TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 05: Fehlerbehandlung

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Laufzeitfehler

***Erinnerung*: Zu Beginn von Kapitel 03a hatten wir uns Fehler systematisch angesehen und dabei unterschieden zwischen Fehlern, die der Compiler findet – und dann den Quelltext gar nicht erst übersetzt – und Fehlern, die erst zur Laufzeit auftreten und dann zum Abbruch der Ausführung des Programm führen.**

Im vorliegenden Kapitel geht es um Laufzeitfehler und darum, wie verhindert werden kann, dass die Ausführung des Programms abgebrochen wird.

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Was ist das allgemein?

- **Kann nicht vom Compiler entdeckt werden**
- **Laufzeitsystem bricht Ablauf ab**
- **Call-Stack mit Zeilennummern wird in der Fehlermeldung angezeigt**
  - ➢ **Man sieht sofort, wo der Fehler aufgetreten ist**

Die ersten beiden Punkte fassen noch einmal zusammen, was ein Laufzeitfehler ist und was er bewirkt. Den dritten Punkt haben Sie sicher schon bei dem einen oder anderen Laufzeitfehler selbst gesehen.

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele für Laufzeitfehler:

- **Durch 0 teilen bei int**
- **Auf Attribut / Methode von null zugreifen**
- **Auf nichtexistierenden Array-Index zugreifen**
- **Downcast auf Referenztyp, so dass der dynamische Typ weder gleich dem statischen Typ noch Subtyp davon ist**

Hier noch einmal ein paar Beispiele von Laufzeitfehlern in Java, die wir auch schon in Kapitel 03a kurz angesprochen hatten.

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Manche Laufzeitfehler durch Sprache ausgeschlossen:

- **Nur Methoden aufrufbar, die im statischem Typ definiert sind**
- **Keine Objekte von abstrakten Klassen oder Interfaces**
- **Durch 0 teilen bei float und double: *_INFINITY und NaN**
- **Arithmetischer Überlauf bei ganzzahligen Typen: Wird einfach „schmerzlos“ ausgeführt**

In wahrscheinlich jeder Sprache sind einige Möglichkeiten für Laufzeitfehler von vornherein ausgeschlossen. Dies sind ein paar Beispiele dafür in Java.

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Manche Laufzeitfehler durch Sprache ausgeschlossen:

- **Nur Methoden aufrufbar, die im statischem Typ definiert sind**
- **Keine Objekte von abstrakten Klassen oder Interfaces**
- **Durch 0 teilen bei float und double: *_INFINITY und NaN**
- **Arithmetischer Überlauf bei ganzzahligen Typen: Wird einfach „schmerzlos“ ausgeführt**

Im Prinzip könnte man ja alle Methoden aufrufen, die im dynamischen Typ vorhanden sind. In einigen objektorientierten Sprache wie Smalltalk ist das tatsächlich so. Aber falls man sich geirrt hat und der dynamische Typ doch nicht derjenige ist, den man erwartet hat, dann existiert die Methode für diesen Typ nicht, und es gibt einen Laufzeitfehler. Beim Design von Java hat man sich hingegen dafür entschieden, diese Art von Laufzeitfehlern von vornherein zu vermeiden, indem der Compiler einen Aufruf einer Methode, die nicht schon im statischen Typ definiert ist, nicht übersetzt.

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Manche Laufzeitfehler durch Sprache ausgeschlossen:

- **Nur Methoden aufrufbar, die im statischem Typ definiert sind**
- **Keine Objekte von abstrakten Klassen oder Interfaces**
- **Durch 0 teilen bei float und double: *_INFINITY und NaN**
- **Arithmetischer Überlauf bei ganzzahligen Typen: Wird einfach „schmerzlos“ ausgeführt**

Auch dies dient der Verhinderung von Laufzeitfehlern, denn mit einem Objekt einer abstrakten Klasse oder eines Interface könnte man ja eine Methode aufrufen, die zwar *definiert*, aber nicht *implementiert* ist.

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Manche Laufzeitfehler durch Sprache ausgeschlossen:

- **Nur Methoden aufrufbar, die im statischem Typ definiert sind**
- **Keine Objekte von abstrakten Klassen oder Interfaces**
- **Durch 0 teilen bei float und double: *_INFINITY und NaN**
- **Arithmetischer Überlauf bei ganzzahligen Typen: Wird einfach „schmerzlos“ ausgeführt**

Für ganze Zahlen ergibt Division durch 0 einen Laufzeitfehler. Bei gebrochenzahligen Typen hat man hingegen in den Klassen Float und Double spezielle Klassenkonstanten POSITIVE_INFINITY und NEGATIVE_INFINITY eingerichtet – in der Folien flapsig *_INFINITY genannt –, die unter anderem als Ergebnis bei Division eines positiven beziehungsweise negativen Wertes durch 0 herauskommen (NaN bei Division 0 durch 0, NaN = not a number).

***Erinnerung*: Kapitel 01b, Abschnitt „Allgemein: Primitive Datentypen“.**

***Vorgriff*: Kapitel 11 zur internen Zahldarstellung.**

Laufzeitfehler

TECHNISCHE UNIVERSITÄT DARMSTADT

Manche Laufzeitfehler durch Sprache ausgeschlossen:

- **Nur Methoden aufrufbar, die im statischem Typ definiert sind**
- **Keine Objekte von abstrakten Klassen oder Interfaces**
- **Durch 0 teilen bei float und double: *_INFINITY und NaN**
- **Arithmetischer Überlauf bei ganzzahligen Typen: Wird einfach „schmerzlos“ ausgeführt**

Bei ganzzahligen Datentypen ist ein typischer Fehler, dass das Ergebnis einer arithmetischen Operation nicht mehr in den Datentyp passt. Zum Beispiel könnte die Summe zweier Zahlen größer als die größte darstellbare Zahl sein. Aus pragmatischen Gründen hat man sich dafür entschieden, diesen Fall nicht abzufangen, sondern es wird ohne Laufzeitfehler einfach das Ergebnis zurückgeliefert, das in einem solchen Fall unsinnigerweise berechnet wird.

***Vorgriff*: Das sehen wir uns in Kapitel 11 an.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Exceptions

Java Oracle Tutorials: Exceptions

Dieses Kapitel hat ein größeres Thema, Exceptions, und dann noch zwei kürzere Abschnitte, die beide in unterschiedlicher Form mit Assertions, also Zusicherungen zu tun haben.

Exceptions sind Fehlerbehandlungen während der ganz normalen Ausführung eines Programms im realen Einsatz. Mit Exceptions kann man dafür sorgen, dass die Abarbeitung eines Programms auch in unerwarteten, unerwünschten Fällen nicht vom Laufzeitsystem beendet wird, sondern statt dessen eine selbstgeschriebene Fehlerbehandlung durchgeführt wird und das Programm danach ordnungsgemäß weiterläuft.

Exception-Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Klasse java.lang.Exception
- Alle von java.lang.Exception direkt oder indirekt abgeleiteten Klassen

Rein formal sind Exception-Klassen ganz einfach definiert: erstens eine Klasse namens Exception in Package java.lang; zweitens alle Klassen, die direkt oder indirekt von Klasse Exception abgeleitet sind. Etliche davon finden sich in der Java-Standardbibliothek in verschiedenen Packages, aber wie wir gleich sehen werden, kann man natürlich auch eigene Exception-Klassen definieren.

TECHNISCHE UNIVERSITÄT DARMSTADT

Exceptions werfen und fangen

Werfen und Fangen ist der Grundmechanismus von Exceptions.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
       throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Eine Exception wird in einer Methode geworfen und in *der* Methode, die diese aufruft, gefangen oder weitergereicht. Hier ein kleines, durchaus realistisches Beispiel, zuerst das Werfen.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
       throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Die gezeigte Methode erwartet, dass c ein Kleinbuchstabe ist, und liefert den zugehörigen Großbuchstaben zurück.

***Erinnerung*: Wir haben diesen Code im Wesentlichen schon einmal in einfacherer Form im Kapitel 01b im Abschnitt zu Konversionen gesehen.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
        throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Hier der erste Unterschied zu bisherigen Methoden: Mit Schlüsselwort throws am Ende des Kopfes einer Methode wird angekündigt, dass diese Methode Exceptions wirft.

