Objektorientierte Programmierung

Kapitel 18: Exceptions II

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

http://www.informatik.uni-halle.de/~brass/oop14/

Inhalt

1. Umgang mit Fehlern
2. Assertions
3. Exception-Klassen
4. throw
5. Programmausführung

Umgang mit Fehlern (1)

- Wenn man Methoden und Klassen entwickelt, muss man mit Fehlersituationen umgehen:
  - mit Fehlern des Aufrufers (Benutzers der Methode/Klasse),
  - mit Fehlern der Umgebung (z.B. USB Stick voll, keine Netzwerk-Verbindung, keine Schreibrechte für Verzeichnis),
  - und mögliche Fehler im eigenen Programmcode bedenken.
- Benutzer sollten bei falschen Eingaben sinnvolle Fehlermeldungen bekommen, und nicht einfach einen Programmabbruch aufgrund einer Java Exception.

  Prüfen von Benutzer-Eingaben ist auch aus Sicherheitsgründen wichtig: Eventuell tut das Programm sonst Dinge, an die der Programmierer nie gedacht hat. Z.B. darf man Benutzereingaben nicht ungeprüft in Datenbank-Befehle einfügen und diese ausführen ("SQL Injection").

Umgang mit Fehlern (2)

- Je früher ein Fehler entdeckt wird, und je spezifischer die Fehlermeldung, desto leichter läßt sich die Ursache finden und der Fehler beseitigen.

  Häufig läuft das Programm nach der falschen Anweisung noch ein Stück weiter, und scheitert dann an anderer Stelle. Es wäre besser, durch entsprechende Tests den Abstand klein zu halten.

- Öfters sind nicht alle Werte, die der Parameter-Typ einer Methode erlauben würde, auch sinnvoll.

  Oder die Methode kann nur in einem bestimmten Zustand des Objektes aufgerufen werden, d.h. man muss eine bestimme Reihenfolge für die Methodenaufrufe einhalten.

- Es empfiehlt sich, dass die Methode selbst entsprechende Prüfungen vornimmt.

  Und natürlich alles gut dokumentiert ist.

Umgang mit Fehlern (3)

- In den bisherigen Beispiel-Programmen wurde meist eine Fehlermeldung ausgedruckt und das Programm beendet.
- Das geht nur bei kleinen Programmen.
- Klassen mit Methoden, die sich so verhalten, sind kaum wiederverwendbar in größeren Programmen:
  - Vielleicht sieht der Benutzer die Standard-Ausgabe gar nicht.

    Bei Programmen mit graphischer Schnittstelle oder Hintergrund-Prozessen.
  - Vielleicht hat das Programm auch nach Auftreten eines Fehlers noch wichtige Dinge zu tun.

    Daten sichern, Rücksetzen von Dateien auf Zustand vor Ausführung des Programms, gesteuerte Anlage in sicheren Zustand herunterfahren, nochmal probieren, sich selbst neu starten, Entschuldigung ausdrucken, Log-Eintrag schreiben, EMail an Programm-Autor schicken, etc.

Umgang mit Fehlern (4)

- Daher ist es in Java üblich, bei festgestellten Fehlern eine Exception (Ausnahme) zu "werfen" (engl. "throw").
- Programmcode beim Aufrufer kann diese Exception "auffangen" (engl. "catch") und behandeln.
- Exceptions erlauben also die Trennung von
  - Feststellen des Fehlers und
  - Reaktion auf den Fehler.

    Dies kann direkt bei der aufrufenden Methode geschehen, oder irgendwo auf dem Weg bis zum Hauptprogramm, oder notfalls in der abstrakten Maschine — die wird das Programm dann aber einfach beenden.
- Wenn die Methode den Fehler selbst behandeln könnte, sollte sie dies natürlich nicht dem Aufrufer aufbürden.

Umgang mit Fehlern (5)

- Exceptions sind auch nützlich, wenn Fehler nicht über den Rückgabewert signalisiert werden können:
  - Wenn alle Werte des Rückgabetyps schon für korrekte Ausführungen der Methode möglich sind,
  - in Konstruktoren (sie können keinen Wert zurückgeben).
- Im Gegensatz zu Rückgabewerten können Exceptions nicht einfach ignoriert werden.
- Da man nicht weiß, wie der Aufrufer auf die Exception reagieren wird (vielleicht probiert er es später nochmal), sollte man das Objekt nicht in einem inkonsistenten Zustand hinterlassen, wenn man eine Exception wirft.

