Java

Collections part 2

Vincent Gerber, Tilman Hinnerichs

Java Kurs

28. Juni 2018

Overview

Repetition

What we learned last time:

- How to use generics
- How to handle Javas lists, sets and iterators

What we will try to achieve today:

- How to use iterators on sets and lists
- How to use maps and what to with them
- What exceptions are and how to handle them

A quiz!

	Set	List

A quiz!

	Set	List
Same item twice in it?		
Ordered?		
Iterable?		
What package to import		
Declaring set type (variable type)		
Building an instance (example)		
Add an item		
Removing an item		

A quiz!

	Set	List
Same item twice in it?	No!	Yes!
Ordered?	No!	Yes!
Iterable?	Yes!	Also yes!
What package to import	import java.util.*	import java.util.*
Declaring set type (variable type)	Set<T> set	List<T>list
Building an instance (example)	= new HashSet<T>()	= new ArrayList<T>
Add an item	set.add(item)	list.add(item)
Removing an item	set.remove(item)	list.remove(item)

Another quiz!

The iterator:

	Iterator
How to declare	
How to build an instance	
First main function (With data type)	
Second main function (With data types)	
Third main function (With data type)	
How to get from collection?	

How to iterate over sets and lists

How to iterate over sets and lists

```
Set<T> mySet = new HashSet<T>();
foreach(T item:mySet){
    item.doSomething();
}

List<T> myList = new ArrayList<T>();
foreach(T item:myList){
item.doSomething();
}

```

Another quiz!

The iterator:

	Iterator
How to declare	Iterator<T> iter
How to build an instance	= new Iterator<T>()
First main function (With data type)	boolean iter.hasNext()
Second main function (With data types)	T iter.next()
Third main function (With data type)	T iter.remove()
How to get from collection(e.g. set)?	set.iterator()

How to iterate over sets and lists using iterators

```java
    Set<T> mySet = new HashSet<T>();
    Iterator<T> myIter = mySet.iterator();

    while(myIter.hasNext()){
        T item = myIter.next();
        item.doSomething();
    }

```

Exercise

- Create an array with 10 elements. Create a list and fill the list with the array elments. Create a set and fill the set with the list elments
- Extend our vending machine with an internal storage

Map

The interface *Map* is not a subinterface of *Collection*.
A map contains pairs of key and value. Each key refers to a value. Two keys can refer to the same value. There are not two equal keys in one map. *Map* is part of the package `java.util`.

```java
    public static void main (String[] args) {

    Map<Integer, String> map =
    new HashMap<Integer, String>();

    map.put(23, "foo");
    map.put(28, "foo");
    map.put(31, "bar");
    map.put(23, "bar"); // "bar" replaces "foo" for key = 23

    System.out.println(map);
    // prints: {23=bar, 28=foo, 31=bar}
    }

```

Key, Set and Values

You can get the set of keys from the map. Because one value can exist multiple times a collection is used for the values.

```java
    public static void main (String[] args) {

    // [...] map like previous slide

    Set<Integer> keys = map.keySet();
    Collection<String> values = map.values();

    System.out.println(keys);
    // prints: [23, 28, 31]

    System.out.println(values);
    // prints: [bar, foo, bar]
    }

```

Iterator

To iterate over a map use the iterator from the set of keys.

```
    public static void main (String[] args) {

    // [...] map, keys, values like previous slide
    Iterator<Integer> iter = keys.iterator();

    while(iter.hasNext()) {
    System.out.print(map.get(iter.next()) + " ");
    } // prints: bar foo bar

    System.out.println(); // print a line break

    for(Integer i: keys) {
    System.out.print(map.get(i) + " ");
    } // prints: bar foo bar
    }

```

Nested Maps

Nested maps offer storage with key pairs.

```java
    public static void main (String[] args) {

    Map<String, Map<Integer, String>> addresses =
    new HashMap<String, Map<Integer, String>>();

    addresses.put("Noethnitzer Str.",
    new HashMap<Integer, String>());

    addresses.get("Noethnitzer Str.").
    put(46, "Andreas-Pfitzmann-Bau");
    addresses.get("Noethnitzer Str.").
    put(44, "Fraunhofer IWU");
    }

```

Maps and For Each

You can interate through the entry set of a map (available before Java 1.8)

```java
Map<String, String> map = ...
for (Map.Entry<String, String> entry : map.entrySet()) {
System.out.println("Key: " + entry.getKey() +
", value" + entry.getValue());
}

```

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

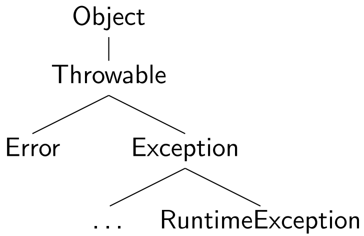
- fill a map with our 10 set elements and use the index as key. Print every item in the map.
- remove every item from every collection step by step and dont use clear
- create a vending machine company. The company stores their vending machine data in a map with place (city, ...) as key and machine as 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.

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.

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.

Introduction

```
public class Calc {

public static void main(String[] args) {

int a = 7 / 0;
// will cause an ArithmeticException

System.out.println(a);
}
}

```

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.

```java
public class Calc {

public static void main(String[] args) {

try {
int a = 7 / 0;
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
}
}
}

```

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

```java
public class Calc {

public static void main(String[] args) {

try {
int a = 7 / 0;
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
e.printStackTrace();
}
}
}

```

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

Stack Trace

```
Division by zero.
java.lang.ArithmeticException: / by zero
at Calc.main(Calc.java:6)
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:498)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)

```

Finally

```java
public class Calc {

public static void main(String[] args) {

try {
int a = 7 / 0;
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
e.printStackTrace();
} finally {
System.out.println("End of program.");
}
}
}

```

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

Propagate Exceptions

Unhandled exceptions can be thrown (propagated).

```
    public static int divide (int divident, int divisor) throws
     ArithmeticException {
    return divident / divisor;
    }

```

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

Propagate Exceptions - Test 1

```java
public class Calc {

public static int divide (int divident, int divisor) throws
 ArithmeticException {
return divident / divisor;
}

public static void main(String[] args) {

int a = 0;
try {
a = Calc.divide(7, 0);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
e.printStackTrace();
}
}
}

```

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.

```
public class DivisionByZeroException extends Exception {

}

```

```
public static int divide (int divident, int divisor) throws
 DivisionByZeroException {
if (divisor == 0) {
throw new DivisionByZeroException();
}
return divident / divisor;
}

```

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

Creating new Exceptions - Test

```java
    public static void main(String[] args) {

    int a = 0;
    try {
    a = Calc.divide(7, 0);
    } catch (DivisionByZeroException e) {
    System.out.println("Division by zero.");
    e.printStackTrace();
    }
    }

```

DivisionByZeroException is checked and therefore has to be handled.