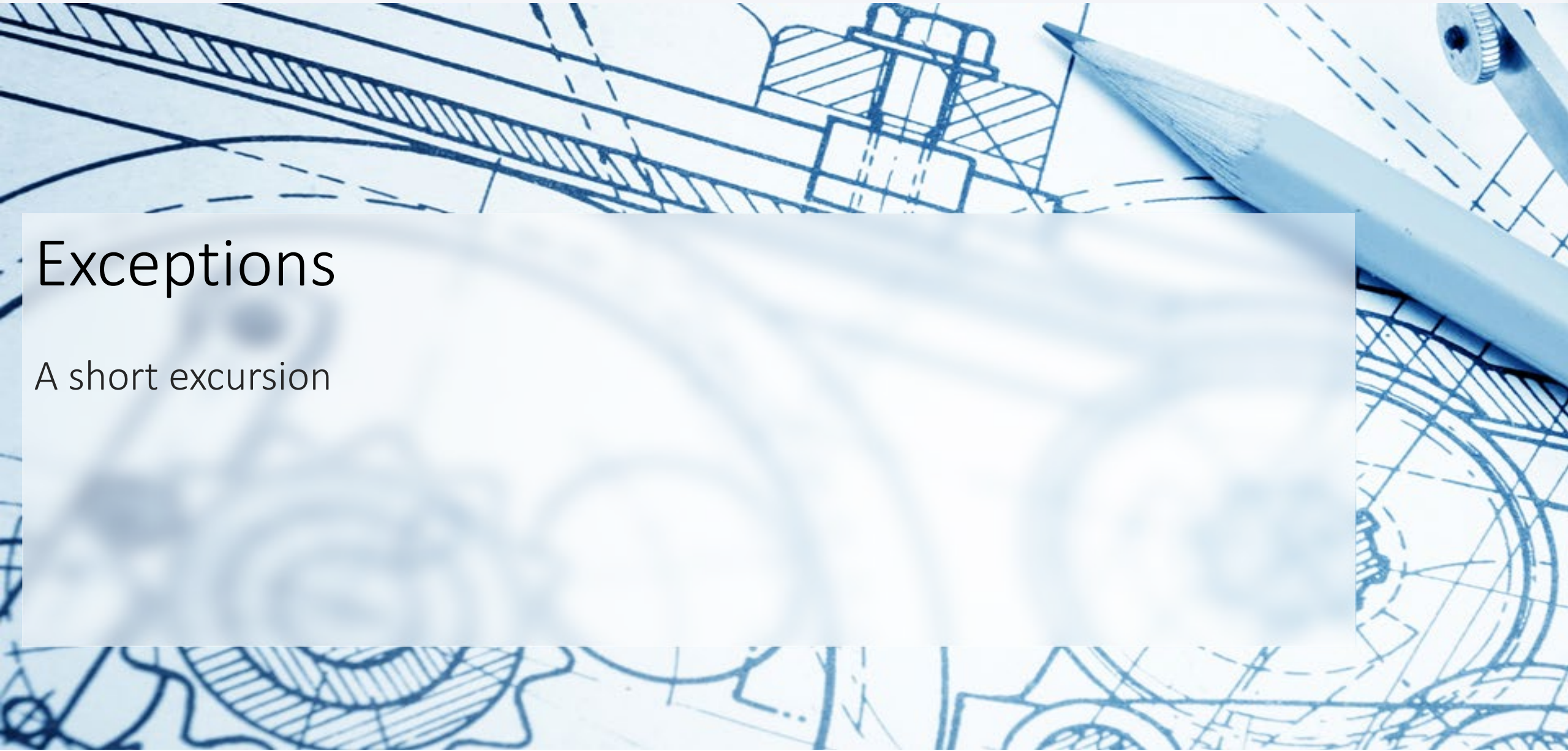


Exceptions

A short excursion

Exceptions in Java

Throwing of Exceptions

- How is an Exception thrown?
- When should you throw an Exception?

Catching of Exceptions

- How does the caller handle an Exception?
- What to do when an Exception can not be handled reasonably?

Recap

Finally an explanation to NullPointerException, ArrayIndexOutOfBoundsException!