Inhalt

1. Umgang mit Fehlern
2. Assertions
3. Exception-Klassen
4. throw
5. Programmausführung

Assertions (1)

- Assertions (deutsch "Zusicherungen") sind Bedingungen, die bei jeder Ausführung des Programms erfüllt sein müssten, wenn das Programm korrekt ist.
- Wenn eine Assertion nicht erfüllt ist, muss ein echter Programmier-Fehler vorliegen.
- Man darf Assertions also nicht verwenden, um
  - Eingaben des Benutzers zu prüfen, oder
  - Bedingungen der Ausführungsumgebung.
- Das ist deswegen wichtig, weil Assertions auch abgeschaltet werden können: Sie stehen dann zwar noch im Programm, werden aber nicht geprüft (das ist sogar der Default).

  Deswegen sollten Assertions auch keine Seiteneffekte enthalten.

Assertions (2)

- Java hat seit Version 1.4 ein spezielles Konstrukt dafür, das `assert`-Statement. Beispiel:

```
assert i >= 0;
```

- Falls die Bedingung falsch ist, wird das Programm mit einem `AssertionError` abgebrochen.

  Dazu ist noch wichtig, dass die Prüfung der Assertions "enabled" ist, s.u.

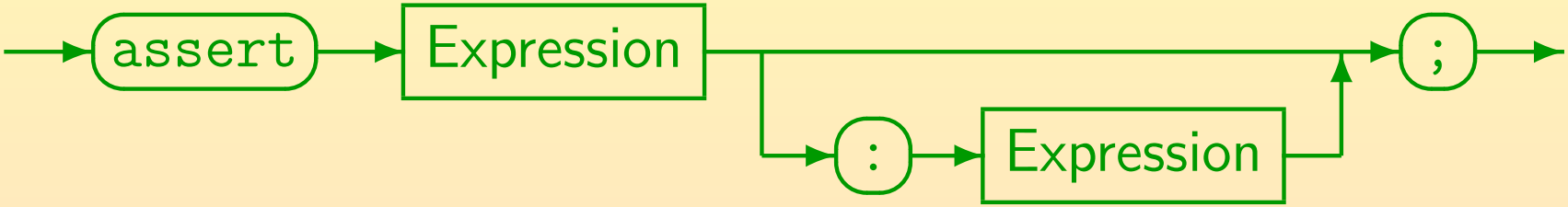
- Ein "`AssertionError`" ist eine "harte Fehler-Ausnahme" (ein `Error`, s.u.). Man fängt sie normalerweise nicht ab, daher wird die Programmausführung beendet der Meldung:

```
Exception in thread "main" java.lang.AssertionError
    at TestClass.main(TestClass.java:3)
```

  Man hat so Datei und Zeile (und den "Stack Trace", also die Methodenaufrufe) und kann sich damit auf die Fehlersuche machen.

Assertions (3)

- AssertStatement:



- Das `assert`-Statement besteht also aus:
  - dem Schlüsselwort `assert`,
  - einem Ausdruck vom Typ `boolean` (die Bedingung, die wahr sein muss, wenn das Programm korrekt ist),
  - optional einem Doppelpunkt "`:`" und einem Ausdruck, der einen Wert liefert, der nach `String` konvertiert werden kann (eine Fehlermeldung).
- Beispiel für Version mit Fehlermeldung:

```
assert i >= 0 : "i ist negativ!";
```

Assertions (4)

- Damit die Assertions geprüft werden, muss man die JVM mit der Option `-ea` (kurz für `-enableassertions`) starten:

  ```
  java -ea TestClass
  ```

  Der Default ist also, dass Assertions nicht geprüft werden.

  Man kann die Prüfung auch nur für bestimmte Pakete anschalten.

- Üblicherweise sind Assertions bei der Programmentwicklung angeschaltet, aber im Produktivbetrieb ausgeschaltet.

  Hoare soll dazu gesagt haben, das wäre, wie wenn man den Gurt nur in der Fahrschule anlegt, aber nicht später im eigentlichen Straßenverkehr.

