3. Exceptions

Hintergrund: Programmieren auf der Basis von Verträgen

Kundenklasse
Methodenaufruf • Verpflichtung zur Einhaltung der Vorbedingung

Lieferantenklasse
Methodendefinition • Vorbedingung • Invarianten • Verpflichtung zur Einhaltung der Nachbedingung

☞ Ausnahme/Exception: Lieferantenklasse kann ihren Vertrag nicht erfüllen.

Hier werden Exceptions ausgelöst!

```
public class ExceptionTest {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);

        System.out.println(a + "/" + b + "=" + (a/b));
    }
}
```

- `java ExceptionTest`
  ☞ ArrayIndexOutOfBoundsException
- `java ExceptionTest 4 0`
  ☞ ArithmeticException
- `java ExceptionTest 4 a`
  ☞ NumberFormatException

Hier auch ...

```
public class ExceptionTest {
    public static void main(String[] args) {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);

        System.out.println(a + "/" + b + "=" + (a/b));
    }
}
```

für den Aufruf

```
java ExceptionTest 4 0
```

?

Exception

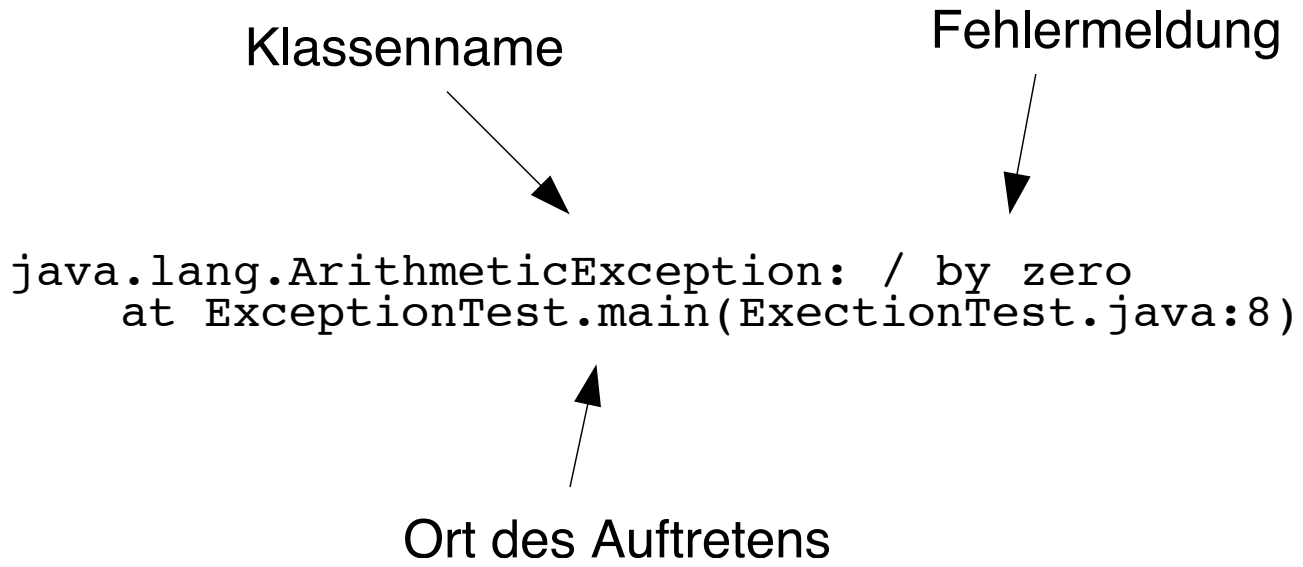
- Eine *Exception* ist ein Objekt, das Informationen über einen Programmfehler enthält.
- Eine Exception wird ausgelöst, um zu signalisieren, dass ein Fehler aufgetreten ist.

Vorteile von Exceptions:

- Für einen Klienten/Kunden (also den Programmierer, der eine Klasse des Lieferanten nutzt) ist es (fast) unmöglich, eine aufgetretene Exception zu ignorieren und einfach weiterzuarbeiten.
- Wenn der Kunde die Exception nicht behandelt, dann wird die laufende Anwendung beendet.

