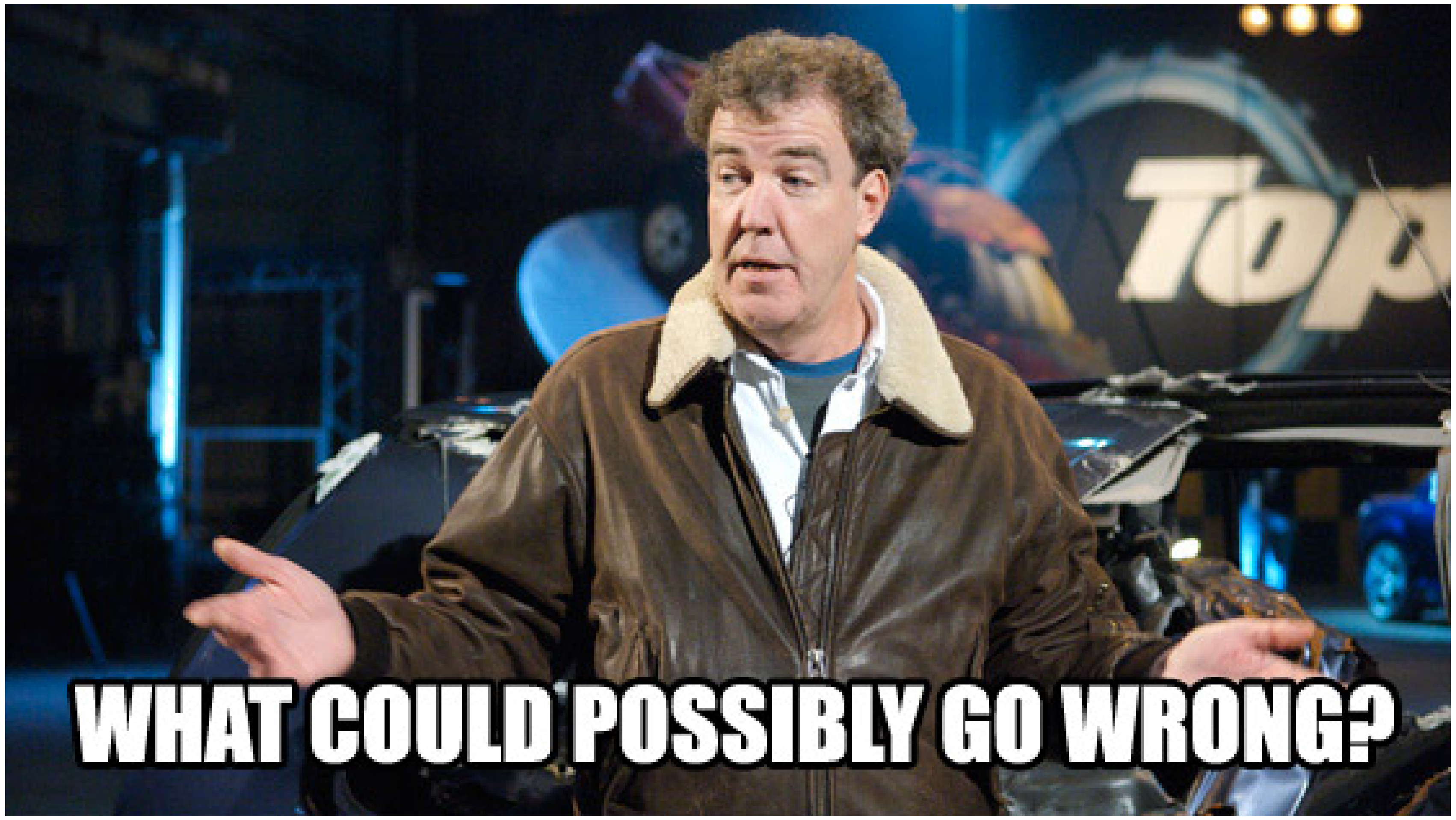
Secure OOP with Java

Lecture Unit - 07

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

Dealing with Failure


Top
WHAT COULD POSSIBLY GO WRONG?

What is a Failure?

Failure is the inability of a system or component to perform its required functions within specified performance requirements.

— IEEE standard computer dictionary (1990)

Reason for Failure

- Abnormal input
- Missing resources
- System failures
- Programming errors
- Logic errors

Bad things can happen, even to good programs.

Java, The Good Parts, 2010
— Jim Waldo

How Can We Handle Failure?

