Java

`abstract` & Exceptions

Marcus Köhler

30. November 2018

Java-Kurs

Überblick

1. `abstract`
   - Intro
2. Exceptions
   - Überblick
   - Exceptions behandeln
   - Exceptions werfen

abstract

Abstrakte Klassen

Abstrakte Klassen repräsentieren eine „Mischung“ aus Interfaces und Vererbung, da sie sowohl Methoden & Konstruktoren als auch „Vorschriften“(Methodensignaturen) enthalten kann.
Abstrakte Klassen sind durch das Keyword `abstract` gekennzeichnet.

```
1 public abstract class AbstractExample {
2
3 }
```

Abstrakte Klassen

Abstrakte Klassen repräsentieren eine „Mischung“ aus Interfaces und Vererbung, da sie sowohl Methoden & Konstruktoren als auch „Vorschriften“(Methodensignaturen) enthalten kann.
Abstrakte Klassen sind durch das Keyword `abstract` gekennzeichnet.

```
1 public abstract class AbstractExample {
2
3 }
```

- Abstrakte Klassen sind gut geeignet um ein „Interface“ mit vielen `default`-Methoden zu realisieren.

Abstrakte Klassen

Abstrakte Klassen repräsentieren eine „Mischung“ aus Interfaces und Vererbung, da sie sowohl Methoden & Konstruktoren als auch „Vorschriften“(Methodensignaturen) enthalten kann.
Abstrakte Klassen sind durch das Keyword `abstract` gekennzeichnet.

```
1 public abstract class AbstractExample {
2
3 }
```

- Abstrakte Klassen sind gut geeignet um ein „Interface“ mit vielen `default`-Methoden zu realisieren.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.

Abstrakte Klassen

Abstrakte Klassen repräsentieren eine „Mischung“ aus Interfaces und Vererbung, da sie sowohl Methoden & Konstruktoren als auch „Vorschriften“(Methodensignaturen) enthalten kann.
Abstrakte Klassen sind durch das Keyword `abstract` gekennzeichnet.

```
1 public abstract class AbstractExample {
2
3 }
```

- Abstrakte Klassen sind gut geeignet um ein „Interface“ mit vielen `default`-Methoden zu realisieren.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.
- Abstrakte Klasse können von anderen abstrakten Klassen erben und können interfaces implementieren.

Abstrakte Klassen

Abstrakte Klassen repräsentieren eine „Mischung“ aus Interfaces und Vererbung, da sie sowohl Methoden & Konstruktoren als auch „Vorschriften“(Methodensignaturen) enthalten kann.
Abstrakte Klassen sind durch das Keyword `abstract` gekennzeichnet.

```
1 public abstract class AbstractExample {
2
3 }
```

- Abstrakte Klassen sind gut geeignet um ein „Interface“ mit vielen `default`-Methoden zu realisieren.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.
- Abstrakte Klasse können von anderen abstrakten Klassen erben und können interfaces implementieren.
- Normale und abstrakte Klassen können von abstrakten Klassen erben.

Methoden

Eine abstrakte Klasse kann sowohl „normale“ Methoden als auch *abstrakte* Methoden haben:

```
1 public abstract class AbstractExample {
2     public void printHello() {
3         System.out.println("Hello");
4     }
5
6     public abstract String getName();
7 }
```

Abstrakte Methoden bestehen(wie bei Interfaces) nur aus der Methodensignatur(und dem Keyword `abstract`). Sobald man eine abstrakte Methode deklariert, muss die enthaltende Klasse ebenfalls als abstrakt markiert werden.

Vererbung

Jede erbende Klasse einer abstrakten Klasse muss entweder alle abstrakten Methoden implementieren oder selbst abstrakt sein. Alle „normalen“ Methoden werden wie gewohnt übernommen.

```
1 public class Example extends AbstractExample {
2     @Override
3     public String getName() {
4         return "Example";
5     }
6 }
```