- Wenn man einen Benchmark gewinnen will, macht es sicher Sinn, alles abzuschalten, was Laufzeit kostet.

- Ansonsten sind zuverlässigere Ergebnisse und ggf. schneller gefundene Fehler wichtiger als eine schnellere Ausführung.

Assertions (5)

- Die `assert`-Anweisung ist kürzer als ein `if` und eine `throw`-Anweisung (s.u.) zum Auslösen einer Exception.
- Sie macht auch sehr klar, dass es sich um einen Programmfehler handelt, falls die Bedingung falsch ist.
- Bei `public` Methoden, die von außerhalb des Paketes aufgerufen werden, könnte es Teil der Spezifikation sein, dass sie die Argumente überprüfen. Dann wäre es fragwürdig, wenn das über Assertions geschieht.

  Es ist ein bißchen wie eine Benutzereingabe, und die sollte auch nicht nur optional geprüft werden. Zumindest muss dieser Unterschied klar dokumentiert sein. Es wird auch gesagt, dass Exceptions spezifischer sein könnten, z.B. `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`. Ich sehe hier aber nicht, inwieweit es Sinn macht, solche Programmfehler unterschiedlich zu behandeln.

Inhalt

1. Umgang mit Fehlern
2. Assertions
3. Exception-Klassen
4. throw
5. Programmausführung

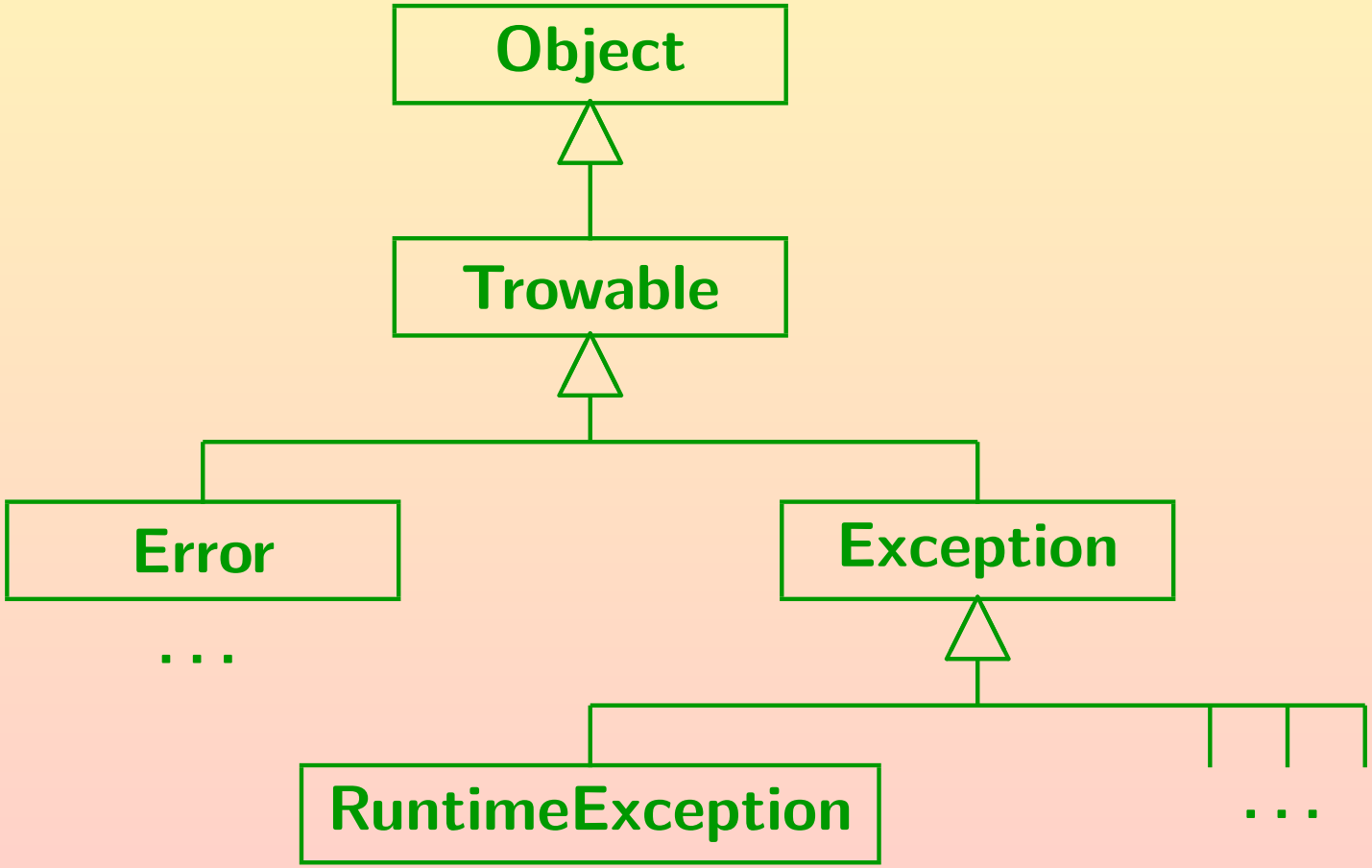
Exception-Klassen (1)

