronously*. The execution of the `main` method continues right after the `t.start()` call, independent of the execution of `run()`. On a multi-core computer, the threads may actually be active at the same time, one thread per core.

There are two major drawbacks of this method:

1. There is no (direct) way to provide arguments to or obtaining *results* from the computation done in the thread. This is also reflected in the fact that `run()` does not take any arguments and has return type `void`.
2. There is no (direct) way to communicate potential exceptions that occur in the thread to the main application; the signature of `run()` can not be modified.

How could these common issues be solved?

Parallel Processing

Working Asynchronously

A possible solution is to store both arguments and results inside the thread or `Runnable`, and relay possible exceptions to whoever tries to retrieve the result.

```
public class JobWithResult implements Runnable {
    private int a, b, result;
    private Exception e;

    JobWithResult(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        try {
            this.result = a / b;
        } catch (Exception e) {
            this.e = e;
        }
    }

    int getResult() throws Exception {
        if (e != null)
            throw e;
        return result;
    }
}
```


Parallel Processing

Working Asynchronously

Which can be used in your main program:

```
JobWithResult jwr = new JobWithResult(4, 0);  
Thread t = new Thread(jwr);  
t.start();  
  
System.out.println("Waiting for thread to complete...");  
t.join();  
  
System.out.println("All done.");  
try {  
    System.out.println(jwr.getResult());  
} catch (Exception e) {  
    System.out.println("Ooops: " + e.toString());  
}
```

The Future of Callables

The code above features three key parts.

1. A `Runnable` which may use external information and stores results as attributes.
Possible exceptions raised in the `run()` method are caught and stored as attribute.
2. The main program creates and starts the thread to execute the `run()` method of the above instance.
3. The main program uses the thread's `join()` to wait for completion, and our modified `Runnable`'s `get()` to obtain the result (or raise a possible exception).

Let's refactor the code to separate out the recurring scheme (the mechanics) from the actual business logic (the contents of `run`).



The Future of Callables

Callable

For (1), we'll use a different interface to reflect a return type.

```
interface Callable<V> {  
    V call();  
}
```



The Future of Callables

Future

For (3), we'll introduce an interface that will allow us to retrieve the *future result* from the `Callable`, once it's done.

```
interface Future<T> {  
    T get();  
}
```



The Future of Callables

Executor

The remaining part (2) that organizes the thread logistics to run the actual task, we'll stick here:

```
interface Executor {  
    <T> Future<T> async(Callable<T> task);  
}
```

As you can see, the `Executor` ties things together. Let's see how we can make the threading work.

The Future of Callables

The incoming `Callable` needs to be wrapped into a thread, while watching for possible exceptions. Furthermore, it must return a `Future` that, on `get()`, waits for the thread to finish and then either throws any exception that occurred or returns the result.

```
class SimpleExecutor implements Executor {
    @Override
    public <T> Future<T> async(Callable<T> task) {
        // create anonymous Future instance
        return new Future<T> () {
            Thread t; // handle on the thread (get needs to wait!)
            T result; // save the result
            ExecutionException e; // in case something goes wrong?

            // constructor block
            {
                // create a new thread and start it
                // the runnable "watches" over the task
                t = new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            result = task.call();
                        } catch (Exception ex) {
                            e = new ExecutionException(ex);
                        }
                    }
                });
                t.start();
            }
        };
    }
}
```



The Future of Callables

...continued

```
@Override
public T get() throws InterruptedException, ExecutionException {
    // wait for it...
    t.join();

    // anything fishy?
    if (e != null)
        throw e;

    // all good, return result!
    return result;
}
};
}
```

The Future of Callables

Putting the pieces together, the `SimpleExecutor` simplifies asynchronous execution tremendously:

```
Executor ex = new SimpleExecutor();
int a = 4, b = 0;
Future<Integer> f1 = ex.async(new Callable<Integer>() {
    @Override
    public Integer call() {
        // can use variables from outer scope
        return a / b;
    }
});

// do other things if you like...

try {
    System.out.println(f2.get());
} catch (ExecutionException e) {
    System.out.println("The thread raised an exception: "
        + e.getCause());
}
```


