

# Java

## Collections part 2

Vincent Gerber, Tilman Hinnerichs

Java Kurs

28. Juni 2018



# Overview

## 1 Repetition

## 2 Exceptions

- Overview
- Catching Exceptions
- Throwing Exceptions

# Repetition

# Quizzing again!

How to use Maps:

Property		Map
Declaring Map		
Building Instance of Map		
Adding items		
Receiving items		

Quiz on keys and values of maps:

Property		Key		Value
Only once in map				
How to receive				
Type of return value				

# Solutions

How to use Maps:

Property	Map
Declaring Map	<code>Map&lt; S, T &gt; map</code>
Building Instance of Map	<code>new HashMap&lt; S, T &gt;()</code>
Adding items	<code>map.put(s,t);</code>
Receiving items	<code>map.get(s);</code>

Quiz on keys and values of maps:

Property	Key	Value
Only once in map	Yes	No
How to receive	<code>keys = map.keySet();</code>	<code>values = map.values();</code>
Type of return value	<code>Set&lt;T&gt;</code>	<code>Collection&lt;T&gt;</code>

# Quiz: How to iterate over a map?

We would like to iterate over the keys:

Possibility 1:

## Quiz: How to iterate over a map?

We would like to iterate over the keys:

Possibility 1:

```
1      Iterator<T> iter = keys.iterator();  
2  
3      while(iter.hasNext()) {  
4          S key = iter.next();  
5          key.doSth();  
6          ...  
7      }  
8
```

## Quiz: How to iterate over a map?

We would like to iterate over the keys:

Possibility 1:

```
1      Iterator<T> iter = keys.iterator();  
2  
3      while(iter.hasNext()) {  
4          S key = iter.next();  
5          key.doSth();  
6          ...  
7      }  
8
```

Possibility 2:



## Quiz: How to iterate over a map?

We would like to iterate over the keys:

Possibility 1:

```
1      Iterator<T> iter = keys.iterator();  
2  
3      while(iter.hasNext()) {  
4          S key = iter.next();  
5          key.doSth();  
6          ...  
7      }  
8
```

Possibility 2:

```
1      for(S key:keys) {  
2          key.doSth();  
3          ...  
4      }  
5
```

# Overview

List	<ul style="list-style-type: none"><li>• Keeps order of objects</li><li>• Easily traversible</li><li>• Search not effective</li></ul>
Set	<ul style="list-style-type: none"><li>• No duplicates</li><li>• No order - still traversible</li><li>• Effective searching</li></ul>
Map	<ul style="list-style-type: none"><li>• Key-Value storage</li><li>• Search super-effective</li><li>• Traversing difficult</li></ul>

# Easy and some more complex exercises

- fill a map with our 10 set elements and use the index as key. Print every item in the map.
- remove every item from every collection step by step and dont use clear
- create a vending machine company. The company stores their vending machine data in a map with place (city, ...) as key and machine as value. They also have an employee list. Each employee should appear only once (use an id). Each employee has a wage and a name. It is possible to filter employees by name or wage and to return every vending machine with city when it is empty. There can be multiple results for one city too.

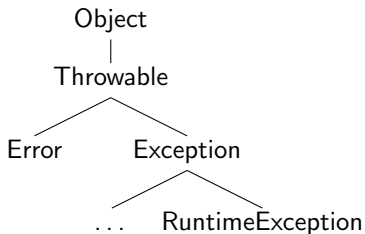
# Exceptions

While running software many things can go wrong. You have to deal with errors or exceptional behavior.

Java offers exception handling out of the box. Exceptions separate error-handling from normal code.

On this slide *exception* means the Java term and *error* a nonspecified general term.

# Hierarchy



Every exception is a subclass of *Throwable*. *Error* is also a subclass of *Throwable* but used for serious errors like *VirtualMachineError*.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html>

# Checked Exceptions

Every exception except *RuntimeException* and its subclasses are **checked exceptions**.

A checked exception has to be handled or denoted.

The cause of this kind of exception is often outside of your program.

E.g. `ArithmeticException`

# Unchecked Exceptions

*RuntimeException* and its subclasses are called **unchecked exceptions**.

Unchecked Exceptions do not have to be denoted or handled, but can be. Often handling is senseless because the program can not recover in case such exception occurs.

The cause of an unchecked exception can be a method call with incorrect arguments. Therefore any method could throw an unchecked exception. Most unchecked exceptions are caused by the programmer.

Errors are also unchecked.

E.g. `ArrayIndexOutOfBoundsException`



# Introduction

```
1 public class Calc {  
2  
3 public static void main(String[] args) {  
4  
5     int a = 7 / 0;  
6     // will cause an ArithmeticException  
7  
8     System.out.println(a);  
9     }  
10 }  
11
```

A division by zero causes an *ArithmeticException* which is a subclass of *RuntimeException*. Therefore *ArithmeticException* is unchecked and does not have to be handled.