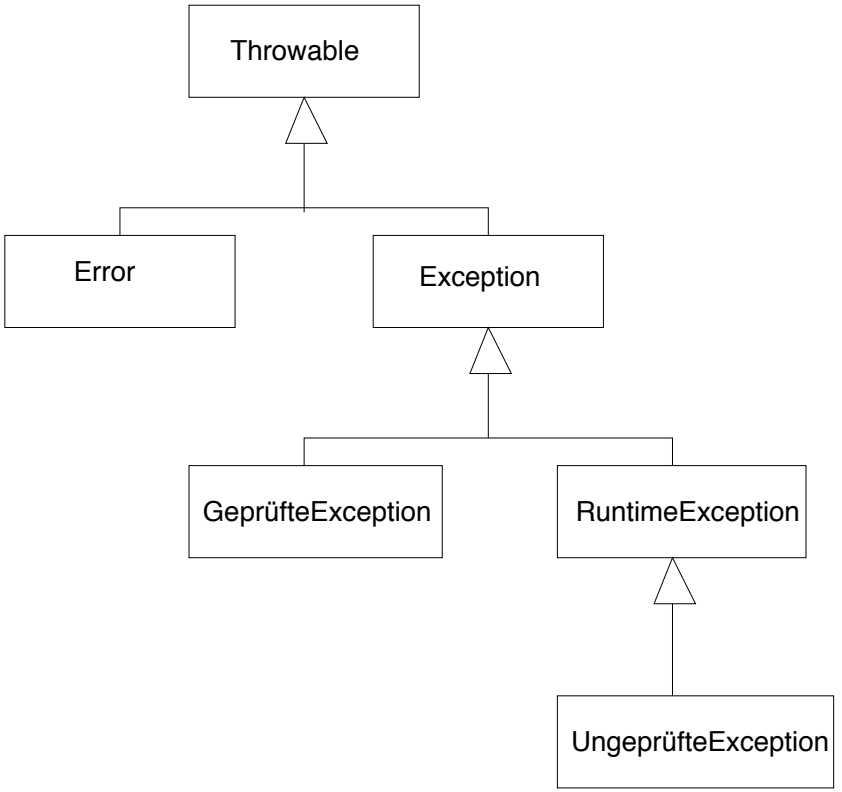
Die Klasse java.lang.Exception

- Exceptions sind Instanzen der Klasse `java.lang.Exception`
- bzw. einer Unterklasse von `java.lang.Exception`.
- `java.lang.Exception` definiert u.a. die folgenden Methoden:
  - `getMessage()`: Rückgabe der Fehlermeldung
  - `printStackTrace()`: Erzeugt die Ausgabe, die wir auf der Konsole sehen.



Exception-Klassen

- Eine Exception ist immer eine Instanz aus einer speziellen Vererbungshierarchie.
- Wir können selbst neue Exception-Typen definieren, indem wir Subklassen dieser Hierarchie erzeugen.
- Subklassen von `java.lang.Error` sind für Fehler des Laufzeitsystems vorgesehen.
- Für neue Exceptions erzeugen wir Subklassen von `java.lang.Exception`.
- Das Paket `java.lang` stellt bereits eine Reihe von vordefinierten Subklassen bereit.



Arten von Exceptions

- Java unterscheidet zwischen *geprüften* und *ungeprüften Exceptions*.
- `RuntimeException` und alle Subklassen der Klasse `RuntimeException` definieren ungeprüfte Exceptions.
- Alle anderen Subklassen von `Exception` definieren geprüfte Exceptions

Ungeprüfte Exceptions

- Dies sind Exceptions, bei deren Verwendung der Compiler keine zusätzlichen Überprüfungen vornimmt.
- Genauer: Der Compiler prüft nicht, ob sich der Klient um diese Exceptions kümmert.
- Ungeprüfte Exceptions sind für Fälle gedacht, die normalerweise nicht auftreten sollten, z.B. ausgelöst durch Programmfehler.

Beispiele:

```
ArrayIndexOutOfBoundsException
NullPointerException
NumberFormatException
ArithmeticException
```

Geprüfte Exceptions

- Geprüfte Exceptions sind Exception-Typen, bei deren Verwendung der Compiler zusätzliche Überprüfungen durchführt und einfordert.
- Ein Klient muss sich um geprüfte Exceptions kümmern.
- Geprüfte Exceptions sind für die Fälle gedacht, bei denen ein Klient damit rechnen sollte, dass eine Operation fehlschlagen kann. Hier sollte der Klient gezwungen werden, sich um den Fehler zu kümmern.

Beispiele:

`java.io.IOException`, `java.io.FileNotFoundException`

Faustregeln

Verwende als Programmierer einer Lieferantenklasse

ungeprüfte Exceptions ...