Throwing an Exception

Fachhochschule Nordwestschweiz

Method declares it can throw an Exception under certain circumstances

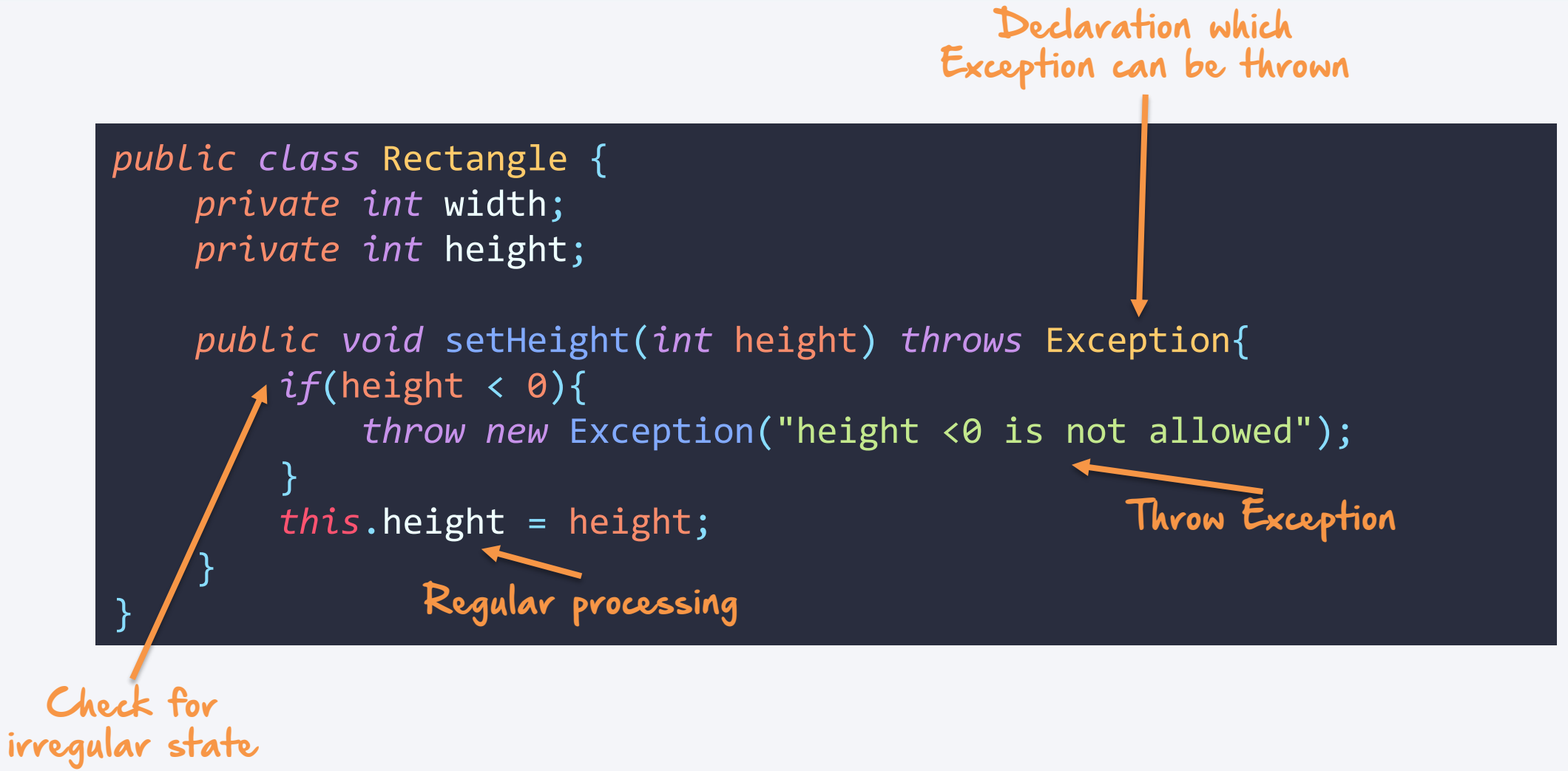
```
public void doSomething() throws Exception{
    if(somethingWentWrong){
        throw new Exception("Uh-oh");
    }
    //normal execution when everything is fine
}
```