- In einem Programm können ganz unterschiedliche Fehler auftreten, die auch unterschiedlich behandelt werden müssen.
- Oft müssen dem Aufrufer auch zusätzliche Daten zum Fehler übermittelt werden.
- In Java werden für Exceptions (Fehler, Ausnahme-Situationen) daher Objekte erzeugt.

  So kann der bereits in der Sprache vorhandene Subklassen-Mechanismus genutzt werden, um auch Exceptions zu klassifizieren. Man kann auch eigene Exception-Klassen definieren.
- Diese Exception-Objekte sind in die Klassen-Hierarchie unterhalb der Klasse "`Throwable`" eingebettet.

  Exceptions werden mit dem Schlüsselwort "`throw`" ausgelöst. Mit dem Schlüsselwort "`catch`" kann man eine Fehlerbehandlung angeben (s.u.).

Exception-Klassen (2)



Exception-Klassen (3)

- Z.B. löst `Integer.parseInt()` im Fehlerfall eine `NumberFormatException` aus, das ist eine Subklasse von `IllegalArgumentException`, das wieder von `RuntimeException`.
- Wenn man auf ein Array außerhalb der Grenzen zugreift, erhält man eine `ArrayIndexOutOfBoundsException`, das ist Subklasse von `IndexOutOfBoundsException`, und wieder von `RuntimeException`.
- Wenn man mit `new FileInputStream(String name)` versucht, eine Datei zum Lesen zu öffnen, und es gibt die Datei nicht, bekommt man eine `FileNotFoundException`, Subklasse von `IOException`, und das wieder von `Exception`.

`IOException` und `FileNotFoundException` gehören zum Paket "`java.io`".

Exception-Klassen (4)

- Der Unterschied zwischen `Error` und `Exception` ist:
  - `Error` sind so schwere Fehler, dass sie in einem normalen Programm nicht behandelt werden können.

    Z.B. `VirtualMachineError`, `AssertionError`. Es ist nicht verboten, dafür eine `catch`-Klausel zu schreiben, aber es ist allgemein empfohlen, das nicht zu tun. Wenn die virtuelle Maschine nicht mehr richtig arbeitet, ist auch unklar, ob sie den `catch`-Block ausführen könnte. Zumindest sollte man das Programm beenden, nachdem man vielleicht noch versucht hat, Daten für ein eventuelles Recovery zu sichern, einen Log-Eintrag zu schreiben, und sich beim Benutzer zu entschuldigen.
  - `Exceptions` sind Ausnahmesituationen, für die sich ein normales Programm interessieren kann.

    Hier besteht eher eine Chance, sich von dem Fehler zu erholen und mit der Programmausführung fortzufahren.

Exception-Klassen (5)

- Es gibt noch eine weitere Unterscheidung:
  - Subklassen von `Error` und `RuntimeException`, die eine Methode erzeugen könnte, müssen im Methodenkopf nicht deklariert werden (siehe Folie 36).