# Try and Catch

Nevertheless the exception can be handled.

```
1 public class Calc {  
2  
3     public static void main(String[] args) {  
4         try {  
5             int a = 7 / 0;  
6         } catch (ArithmeticException e) {  
7             System.out.println("Division by zero.");  
8         }  
9     }  
10 }  
11
```

The **catch**-block, also called exception handler, is invoked if the specified exception (`ArithmeticException`) occurs in the **try**-block.

In general there can be multiple catch-blocks handling multiple kinds of exceptions.

# Stack Trace

```
1 public class Calc {  
2  
3     public static void main(String[] args) {  
4         try {  
5             int a = 7 / 0;  
6         }  
7         catch (ArithmeticException e) {  
8             System.out.println("Division by zero.");  
9             e.printStackTrace();  
10        }  
11    }  
12 }  
13
```

The stack trace shows the order of method calls leading to point where the exception occurs.

# Stack Trace

```
1 Division by zero.  
2 java.lang.ArithmeticException: / by zero  
3 at Calc.main(Calc.java:6)  
4 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
5 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.  
  java:62)  
6 at sun.reflect.DelegatingMethodAccessorImpl.invoke(  
  DelegatingMethodAccessorImpl.java:43)  
7 at java.lang.reflect.Method.invoke(Method.java:498)  
8 at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)  
9
```

This can also be seen when taking a look at your IDEs output.

# Finally

```
1 public class Calc {  
2  
3     public static void main(String[] args) {  
4  
5         try {  
6             int a = 7 / 0;  
7         }  
8         catch (ArithmeticException e) {  
9             System.out.println("Division by zero.");  
10            e.printStackTrace();  
11        }  
12        finally {  
13            System.out.println("End of program.");  
14        }  
15    }  
16 }  
17
```

The **finally**-block will always be executed, regardless if an exception occurs.

# Propagate Exceptions

Unhandled exceptions can be thrown (propagated).

```
1 public static int divide (int dividend, int divisor) throws  
   ArithmeticException {  
2     return dividend / divisor;  
3 }  
4
```

The method `int divide(...)` propagates the exception to the calling method denoted by the keyword **throws**.

Take a look at the explicit notation of the expected error.

# Propagate Exceptions - Test 1

```
1 public class Calc {
2
3     public static int divide (int dividend, int divisor) throws
ArithmeticException {
4         return dividend / divisor;
5     }
6
7     public static void main(String[] args) {
8
9         int a = 0;
10        try {
11            a = Calc.divide(7, 0);
12        } catch (ArithmeticException e) {
13            System.out.println("Division by zero.");
14            e.printStackTrace();
15        }
16    }
17 }
18
```

# Propagate Exceptions - Test 2

```
7 public static void main(String[] args) {  
8  
9     int a = 0;  
10    try {  
11        a = Calc.divide(7, 0);  
12    } catch (ArithmeticException e) {  
13        System.out.println("Division by zero.");  
14        e.printStackTrace();  
15    }  
16 }  
17
```

In this example there are two jumps in the stack trace:

```
java.lang.ArithmeticException: / by zero  
at Calc.divide(Calc.java:4)  
at Calc.main(Calc.java:11)
```



# Java API

The Java API shows<sup>1</sup> if a method throws exceptions. The notation `throws exception` means that the method can throw exceptions in case of an unexpected situation. It does not mean that the method throws exception every time.

Check if the Exception is a subclass of *RuntimeException*. If not the exception has to be handled or rethrown.

---

<sup>1</sup><https://docs.oracle.com/javase/7/docs/api/>

# Creating new Exceptions

You can create und use your own exception class.

```
1 public class DivisionByZeroException extends Exception {  
2  
3 }  
4
```

```
1 public static int divide (int dividend, int divisor) throws  
DivisionByZeroException {  
2     if (divisor == 0) {  
3         throw new DivisionByZeroException();  
4     }  
5     return dividend / divisor;  
6 }  
7
```

Exceptions can be thrown manually with the keyword **throw**.

# Creating new Exceptions - Test

```
1 public static void main(String[] args) {  
2  
3     int a = 0;  
4     try {  
5         a = Calc.divide(7, 0);  
6     } catch (DivisionByZeroException e) {  
7         System.out.println("Division by zero.");  
8         e.printStackTrace();  
9     }  
10 }  
11
```

*DivisionByZeroException* is checked and therefore has to be handled.

# Exercise

- Create a basic authentication system with users who can register, login, logout and change user informations. Requirements: username, password, email,... (password=min 6 letters, username= 4 - 10 letters) Each invalid input will result in an exception.

# Exercise

- Create a three dimensional warehouse with crates. Each crate will be stored at  $x=?$   $y=?$  and  $z=?$  coordinates and each user should be able to auto store crates or to manually place them. Crates can't float. Every invalid state will throw an exception. The system is able to find free spots and find/remove stored crates. It is capable to work with different sizes.

# Exercise

- Combine both systems. Our warehouse systems consists of employees with user accounts and a storage. Only authenticated employees will be able to interact with the system. Every employee will be an user with some extra attributes like loan, years spent and so on. Throw an exception when invalid actions appear.