- Return to a safe state and enable the user to execute other commands, or
- allow the user to save all work and terminate the program gracefully.

Sentinel Values

aka **flag value, signal value**

- Values which may be used as valid return type
- BUT they signal a failure to fulfill the task
- Unfortunately they are easily overlooked/ignored
- Examples: `null`, `-1`

```
int indexOf(int[] values, int target) {
    for (int i = 0; i < values.length; i++) {
        if (values[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

Different Meanings of Exception

- Occurrence of an exceptional condition
- Creation of a Java object to represent the exceptional condition
- Throwing the exception to the exception handler

Exception Handling

- There is a clear distinction between the "normal" code flow and exceptional situations.
- Java uses objects to signal an exception during runtime.
- This object encapsulates the details of an error in the program.
- The handling of the exception may be delegated to the calling methods.

Exception Handling in Java

1. Throwing exceptions → `throw`
2. Catching exceptions → `try/catch/finally`
3. Declaring exceptions → `throws`

throw

- Used to signal an exception during runtime.
- `throw` interrupts the 'normal' application flow.

1. Create an exception object.
   - Anything that is an instance of `Throwable` can be used.
2. Use `throw` to start exception handling process.

```java
int indexOf(int[] values, int target) {
    for (int i = 0; i < values.length; i++) {
        if (values[i] == target) {
            return i;
        }
    }
    throw new NoSuchElementException();
}
```

```java
void setValue(String value) {
    if (value == null) {
        throw new NullPointerException();
    }
    this.value = value;
}
```

try/catch/finally

- Deals with an exception **during runtime**.
- `try` encloses expressions which may throw an exception.
- `catch` defines the reaction to specific exception types.
- `finally` executes whether or not an exception was caught.

try/catch/finally

If any code inside the `try` block throws an exception

1. the program skips the remainder of the code in the `try` block and
2. executes the code in the matching `catch` block.

> If there is no matching `catch` block the method exits immediately.

→ If none of the code inside the `try` block throws an exception, the `catch` blocks are all skipped.

```java
try {
    int position = indexOf(new int[] {1, 5, 6, 10}, 9)
    System.out.println("Position of 9 is " + position);
} catch (NoSuchElementException e) {
    System.err.println("9 could not be found.")
}
```

```
try {
    // some db operations
} catch (DbConnectionException e) {
    LOG.error("Could not connect to database.");
} catch (UpdateFailedException e) {
    LOG.error("Could not update data.");
} catch (CommitFailedException e) {
    LOG.error("Could not commit changes to database.");
}
```

Arranging Multiple catch Blocks

Multiple catch blocks must be arranged from the most specific exception type to the most generic exception type.

```
try {
    // some io access
} catch (FileNotFoundException e) {
    // handle exception
} catch (IOException e) {
    // handle exception
} catch (Exception e) {
    // handle exception
}
```

Multi-Catch

```
try {
    // some db operations
} catch (DbConnectionException e) {
    LOG.error("Could not connect to database.");
} catch (UpdateFailedException | CommitFailedException e) {
    LOG.error("Could not save to database.");
}
```

finally

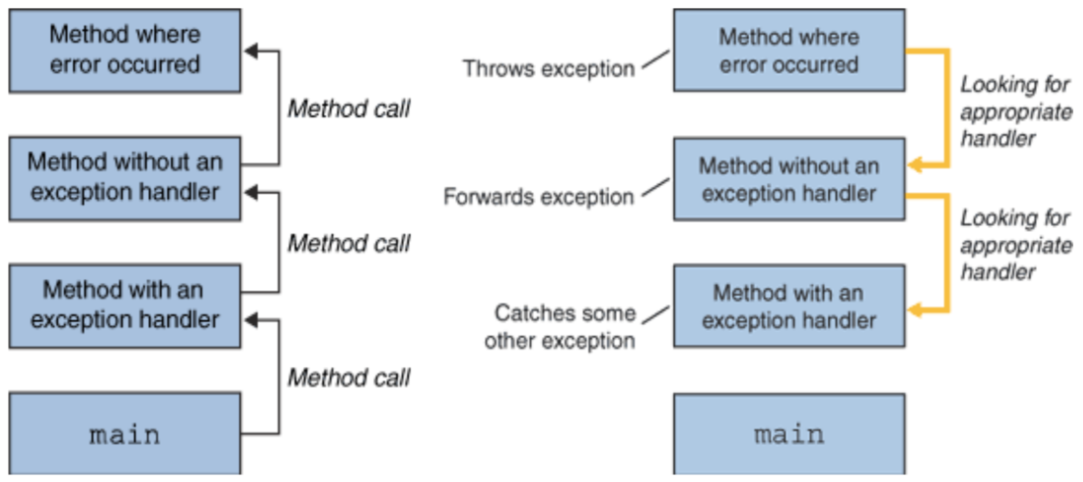
```
int[] values = {1, 5, 6, 10};
try {
    int position = indexOf(values, 9)
    System.out.println("Position of 9 is " + position);
} catch (NoSuchElementException e) {
    System.err.println("9 could not be found.")
} finally {
    System.out.println("There is a total of " + values.length + " elements.");
}
```

try/finally