    Die Idee ist, das solche Fehler im Prinzip überall auftreten können (es handelt sich ja um Programmierfehler). Z.B. könnte jeder Array-Zugriff eine `ArrayIndexOutOfBounds`-Exception erzeugen. Es ist aber nicht verboten, solche Exceptions auch zu deklarieren. Z.B. weist die Dokumentation zu `String.parseInt()` darauf hin, dass eine `NumberFormatException` erzeugt werden könnte, obwohl dies eine indirekte Subklasse von `RuntimeException` ist.
  - Die anderen Exceptions ("checked exceptions") müssen dagegen im Methodenkopf deklariert werden.

    Sie sind spezifischer für bestimmte Fehlersituationen, und man kann erwarten, dass der Aufrufer sie behandeln will.

Exception-Klassen (6)

- `Throwable` hat eine Reihe nützlicher Methoden, z.B.:

  Siehe [http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html]

  - `String getMessage()`
    Liefert einen Fehlermeldungstext ("detail message").
  - `String toString()`
    Name der Exeption-Klasse, ":", "detail message".
  - `void printStackTrace()`
    Zeigt Schachtelung der Methoden, deren Aufrufe zum Auslösen der Exception geführt haben.
  - `Throwable getCause()`
    Liefert die Exception, die zu dieser Exception geführt hat.

    Exceptions werden manchmal verkettet, so dass ein spezieller Fehler für eine bestimmte Operation nach oben in einer größeren und anwendungs-näheren Kategorie weitergegeben wird.

Exception-Klassen (7)

- Wenn man eine Exception auslösen will, muss man ein Objekt der entsprechenden Exception-Klasse erzeugen, und darauf den `throw`-Befehl (s.u.) anwenden.
- `Throwable` hat vier Konstruktoren, davon kann man die ersten zwei für die meisten Subklassen erwarten:

  Die anderen beiden nur, wenn Verkettung von Exceptions (s.u.) Sinn macht. Ausnahme z.B.: `java.text.ParseException(String s, int offset)` einziger Konstruktor dieser Klasse. Konstruktoren werden ja nicht vererbt, insofern bestimmt jede Klasse selbst, welche Konstruktoren sie hat.

  - `Throwable()`
  - `Throwable(String message)`
  - `Throwable(String message, Throwable cause)`
  - `Throwable(Throwable cause)`

Exceptions, die man kennen sollte (1)

- Das folgende Beispiel-Programm demonstriert einige Exceptions (Laufzeit-Fehler), die bereits in früheren Kapiteln genannt wurden.

  So, wie ein erfahrener Programmierer die häufigsten Fehlermeldungen des Compilers kennt, sollte auch bei typischen Exceptions sofort klar sein, was sie bedeuten (die eigentliche Ursache zu finden, kann dann etwas dauern).

- Im Beispiel-Programm wird für jede Methode im Kopf deklariert, welche Exception sie erzeugen kann.

  Die Methoden sind so gemacht, dass sie die Exception auch wirklich erzeugen.

- Die Ausgabe-Anweisung wird jeweils nicht ausgedruckt, weil die Ausführung der Methode aufgrund der Exception abgebrochen wird.

  Es findet ein nicht-lokaler Sprung zum `catch`-Block im Hauptprogramm statt.

Exceptions, die man kennen sollte (2)

```
(1)  // Fuer einige Beispiele ist eine Klasse
(2)  // mit Subklasse noetig:
(3)  class Ober { int a = 1; }
(4)  class Unter extends Ober { }
(5)
(6)  class ExceptionTest {
(7)
(8)      static void test1()
(9)          throws ArrayIndexOutOfBoundsException
(10)     {
(11)         int[] a = new int[5];
(12)         int i = 5;
(13)         int n = a[i]; // Exception
(14)         System.out.println("Nie ausgeführt.");
(15)     }
(16)
```

Exceptions, die man kennen sollte (3)

```
(17)    static void test2()
(18)        throws StringIndexOutOfBoundsException
(19)    {
(20)        String s = "abc";
(21)        int i = 3;
(22)        char c = s.charAt(i); // Exception
(23)        System.out.println("Nie ausgeführt.");
(24)    }
(25)
(26)    static void test3()
(27)        throws NullPointerException
(28)    {
(29)        Ober o = null;
(30)        o.a = 2; // Exception
(31)        System.out.println("Nie ausgeführt.");
(32)    }
(33)
```