Callables, Futures and Executors since Java 5

Since Java 5, these three parts are realized as [Callable](#) (1), [Executor](#) (2) and [Future](#) (3). The "real" interfaces are slightly different to allow a more faceted use.

```
interface Callable<V> {
    V call();
}
```

```
interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    V get();
    V get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

```
interface ExecutorService extends Executor {
    void execute(Runnable command); // for convenience
    <T> Future<T> submit(Callable<T> task);
    Future<?> submit(Runnable task);
    <T> Future<T> submit(Runnable task, T result);
    // ...
}
```

Executor

Java 5 also introduced a [number of different executors](#), here are a few examples:

```
Executor executor = Executors.newSingleThreadExecutor();

// executor = Executors.newCachedThreadPool(); // reuses threads
// executor = Executors.newFixedThreadPool(5); // use 5 threads

executor.execute(new Runnable() {
    public void run() {
        System.out.println("Hello world!");
    }
});
```



Asynchronous on Steroids

The previously introduced `Executor` allows to asynchronously execute a `Callable`, only waiting (blocking) on the executing thread when calling `get()` on the `Future`.

This is convenient enough for basic use cases (e.g. re-query the database after a swipe-down gesture on mobile), but quickly reaches its limits for more complex operations, where the result of one operation gets processed by another.

For the remainder of this chapter, let's consider this rather frequent example: For a cloud-backed application, you write a `displayStatus()` method to

- sign in with username and password; on success you gain a token to
- retrieve the user's status; once received,
- greet the user.

Chaining Executors

```
public class Workflow {
    static void displayStatus() throws ExecutionException, InterruptedException {
        final String user = "mustermann";
        final String pass = "12345"; // spaceballs, anyone? :-)

        // log in...
        Future<String> f1 = Executor.async(new Callable<String>() {
            public String call() {
                System.out.println("Authenticating with " + user + ":" + pass);
                return "secrettoken";
            }
        });

        final String token = f1.get();

        // retrieve user
        Future<String> f2 = Executor.async(new Callable<String>() {
            public String call() {
                System.out.println("Retrieving user details with token " + token);
                return "lightly sleep deprived, should get haircut";
            }
        });

        final String details = f2.get();

        System.out.println("Welcome " + user + "! You look " + details);
    }

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        displayStatus(); // blocks until completed!!
    }
}
```

Chaining with CompletableFuture

While this works, the `displayStatus()` method blocks until the status is displayed, thus blocking the remainder of the application from being responsive. Also, reacting to execution exceptions becomes tricky: which parts should be guarded by `try-catch`?

It would be desirable to be able to chain asynchronous actions that depend on each other. The [CompletableFuture](#) class added in Java 8 provides exactly that:

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {
    static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
        // ...
    }
    <U> CompletionStage<U> thenApplyAsync(Function<? super T, ? extends U> fn) {
        // ...
    }
    <U> CompletionStage<U> thenAcceptAsync(Consumer<? super T> action) {
        // ...
    }
    CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn) {
        // ...
    }
    // and much more...
}
```

Chaining with CompletableFuture

The `supplyAsync` accepts a `Supplier<U>`, which is almost identical to `Callable<T>`, but rooted in `java.util.function` for semantic reasons. This method is used to create a `CompletableFuture` that can then be chained with other actions.