***Erinnerung*: Im Kapitel zu Methoden, Abschnitt zum Kopf einer Methode, haben wir schon kurz gesehen, dass zwischen Parameterliste und Rumpf eine throws-Klausel stehen kann.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
        throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Nach throws stehen die Exception-Klassen, die von der Methode toUpperCase potentiell geworfen werden. In diesem Beispiel wird nur eine einzige Exception-Klasse hier deklariert, nämlich die Basisklasse Exception selbst. Wir werden später Beispiele mit mehreren Exception-Klassen – also von Exception abgeleiteten Klassen – in der throws-Klausel sehen.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
        throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Der boolesche Ausdruck in dieser if-Abfrage ist genau dann wahr, wenn das Zeichen im char-Parameter c *kein* Kleinbuchstabe ist. In diesem Fall würde die Methode natürlich ein Unsinnsergebnis zurückliefern. Dass die Parameter eine essentielle Vorbedingung nicht erfüllen, ist eine typische Situation, um eine Exception zu werfen.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
      throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Mit Schlüsselwort throw wird eine Exception nun tatsächlich geworfen.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
        throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Eine Exception ist ganz konkret ein Objekt der gewählten Exception-Klasse und muss daher mit new erst erzeugt werden, bevor sie geworfen werden kann.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
        throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Klasse Exception hat einen Konstruktor mit einem String als Parameter. Damit können wir der Exception eine Botschaft mit auf ihren Weg geben.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
      throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Beachten Sie, dass das throws oben im Methodenkopf und das throw in der Anweisung zwei verschiedene Schlüsselwörter sind. Bei beiden passt die englische Verbform jeweils: Im Methodenkopf wird mit throws angekündigt, dass die Methode potentiell eine Exception wirft, und throw im Methodenrumpf ist zu verstehen als Imperativ.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
        throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Im Methodenkopf wird gesagt, welche Exception-Klassen geworfen werden dürfen. In der throw-Anweisung muss das geworfene Objekt von einer dieser Klassen oder davon abgeleiteten Klassen sein. In diesem kleinen Beispiel ist nur eine Exception-Klasse deklariert, und ein Objekt von der wird auch geworfen, das passt also.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUpperCase ( char c ) throws Exception {
    if ( c < ´a´ || c > ´z´ )
      throw new Exception ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Was passiert nun bei Programmausführung, wenn eine throw-Anweisung ausgeführt wird?

Mit einer throw-Anweisung wird wie mit einem return die Ausführung der Methode sofort beendet. Im Unterschied zu return wird kein Wert zurückgeliefert, sondern das in der Anweisung erzeugte Exception-Objekt wird *geworfen*.

Was das eigentlich heißt, wird klarer, wenn wir uns jetzt ansehen, wie die geworfene Exception dann gefangen wird. Das geschieht, wie gesagt, in der Methode, von der aus die Methode toUpperCase aufgerufen worden ist.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Eine solche beispielhafte Methode schauen wir uns als nächstes an, genauer gesagt nur den Ausschnitt, der den Aufruf der Methode toUpperCase enthält.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

In x wird der Parameterwert für den Aufruf von toUpperCase gespeichert. Der ist aber dummerweise kein Kleinbuchstabe.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

In y soll – eigentlich – der Großbuchstabe zu x gespeichert werden. Das wird natürlich schiefgehen, da in x nun einmal kein Kleinbuchstabe steht.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Hier ist der Aufruf. Wir nehmen an, dass abc eine Variable von der Klasse ist, in der die Methode toUpperCase definiert ist.

***Nebenbemerkung*: Methode toUpperCase wäre normalerweise static definiert und könnte – und sollte – dann auch mit dem Namen der Klasse aufgerufen werden. Aber dann hätte der ganze Methodenrumpf von toUpperCase auf den vorhergehenden Folien nicht in derselben Schriftgröße auf eine Zeile gepasst.**

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
    y = abc.toUpperCase ( x );
    System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
    System.out.print ( exc.getMessage() );
}
```

Wenn für eine Methode definiert ist, dass sie potentiell Exceptions wirft – egal von welcher Exception-Klasse –, dann kann diese Methode nicht einfach so aufgerufen werden, sondern der Aufruf muss in einen try-Block wie hier. In einem solchen try-Block könnten auch mehrere Methoden aufgerufen werden, die Exceptions werfen, und es können auch zusätzlich beliebig viele andere Anweisungen stehen, die *keine* Exceptions werfen.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Wenn Methode toUppercase *keine* Exception wirft, dann wird Methode toUpperCase ordnungsgemäß mit einer Rückgabe beendet, die in y gespeichert wird, und diese Schreibausgabe unmittelbar danach wird ausgeführt. In diesem Fall wird der catch-Block übersprungen, und hinter dem catch-Block geht es weiter mit der Ausführung des Programms. Wirft Methode toUpperCase hingegen eine Exception, dann wird die Ausführung des try-Blockes zusammen mit der Ausführung von toUpperCase sofort beendet, insbesondere die Schreibanweisung *nicht* ausgeführt, und als nächstes wird der catch-Block ausgeführt.

Eine entfernte Ähnlichkeit zu if-else ist sicherlich unverkennbar.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Ein try-Block kommt *nie* allein, sondern *immer* mit einem oder manchmal auch mehreren catch-Blöcken, die unmittelbar danach kommen (auch ein finally-Block ist möglich, wird aber hier nicht behandelt). Wir kommen später zu dem Fall von mehreren catch-Blöcken, hier erst einmal nur einer.

Zwischen den Blöcken dürfen Whitespaces sein, wie sie im Kapitel zu lexikalischen Bestandteilen besprochen wurden, sonst darf nichts zwischen try-Block und catch-Block beziehungsweise zwischen zwei aufeinanderfolgenden catch-Blöcken sein.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Das ist die Stelle, an der die geworfene Exception wieder auftaucht. Die in toUpperCase geworfene und hier nun gefangene Exception bekommt einen Namen. Wir nehmen den vielleicht etwas phantasielosen, aber häufig gewählten Namen exc.

Der Scope des Identifiers exc ist der catch-Block.

***Erinnerung*: In Kapitel 03a hatten wir den Scope einer Definition eingeführt als den Bereich im Quelltext, in dem der Identifier gemäß seiner Definition verwendet werden kann.**

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Die Anweisungen im catch-Block – in diesem kleinen Beispiel nur eine einzige Anweisung – werden genau dann ausgeführt, wenn im try-Block eine Exception geworfen wurde. Wurde im try-Block *keine* Exception geworfen, dann wird der gesamte catch-Block, wie gesagt, ohne Effekt einfach übersprungen.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Klasse Exception hat eine Methode getMessage, die genau den String zurückliefert, den wir dem Objekt im Konstruktor beim Werfen mitgegeben haben. Als wir uns den Konstruktor eben angesehen haben, hatten wir diesen String die *Botschaft* genannt, die dem Objekt mitgegeben wird. Das passt: message ist das englische Wort für Botschaft.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   System.out.print ( exc.getMessage() );
}
```

Wir sehen schon an diesem kleinen Beispiel ganz gut, wofür Exceptions generell da sind. In der Ausführung einer Methode kann eine Ausnahmesituation auftreten, mit der diese Methode nicht umzugehen weiß. Dann beendet sich diese Methode sofort und delegiert das Problem an die aufrufende Methode, indem eine Exception geworfen wird. Die aufrufende Methode fängt die Exception und behandelt im catch-Block das Problem.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
   y = abc.toUpperCase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( Exception exc ) {
   exc.printStackTrace();
}
```

Noch eine kleine Variation, die nur bei der Programmentwicklung empfehlenswert ist und im realen Einsatz des Programms besser entfernt oder auskommentiert sein sollte, damit der Endnutzer Ihres Programms nicht mit technischen Details behelligt wird: Klasse Exception hat eine Methode printStackTrace, mit der der Call-Stack auf dem Bildschirm ausgegeben wird, und zwar der Inhalt des Call-Stack zu dem Zeitpunkt, an dem die Exception geworfen wurde.

***Erinnerung*: Den Call-Stack hatten wir uns im Kapitel 01e, Abschnitt „Ausführung von Methoden“, angesehen.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUppercase ( char c )
throws IndexOutOfBoundsException {
    if ( c < ´a´ || c > ´z´ )
       throw new IndexOutOfBoundsException
                                  ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Eben haben wir beispielhaft mit der Klasse Exception gearbeitet. Eine der indirekt von Exception abgeleiteten Klassen in der Java-Standardbibliothek heißt IndexOutOfBoundsException. Falls diese Exception-Klasse geworfen werden soll, sieht alles genauso aus, nur dass der Name der Exception-Klasse ein anderer ist.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public char toUppercase ( char c )
throws IndexOutOfBoundsException {
    if ( c < ´a´ || c > ´z´ )
       throw new IndexOutOfBoundsException
                              ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Klasse IndexOutOfBoundsException hat ebenfalls einen Konstruktor mit einem String-Parameter, der wieder die Botschaft ist.

***Erinnerung*: Konstruktoren werden nicht vererbt, daher ist dieser Konstruktor in Klasse IndexOutOfBoundsException definiert.**

Er macht aber dasselbe wie der Konstruktor gleicher Signatur von Exception.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
    y = abc.toUppercase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( IndexOutOfBoundsException exc ) {
    System.out.print ( exc.getMessage() );
}
```

Wie beim Werfen der Exception, hat sich auch beim Fangen der Exception nichts außer dem Namen der Exception-Klasse geändert.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char x = ´3´;
char y;
try {
    y = abc.toUppercase ( x );
   System.out.print ( “Reached this line?“ );
}
catch ( IndexOutOfBoundsException exc ) {
    System.out.print ( exc.getMessage() );
}
```

Klasse IndexOutOfBoundsException hat Methode getMessage von Klasse Exception geerbt.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/**
• @param  c  the character to be converted to upper case
• @throws  class IndexOutOfBoundsException if c ist not a
                lower-case letter
• @return  upper case of c
*/

public char toUpperCase ( char c ) throws IndexOutOfBoundsException {
    if ( c < ´a´ || c > ´z´ )
        throw new IndexOutOfBoundsException ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

***Erinnerung*: In Kapitel 01e, Abschnitt zu Methoden mit Parametern, hatten wir javadoc für Methoden gesehen.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/**
• @param  c  the character to be converted to upper case
• @throws  class IndexOutOfBoundsException if c ist not a
           lower-case letter
• @return  upper case of c
*/

public char toUpperCase ( char c ) throws IndexOutOfBoundsException {
    if ( c < ´a´ | | c > ´z´ )
        throw new IndexOutOfBoundsException ( “No lower-case letter!“ );
    return (char)( c – ´a´ + ´A´ );
}
```