Exceptions, die man kennen sollte (4)

```
(34)    static void test4()
(35)        throws ClassCastException
(36)    {
(37)        Ober o = new Ober();
(38)        Unter u = (Unter) o; // Exception
(39)        System.out.println("Nie ausgeführt.");
(40)    }
(41)
(42)    static void test5()
(43)        throws ArrayStoreException
(44)    {
(45)        Ober o[] = new Unter[5];
(46)        o[0] = new Ober(); // Exception
(47)        System.out.println("Nie ausgeführt.");
(48)    }
(49)
```

Exceptions, die man kennen sollte (5)

```
(50)    static void test6()
(51)        throws ArithmeticException
(52)    {
(53)        int i = 0;
(54)        int n = 5 / i; // Exception
(55)        // Bei double keine Exception (-> NaN)
(56)        System.out.println("Nie ausgeführt.");
(57)    }
(58)
(59)    static void test7()
(60)        throws NumberFormatException
(61)    {
(62)        String s = "abc";
(63)        int n = Integer.parseInt(s); // Excep.
(64)        System.out.println("Nie ausgeführt.");
(65)    }
(66)
```

Exceptions, die man kennen sollte (6)

```
(67)      public static void main(String[] args) {
(68)         for(int i = 1; i <= 7; i++) {
(69)            System.out.print("test"+i+"(): ");
(70)            try {
(71)               switch(i) {
(72)                  case 1: test1();
(73)                  case 2: test2();
...                   ...
(79)               }
(80)            }
(81)            catch(Exception e) {
(82)               System.out.println(e);
(83)            }
(84)         }
(85)      }
(86)
(87)   }
```

Inhalt

1. Umgang mit Fehlern
2. Assertions
3. Exception-Klassen
4. throw
5. Programmausführung

Auslösen von Exceptions mit throw (1)

- Man löst eine Exception aus mit der Anweisung

  throw ⟨Expression⟩;

- Dabei muss die Expression ein Objekt der Klasse `Throwable` oder einer ihrer Unterklassen liefern.

- Beispiel:

```
String meldung = "Falsche Kanal-Nummer: " + kanal;
throw new IndexOutOfBoundsException(meldung);
```

  Dieses Beispiel stammt aus einer Feuerwerks-Anwendung. Eine Zündanlage für die elektrische Zündung von Feuerwerksartikeln hat eine gewisse Anzahl Kanäle, also Klemmen für Zündstromkreise. Wenn man eine Methode zum Speichern der Daten für ein Feuerwerk mit einer ungültigen Kanalnummer aufruft, scheint `IndexOutOfBoundsException` passend zu sein. `IllegalArgumentException` würde natürlich auch gehen.

Auslösen von Exceptions mit `throw` (2)

- Wie `break` oder `return` unterbricht `throw` den normalen Fluss der Ausführung, direkt folgende Statements würden niemals ausgeführt.

  Ein `throw` könnte also z.B. am Ende eines `if/else`-Blockes stehen.

  Ein `throw` am Ende des Methoden-Rumpfes ist auch möglich. Man braucht dann kein `return`, weil die Methode nicht normal verlassen wird.

- Man kann auch eine mit `catch` "aufgefangene" Exception erneut auslösen, um sie an die aufrufende Methode weiterzugeben:

```
throw e;
```

  Dies ist üblich, wenn man die Exception nicht wirklich behandeln kann, aber doch etwas tun muss. Wenn das für beliebige Exceptions gilt, wäre der `finally`-Block passender.

Exception-Deklaration für Methoden (1)

- Der Aufrufer einer Methode muss eventuell berücksichtigen, dass die Methode auch nicht-normal enden kann.
- Ein normaler Methoden-Aufruf kann dann auch dazu führen, dass die eigene Methode unerwartet verlassen wird.
- Wenn man z.B. eine Datenstruktur ändert, und temporär einen ungültigen Zwischenzustand hat, ist es wichtig, dass die Ausführung nicht einfach abgebrochen wird, bevor man wieder einen gültigen Zustand hergestellt hat.
- Deswegen sollte im Methoden-Kopf deklariert werden, welche Exceptions die Methode erzeugen kann.
- Für `Error` und `RuntimeException` und ihre Subklassen ist dies optional, für alle anderen Exceptions Pflicht.