- ☞ ... in Situationen, die zu einem Programmabbruch führen sollten.
- ☞ ... bei Fehlern, die vermeidbar gewesen wären.

und geprüfte Exceptions ...

- ☞ ... bei Fehlern, die vom Klienten behandelt werden können.
- ☞ ... für Fälle, die außerhalb des Einflusses des Programmierers liegen.

Auswirkungen einer Exception

- Wenn eine Exception ausgelöst wird, wird die Ausführung der auslösenden Methode sofort beendet.
- In diesem Fall muss kein Wert von der Methode zurückgeliefert werden, auch wenn die Methode nicht als `void` deklariert wurde.
- Die Anweisung, die die auslösende Methode aufgerufen hat, konnte nicht korrekt abgearbeitet werden.
- Es besteht die Möglichkeit, solche Exceptions zu *fangen*.

Kontrollfluss bei Exceptions

```
Klasse obj = new Klasse();
try {
   . . .
   obj.method();
   . . .
}
catch ( MeineException e ) {
   /* spez. F.-Behandlung */
}
catch ( Exception e ) {
   /* allg. F.-Behandlung */
}
finally {
   /* Aufräumarbeiten */
}
...
```

```
public class Klasse
{
   public void method()
          throws MeineException
   {
      . . .
      /* Fehlererkennung */
      throw new MeineException("...");
      . . .
   }
}
```

Die Konstrukte try, catch, finally, throw und throws

`try`
: Definiert einen Block, innerhalb dessen Exceptions auftreten können.

`catch`
: Definiert einen Block, der die Fehlerbehandlung für eine Ausnahme durchführt.

`finally`
: Definiert einen Block, der Aufräumarbeiten durchführt.

`throw`
: Erzeugt eine Ausnahme.

`throws`
: Deklariert eine Ausnahme für eine Methode.

Geprüfte Exceptions: throws

- Der Compiler fordert, dass eine Methode, die eine geprüfte Exception auslösen (oder weiterreichen) kann, dies im Methodenkopf angibt.
- Syntaxbeispiel:

```
public void speichereInDatei(String dateiname)
   throws IOException
```

- Die Angabe von ungeprüften Exception-Typen ist erlaubt aber nicht erforderlich.
- Hinter `throws` können durch Komma getrennt auch mehrere Exception-Typen angegeben werden.

```
public void kopiereDatei(String quelldatei, String zieldatei)
   throws FileNotFoundException, IOException
```

Exception-Handler

- Ein *Exception-Handler* ist ein Programmabschnitt, der Anweisungen schützt, in denen eine Exception auftreten könnte.
- Der Exception-Handler definiert Anweisungen zur Meldung oder zum Wiederaufsetzen nach einer aufgetretenen Exception.

Syntax:

```
try {
   geschützte Anweisungen
}
catch (ExceptionTyp e) {
   Anweisungen für die Behandlung
}
finally {
   Anweisungen für alle Fälle
}
```

Zwang zur Behandlung von Exceptions

- Eine Anweisung innerhalb der Methode `caller()` ruft eine Methode `method()` auf, die eine geprüfte Exception `E` auslösen kann.
- Dann muss für `caller()` folgendes gelten:
  Entweder in `caller()` wird `E` durch einen Exception-Handler behandelt
  oder `caller()` macht mittels `throws` deutlich, dass die Exception `E` weitergereicht wird.

```
public class Klasse {
    void method() throws SomeException { ... }
}
```

Falsch:

```
public class AndereKlasse {
    void caller() {
        Klasse k = new Klasse();
        k.method();
    }
}
```

Richtig:

```
public class AndereKlasse {
    void caller() throws SomeException {
        Klasse k = new Klasse();
        k.method();
    }
}
```

oder ...

... in `caller()` einen Exception-Handler für `SomeException` verwenden:

```
public class AndereKlasse {
    void caller() {
        Klasse k = new Klasse();
        try {
            k.method();
        }
        catch (SomeException e) {
            ...
        }
        ...
    }
}
```

Propagierung von Exceptions

- Eine ausgelöste Exception `E` wird entlang der Aufrufhierarchie propagiert, bis ein geeigneter Exception-Handler gefunden wurde.
- Ein Exception-Handler ist geeignet, wenn er eine `catch`-Klausel hat, deren Exception-Typ gleich `E` ist oder ein Obertyp davon.
- Die `catch`-Klauseln werden daraufhin in der Reihenfolge geprüft, wie sie im Quelltext angegeben sind.
- Die erste passende `catch`-Klausel wird genommen!