Diese **Klausel hatten wir dabei noch nicht gesehen. Ihre Intention sollte offensichtlich sein: Dokumentation, unter welchen Umständen welche Exceptionklasse geworfen wird.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Methoden mit throws-Klauseln überschreiben

***Erinnerung*: In Kapitel 03c, Abschnitt „Signatur und Überschreiben / Überladen von Methoden", hatten wir uns angeschaut, inwieweit die Methodenköpfe der überschreibenden und der überschriebenen Methode voneinander abweichen dürfen. Dabei hatten wir auch schon erste Andeutungen zu Exceptions gemacht, die wir jetzt aufarbeiten.**

Methoden überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m () throws Exception { ..... }
}

public class Y extends X {
   public void m () throws IndexOutOfBoundsException {
      .....
   }
}
```

Wieder ein kleines, rein illustratives Beispiel.

Methoden überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m () throws Exception { ..... }
}

public class Y extends X {
   public void m () throws IndexOutOfBoundsException {
      .....
   }
}
```

Die Regel ist ganz einfach: In der überschreibenden Methode darf eine Exception-Klasse ersetzt werden durch eine direkt oder indirekt abgeleitete Exception-Klasse. Zum Beispiel darf Exception in der Methode m der Basisklasse ersetzt werden durch IndexOutOfBoundsException in Methode m der abgeleiteten Klasse.

Methoden überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y();
try {
    a.m();
}
catch ( Exception exc ) {
    ..........
}
```

Denn dann gibt es keine Probleme in Situationen, in denen statischer und dynamischer Typ unterschiedlich sind.

Methoden überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y();
try {
    a.m();
}
catch ( Exception exc ) {
    ..........
}
```

Im catch-Block muss diejenige Exception-Klasse gefangen werden, die die Methode m in X wirft, also Klasse Exception. Wenn nun Y der dynamische Typ von a ist, wird eine IndexOutOfBoundsException geworfen. Aber das ist kein Problem, denn wie immer gilt: Da, wo die Basisklasse erwartet wird, darf auch ein Objekt der abgeleiteten Klassen dahinterstecken.

TECHNISCHE UNIVERSITÄT DARMSTADT

Exceptions aus Lambda-Ausdrücken

***Erinnerung*: Kapitel 04c, Abschnitt zu Functional Interfaces und Lambda-Ausdrücken.**

Exceptions aus Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface MyBinaryIntFunction {
   int apply ( int m, int n ) throws Exception;
}

MyBinaryIntFunction div =
    ( m, n ) ->
    { if ( n == 0 ) throw new Exception(); return m / n; };
```

Ein kleines, aber sinnvolles Beispiel dazu.

Exceptions aus Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface MyBinaryIntFunction {
   int apply ( int m, int n ) throws Exception;
}

MyBinaryIntFunction div =
    ( m, n ) ->
    { if ( n == 0 ) throw new Exception(); return m / n; };
```

Die funktionale Methode enes Functional Interface darf natürlich auch Exceptions werfen.

Exceptions aus Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface MyBinaryIntFunction {
   int apply ( int m, int n ) throws Exception;
}

MyBinaryIntFunction div =
    ( m, n ) ->
    { if ( n == 0 ) throw new Exception(); return m / n; };
```

Im Rumpf des Lambda-Ausdruckes kann dann der Wurf der Exception wie überall sonst in Methodenrümpfen stehen. Damit sind Exceptions in Lambda-Ausdrücken auch schon abgehakt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Eigene Exception-Klassen definieren

Exception-Klassen sind nicht so viel anders als andere Klassen. Etliche finden sich in der Java-Standardbibliothek, so wie die schon gesehenen Exception und indexOutOfBoundsException aber darüber hinaus können wir auch selbst Exception-Klassen bauen.

Eigene Exception-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WitchingHourException extends Exception {
    public WitchingHourException () {
        super ( “It\‘s witching hour!“ );
    }
}
```

Nun schreiben wir also unsere erste eigene Exception-Klasse. Eine Exception *dieser* Klasse soll geworfen werden, wenn ein bestimmtes Programmstück während der angelsächsischen Geisterstunde ausgeführt wird, also zwischen drei und vier Uhr morgens.

Eigene Exception-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WitchingHourException extends Exception {
    public WitchingHourException () {
        super ( “It\'s witching hour!“ );
    }
}
```

Wie gesagt, ist neben der Klasse Exception selbst jede andere Klasse genau dann eine Exception-Klasse, wenn sie von Klasse Exception direkt oder indirekt abgeleitet ist. In diesem Fall leiten wir *direkt* von Klasse Exception ab.

Eigene Exception-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WitchingHourException extends Exception {
    public WitchingHourException () {
        super ( “It\'s witching hour!“ );
    }
}
```

Für diese Exception-Klasse wissen wir schon, wie die Botschaft lauten soll. Daher brauchen wir einen Konstruktor ohne Argumente, der für jedes Objekt immer dieselbe Botschaft setzen soll.

***Erinnerung*: Konstruktoren werden nicht vererbt, daher hat WitchingHourException nur diesen einen Konstruktor, insbesondere nicht den Konstruktor mit einem String-Parameter, den wir schon bei Klasse Exception verwendet hatten.**

Eigene Exception-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WitchingHourException extends Exception {

    public WitchingHourException () {
        super ( “It\'s witching hour!“ );
    }
}
```

Der parameterlose Konstruktor der neuen Klasse ruft dafür einfach nur den Konstruktor mit String-Parameter von Klasse Exception auf. Die Botschaft ist also immer dieselbe, nämlich dass Geisterstunde ist.

Eigene Exception-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class WitchingHourException extends Exception {
    public WitchingHourException () {
        super ( “It\'s witching hour!“ );
    }
}
```

***Erinnerung*: Zeichen, die bei char- und String-Literalen eine Sonderbedeutung haben, muss generell ein Backslash vorangestellt werden.**

***Nebenbemerkung*: In diesem konkreten Fall könnte man den Backslash weglassen, weil der Compiler erkennen kann, dass ein Hochkomma in einem String-Literal unmöglich die Sonderbedeutung haben kann, ein Zeichenliteral zu umgrenzen. Aber wir demonstrieren hier lieber noch einmal die allgemeine Regel.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
   Calendar cal = Calendar.getInstance();
   int hour = cal.get ( Calendar.HOUR_OF_DAY );
   if ( hour == 3 )
     throw new WitchingHourException();
   return hour;
}
```

Und so können wir diese Exception-Klasse nun verwenden. Die hier gezeigte Methode soll die aktuelle Stunde zurückliefern, also beispielsweise die Zahl 15, falls die Methode im Zeitraum ab 15 Uhr, aber vor 16 Uhr ausgeführt wird.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
    Calendar cal = Calendar.getInstance();
    int hour = cal.get ( Calendar.HOUR_OF_DAY );
    if ( hour == 3 )
      throw new WitchingHourException();
    return hour;
}
```

Aber der Entwickler dieser Methode ist abergläubisch und liefert die Geisterstunde nicht zurück, sondern wirft stattdessen eine Exception.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
   Calendar cal = Calendar.getInstance();
   int hour = cal.get ( Calendar.HOUR_OF_DAY );
   if ( hour == 3 )
     throw new WitchingHourException();
   return hour;
}
```

***Erinnerung*: Wir hatten die Klasse java.util.Calendar schon im Kapitel 03c gesehen, Abschnitt über Konstruktoren. Sie organisiert Zeitpunkte bestehend aus Jahr, Monat, Tag, Stunde, Minute, Sekunde und Millisekunde.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
   Calendar cal = Calendar.getInstance();
   int hour = cal.get ( Calendar.HOUR_OF_DAY );
   if ( hour == 3 )
     throw new WitchingHourException();
   return hour;
}
```

Die Klassenmethode getInstance liefert ein Calendar-Objekt für den aktuellen Zeitpunkt gemäß Systemuhr zurück, auf das dann die Variable cal verweist.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
   Calendar cal = Calendar.getInstance();
   int hour = cal.get ( Calendar.HOUR_OF_DAY );
   if ( hour == 3 )
     throw new WitchingHourException();
   return hour;
}
```

Methode get erhält eine ganze Zahl, die das Zeitattribut identifiziert, dessen Wert zurückgeliefert werden soll. Das Klassenattribut HOUR_OF_DAY ist die Stunde im 24-Stunden-Modus.

***Nebenbemerkung*: Calendar hat etliche Attribute. Aus diesem Grund sind die get- und die set-Methode bei Calendar etwas anders konzipiert, als wir es bisher kennengelernt haben, nämlich:**

Das Attribut ist hier nicht im Namen der get- und der set-Methode einkodiert, sondern wird durch einen zusätzlichen int-Parameter angegeben, der bequemerweise auch schon als Klassenkonstante vordefiniert ist.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
   Calendar cal = Calendar.getInstance();
   int hour = cal.get ( Calendar.HOUR_OF_DAY );
   if ( hour == 3 )
     throw new WitchingHourException();
   return hour;
}
```

Wie gesagt, die Stunde zwischen drei und vier Uhr morgens ist die angelsächsische Geisterstunde.

