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	Map< S, T > map
Building Instance of Map	new HashMap< S, T >()
Adding items	map.put(s,t);
Receiving items	map.get(s);

Quiz on keys and values of maps:

Property	Key	Value
Only once in map	Yes	No
How to receive	keys = map.keySet();	values = map.values();
Type of return value	Set¡T¿	Collection¡T¿

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             T item = iter.next();
5             item.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             T item = iter.next();
5             item.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            T item = iter.next();
5            item.doSth();
6            ...
7        }
8
```

Possibility 2:

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

Overview

List	• Keeps order of objects • Easily traversible • Search not effective
Set	• No duplicates • No order - still traversible • Effective searching
Map	• Key-Value storage • Search super-effective • Traversing difficult

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.

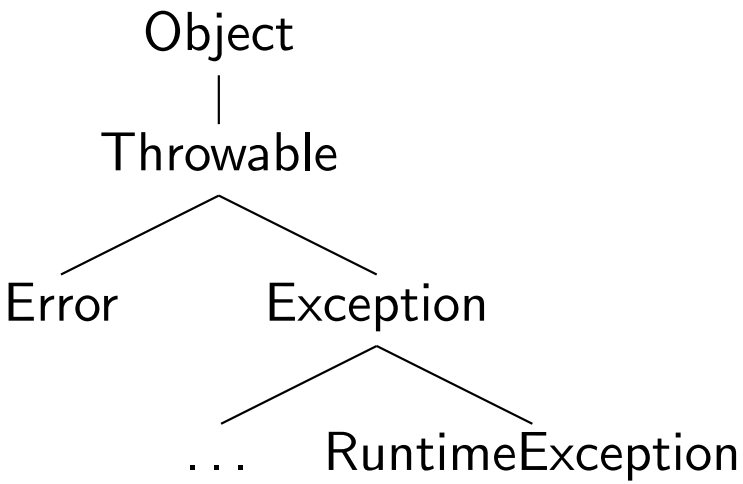
Exceptions

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 seperate 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 programer.

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  } catch (ArithmeticException e) {
8  System.out.println("Division by zero.");
9  e.printStackTrace();
10 } finally {
11 System.out.println("End of program.");
12 }
13 }
14 }
15
```

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 divident, int divisor) throws
      ArithmeticException {
2    return divident / divisor;
3    }
4
```

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

Propagate Exceptions - Test 1

```
1   public class Calc {
2
3   public static int divide (int divident, int divisor) throws
     ArithmeticException {
4   return divident / 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[1] 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.

[1] `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 divident, int divisor) throws
   DivisionByZeroException {
2  if (divisor == 0) {
3  throw new DivisionByZeroException();
4  }
5  return divident / 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.