Java

Exceptions

FSR Informatik

October 11, 2016

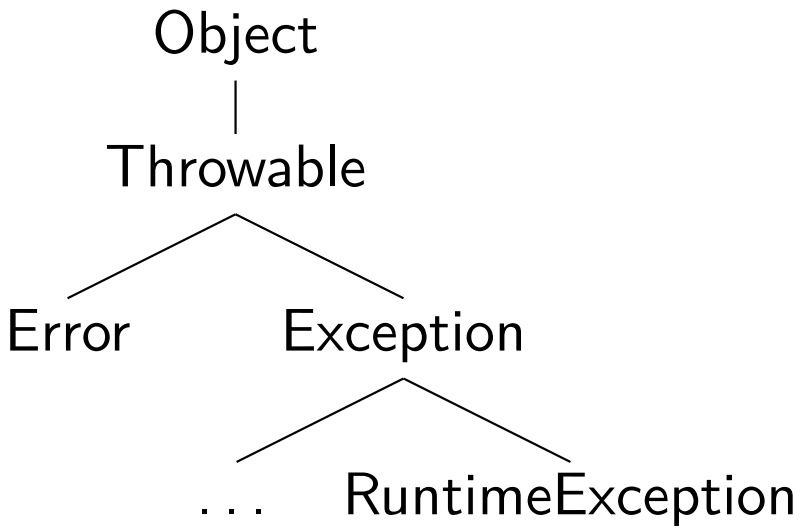
Overview

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 caused by the programer.

Errors are also unchecked.

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
5           try {
6               int a = 7 / 0;
7           } catch (ArithmeticException e) {
8               System.out.println("Division by zero.");
9           }
10      }
11  }
12
```

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
5               try {
6                   int a = 7 / 0;
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.

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 hops 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.