***Nebenbemerkung*: In einem realen Programm würde man die Zahl 3 nicht wie hier einfach als Literal an die Stelle schreiben, an der die Zahl benutzt wird, sondern *einmal* an einer geeigneten Stelle als Konstante mit sprechendem Namen definieren und nur diesen Namen verwenden. Denn erstens ist dann bei jeder Verwendung dieser Zahl durch den sprechenden Namen klar, was sie bedeutet; zweitens kann sie leicht – einfach durch Änderung der Definition – geändert werden, falls man beispielsweise die deutsche Geisterstunde, also Stunde null haben möchte. Entscheidend ist aber die geringere Fehleranfälligkeit: Stehen überall nur Literale anstelle von sprechenden Namen, dann muss man sich bei jedem Vorkommen etwa der Zahl 3 überlegen, ob das jetzt die Geisterstunde und daher zu ändern ist, oder ob ein Vorkommen der Zahl drei statt dessen für die drei Musketiere oder die drei Fragezeichen steht und daher *nicht* zu ändern ist, wenn nur die Geisterstunde geändert werden soll.**

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
    Calendar cal = Calendar.getInstance();
    int hour = cal.get ( Calendar.HOUR_OF_DAY );
    if ( hour == 3 )
      throw new WitchingHourException();
    return hour;
}
```

Wenn die aktuelle Uhrzeit in der Geisterstunde liegt, dann wird wie geplant eine WitchingHourException geworfen. Wie wir schon gesehen haben, ist die Parameterliste des Konstruktors leer, da die Botschaft im Konstruktor festgelegt ist.

Exception werfen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int hourOfNow ()
throws WitchingHourException {
    Calendar cal = Calendar.getInstance();
    int hour = cal.get ( Calendar.HOUR_OF_DAY );
    if ( hour == 3 )
      throw new WitchingHourException();
   return hour;
}
```

Außerhalb der Geisterstunde wird – wie der Name der Methode schon sagt – die aktuelle Stunde zurückgeliefert.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int hourOfNow;
try {
   hourOfNow = abc.hourOfNow ();
}
catch ( WitchingHourException exc ) {
   System.out.print ( exc.getMessage() );
}
```

So würde ein Programmstück aussehen, in dem die Methode hourOfNow aufgerufen und die Exception gefangen wird, völlig analog zu den vorherigen Beispielen.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int hourOfNow;
try {
   hourOfNow = abc.hourOfNow ();
}
catch ( WitchingHourException exc ) {
   System.out.print ( exc.getMessage() );
}
```

Wir nehmen wieder an, dass abc eine Variable der Klasse ist, zu der die Methode hourOfNow gehört.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int hourOfNow;
try {
   hourOfNow = abc.hourOfNow ();
}
catch ( WitchingHourException exc ) {
   System.out.print ( exc.getMessage() );
}
```

Die Art von Exception, die im try-Block geworfen werden kann und daher zu fangen ist, ist eben die WitchingHourException.

Exception fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int hourOfNow;
try {
   hourOfNow = abc.hourOfNow ();
}
catch ( WitchingHourException exc ) {
   System.out.print ( exc.getMessage() );
}
```

Die von Klasse Exception an Klasse WitchingHourException vererbte Methode getMessage liefert die Botschaft, mit der wir im Konstruktor von WitchingHourException den Konstruktor von Exception aufgerufen hatten, also auf Englisch den Satz, dass Geisterstunde ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Mehrere Exception-Klassen

Wir hatten schon gesagt, dass eine Methode potentiell auch Exceptions von mehreren Exception-Klassen werfen kann. Natürlich wird im Fall des Falles nur eine Exception geworfen und die ist nur von einer Klasse. Aber welche Klasse das ist, steht nicht von vornherein fest, es muss nur eine in der Aufzählung sein.

Mehrere Exception-Klassen werfen


```
public void printHour ( int hour ) throws
IndexOutOfBoundsException, WitchingHourException {
   String excStr = “Invalid hour : “ + hour;
   if ( hour < 0 || hour >= 24 )
      throw new IndexOutOfBoundsException ( excStr );
   if ( hour == 3 )
      throw new WitchingHourException ();
   System.out.print ( hour );
}
```

Dazu eine kleine Variation des letzten Beispiels.

Mehrere Exception-Klassen werfen


```
public void printHour ( int hour ) throws
IndexOutOfBoundsException, WitchingHourException {
   String excStr = "Invalid hour : " + hour;
   if ( hour < 0 || hour >= 24 )
      throw new IndexOutOfBoundsException ( excStr );
   if ( hour == 3 )
      throw new WitchingHourException ();
   System.out.print ( hour );
}
```

Konkret werden die potentiell geworfenen Exception-Klassen alle, durch Kommas getrennt, hinter das Schlüsselwort throws geschrieben, vor der öffnenden geschweiften Klammer für den Methodenrumpf. Die Reihenfolge ist egal.

Mehrere Exception-Klassen werfen


```
public void printHour ( int hour ) throws
IndexOutOfBoundsException, WitchingHourException {
  String excStr = "Invalid hour : " + hour;
  if ( hour < 0 || hour >= 24 )
     throw new IndexOutOfBoundsException ( excStr );
  if ( hour == 3 )
     throw new WitchingHourException ();
  System.out.print ( hour );
}
```

Hier wird die Botschaft für die erste Exception-Klasse zusammengebaut.

***Erinnerung*: Wir hatten schon im Abschnitt zu Strings im Kapitel 03b gesehen, dass der Plus-Operator nicht nur für arithmetische Datentypen definiert ist, sondern auch für die Klasse String. Sein Ergebnis ist die *Konkatenation* der beiden Operanden, das heißt, ein neuer String, der genau so lang ist wie die beiden Operanden zusammen, und sein Inhalt besteht aus den Zeichen des ersten Operanden gefolgt von den Zeichen des zweiten Operanden.**

Mehrere Exception-Klassen werfen


```
public void printHour ( int hour ) throws
IndexOutOfBoundsException, WitchingHourException {
   String excStr = "Invalid hour : " + hour;
   if ( hour < 0 || hour >= 24 )
      throw new IndexOutOfBoundsException ( excStr );
   if ( hour == 3 )
      throw new WitchingHourException ();
   System.out.print ( hour );
}
```

Wenn der Parameter nicht im Bereich 0 bis 23 liegt, also nicht die Nummer einer Stunde ist, dann wird eine IndexOutOfBoundsException geworfen. Diese Exception-Klasse aus der Java-Standardbibliothek ist sicherlich vom Namen her gut für diese spezielle Verwendung in Methode printHour gewählt, der Name passt ja ganz gut.

Mehrere Exception-Klassen werfen


```
public void printHour ( int hour ) throws
IndexOutOfBoundsException, WitchingHourException {
  String excStr = "Invalid hour : " + hour;
  if ( hour < 0 || hour >= 24 )
     throw new IndexOutOfBoundsException ( excStr );
  if ( hour == 3 )
     throw new WitchingHourException ();
  System.out.print ( hour );
}
```

Zur Geisterstunde wird dann eine WitchingHourException geworfen. Allein der Name der Exception-Klasse für sich sagt also schon aus, welche Art von Ausnahmesituation aufgetreten ist. Das ist der Grund dafür, dass recht viele Exception-Klassen schon vordefiniert sind, die sich großenteils eigentlich gar nicht wesentlich voneinander unterscheiden, außer im Namen.

Wenn man selbst Exception-Klassen definiert, sollte man dieser Idee folgen, für verschiedene Arten von Ausnahmesituationen jeweils eine eigene Exception-Klasse zu schreiben – wobei man natürlich wie immer Übertreibungen vermeiden und die Anzahl der Exception-Klassen nicht sozusagen explodieren lassen sollte.

Mehrere Exception-Klassen werfen


```
public void printHour ( int hour ) throws
IndexOutOfBoundsException, WitchingHourException {
   String excStr = "Invalid hour : " + hour;
   if ( hour < 0 || hour >= 24 )
      throw new IndexOutOfBoundsException ( excStr );
   if ( hour == 3 )
      throw new WitchingHourException ();
   System.out.print ( hour );
}
```

Falls weder der eine noch der andere Ausnahmefall eintritt, macht die Methode das, was sie eigentlich tun soll, nämlich den Wert des Parameters auf den Bildschirm zu schreiben.

***Nebenbemerkung*: Das ist leider ein gar nicht so seltenes Bild: Die Methode besteht mehr aus Ausnahmebehandlungen als aus der eigentlichen Funktionalität.**

Mehrere Exception-Klassen fangen


```
Calendar cal = Calendar.getInstance();
int hour = cal.get ( Calendar.HOUR_OF_DAY );

try {
   x.printHour ( hour );
}
catch ( IndexOutOfBoundsException exc ) { ..... }
catch ( WitchingHourException exc ) { ..... }
```

Wenn mehrere Exception-Klassen geworfen werden können, dann müssen auch mehrere Exception-Klassen separat voneinander gefangen und behandelt werden können.