Exception-Deklaration für Methoden (2)

- Beispiel:

```
void startZeit(int kanal)
    throws IndexOutOfBoundsException {
    ...
}
```

- Diese Exception müsste nicht unbedingt deklariert werden, da sie eine Subklasse von `RuntimeException` ist.
- Man kann auch mehrere Exception-Klassen angeben, durch Komma getrennt.

  Die "`throws`"-Klausel sagt nur aus, dass die Methode möglicherweise eine Exception dieses Typs erzeugen könnte. Man bekommt keine Fehlermeldung, wenn dieser Fall tatsächlich nie eintreten kann. Es wäre aber schlechter Stil, unmögliche Exceptions zu deklarieren. Man erzeugt damit auch eine zusätzliche Last für den Aufrufer (siehe nächste Folie).

Exception-Deklaration für Methoden (3)

- Wenn man eine Methode `m()` aufruft, die eine Exception vom Typ `E` erzeugen kann, muss man
  - die Exception selbst behandeln, d.h. `m()` in einem `try`-Block aufrufen, zu dem ein Exception-Handler für `E` (oder eine Oberklasse von `E`) gehört, oder
  - deklarieren, dass die eigene Methode Exceptions vom Typ `E` (oder einer Oberklasse von `E`) erzeugen kann.
- Dies ist die sogenannte "catch-or-throw" Regel.
- Vergisst man es, erhält man eine Fehlermeldung der Art:

```
ExDeclErr.java:4:
        unreported exception java.io.IOException;
        must be caught or declared to be thrown
```

Inhalt

1. Umgang mit Fehlern
2. Assertions
3. Exception-Klassen
4. throw
5. Programmausführung

Abruptes Ende der Ausführung (1)

- Die Java Sprachspezifikation unterscheidet zwischen normaler und abrupter Beendigung von Statements (Anweisungen) und Expressions (Wertausdrücken).
- Eine abrupte Beendigung von Statements ist möglich durch ein `return`, `break`, `continue`-Statement, oder durch Auftreten einer Exception.
- Die Auswertung von Expressions (Wertausdrücken) kann nur durch das Auftreten von Exceptions abrupt beendet werden.
- Ein abruptes Ende bedeutet, die normal folgenden Statements bzw. Teilausdrücke nicht ausgeführt werden, sondern zu einer anderen Stelle gesprungen wird.

Abruptes Ende der Ausführung (2)

- Sei z.B. folgende Anweisung betrachtet:

```
y = f(x) + g(x);
```

- Falls der Aufruf von `f` eine Exception auslöst, wird `g` nicht mehr aufgerufen, und die Zuweisung an `y` findet nicht statt.

  Java hat (im Gegensatz zu C++) eine definierte Auswertungsreihenfolge, so dass z.B. klar ist, dass `f` vor `g` aufgerufen wird. Falls `g` Seiteneffekte hat (eine Ausgabe macht oder Attribute von Objekten oder Klassen ändert), werden diese sicher nicht ausgeführt.

- Auch im gleichen Block folgende Anweisungen werden nicht mehr ausgeführt.

  Auch umgebene Statements enden. Wenn diese Zuweisung z.B. im Rumpf einer Schleife steht, wird die Schleife sofort abgebrochen (es sei denn, dazwischen wäre eine `try`-Anweisung mit einer passenden `catch`-Klausel, s.u.).

Wiederholung: Fangen von Exceptions (1)

- Falls die durch eine Exception abgebrochene Anweisung in einem `try`-Block geschachtelt ist, werden die `catch`-Klauseln dieser Anweisung von oben nach unten geprüft, ob die dort angegebene Klasse eine Oberklasse der Exception ist.
  - Ist diese Bedingung erfüllt, so wird der zugehörige Block ausgeführt, und die Exception gilt als behandelt.

    Das Ausführung wird dann nach dem `try`-Statement fortgesetzt (falls vorhanden, wird dazwischen noch `finally` ausgeführt).
  - Ist in keiner der `catch`-Klauseln eine Oberklasse der aufgetretenen Exception angegeben, ist sie nicht behandelt, und es wird weiter außen (in einer anderen `try`-Anweisung) nach einem passenden Exception Handler gesucht.

    Auch hier wird vorher noch `finally` ausgeführt. Falls es in der gleichen Methode kein (anders) `try` gibt, wird beim Aufrufer gesucht.