```
try {
    // normal flow statements
} finally {
    // execute no matter if the try block finishes or not
}
```

The `finally` block is guaranteed to be executed (except when the thread dies or `System.exit()` is called).

Method Stack



Method where error occurred
Method without an exception handler
Method with an exception handler
main
Method call
Method call
Method call
Throws exception
Forwards exception
Catches some other exception
Method where error occurred
Method without an exception handler
Method with an exception handler
main
Looking for appropriate handler
Looking for appropriate handler

```
void main() throws E3 {
    try {
        a();
    } catch (E1 e) {
        // exception handling
    }
}

void a()  throws E1, E3 {
    try {
        b();
    } catch (E2 e) {
        // exception handling
    } finally {
        // things that must be done
    }
}

void b() throws E1, E2, E3 {
    if (condition) {
        throw new E1();
    } else if (otherCondition) {
        throw new E2();
    } else {
        throw new E3();
    }
}
```

Uncaught Exception

- An exception for which the Java Runtime does not find a programmer-defined exception handler.
- All uncaught exceptions are handled by the Java Runtime itself.
  - It catches the uncaught exception.
  - It prints the error stack to the standard error output stream.
  - Then it halts the Java application.

Causes of Exceptions

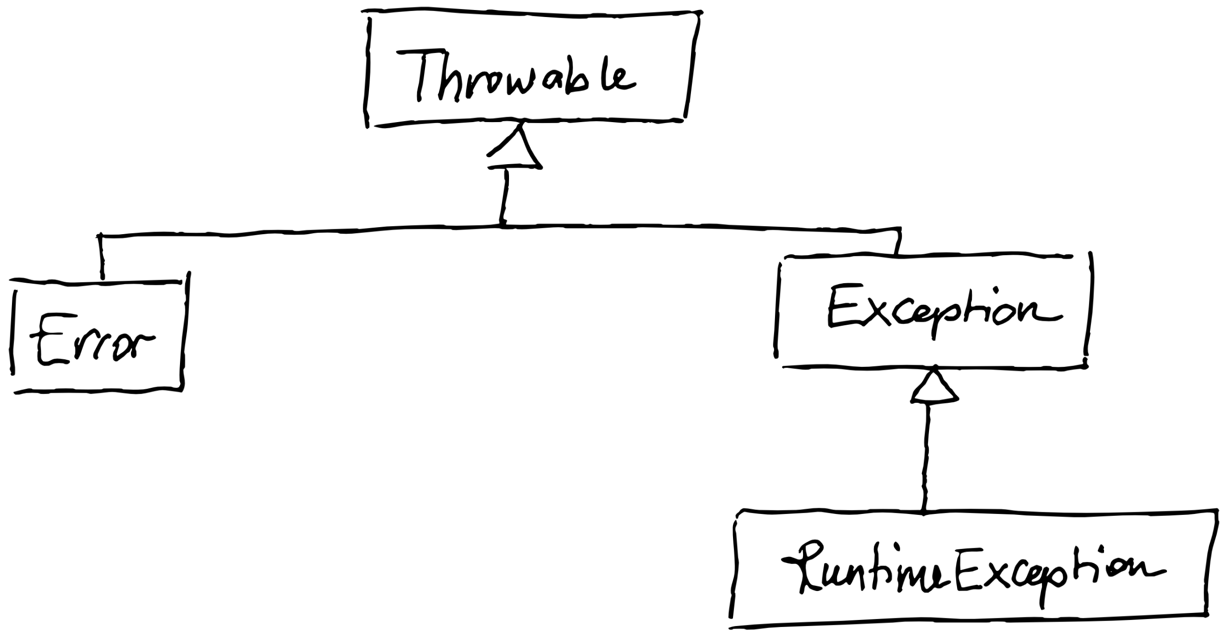
- `throw` statement
- Abnormal execution conditions
  - Violation of semantics (e. g. integer divide by zero)
  - Error during loading, linking, or initializing of the program
  - Internal errors or resource limitations of the Java Virtual Machine (`OutOfMemoryError`, `StackOverflowError`)