Mehrere Exception-Klassen fangen


```
Calendar cal = Calendar.getInstance();
int hour = cal.get ( Calendar.HOUR_OF_DAY );

try {
   x.printHour ( hour );
}
catch ( IndexOutOfBoundsException exc ) { ..... }
catch ( WitchingHourException exc ) { ..... }
```

Wir rufen wieder die genaue Stunde ab. In diesem Fall wissen wir, dass IndexOutOfBoundsException garantiert nicht auftreten wird, denn die aktuelle Stunde ist immer ein Wert zwischen 0 und 23, jeweils inklusive. Dennoch müssen wir auch diese Exception-Klasse fangen, sonst beendet der Compiler die Übersetzung mit einer Fehlermeldung.

Mehrere Exception-Klassen fangen


```
Calendar cal = Calendar.getInstance();
int hour = cal.get ( Calendar.HOUR_OF_DAY );
try {
    x.printHour ( hour );
}
catch ( IndexOutOfBoundsException exc ) { ..... }
catch ( WitchingHourException exc ) { ..... }
```

Die Methode printHour, die wir eben implementiert haben und die die beiden verschiedenen Exception-Klassen wirft, wird nun in einem try-Block aufgerufen.

Mehrere Exception-Klassen fangen


```
Calendar cal = Calendar.getInstance();
int hour = cal.get ( Calendar.HOUR_OF_DAY );

try {
   x.printHour ( hour );
}
catch ( IndexOutOfBoundsException exc ) { ..... }
catch ( WitchingHourException exc ) { ..... }
```

Zu jeder der beiden Exception-Klassen gibt es einen eigenen catch-Block, ...

Mehrere Exception-Klassen fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Calendar cal = Calendar.getInstance();
int hour = cal.get ( Calendar.HOUR_OF_DAY );

try {
   x.printHour ( hour );
}
catch ( IndexOutOfBoundsException exc ) { ..... }
catch ( WitchingHourException exc ) { ..... }
```

... In dem jeweils implementiert wird, was im Falle des Wurfs dieser Exception passieren soll. Die Details interessieren uns hier nicht.

Mehrere Exception-Klassen fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void printHour ( int hour ) throws Exception {
  String excStr = "Invalid hour : " + hour;
  if ( hour < 0 || hour >= 24 )
     throw new IndexOutOfBoundsException ( excStr );
  if ( hour == 3 )
     throw new WitchingHourException ();
  System.out.print ( hour );
}
```

Die mit throw geworfenen Exception-Klassen müssen nicht exakt diejenigen sein, die explizit mit throws deklariert sind. Zum Beispiel könnte für die Methode printHour anstelle der beiden tatsächlich geworfenen Exception-Klassen einfach die gemeinsame Basisklasse Exception deklariert sein.

Mehrere Exception-Klassen fangen


```
public void printHour ( int hour ) throws Exception {
  String excStr = "Invalid hour : " + hour;
  if ( hour < 0 || hour >= 24 )
     throw new IndexOutOfBoundsException ( excStr );
  if ( hour == 3 )
     throw new WitchingHourException ();
  System.out.print ( hour );
}
```

Die allgemeine Regel ist: Für jede tatsächlich geworfene Exception-Klasse muss entweder diese selbst oder eine direkte oder indirekte Basisklasse in der throws-Klausel deklariert sein. Exception ist natürlich eine Basisklasse für beide tatsächlich geworfenen Exception-Klassen, also deckt die Deklaration von Klasse Exception in der throws-Klausel beide tatsächlich geworfenen Exception-Klassen ab.

Auch hier zeigt sich wieder das allgemeine Prinzip, dass hinter einer Variablen der Basisklasse immer ein Objekt einer direkt oder indirekt abgeleiteten Klasse stecken kann.

Mehrere Exception-Klassen fangen


```
try {
   abc.printHour ( hour );
}
catch ( WitchingHourException exc ) { ..... }
catch ( RuntimeException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Die gefangenen Exception-Klassen wiederum müssen weder eins-zu.-eins den deklarierten noch eins-zu-eins den tatsächlich geworfenen Exception-Klassen entsprechen.

Mehrere Exception-Klassen fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try {
    abc.printHour ( hour );
}
catch ( WitchingHourException exc ) { ..... }
catch ( RuntimeException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Bei der WitchingHourException wählen wir in diesem Beispiel eine Eins-zu-eins-Korrespondenz.

Mehrere Exception-Klassen fangen


```
try {
   abc.printHour ( hour );
}
catch ( WitchingHourException exc ) { ..... }
catch ( RuntimeException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Aber bei der IndexOutOfBoundsException gehen wir rein zur Illustration anders vor. Die Klasse RuntimeException ist das Bindeglied zwischen Klasse Exception und Klasse IndexOutOfBoundsException. Das heißt, RuntimeException ist direkt von Exception abgeleitet, und IndexOutOfBoundsException ist direkt von RuntimeException abgeleitet.

Also: Wenn wir *explizit* RuntimeException fangen, dann fangen wir *implizit* auch IndexOutOfBoundsException. Würde Methode printHour auch RuntimeException oder andere von RuntimeException abgeleitete Exception-Klassen werfen, würden diese ebenfalls durch diesen catch-Block gefangen.

Mehrere Exception-Klassen fangen


```
try {
   abc.printHour ( hour );
}
catch ( WitchingHourException exc ) { ..... }
catch ( RuntimeException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Da für Methode printHour deklariert ist, dass Klasse Exception geworfen wird, muss auch jede Exception gefangen werden, die von Klasse Exception oder irgendeiner direkt oder indirekt von Klasse Exception abgeleiteten Klasse ist. Zwar wissen *wir*, dass es keine anderen außer WitchingHourException und IndexOutOfBoundsException gibt. Aber der Java-Compiler kann das nicht unbedingt wissen und verlangt daher grundsätzlich immer die vollständige Abdeckung der throws-Klausel von printhour.

Mehrere Exception-Klassen fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try {
   abc.printHour ( hour );
}
catch ( WitchingHourException exc ) { ..... }
catch ( IndexOutOfBoundsException exc ) { ..... }
catch ( RuntimeException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Eine kleine Variation des Beispiels, um noch einen weiteren wichtigen Aspekt zu illustrieren. Eine Exception kann ja potentiell auf mehrere catch-Blöcke passen. In diesem Beispiel würde eine IndexOutOfBoundsException auf die drei gelb unterlegten catch-Blöcke passen.

Die allgemeine Regel ist: Wenn eine Exception geworfen wird und mehrere catch-Blöcke infrage kommen, dann wird immer der erste infrage kommende catch-Block ausgeführt, die anderen nicht. Daher ist es allgemein sinnvoll, so wie hier, die catch-Blöcke so zu reihen, dass immer jeweils die abgeleitete Klasse vor der Basisklasse kommt. Denn käme der catch-Block für die Basisklasse vor dem catch-Block für die abgeleitete Klasse, dann würde der letztere nie ausgeführt werden können – der erstere würde ihm alle infrage kommenden Exceptions sozusagen „wegschnappen“.

Mehrere Exception-Klassen fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try {
   abc.printHour ();
}
catch ( AException | BException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Noch eine letzte Variation unseres Beispiels, nun mit zwei fiktiven Exception-Klassen zur Illustration. Es kann schon einmal vorkommen, dass die catch-Blöcke für zwei verschiedene Exception-Klassen genau dasselbe tun sollen. Dann wäre es natürlich unbequem und auch fehleranfällig, wenn man zweimal exakt denselben catch-Block hinschreiben müsste. Stattdessen gibt es wie hier gezeigt die Möglichkeit, zwei Exception-Klassen in einem catch-Block zu verarbeiten.

Mehrere Exception-Klassen fangen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try {
    abc.printHour ();
}
catch ( AException | BException exc ) { ..... }
catch ( Exception exc ) { ..... }
```

Dabei werden die Exception-Klassen durch senkrechte Striche voneinander getrennt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Exceptions weiterreichen statt fangen

Bis jetzt wurden Exceptions in der aufrufenden Methode immer gefangen. Das ist aber nicht zwingend notwendig und auch nicht immer sinnvoll.

Exception weiterreichen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void printHourOfNow ()
throws WitchingHourException {
    Calendar cal = Calendar.getInstance();
    int hour = cal.get ( Calendar.HOUR_OF_DAY );
    x.printHour ( hour );
}
```

Um das zu zeigen, schreiben wir jetzt die ganze Methode hin, in der printhour aufgerufen wird, und nennen sie printHourOfNow.

Es geht darum, dass die aufrufende Methode printHourOfNow die Exception nicht fangen muss, sondern alternativ diese Exception ihrerseits an die Methode weiterreichen kann, die wiederum die Methode printHourOfNow aufgerufen hat.

Exception weiterreichen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void printHourOfNow ()
throws WitchingHourException {
    Calendar cal = Calendar.getInstance();
    int hour = cal.get ( Calendar.HOUR_OF_DAY );
    x.printHour ( hour );
}
```

Der Aufruf von printHour kann zwar eine WitchingHourException werfen, ist aber dennoch nicht in einem try-Block, da die Exception ja nicht in printHourOfNow gefangen wird.