Wiederholung: Fangen von Exceptions (2)

- Beispiel: `f(x)` wirft eine `IndexOutOfBoundsException`:

```
try {
    y = f(x) + g(x);
    System.out.println("Im try-Block folgend");
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Exception Handler 1");
} catch(IndexOutOfBoundsException e) {
    System.out.println("Exception Handler 2");
} catch(RuntimeException e) {
    System.out.println("Exception Handler 3");
} finally {
    System.out.println("Finally");
}
System.out.println("Normal weiter ...");
```

Wiederholung: Fangen von Exceptions (3)

- Im Beispiel erfolgt folgende Ausgabe:
  - `Exception Handler 2`
  - `Finally`
  - `Normal weiter ...`
- Obwohl die Klasse der Exception auch eine Subklasse `RuntimeException` ist, wird der Exception Handler 3 nicht ausgeführt.

  Es wird immer der erste passende Exception Handler genommen. Man muss die Unterklassen also vor den Oberklassen schreiben, sonst werden die Handler für die Unterklassen nie ausgeführt.
- Selbst wenn die Exception im `catch`-Block neu geworfen werden würde, werden im gleichen `try` folgende Exception Handler nicht mehr aktiviert.

Wiederholung: Fangen von Exceptions (4)

- Beispiel: `f(x)` wirft eine `IOException`:

```
try {
    y = f(x) + g(x);
    System.out.println("Im try-Block folgend");
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Exception Handler 1");
} catch(IndexOutOfBoundsException e) {
    System.out.println("Exception Handler 2");
} catch(RuntimeException e) {
    System.out.println("Exception Handler 3");
} finally {
    System.out.println("Finally");
}
System.out.println("Normal weiter ...");
```

Wiederholung: Fangen von Exceptions (5)

- Falls in der Methode selbst kein passender Exception Handler gefunden wird, wird die Ausführung der Methode abrupt beendet, und zwar mit dieser Exception.
- Es ist ganz typisch, dass ein gewisser Teil der Aufruf-Hierarchie so beendet wird, bis zu einer Methode, die "neu aufsetzen kann", z.B.
  - den Rest der Eingabezeile / Datensatzes überspringen, und mit der nächsten Zeile weitermachen,
  - den Benutzer fragen, ob die Aktion nochmal versucht werden soll (ggf. mit einem anderen Dateinamen etc.),
  - den Zustand vor der misslungenden Aktion wieder herstellen, und den nächsten Befehl ausführen.

Beispiel: Nichtlokaler Sprung (1)

```
(1)  class Sprung {
(2)
(3)      static int g() {
(4)          System.out.println("In g() ...");
(5)          int i = 0;
(6)          int n = 1 / i; // Hier passiert's
(7)          System.out.println("Nicht gedruckt.");
(8)          return 1; // Nicht ausgeführt
(9)      }
(10)
(11)     static int f() {
(12)         System.out.println("In f() ...");
(13)         int j = g();
(14)         System.out.println("Nicht gedruckt");
(15)         return 2; // Auch nicht ausgeführt
(16)     }
(17)
```

Beispiel: Nichtlokaler Sprung (2)

```
(18)        public static void main(String[] args) {
(19)            System.out.println("Anfang.");
(20)            try {
(21)                int k = f();
(22)                System.out.println("Nicht gedruckt");
(23)            }
(24)            catch(Exception e) {
(25)                System.out.println("Exception!");
(26)                e.printStackTrace();
(27)            }
(28)            System.out.println("Ende.");
(29)        }
(30)    }
```

Beispiel: Nichtlokaler Sprung (3)

- Das Programm erzeugt folgende Ausgaben:

```
Anfang.
In f() ...
In g() ...
Exception!
java.lang.ArithmeticException: / by zero
        at Sprung.g(Sprung.java:6)
        at Sprung.f(Sprung.java:13)
        at Sprung.main(Sprung.java:21)
Ende.
```

- Weil `f()` und `g()` jeweils keinen Exception Handler haben, springt die Ausführung von Zeile 6 zu Zeile 24 und verlässt dabei zwei Methodenaufrufe auf nicht-normalem Wege.

  Deswegen spricht man auch von einem "nichtlokalen Sprung".