Throwing of the Exception Method is immediately left

Executed if no Exception needed to be thrown

Irregular State: Throw Exception

Fachhochschule Nordwestschweiz



Catching Exceptions

Fachhochschule Nordwestschweiz

```
try {
  //call method that could throw an Exception
} catch (Exception ex) {
  //try to safe or recover
}
```

Same Exception-Class as declared in the method

Instructions for regular processing

Example


```
public Rectangle createRectangle(){
    int width;
    int height;

    Scanner s = new Scanner(System.in);
    height = s.nextInt();
    width = s.nextInt();
    Rectangle rectangle = new Rectangle();
    try{
        rectangle.setHeight(height);
    }catch(Exception ex){
        //What reasonable thing could we do here
    }
    return rectangle;
}
```

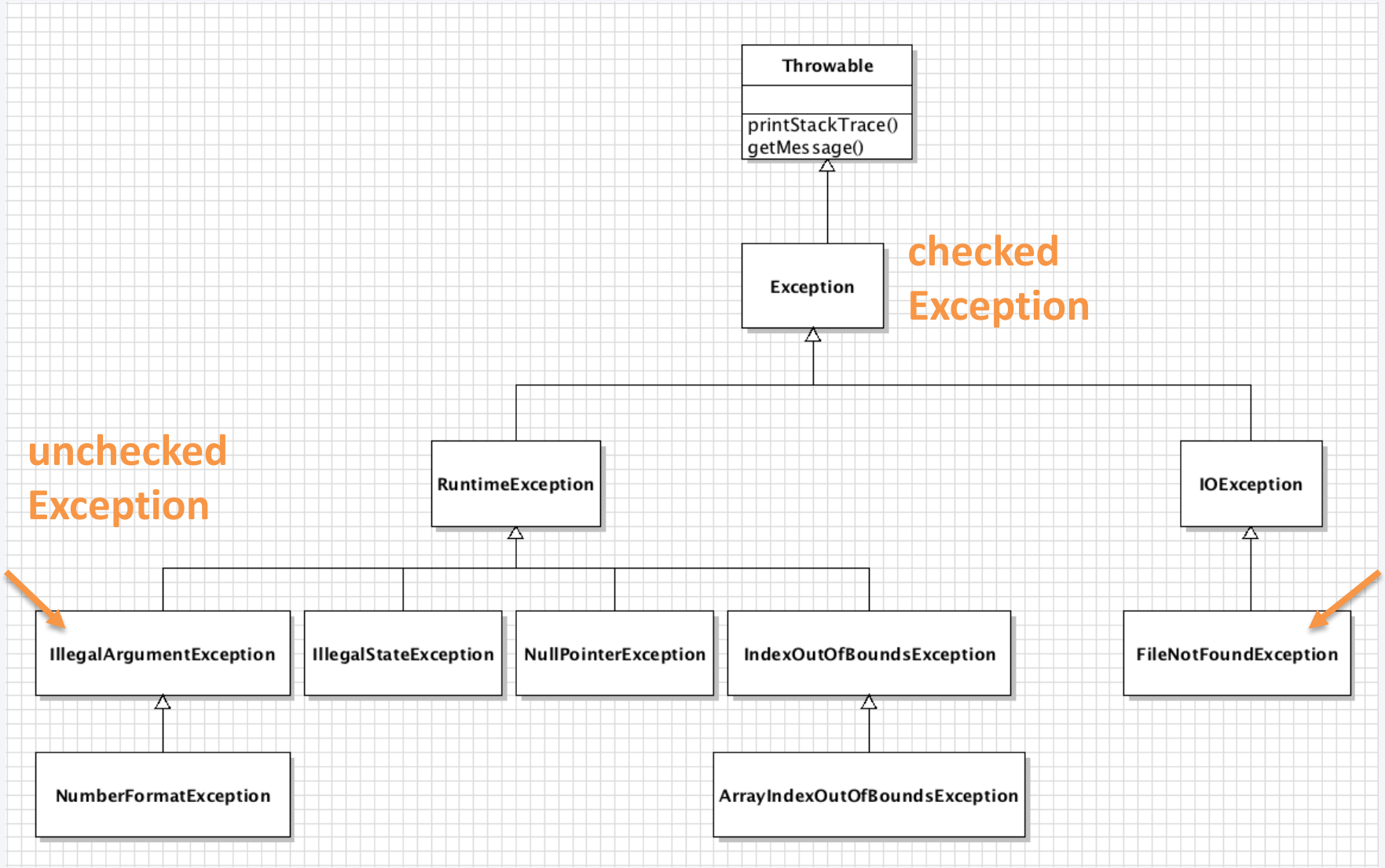
But wait…?