Exception weiterreichen

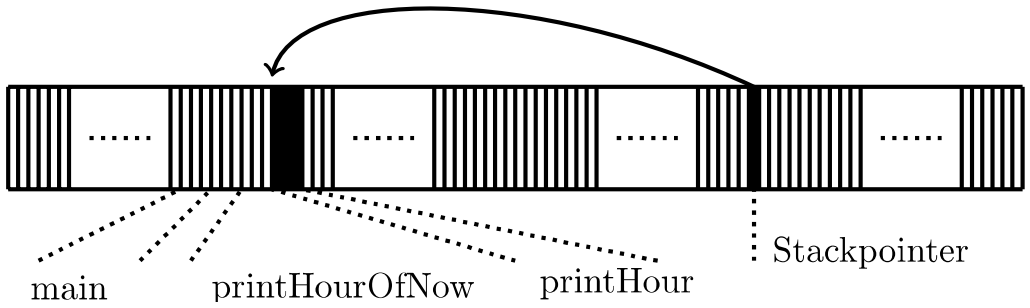
TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void printHourOfNow ()
throws WitchingHourException {
    Calendar cal = Calendar.getInstance();
    int hour = cal.get ( Calendar.HOUR_OF_DAY );
    x.printHour ( hour );
}
```

Stattdessen deklariert die Methode printHourOfNow ihrerseits die von printHour geworfene Exception-Klasse. Jetzt muss die Methode, die printHourOfNow aufgerufen hat, entweder die Exception mit try und catch fangen oder ihrerseits weiterreichen.

Exception weiterreichen

TECHNISCHE UNIVERSITÄT DARMSTADT



Im Kapitel 03c, Abschnitt Abarbeitung von Methoden, hatten wir schon den Call-Stack gesehen. Diese Bildsprache können wir auch hier verwenden: Wenn eine Exception in printHour geworfen wird, dann wird sofort der Frame für printHour vom Call-Stack genommen und der Stackpointer entsprechend auf den Anfang des Frames von printHourOfNow zurückgesetzt. Da printHourOfNow die Exception nicht fängt, wird der Frame für printHourOfNow ebenfalls vom Call-Stack genommen.

Das geht so lange weiter, bis eine Methode die Exception fängt. Diese Methode wird dann weiter mit dem zugehörigen catch-Block abgearbeitet.

Exception weiterreichen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void main ( String[] args ) {

    ..........

}
```

Das Weiterreichen von Exceptions kann natürlich nicht endlos weitergehen. Es gibt immer eine alleroberste, zuerst aufgerufene Methode, zum Beispiel main bei vielen Java-Interpretern. Diese Methode main muss bei vielen Java-Interpretern exakt diesen Methodenkopf haben. Insbesondere ist bei solchen Java-Interpretern keine throws-Klausel erlaubt. In diesem Fall kann main grundsätzlich keine Exceptions weiterreichen. Spätestens main muss dann also jede Exception fangen, die nicht vorher gefangen wurde, sonst geht das Programm nicht durch den Compiler.

TECHNISCHE UNIVERSITÄT DARMSTADT

try-with-resources

Manchmal öffnet ein Programm bestimmte Ressourcen, die immer nur von einem Programm zugleich geöffnet werden können. Eine solche Ressource muss unter allen Umständen geschlossen werden, wenn sie nicht mehr vom Programm gebraucht wird, und zwar so schnell wie möglich. Das ist bei try-catch ein bisschen problematisch, denn es gibt viele Wege, einen try-catch-Block zu verlassen, und auf jedem dieser Wege muss die Ressource geschlossen werden. Dafür ist try-with-resources entwickelt worden.

try-with-resources

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( Printer printer = ..... ) {
    .....
}
catch ( ..... ) { ..... }
catch ( ..... ) { ..... }
```

In diesem übermäßig simplifizierten und nur zur Illustration gedachten Fall ist ein Drucker die Ressource. Eine solche Ressource muss unbedingt wieder freigegeben werden, sobald sie nicht mehr für das Programm verwendet wird, damit andere Programme sie öffnen können. Dazu implementieren alle Klassen, die solche unteilbaren Ressourcen in einem Java-Programm repräsentieren, das Interface AutoCloseable in java.lang, dessen parameterlose void-Methode close die Ressource freigibt.

Dafür zu sorgen, dass Methode close in jedem denkbaren Fall aufgerufen wird, ist insbesondere bei try-catch-Blöcken ein Problem. da sie ja auf unterschiedlichen Wegen verlassen werden können. Man muss daran denken, im try-Block und in jedem der catch-Blöcke Methode close mit jeder betroffenen Ressource aufzurufen. Dass es auch anders geht, sehen wir auf dieser Folie.

try-with-resources

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( Printer printer = ..... ) {
    .....
}
catch ( ..... ) { ..... }
catch ( ..... ) { ..... }
```

Denn alternativ kann man die Ressource auch in einer Anweisung öffnen, die Sie unter dem Begriff try-with-resources in Java-Dokumentationen finden werden. In runden Klammern unmittelbar hinter dem Schlüsselwort try – also unmittelbar vor dem eigentlichen try-Block – wird die Ressource geöffnet. Der Compiler setzt automatisch Aufrufe von Methode close ein, so dass bei jeder Möglichkeit, den try-catch-Block zu verlassen, auch die Ressource geschlossen wird, ohne dass wir uns selbst darum kümmern müssen. Voraussetzung ist aber, wie gesagt, dass die Ressource das Interface AutoCloseable entsprechend implementiert.

Man kann im farblich unterlegten Bereich mehrere Ressourcen öffnen, getrennt durch Semikolons voneinander.

TECHNISCHE UNIVERSITÄT DARMSTADT

Runtime Exceptions

Eigentlich hatten wir gesagt, dass alle Operationen, die Exceptions werfen können, in try-Blöcke gehören, weil *alle* möglicherweise geworfenen Exceptions dann auch gefangen werden müssen. Das stimmt so nicht ganz, es gibt da eine bewusst und vorsätzlich eingebaute Sicherheitslücke in Java.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Hat eine Methode eine throw-Klausel, muss jeder Aufruf in einen try-Block

➔ Alle Exceptions werden gefangen

Ausnahme:

- **Exceptions von java.lang.RuntimeException**
- **bzw. unmittelbar oder mittelbar davon abgeleitete Klassen**

Oben auf dieser Folie sehen Sie nochmals die allgemeine Regel.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Hat eine Methode eine throw-Klausel, muss jeder Aufruf in einen try-Block

➔ Alle Exceptions werden gefangen

Ausnahme:

- **Exceptions von java.lang.RuntimeException**
- **bzw. unmittelbar oder mittelbar davon abgeleitete Klassen**

Es gibt aber eine Ausnahme: Exceptions vom Typ RuntimeException beziehungsweise von davon direkt oder indirekt abgeleiteten Klassen müssen *nicht* gefangen werden.

Offensichtlich ist das eine gefährliche Lücke. Tatsächlich resultieren *alle* Programmabbrüche zur Laufzeit daraus, dass eine Exception dieser Art geworfen, aber nicht gefangen wurde. Wann immer das Laufzeitsystem den Lauf eines *Ihrer* Programme abgebrochen hat, lag das an einer RuntimeException. Warum diese Lücke im Sicherheitssystem?

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- ArithmeticException
- ClassCastException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Der Sinn dieser gefährlichen Lücke wird klarer, wenn wir uns einmal die Klassen in der Java-Standardbibliothek genauer ansehen, die von RuntimeException abgeleitet sind. Hier nur ein paar repräsentative Beispiele.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- **ArithmeticException**
- **ClassCastException**
- **IndexOutOfBoundsException**
- **NegativeArraySizeException**
- **NullPointerException**

Eine Arithmetic Exception wird geworfen, wenn ein arithmetischer Fehler auftritt, zum Beispiel eine ganzzahlige Division durch 0. Wenn diese Exception gefangen werden müsste, dann müsste *jede* Division mit einem nicht zur Kompilierzeit evaluierbaren Divisor in einen try-Block, denn *jede* Division könnte theoretisch diese Exception werfen. Ein typisches numerisches Programm wäre dann so gespickt mit try und catch, dass man den eigentlichen Code kaum noch lesen könnte.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m1 ( int n ) {
   if ( n == 0 )
      return 0;
   return m2 ( n );
}
```

```
public int m2 ( int m ) {
   return 1 / m;
}
```

Eine Idee wäre, dass der Compiler das Fangen der Exception nur dort verlangt, wo der Compiler nicht selbst sehen kann, dass der Divisor garantiert nicht 0 ist. Aber schon dieses sehr einfache Beispiel demonstriert, dass der Compiler ganz schnell keine Chance mehr hat, irgendetwas zu sehen. Wir mit unserer menschlichen Draufsicht sehen hingegen sofort, dass der Divisor niemals 0 sein kann.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- ArithmeticException
- ClassCastException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Zurück zur Liste der von RuntimeException abgeleiteten Klassen. ClassCastException wird geworfen, wenn ein Downcast nicht geklappt hat.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }
public class Y1 extends X { ..... }
public class Y2 extends X { ..... }
public class Z extends Y1 { ..... }
```

```
X a = new X ();
X b = new Y1 ();
X c = new Y2 ();
X d = new Z ();
```

(Y1)b	(Y1)a
(Y1)d	(Y1)c

Ein Downcast klappt, wenn das Objekt entweder vom Zieltyp des Downcast oder von einem direkt oder indirekt vom Zieltyp abgeleiteten Typ ist. In diesem kleinen Beispiel kann b nach Y1 konvertiert werden, weil das Objekt hinter b vom Zieltyp ist. Und bei d klappt es ebenfalls, weil der Typ Z des Objekts vom Zieltyp Y1 des Downcast abgeleitet ist.

