Java

Exceptions

Hannes Ueck, Jakob Krude

7. Januar 2021

Java-Kurs

Overview

Exceptions

- Overview
- Catching Exceptions
- Throwing Exceptions

Debugging

- Overview
- Breakpoints

Testing

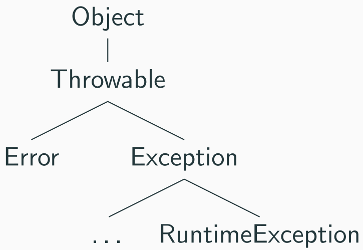
Exceptions

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.

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

```java
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.

```
    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.

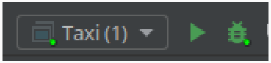
Debugging

A debugger helps the programmer to trace errors in their code.

Normally a program executes every line of code and stops afterwards.
With a debugger you can watch every state of the execution.
The inspection of a specific state can help to find errors.

Debug Perspective

Intellij offers support for debugging.



The bug icon starts the debugger. You can also press Shift + F9.

Breakpoint

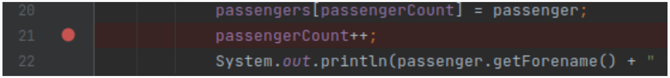
A breakpoint marks a specific line in the code.

While debugging the excecution of the program will stop before every marked line. At this point you see every current variable and object with their current values.

The debugger steps through the code breakpoint by breakpoint.

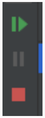
Toggle Breakpoints

With Ctrl + 8 you can add or remove a breakpoint to the current line. Or just click next to the line number.



```
passengers[passengerCount] = passenger;
passengerCount++;
System.out.println(passenger.getForename() + "
```

Breakpoint at the line: `passengerCount++;`



The green right arrow or F9 will execute code until next breakpoint.
The red square or Ctrl + 2 will end the debugging.

Stepping through the Code - 1

If your code contains breakpoints Intellij will open a debug toolbar automatically when you start the debugger.

```
 1      public class TestDebug {
 2  
 3          public static void main(String[] args) {
 4  
 5              for(int i = 0; i < 13; i++) {
 6                  int a = 0;
 7                  a++;
 8                  System.out.println(a);
 9              }
10          }
11      }
12  
```

Assume breakpoints in line 6, 7 and 8. Start the debugger.

Stepping through the Code - 2

- Step Over (F8):
  step through the code line by line
- Step Into (F7):
  step into the function called in the current line of code
- Step Out (Shift + F8):
  step out of current function

While stepping through the code variables appear and change their values.

Testing

Asserting values of variables

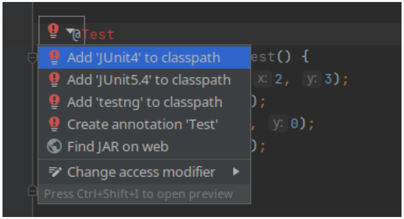
```
Using JUnit for testing code:
@Test
public void multiplyTest(){
    int z = multiply(2,3);
    assertEquas(6,z);
}

```

With the annotation @Test you tell java that JUnit is supposed to be used. The assertEquals function takes two values.

- first parameter: expected value by programmer
- second parameter: actual value in the program

Importing JUnit

When you use the annotation @Test for the first time, Intellij will propose to import it. Just press enter, and you are good to go.

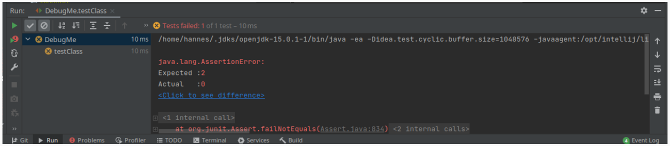


Add 'JUnit4' to classpath
Add 'JUnit5.4' to classpath
Add 'testng' to classpath
Create annotation 'Test'
Find JAR on web
Change access modifier
Press Ctrl+Shift+I to open preview

```
@Test
...est() {
... x: 2,  y: 3);
...);
...,  y: 0);
...);
```

Result of the Test

If you run a test, Intellij will open a view at the bottom of the screen.



```
/home/hannes/.jdks/openjdk-15.0.1-1/bin/java -ea -Didea.test.cyclic.buffer.size=1048576 -javaagent:/opt/intellij/li

java.lang.AssertionError:
Expected :2
Actual   :0
<Click to see difference>

<1 internal call>
    at org.junit.Assert.failNotEquals(Assert.java:834) <2 internal calls>
```

You can see if a test failed and what the actual value was.