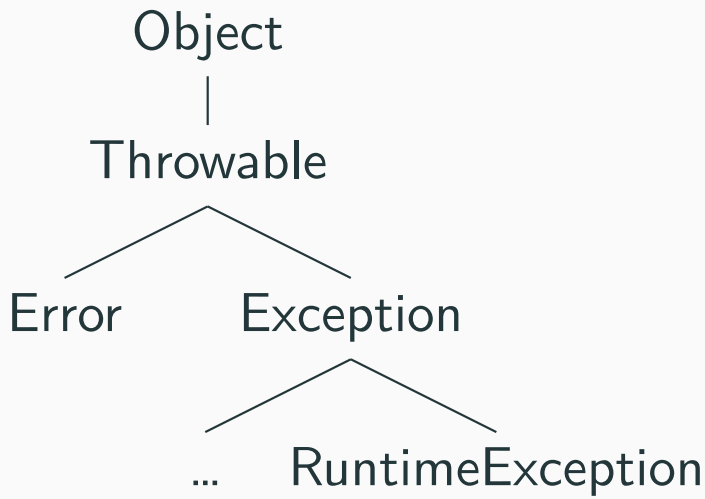
Exceptions

Exceptions

Idealerweise würden bei der Ausführung eines Programms nie Fehler oder generell unerwartetes Verhalten auftreten. Da die Welt allerdings nicht ideal ist, muss man sich als Programmierer (leider) mit solchen Situationen auseinandersetzen.

Java unterstützt Fehlerbehandlung nativ, mit sogenannten *Exceptions*. Exceptions separieren Fehlersituationen und -behandlung von „normalem“ Code.

Hierarchie



Jede Exception ist eine Subklasse von `Throwable`. `Error` ist ebenfalls eine Subklasse von `Throwable`, ist aber für schwerwiegende Fehler innerhalb der JVM bzw. des JRE reserviert. Da jede Exception auch „throwable“ ist, sagt man auch, dass Exceptions *geworfen* werden.

`https://docs.oracle.com/javase/10/docs/api/java/lang/Throwable.html`

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

- Checked exceptions müssen entweder sofort behandelt oder weitergegeben werden(dazu später mehr)

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

- Checked exceptions müssen entweder sofort behandelt oder weitergegeben werden(dazu später mehr)
- Die Ursache solcher Exceptions liegt meistens außerhalb des Programms(z.B. Dateien oder Datenbanken)

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

- Checked exceptions müssen entweder sofort behandelt oder weitergegeben werden(dazu später mehr)
- Die Ursache solcher Exceptions liegt meistens außerhalb des Programms(z.B. Dateien oder Datenbanken)

`RuntimeException` und ihre Subklassen dagegen sind sogenannte *unchecked Exceptions*.

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

- Checked exceptions müssen entweder sofort behandelt oder weitergegeben werden(dazu später mehr)
- Die Ursache solcher Exceptions liegt meistens außerhalb des Programms(z.B. Dateien oder Datenbanken)

`RuntimeException` und ihre Subklassen dagegen sind sogenannte *unchecked Exceptions*.

- Unchecked Exceptions müssen nicht zwingend behandelt oder weitergegeben werden.

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

- Checked exceptions müssen entweder sofort behandelt oder weitergegeben werden(dazu später mehr)
- Die Ursache solcher Exceptions liegt meistens außerhalb des Programms(z.B. Dateien oder Datenbanken)

`RuntimeException` und ihre Subklassen dagegen sind sogenannte *unchecked Exceptions*.

- Unchecked Exceptions müssen nicht zwingend behandelt oder weitergegeben werden.
- Solche Exceptions treten meistens aufgrund eines Fehlers im Programmcode auf, wie z.B. eine NullPointerException.

(Un-)Checked Exceptions

Alle Exceptions außer `RuntimeException` und ihrer Subklassen sind sogenannte *checked Exceptions*.

- Checked exceptions müssen entweder sofort behandelt oder weitergegeben werden(dazu später mehr)
- Die Ursache solcher Exceptions liegt meistens außerhalb des Programms(z.B. Dateien oder Datenbanken)

`RuntimeException` und ihre Subklassen dagegen sind sogenannte *unchecked Exceptions*.

- Unchecked Exceptions müssen nicht zwingend behandelt oder weitergegeben werden.
- Solche Exceptions treten meistens aufgrund eines Fehlers im Programmcode auf, wie z.B. eine NullPointerException.
- Prinzipiell kann jede Methode eine solche Exception erzeugen.