Bei a und c wird hingegen jeweils eine ClassCastException geworfen, weil die Objekte hinter a und c weder vom Zieltyp Y1 selbst noch von Y1 abgeleitet sind.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- ArithmeticException
- ClassCastException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Das nächste Beispiel, IndexOutOfBoundsException, wird geworfen, wenn in einem Array oder String auf einen Index zugegriffen wird, der gar nicht existiert. Diese Exception wird außerdem von ein paar weiteren Klassen in der Java-Standardbibliothek geworfen. Wir haben sie auch schon für eigenen Code sozusagen zweckentfremdet.

Auch hier wieder: Müsste diese Exception gefangen werden, dann müsste jeder Zugriff auf eine Arraykomponente in einen try-catch-Block.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- ArithmeticException
- ClassCastException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Der Name sagt es schon: Ein Objekt *dieser* Exception-Klasse wird geworfen, wenn wir versuchen, ein Arrayobjekt einzurichten, und dabei eine negative Länge angeben.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- ArithmeticException
- ClassCastException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Zum Schluss des Abschnitts über Exceptions schauen wir uns noch NullPointerException genauer an.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Wie der Name schon sagt, hat diese Exception etwas damit zu tun, dass man ja Variable von Referenztypen auf den symbolischen Wert null setzen kann.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Zum Beispiel wird eine NullPointerException geworfen, wenn auf ein Attribut zugegriffen wird, obwohl die Variable den Wert null hat.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Oder wenn in derselben Situation eine Methode aufgerufen wird.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Analog bei Variablen von Arraytypen.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Der Zugriff auf die Länge des Arrays führt natürlich ebenfalls zum Wurf einer NullPointerException.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Sogar bei Exceptions: Wird versucht, null anstelle eines Objektes zu werfen, so wird statt dessen eine NullPointerException geworfen.

Runtime Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

NullPointerException:

```
X a = null;
int[] b = null;
Exception exc = null;
```

```
a.i = 1;
a.m();
b[0] = 2;
int j = b.length;
throw exc;
```

Müsste NullPointerException gefangen werden, dann müssten *alle* Anweisungen dieser Art in try-Blöcke nebst catch-Blöcken – der Code wäre nicht mehr zu lesen.

Sprechende Namen!

TECHNISCHE UNIVERSITÄT DARMSTADT

ArithmeticException

ClassCastException

IndexOutOfBoundsException

NegativeArraySizeException

NullPointerException

.....

Beachten Sie, dass diese Klassen sehr vorbildliche Beispiele dafür sind, dass alle Bezeichner möglichst selbsterklärend gewählt sein sollten.

TECHNISCHE UNIVERSITÄT DARMSTADT

Throwable, Error und Assert-Anweisung

Wie zu Beginn dieses Kapitels angedeutet, kommen wir jetzt zu zwei anderen Konstrukten neben Exceptions, die relativ leicht zu verstehen und daher schnell abzuhandeln sind. Im Gegensatz zu Exceptions sind diese beiden Konstrukte eher für die Testphase gedacht, nicht für den realen Einsatz des Programms.

Als erstes betrachten wir die assert-Anweisung. Zu ihr kommen wir vom Thema Exception über die Zwischenthemen Throwable und Error.

Throwable und Error

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Throwable { ..... }

public class Exception extends Throwable { ..... }

public class Error extends Throwable { ..... }
```

Die Klasse Exception ist direkt abgeleitet von einer Klasse Throwable, und von Throwable ist noch eine Klasse Error abgeleitet. Alle drei sind im Package java.lang, das ja bekanntlich nicht explizit importiert werden muss.

Throwable und Error

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Throwable { ..... }

public class Exception extends Throwable { ..... }

public class Error extends Throwable { ..... }
```

Der ganze Mechanismus mit try und catch, eigentlich alles, was wir bisher in diesem Kapitel gesagt haben, funktioniert generell bei Throwable und allen davon direkt oder indirekt abgeleiteten Klassen ...

Throwable und Error

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Throwable { ..... }

public class Exception extends Throwable { ..... }

public class Error extends Throwable { ..... }
```

... und daher auch bei Klasse Exception und allen von ihr direkt oder indirekt abgeleiteten Klassen.

Throwable und Error

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Throwable { ..... }

public class Exception extends Throwable { ..... }

public class Error extends Throwable { ..... }
```

Klassen, die von Error direkt oder indirekt abgeleitet sind, werden typischerweise vom Laufzeitsystem geworfen in Fällen, in denen der Fehler nach menschlichem Ermessen so gewichtig ist, dass wohl keine sinnvolle Fehlerbehandlung mehr möglich ist, sondern Programmabbruch die beste Lösung zu sein scheint. Daher müssen Error-Klassen generell nicht gefangen werden.

Dass Error-Klassen in der Regel auch tatsächlich nicht mit try-catch gefangen werden, ist auch der Grund, warum in Einführungen wie hier in der Regel vor allem Klasse Exception behandelt wird, Throwable oder Error allenfalls am Rande.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Error extends Throwable { ..... }

public class AssertionError extends Error {
    .....
}
```

Wir interessieren uns für eine spezielle von Error abgeleitete Klasse namens AssertionError. Die Sprache Java bietet eine sehr bequeme Sonderform, mit AssertionError umzugehen, und genau das ist die angekündigte assert-Anweisung.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );
```

```
assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

Oben sehen Sie ein ganz normales Beispiel für eine throw-Anweisung im Rahmen einer if-Abfrage, wie wir es mehrfach für Exception-Klassen gesehen haben.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );
```

```
assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

Der Unterschied ist, dass keine Exception-Klasse, sondern AssertionError verwendet wird. Wie gesagt, müssen Error-Klassen, also auch AssertionError, nicht mit try-catch gefangen werden. Es ist auch so gedacht, dass AssertionError nicht gefangen wird, sondern zu sofortigem Programmabbruch mit einer hoffentlich aussagekräftigen Fehlerdiagnostik führt.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );
```

```
assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

Wie für die Klasse String, gibt es in Java auch für die Klasse AssertionError eine verkürzte Schreibweise. Und zwar kann man die obige Anweisung mit exakt derselben Bedeutung auch so wie unten schreiben. Dies ist ein Beispiel für eine assert-Anweisung; assert-Anweisungen sind bedingte Würfe von AssertionError.

Ein wesentlicher Unterschied zwischen oben und unten springt natürlich sofort ins Auge: Die assert-Anweisung ist kürzer und übersichtlicher.

Der vielleicht entscheidende Unterschied ist aber, dass assert-Anweisungen beim Kompilieren an- oder abgeschaltet werden können mit entsprechenden Setzungen für den Compiler. Das heißt, in der Testphase kann man die assert-Anweisungen in einem Programm anschalten, um Fehler zu finden, nach der Testphase kann man sie ausschalten.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );
```

```
assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

***Nebenbemerkung*: Warum ausschalten, warum nicht einfach auch im realen Einsatz des Programms immer eingeschaltet mitlaufen lassen? *Eine* Antwort ist: Weil man assert-Anweisungen durchaus auch in der Testphase sinnvoll für Tests verwenden kann, die nicht echte Fehler abtesten, sondern ob bestimmte Konstellation tatsächlich so wie gedacht sind. So etwas sollte im realen Einsatz natürlich nicht zu einem Programmabbruch führen.**

Für Fehlertests, die auch im realen Einsatz des Programms mitlaufen sollen, sind Exceptions das Mittel der Wahl.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );
```

```
assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

Bei der Bedingung müssen wir aufpassen: Die Bedingung in der assert-Anweisung ist genau die Negation der Bedingung in der if-Abfrage. Das ist auch logisch, denn die Bedingung in der assert-Anweisung ist ja das, was gemäß Bedeutung des Verbs to assert eben *zugesichert* werden soll, also genau der Fall, in dem *kein* Error zu werfen ist.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( n < 0 || n % 2 == 1 )
    throw new AssertionError ( “Bad n!“ );
```

```
assert n >= 0 && n % 2 == 0 : “Bad n!“;
```

Auch die Schreibweise für die Fehlerbotschaft ist leicht verkürzt: Der String wird nach einem Doppelpunkt ohne Klammern oder anderes einfach hingeschrieben.