```java
int x = 4;
int y = 0;

int result = x / y;
```

→ Ausnahme java.lang.ArithmeticException: / by zero

Exception Hierarchy



java.lang.Throwable

- Base class for all errors and exceptions in Java.
- Only instances of `Throwable` and its subclasses may be used in Java exception handling.
- `Throwable` objects contain a stacktrace.

java.lang.Error

- Superclass of all the exceptions from which an ordinary program are not expected to recover.
- Examples
  - `java.lang.AssertionError`
  - `java.lang.OutOfMemoryError`
  - `java.lang.StackOverflowError`

java.lang.Exception

- Superclass of all exceptions an ordinary program may wish to recover.
- Base class for custom exception classes.

java.lang.RuntimeException

- Examples
  - ArithmeticException
  - ClassCastException
  - IllegalArgumentException
  - IllegalStatException
  - NullPointerException

Unchecked Exceptions

- `java.lang.RuntimeException` and all of its subclasses
- `java.lang.Error` and all its subclasses

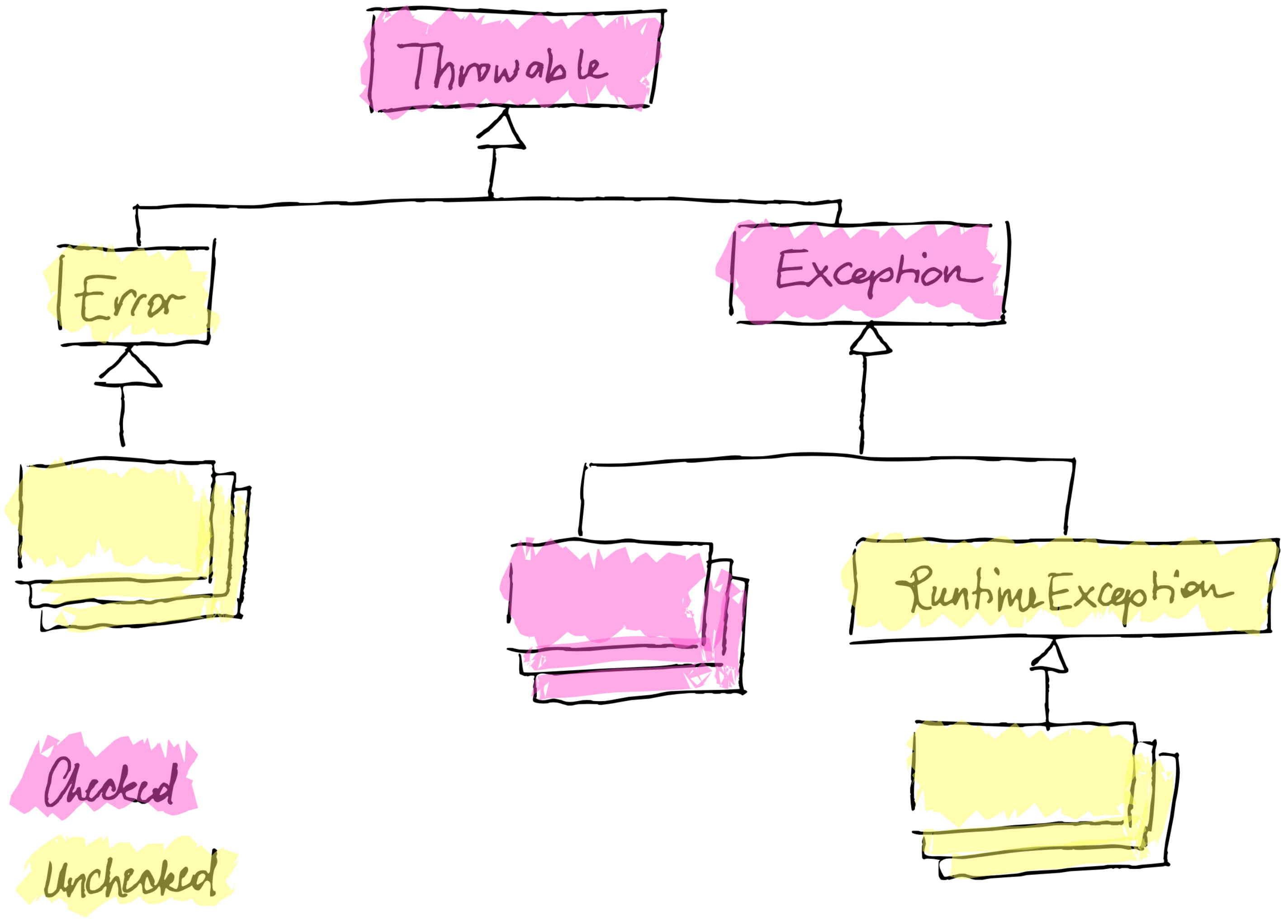
- The handling of unchecked exception is not enforced during compilation
- But they may be handled like any other (checked) exception

Checked Exceptions

- `java.lang.Throwable` and all of its subclasses, except
  - `java.lang.RuntimeException` and all of its subclasses and
  - `java.lang.Error` and all its subclasses

Checked exception needs to be handled in some form

- `catch` clause
- `throws` declaration


Throwable
Error
Exception
RuntimeException
Checked
Unchecked

throws