Why did we use so few try-catch-blocks?

We get tons of Exceptions like `NullPointerException` or `ArrayIndexOutOfBoundsException`

Do not all Exceptions have to be caught in a try-catch-block?

Excerpt of the Exception-Classhierarchy



Checked Exception – Checking through the Compiler

Fachhochschule Nordwestschweiz

```
public class FileExceptionExample {
    ...
    public void readFile() throws FileNotFoundException{
        ...
        throw new FileNotFoundException("no file found");
        ...
    }

    public void someMethod(){
        try{
            readFile();
        }catch(FileNotFoundException ex){
          //do what is possible
        }
    }

    public void someOtherMethod() throws FileNotFoundException{
        readFile();
    }
}
```

All thrown exceptions need to be declared

Catching of declared Exceptions

Forward a declared Exception that can not be handled appropriately

Unchecked Exception – Handling of runtime errors

RuntimeExceptions do not need to be declared

```
public void setSize(int size){
    if((size <= 8) || (size >= 120)){
        throw new IllegalArgumentException("Wrong size!");
    }else{
        this.size = size;
    }
}
```

Checked vs. Unchecked Exceptions

Checked Exceptions (checked by the Compiler)

- For foreseeable exceptional situations
  - Server is not running
  - File is not available
  - General: An external resource is not available

Unchecked Exception (Runtime Error)

- As a reaction to a coding error
- Error with the caller
  - Illegal parameters

```
throw new IllegalArgumentException();
throw new ArrayIndexOutOfBoundsException();
throw new NullPointerException();
```

- Error within a method

```
throw new IllegalStateException();
```

Control flow in try-catch-blocks

Fachhochschule Nordwestschweiz

Try-block is executed first

```
   try{
1)     int result = calculateSomething();
2)     int x = result * result;
   }catch(Exception ex){
       System.out.println("Error!");
   }
3)System.out.println("Done");
```

CalculateSomething can throw an Exception

The catch-block is not executed

Then code below the catch block

If "try" is successful (throws no Exception)

Control flow in try-catch-blocks

calculateSomething fails. The rest of the try-block is not executed

Is executed after the catch-block

```
  try{
1)    int result = calculateSomething();
      int x = result * result;
  }catch(Exception ex){
2)      System.out.println("Error!");
  }
3)System.out.println("Done");
```

CalculateSomething can throw an Exception

The catch-block is executed

If "try" fails (throws an Exception)

The **finally** keyword


```
public void cook(){
    try{
        turnOnStove();
        bakeACake();
    }catch(Exception ex){
        System.out.println("Something went wrong!");
    }finally {
        turnOffStove();
    }
}
```

The code in the finally-block is always executed

Control Flow with Exceptions

Experiment with **ExceptionTest** (on AD)

test = “nein”

test = “ja”

Stepwise Debugging of the class

Exceptions Recap

A method can throw an Exception, if something goes wrong at runtime

A method throws an Exception using the **throw** keyword, followed by a new Exception-Object

Exceptions of the type RuntimeException are **not** checked by the compiled
- Neither do they need to be declared or handled by a try/catch-block
- Are used when programming errors occur

Checked Exceptions are checked by the compiler
- The **Exception** needs to be declared in the method
- The calling code needs to be wrapped in a try/catch-block…
- … or continue to throw the Exception up the chain
- Checked Exceptions are used when calls to external resources fail

JDK contains loads of different Exceptions
- It is usually not necessary to implement your own RuntimeException-class
- Checked Exceptions are usually application specific subclasses of **Exception**