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int factorial  ( int n ) {
    assert n >= 0;
    int result = 1;
    for ( int i = 1; i <= n; i++ )
        result *= i;
    return result;
}
```

Hier sehen Sie ein realistisches Anwendungsbeispiel mit der einen Einschränkung, dass man in diesem Fall vielleicht doch eher eine Exception-Klasse wählen würde. Aber es mag auch Situationen geben, in denen man einen solchen Test nicht im realen Einsatz, sondern nur in der Testphase haben möchte, und dann ist die assert-Anweisung die einfachste Lösung.

***Vorgriff*: Eine solche Situation, die in diesem Beispiel aber sicher nicht zutrifft, wäre etwa, wenn die Methode so schnell ausführbar wäre, dass die assert-Anweisungen einen nicht zu vernachlässigenden Anteil an der Zeit haben, die die Abarbeitung der Methode kostet. Wenn diese Methode dann noch so häufig ausgeführt wird, dass sie einen Großteil der Gesamtlaufzeit des Programms in Anspruch nimmt, dann macht sich die Laufzeit der assert-Anweisungen deutlich negativ bemerkbar. Generell schauen wir uns Laufzeitproblematiken im Kapitel zu effizienter Software an.**

AssertionError

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int factorial  ( int n ) {
   assert n >= 0;
   int result = 1;
   for ( int i = 1; i <= n; i++ )
       result *= i;
   return result;
}
```

Die Fakultätsfunktion ist ja nur auf nichtnegativen ganzen Zahlen definiert. Dass der Parameter tatsächlich ganzzahlig ist, ist in Java durch den Compiler sichergestellt. Der Nutzer der Methode factorial kann aber vielleicht dennoch aus Versehen eine negative ganze Zahl als Parameter übergeben. Daher ist es zumindest in der Testphase sinnvoll, solche Fehler an einer möglichst früh ausgeführten Stelle zu entdecken.

Ohne diese assert-Anweisung würde eine negative Zahl n sich nämlich nur dadurch bemerkbar machen, dass das Resultat 1 ist. Es kann durchaus lange dauern, bevor es in der Testphase oder – schlimmer – erst im realen Einsatz auffällt, dass der Rückgabewert, mit dem dann munter weitergerechnet wird, Murks ist. Und dann kann es noch einige Mühe erfordern, den Fehler, der dann ja ganz woanders sichtbar würde, zu seinem Ursprung zurückzuverfolgen.

TECHNISCHE UNIVERSITÄT DARMSTADT

JUnit-Tests

Die soeben betrachteten assert-Anweisungen sind eine Möglichkeit, *innerhalb* einer Methode Tests einzubauen. Mindestens genauso wichtig ist es aber, die Korrektheit von Methoden *als Ganzes* zu testen. Als Quasi-Standard haben sich dafür JUnit-Tests bei Java etabliert.

***Nebenbemerkung*: Tests, die wie Exceptions und assert-Anweisungen in Einheiten sozusagen hineinschauen, nennt man *White*-Box-Tests; Tests, die wie Junit-Tests Einheiten als Ganzes hernehmen – eben als Black Boxes –, nennt man entsprechend *Black*-Box-Tests.**

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
```

Um JUnit-tests schreiben zu können, müssen wir ein paar Funktionalitäten importieren.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
```

***Erinnerung*: Im Kapitel 03c, Abschnitt zu Klassenmethoden, hatten wir schon gesehen, dass Klassenmethoden mit Schlüsselwort static so importiert werden können, dass sie ohne Klassennamen aufgerufen werden können.**

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
```

Die eigentliche assert-Funktionalität finden Sie als Klassenmethoden in dieser Klasse. Dies sind nur zwei Beispiele für das, was diese Klasse bietet, nämlich genau die beiden Methoden, die wir gleich im Anwendungsbeispiel verwenden werden.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
```

Und die Funktionalität, diese Tests dann auch wirklich durchzuführen und ihre Durchführung zu steuern, finden Sie als ganze Klassen in *diesem* Package.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import painful.tax.*;



public class TaxCalculatorTest {
    ..........
}
```

Im Gegensatz zu Exceptions und assert-Anweisungen schreibt man JUnit-Tests nicht in den zu testenden Quelltext hinein, sondern in eine separate Quelldatei.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import painful.tax.*;


public class TaxCalculatorTest {
    ..........
}
```

Die zu testenden Einheiten, zumeist Klassen, importiert man in der Testdatei wie üblich.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import painful.tax.*;


public class TaxCalculatorTest {
    ..........
}
```

Und in der Testdatei definiert man dann eine Klasse, deren Methoden die Tests durchführen.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@Test
public void testARealCase () {
    int tfn = TaxData.taxFileNumber
                        ( “Doe“, “John“,  ..... );
    assertEquals
          ( 1234.56, TaxCalculator.calculate ( tfn ) );
}
```

So sieht eine solche Testmethode dann aus. Diese Methode ist Teil der soeben andeutungsweise definierten Klasse TaxCalculatorTest.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@Test
public void testARealCase () {
    int tfn = TaxData.taxFileNumber
                    ( “Doe“, “John“,  ..... );
    assertEquals
        ( 1234.56, TaxCalculator.calculate ( tfn ) );
}
```

Wird JUnit mit der Klasse TaxCalculatorTest aufgerufen, dann werden alle Methoden, die diese Annotierung haben, nacheinander aufgerufen.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@Test
public void testARealCase () {
   int tfn = TaxData.taxFileNumber
                    ( “Doe“, “John“,  ..... );
   assertEquals
         ( 1234.56, TaxCalculator.calculate ( tfn ) );
}
```

Die importierte Klassenmethode assertEquals hat zwei Parameter und liefert genau dann einen Fehler, wenn die beiden Parameter unterschiedliche Werte haben. Damit kann man also wie hier testen, ob das Ergebnis einer Berechnung das erwartete Ergebnis ist.

Für gebrochenzahlige Parameter gibt es assertEquals mit *drei* Parametern: Der dritte Parameter ist der maximale Wert, um den die ersten beiden Parameter sich unterscheiden dürfen, um noch als gleich angesehen werden.

***Vergleich mit Racket*: Die Methode assertEquals auf der Folie entspricht genau der Funktion check-expect; die soeben beschriebene Methode assertEquals mit drei Parameterm für gebrochenzahlige Werte entspricht genau check-within.**

***Nebenbemerkung*: Für Referenztypen ist die Methode assertEquals ebenfalls definiert, und zwar mit einem Test auf *Wert*gleichheit mit Methode equals von Klasse Object, siehe Abschnitt zu java.lang.Object in Kapitel ß3b. Für Test auf *Objektidentität* – also ob dieselbe Adresse in beiden getesteten Referenzen steht – gibt es assertSame.**

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@BeforeEach
public void testWhetherReady () {
    assertTrue ( TaxCalculator.isReady() );
}
```

Man kann eine Methode in einer Testklasse auch anders annotieren. Es gibt noch einige weitere, die wir hier nicht behandeln. Sie werden mühelos unzählige Auflistungen im Internet finden und auch problemlos selbst verstehen. Hier nur ein sehr häufig verwendetes Beispiel.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@BeforeEach
public void testWhetherReady () {
    assertTrue ( TaxCalculator.isReady() );
}
```

Methoden mit dieser Annotation werden vor jeder einzelnen Testmethode jeweils einmal unmittelbar davor ausgeführt.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@BeforeEach
public void testWhetherReady () {
    assertTrue ( TaxCalculator.isReady() );
}
```

Damit ist in diesem Beispiel gewährleistet, dass der TaxCalculator tatsächlich vor jeder Testmethode auch bereit ist. Methode assertTrue hat einen booleschen Parameter und ergibt einen Fehler, wenn der aktuale Wert des Parameters false ist.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
public void testDiv ( ) {
  assertThrows
    ( ArithmeticException.class,
      () -> { int m = 0, n = 0, x = m / n; } );
}
```

Zum Schluss soll noch eine weitere wichtige Variante des Fehlertests kurz erwähnt werden: assertThrows. Dieser Test wird angewendet in Situationen, in denen eine Exception zu werfen ist. Er gibt eine Fehlermeldung, falls die Exception wider Erwarten doch nicht geworfen wird.

***Vergleich mit Racket*: In einem gewissen Sinne ist assertThrows mit check-error vergleichbar.**

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
public void testDiv ( ) {
   assertThrows
      ( ArithmeticException.class,
        () -> { int m = 0, n = 0, x = m / n; } );
}
```

Die Klassenmethode assertThrows ist von einer Klasse namens Assertions zu importieren.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
public void testDiv ( ) {
  assertThrows
    ( ArithmeticException.class,
      () -> { int m = 0, n = 0, x = m / n; } );
}
```

Der erste Parameter gibt die erwartete Exception-Klasse an.

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
public void testDiv ( ) {
   assertThrows
      ( ArithmeticException.class,
        () -> { int m = 0, n = 0, x = m / n; } );
}
```

An dieser Stelle nehmen wir erst einmal nur zur Kenntnis, dass wir noch eine vordefinierte Klassenkonstante der Exception-Klasse ansprechen müssen, die wir bisher noch gar nicht gesehen haben.

***Nebenbemerkung*: Diese Klassenkonstante ist im Prinzip genau die Zusatzinformation für jede Klasse, die im Abschnitt namens „Allgemein: Methoden vererben / überschreiben“ im Kapitel 01f angedeutet wurde.**

JUnit-Tests

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
public void testDiv ( ) {
   assertThrows
      ( ArithmeticException.class,
        () -> { int m = 0, n = 0, x = m / n; } );
}
```

Der formale Typ des zweiten Parameters ist ein Functional Interface namens Executable aus dem Package java.lang.reflect, dessen funktionale Methode keine Parameter und Rückgabetyp void hat und daher einen Lambda-Ausdruck wie diesen als aktualen Parameter erlaubt.

Natürlich wird man in der Regel nicht einen eingebauten Java-Operator, sondern den Aufruf einer selbstgeschriebenen Methode testen; der Divisionsoperator ist hier nur gewählt, um das Beispiel möglichst einfach zu halten.

Damit sind JUnit-Tests und das ganze Thema Fehlerbehandlung beendet.