Auswahl des `catch`-Blocks

☞ Die erste passende `catch`-Klausel wird genommen!

falsch:

```
try {
   ...
}
catch (Exception e) {
   ...
}
catch (SpezielleException se) {
   ...
}
```

richtig:

```
try {
   ...
}
catch (SpezielleException se) {
   ...
}
catch (Exception e) {
   ...
}
```

Aufräumen mit `finally`

- Nach einer oder mehreren `catch`-Klauseln kann optional eine `finally`-Klausel folgen.
- Die Anweisungen im Block zu `finally` werden immer ausgeführt:
  - Wenn keine Exception ausgelöst wurde,
  - wenn eine Exception ausgelöst und durch eine `catch`-Klausel behandelt wird und auch
  - wenn eine Exception ausgelöst wird, für die keine passende `catch`-Klausel vorhanden ist (d.h. die Exception wird weitergereicht).
- Typische Anwendung: Freigabe von Ressourcen unabhängig vom Auftreten eines Fehlers.

Auslösen von Exceptions: throw

- Exceptions werden mit Hilfe der `throw`-Klausel ausgelöst.
- Hinter `throw` gibt man ein Exception-Objekt an.
- Typischerweise erzeugt man hierzu ein neues Exception-Objekt mittels `new`.
- Als Konstruktorparameter ist eine Fehlermeldung zulässig.

Beispiel:

```
throw new Exception("Parameterwert ungueltig!");
```

Maßgeschneiderte Exceptions

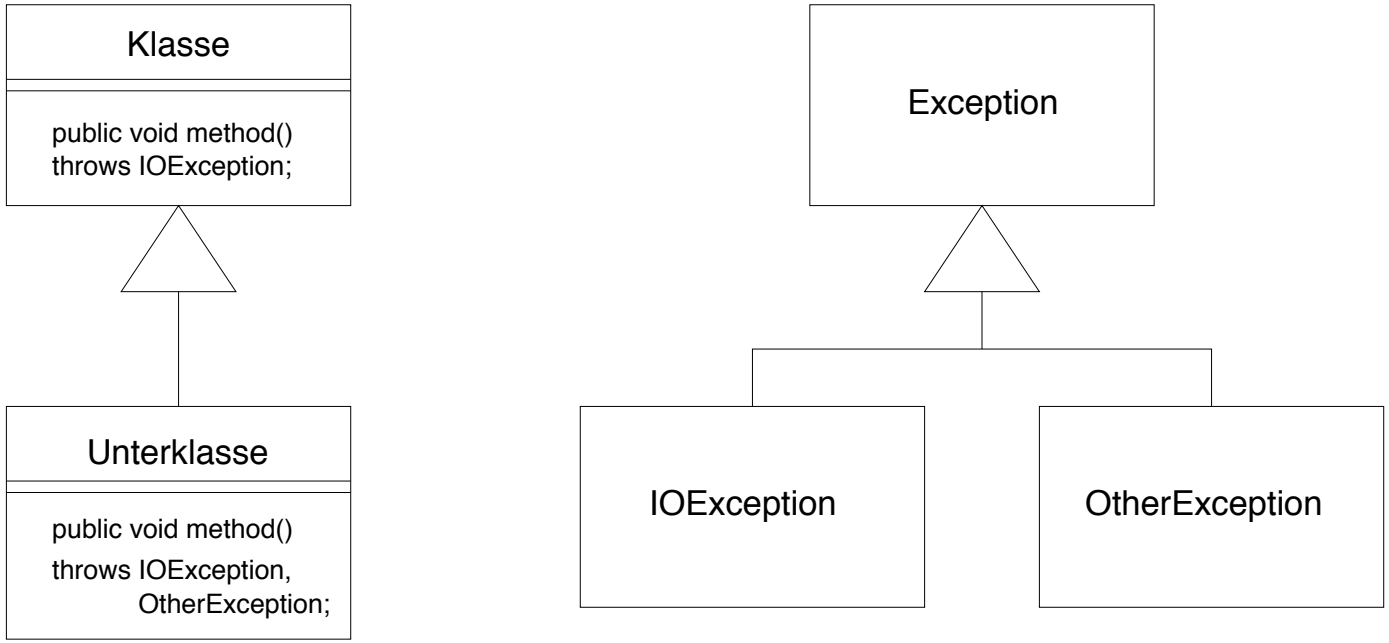
- Für eigene geprüfte Exceptions: Klasse definieren, die abgeleitet von `Exception` ist.
- Ansonsten gelten die üblichen Regeln der Vererbung.

```
public class MyException extends Exception {
   ...
   public MyException() {
      super();
   }

   public MyException(String msg) {
      super(msg);
   }
   ...
}
```

Exceptions und Vererbung

Das folgende Design ist in Java (vernüftigerweise) nicht erlaubt! Warum?