Intro

```java
public class Calc {

    public static void main(String[] args) {

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

        System.out.println(a);
    }
}
```

Division durch 0 wirft eine `ArithmeticException`, was eine Subklasse von `RuntimeException` ist. Deshalb ist `ArithmeticException` eine unchecked Exception und muss nicht zwingend behandelt werden.

try und catch

Sobald eine Exception auftritt, stürzt das Programm ab, sofern sie nicht behandelt wird. Auch wenn eine Exception nicht behandelt werden muss, kann man sie trotzdem mittels eines `try-catch`-Blocks behandeln:

```
1  public class Calc {
2      public static void main(String[] args) {
3
4          try {
5              int a = 7 / 0;
6          } catch (ArithmeticException e) {
7              System.out.println("Division by zero.");
8          }
9      }
10 }
```

Der `catch`-Block, auch *Exception handler* genannt, wird aufgerufen, wenn die angegebene Exception im `try`-Block auftritt.
Man kann innerhalb eines `catch`-Blocks auch mehrere Exceptions abfangen.

try und catch

Man kann (sollte aber nicht!) jede mögliche Exception mit einem `try-catch` abfangen:

```
1 try{
2     ...
3 } catch (Exception e) { ... }
```

Auch wenn es erlaubt ist, geht bei solchen Konstrukten sehr schnell verloren, was der genaue Fehler war.

Stacktraces

```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();
        }
    }
}
```

Ein *Stacktrace* gibt alle Methodenaufrufe(d.h. den *call-stack*) wieder, die zum Zeitpunkt der Exception aktiv waren.

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)
```

try-catch-finally

Bei einigen Ressourcen(z.B. Dateien) ist es wichtig, dass bestimmte beendende Methoden auch dann aufgerufen werden, wenn etwas schiefgeht. Hierzu gibt es den `finally`-Block:

```
1  public class Calc {
2      public static void main(String[] args) {
3          try {
4              int a = 7 / 0;
5          } catch (ArithmeticException e) {
6              System.out.println("Division by zero.");
7              e.printStackTrace();
8          } finally {
9              System.out.println("End of program.");
10         }
11     }
12 }
```

Ein `finally`-Block wird immer ausgeführt, egal ob tatsächlich eine Exception auftritt oder nicht.

Exceptions weitergeben

Unhandled exceptions können auch wieder geworfen(weitergegeben) werden. Dies wird mithilfe des Keywords `throws` am Ende der Methodensignatur getan:

```
1 public static int divide (int dividend, int divisor) throws
      ArithmeticException {
2     return dividend / divisor;
3 }
```

Durch das Keyword `throws` wird jede Exception, die den passenden Typ hat und innerhalb der Methode `int divide(...)` auftritt an die aufrufende Methode weitergegeben.

Exceptions weitergeben - Test

```
1  public class Calc {
2      public static int divide (int dividend, int divisor) throws
       ArithmeticException {
3          return dividend / divisor;
4      }
5
6      public static void main(String[] args) {
7
8          int a = 0;
9          try {
10             a = Calc.divide(7, 0);
11         } catch (ArithmeticException e) {
12             System.out.println("Division by zero.");
13             e.printStackTrace();
14         }
15     }
16 }
```

Exceptions weitergeben - Test

```java
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();
    }
}
```

In diesem Beispiel ist eine neue Methode(die aufrufende Methode) zum Stacktrace dazu gekommen: `java.lang.ArithmeticException: / by zero`

```
at Calc.divide(Calc.java:4)
at Calc.main(Calc.java:11)
```

Eigene Exceptions

Wie auch sonst alles in Java, kann man auch seine eigenen Exceptions erstellen und nutzen:

```
1 public class DivisionByZeroException extends Exception {
2 }
```

```
1 public static int divide (int dividend, int divisor) throws
      DivisionByZeroException {
2     if (divisor == 0) {
3         throw new DivisionByZeroException();
4     }
5     return divident / divisor;
6 }
```

Exceptions können „von Hand“ mit dem Keyword `throw` geworfen werden.

Eigene 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();
    }
}
```

Da `DivisionByZeroException` direkt von `Exception` erbt, ist sie eine checked Exception.

Übungsaufgabe

Such dir eine oder zwei Exceptions aus der Java Reference und schreibe Code, in dem sie auftritt.
Füge dann den Code hinzu, der diese Exception a) behandelt und den Stacktrace ausgibt und b) sie an die aufrufende Methode weitergibt.