Java

Exceptions

Tobias Hanf, Manik Khurana

29. November 2021

Java-Course

Overview

1. Additional Control Structure
    - Switch
    - Break & Continue
    - return
2. Exceptions
    - Overview
    - Catching Exceptions
    - Throwing Exceptions

Additional Control Structure

Differentiate

```
1         public static void main (String[] args) {
2
3             int address = 2;
4
5             if (address == 1) {
6                 System.out.println("Dear Sir,");
7             } else if (address == 2) {
8                 System.out.println("Dear Madam,");
9             } else if (address == 4) {
10                System.out.println("Dear Friend,");
11            } else {
12                System.out.println("Dear Sir/Madam,");
13            }
14        }
15
```

Differentiate with Switch

```
1        public static void main (String[] args) {
2
3            int address = 2;
4
5            switch(address) {
6                case 1:
7                System.out.println("Dear Sir,");
8                break;
9                case 2:
10               System.out.println("Dear Madam,");
11               break;
12               case 4:
13               System.out.println("Dear Friend,");
14               break;
15               default:
16               System.out.println("Dear Sir/Madam,");
17               break;
18           }
19       }
20
```

Differentiate with Switch

Depending on a variable you can switch the execution paths using the keyword **switch**. This works with `int`, `char` and `String`.

The variable is compared with the value following the keyword case. If they are equal the program will enter the corresponding case block. If nothing fits the program will enter the default block.

```
1        public static void main (String[] args) {
2            switch(intVariable) {
3                case 1:
4                doSomething();
5                break;
6                default:
7                doOtherThings();
8                break;
9            }
10       }
11
```

Break

After the last command of the case block you can tell the program to leave using **break**.
Without **break** the program will continue regardless of whether a new case started, like in the example below.

```
1         public static void main (String[] args) {
2
3             switch( 1 ) {
4                 case 1:
5                 System.out.println("enter case 1");
6                 case 2:
7                 System.out.println("enter case 2");
8                 break;
9                 default:
10                System.out.println("enter default case");
11                break;
12            }
13        }
14
```

Break

The keyword **break** also stops the execution of loops.

```
1         public static void main (String[] args) {
2
3             for (int i = 1; 1 < 10; i++) {
4                 System.out.println("i = " + i);
5                 if (i == 3) {
6                     break;
7                 }
8             }
9         }
10
```

Continue

The keyword **continue** jumps to the next loop step.

```
1         public static void main (String[] args) {
2
3             for (int i = 1; 1 < 10; i++) {
4                 if (i == 3) {
5                     continue;
6                 }
7                 System.out.println("i = " + i);
8             }
9         }
10
```

return

Return statement gives back data

```
        class Numbers {
            private int a = 4;
            private int b = 5;

            public Number() {}

            public int addNumbers() {
                return a + b;
            }
        }

        ...

        Numbers numbers = new Numbers();
        int return = numbers.addNumbers();

```

Return works with every primitiv and complex data type.

return

```
1         public String getName() {
2             return "Klaus";
3         }
4
5         private Calculator calc;
6         public Calculator getCalcualtor() {
7             return calc;
8         }
9
```

return

Functions of type void do not have a return value. They are used for e.g. Setters

```
1         public void setNumber(int number) {
2             this.number = number;
3         }
4
```

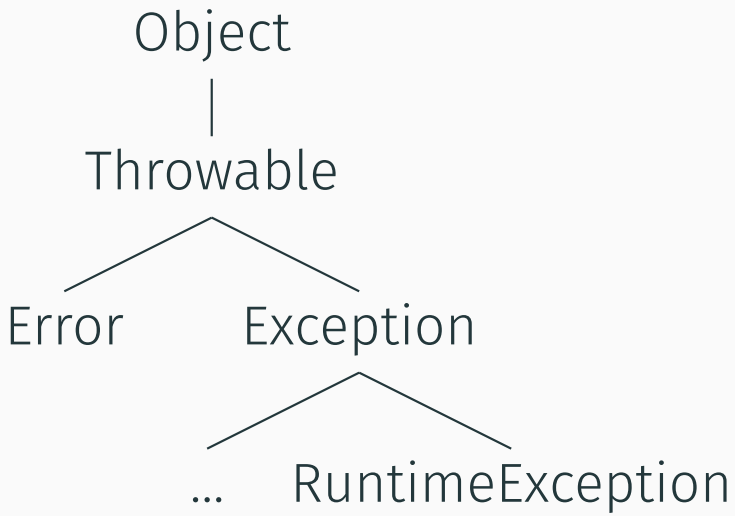
Exceptions

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/en/java/javase/11/docs/api/java.base/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

```
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.

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
```

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/en/java/javase/11/docs/api/index.html`

Creating new Exceptions

You can create and 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.