```java
int indexOf(int[] values, int target) throws Exception {
    for (int i = 0; i < values.length; i++) {
        if (values[i] == target) {
            return i;
        }
    }
    throw new Exception("Element not found");
}
```

```java
void checkPosition() {
    try {
        int position = indexOf(new int[] {1, 5, 6, 10}, 9)
        System.out.println("Position of 9 is " + position);
    } catch (Exception e) {
        System.err.println("9 could not be found.")
    }
}
```

```
void checkPosition() throws Exception {
    int position = indexOf(new int[] {1, 5, 6, 10}, 9)
    System.out.println("Position of 9 is " + position);
}
```

Documenting Exceptions

- JavaDoc `@throws` (or `@exception`)
- All checked exceptions
- Unchecked exceptions the caller might want to catch

Stacktrace

```
Exception in thread "main" java.lang.NullPointerException
    at java.util.HashMap.merge(HashMap.java:1216)
    at java.util.stream.Collectors.lambda$toMap$168(Collectors.java:1320)
    at java.util.stream.Collectors$$Lambda$5/1528902577.accept(Unknown Source)
    at java.util.stream.ReduceOps$3ReducingSink.accept(ReduceOps.java:169)
    at java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1359)
    at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:512)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:502)
    at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
    at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:499)
    at Main.main(Main.java:48)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:483)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
```

Chained Exceptions

```
try {
    fail();
} catch (Exception e) {
    System.err.println("Something went wrong: " + e.getMessage());
    throw new RuntimeException("I don't know what to do with this.", e);
}
```

```
Exception in thread "main" java.lang.RuntimeException: I don't know what to do with this.
    at fail.Chained.main(Chained.java:14)
Caused by: java.lang.Exception: Sorry.
    at fail.Chained.fail(Chained.java:21)
    at fail.Chained.main(Chained.java:11)
```

try-with-resources

```
MyResource resource = new MyResource();
try {
    resource.hardWork();
} finally {
    resource.close();
}
```

```
public class MyResource implements AutoCloseable {

    @Override
    public void close() {
        System.out.println("Clean up done.");
    }

}
```

```
try (MyResource resource = new MyResource()) {
    resource.hardWork();
}
```

Custom Exceptions

```java
package custom;

public class CustomException extends Exception {
    private static final long serialVersionUID = 8765666983770012913L;

    private Object context;

    public CustomException(String message, Object context) {
        super(message);
        this.context = context;
    }

    public CustomException(String message, Object context, Throwable cause) {
        super(message, cause);
        this.context = context;
    }

    public CustomException(Object context, Throwable cause) {
        super(cause);
        this.context = context;
    }

    public Object getContext() {
        return context;
    }
}
```

Best Practices

The Dos

- If there is a runtime failure, do throw an exception.
  - Do not use return values to signal failures during runtime.
  - Prefer already existing exception classes over custom exceptions.
- Not every statement does need its own try/catch block.
- Make good use of the exception hierarchy.

The Dos

- Propagating exceptions is not a sign of shame.
- Fail early, fail hard.

→ **"throw early, catch late"**

The Dont’s

- Do not ignore or swallow exceptions.
  - If you cannot handle the exception pass it on.
- Do not use exceptions to implement simple runtime checks.

```java
try {
    someMethodCall();
} catch (Throwable e) {
}
```

```java
try {
    someMethodCall();
} catch (IOException e) {
    throw new IllegalStateException("Could not read application config.", e);
}
```

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at