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {  
    static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {  
        // ...  
    }  
    ...  
}
```

Chaining with CompletableFuture

Note the generic typing of `thenApply` and `thenAccept`: Although the `CompletableFuture` is generic in `T`, both methods return `CompletionStage<U>`, i.e. a (possibly) different generic type.

The `thenApply` method takes a `@FunctionalInterface` [Function](#) to map values of `? super T` to `? extends U`.

The `thenAccept` method takes a `@FunctionalInterface` [Consumer](#) to act on `? super T`.

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {  
    <U> CompletionStage<U> thenApplyAsync(Function<? super T, ? extends U> fn) {  
        // ...  
    }  
    <U> CompletionStage<U> thenAcceptAsync(Consumer<? super T> action) {  
        // ...  
    }  
    // and much more...  
}
```

Chaining with CompletableFuture

The `exceptionally` method accepts a function that maps a `Throwable` to ? extends `T` to handle exceptions gracefully by providing alternate input to the next in line.

```
class CompletableFuture<T> implements CompletionStage<T>, Future<T> {  
    CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn) {  
        // ...  
    }  
    // and much more...  
}
```

Since all those are functional interfaces, we can use method references and lambda notation for a cleaner codebase.



Chaining with CompletableFuture

An Example

```
CompletableFuture<?> cf = CompletableFuture.supplyAsync(
    () -> "host:12345")
    .thenApplyAsync(creds -> {
        System.out.println("Authenticating with " + creds);
        return "secrettoken";
    })
    .thenApplyAsync(token -> {
        System.out.println("Retrieving status with token=" + token);
        return "in the mood for holidays";
    })
    .thenAccept(status -> System.out.println(status))
    .exceptionally(ex -> { System.out.println(
        "Oops, something went wrong: " + ex); return null; });

System.out.println("All done!");
```



Chaining with CompletableFuture

You can even combine (synchronize) multiple `CompletableFuture` by using its

```
public <U,V> CompletableFuture<V> thenCombineAsync(  
    CompletionStage<? extends U> other,  
    BiFunction<? super T,? super U,? extends V> fn)
```



Other Concurrency Utilities

Since Java 5, there are a number of other concurrency utilities available.

- `java.util.concurrent` provides thread safe collections
 - `BlockingQueue`: *FIFO* queue that blocks on write-if-full and read-if-empty
 - `ConcurrentMap`: map with atomic read/write operations
- `ThreadLocalRandom` provides random numbers with per-thread randomization
- `AtomicInteger` allows atomic integer increment/decrement
- `CountDownLatch` allows for atomic counting and waiting

CountDownLatch

A frequent use for the `CountDownLatch` is batch processing across multiple threads:

```
List<String> filesToProcess = ...;

CountDownLatch latch = new CountDownLatch(filesToProcess.size());
ExecutorService ex = Executors.newFixedThreadPool(5);

for (String f : filesToProcess) {
    ex.submit(() -> {
        System.out.println("processing file " + f);

        // countdown when done!
        latch.countDown();
    });
}

latch.await();

// shutdown the threads to allow main() to quit.
ex.shutdown();
```



Summary (1)

Use threads to run code asynchronously and in parallel.

- *asynchronously* means the caller/delegator immediately continues execution after queueing the task
- *parallel* means that more than one method is executed at the same time; asynchronous methods are typically executed in parallel.
- use `join` to wait on threads to complete

Be extra careful when threads *share resources*.

- use thread-safe containers
- protect critical sections with `synchronized`
- minimize blocking time and avoid deadlocks with inter-thread communication (`wait`, `yield`, `notify`, `notifyAll`)



Summary (2)

Use `Future` to retrieve results and handle exceptions within the threads.

- `get()` will block until the thread has completed (and raise possible exception)
- `cancel()` terminates the execution of the task
- `isDone()` returns `true` if the task completed

Use `CompletableFuture` for elegant asynchronous programming.

- use `supplyAsync` to create a chainable `CompletableFuture`
- use `thenApplyAsync` to transform intermediate results
- use `thenAcceptAsync` to consume final results (end of chain!)
- use `thenCombineAsync` to join multiple `CompletableFuture`

Use `ExecutorServices` when batch-processing large quantities of data, e.g. importing multiple files, scaling images, downloading resources, etc.

[illegible]