Exceptions und Vererbung (2)

- Durch die `throws`-Klausel werden Exceptions Bestandteil des Vertrages zwischen Klient und Lieferant.
- Sie regeln, was im Falle einer Vertragsverletzung passiert.

☞ Prinzip der Ersetzbarkeit: Spezialisierte Methoden dürfen keine neuen geprüften Exceptions definieren.

- Konsequenz: Das Design der vorangegangenen Folie ist in Java nicht erlaubt!
- Korrekte Lösung: `method()` von `Unterklasse` darf
  - `IOException` auslösen,
  - eine Spezialisierung von `IOException` auslösen oder
  - keine Exception auslösen.
- Und wenn `method()` von `Unterklasse` andere Methoden benutzt, die andere geprüfte Exceptions als `IOException` auslösen können?

- Dann muss `method()` von `Unterklasse` diese Exceptions fangen!
- Eine Umsetzung auf `IOException` ist dabei erlaubt.

```
public class Unterklasse {
    ...
    public void method() throws IOException {
        ...
        try {
            this.otherMethod();
        }
        catch(OtherException e) {
            throw new IOException(...);
        }
        ...
    }
}
```

Schnittstellen und Exceptions

Zur Schnittstellendefinition gehört auch die Deklaration von möglichen Exceptions.

Folgende Konstruktion ist nicht erlaubt:

```
interface Schnittstelle {
   void methode();
}

public class Klasse implements Schnittstelle {
   public void methode() throws Exception { ... }
}
```

☞ Widerspruch zum Prinzip der Ersetzbarkeit.

Lösung: Die Exception muss in der Schnittstelle deklariert werden!

```
interface Schnittstelle {
   void methode() throws Exception;
}
```

- Damit dürfte die Implementierung von `methode()` `Exception` oder eine beliebige Unterklasse von `Exception` auslösen.
- Nicht gestattet: allgemeinere Exceptions oder weitere Exceptions

Und wenn die Definition von `Schnittstelle` nicht angepasst werden kann (z.B. weil `Schnittstelle` aus einer Bibliothek stammt)?

- Dann darf die implementierende Klasse für `methode()` keine geprüften Exceptions auslösen oder weiterreichen!
- Innerhalb von `methode()` müssen alle möglichen geprüften Exception gefangen werden!

Assertions

- Seit Java 1.4 gibt es in Java das Konzept der *Zusicherung (assertion)*.
- Eine Zusicherung ist ein Ausdruck, mit dem Einschränkungen definiert werden, welche die erlaubten Zustände oder das Verhalten von Objekten betreffen.
- Zusicherungen gehören eigentlich zur Spezifikation bzw. Modellierung.
- Zusicherungen im Quelltext prüfen, ob die Spezifikation verletzt ist (Korrektheit).
- Verletzte Zusicherungen lösen in Java ein Objekt vom Typ `AssertionError` aus (Untertyp von `Error`).
- Eine verletzte Zusicherung heißt: Das Programm bzw. die Klasse erfüllt nicht die geforderte Spezifikation.
- Man beachte: `Error` statt `Exception`!

Zusicherungen im Quelltext

- Die Überprüfung erfolgt mit dem Schlüsselwort `assert`:

  ```
  assert boolscher-Ausdruck;
  ```

- Liefert der boolesche Ausdruck den Wert `true`, wird das Programm fortgesetzt.
- Ansonsten wird ein `AssertionError` ausgelöst.
- Optional kann für eine Assertion ein Fehlertext angegeben werden:

  ```
  assert boolscher-Ausdruck : String-Ausdruck;
  ```

Zusicherungen compilieren und aktivieren

- Sie benötigen eine Java-Version ≥ 1.4.
- Damit Kompatibilität zu älteren Versionen gewahrt ist, müssen Assertions für den Compiler durch die Option `-source` aktiviert werden:

```
javac -source 1.4 Klasse.java
```

- Assertions sind auch in der virtuellen Maschine standardmäßig nicht aktiviert. Mit Hilfe der Option `-ea` werden die Assertions aktiviert.

```
java -ea Klasse
```