Objektorientierte Programmierung

Kapitel 9: Exceptions I

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

Motivation (1)

- Methoden können manchmal mit Eingabewerten oder in Situationen aufgerufen werden, bei denen sie ihre Aufgabe nicht erfüllen können.
- Z.B. gibt es in der Bibliotheks-Klasse `Integer` [http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html] die statische Methode `parseInt`:

```
static int  parseInt(String s)
            Parses the string argument as a
            signed decimal integer.
```

- Diese Methode wandelt eine Zeichenkette, die aus Ziffern besteht, in die entsprechende Zahl um:

```
int i = Integer.parseInt("123");
```

- Was ist aber mit `Integer.parseInt("abc")`?

Motivation (2)

- Typische (früher übliche) Behandlungsmöglichkeiten:
  - Es wird ein Fehlerwert zurückgeliefert, z.B. `-1`.

    Das ist hier nicht so einfach, siehe nächste Folie.
  - Das Programm wird mit einer Fehlermeldung beendet.
- Beide Möglichkeiten haben wesentliche Nachteile (siehe folgende Folien).
- Deswegen haben moderne Sprachen üblicherweise einen Exception-Mechanismus (engl. “exception”: Ausnahme).

  Auch dies ist allerdings keine perfekte Lösung (s.u.).
- Die wesentliche Idee ist dabei, die Fehlererkennung und die Fehlerbehandlung zu trennen.

Motivation (3)

Probleme mit Fehlerwert:

- Manchmal gibt es keinen unbenutzten Wert, den man zur Signalisierung von Fehlern benutzen kann.

  `Integer.parseInt()` kann jeden möglichen `int`-Wert liefern. In solchen Situationen müßte man ein Objekt mit zwei Komponenten liefern: Dem eigentlichen Rückgabewert und dem Status-Indikator (Fehler oder ok).

- Diese Art der Fehlerbehandlung ist mühsam, weil bei jedem Aufruf abgefragt werden muss, ob er erfolgreich war.

  Das Programm wird dadurch deutlich länger.

- Diese Abfrage kann leicht vergessen werden.
  Dann wird mit einem falschen Wert weitergerechnet.

Motivation (4)

Probleme mit Programm-Abbruch:

- Die Methode, die den Fehler feststellt, weiß nicht, ob der Aufrufer noch etwas Wichtiges zu tun hat.
- Z.B. bei einem Editor / Spiel: Noch einmal abspeichern.

  Am besten in eine temporäre Datei, weil die Daten im Puffer ja durch den Fehler beschädigt sein könnten, und man den letzten sicheren Stand nicht überschreiben will. Aber wenn der Benutzer lange mit dem Programm gearbeitet/gespielt hat, möchte er möglichst vermeiden, dass durch einen Programmierfehler alles umsonst war. Bei dem Spiel Nethack wurde festgestellt, dass der in solchen Fällen gesicherte Spielstand für das Debugging oft nützlich war.
- Bei einem Kernkraftwerk: Anlage in einen sicheren Betriebszustand herunterfahren.

Motivation (5)

Probleme mit Programm-Abbruch, Forts.:

- Wenn man als Aufrufer den Programm-Abbruch verhindern will, müsste man die Zeichenkette vorher prüfen.
- Dabei würde man die Arbeit, die die Methode leisten soll, aber zu einem wesentlichen Teil duplizieren.
- Nicht immer läßt sich die Fehler-Möglichkeit so deutlich vorhersehen:
  - Man will ein Objekt anlegen, aber es ist nicht mehr genügend Hauptspeicher da.
  - Man will in eine Datei schreiben, aber die Platte ist voll.
  - Man will eine Datei zum Schreiben eröffnen, aber die Zugriffsrechte reichen nicht.

Es gibt keine perfekte Lösung!

Probleme mit Exceptions:

- Wenn eine Ausnahme oder ein Fehler auftritt, springt die Programmausführung zu einem "Exception Handler", das ist ein Block mit Programmcode, der das Problem lösen soll.
- Dabei wird anderer Programmcode, der normalerweise ausgeführt würde, übersprungen.
- Das könnte Datenstrukturen in halb-fertigem oder inkonsistenten Zustand hinterlassen.
- Allerdings hat Java Mechanismen, die das Problem abmildern:
  - Exceptions müssen in Methodenköpfen deklariert werden.
  - Die `finally`-Klausel ist für Aufräumarbeiten vorgesehen.

Inhalt

try-catch (1)

- Wenn man damit rechnet, dass eine Exception in einer Anweisung auftreten kann, und man darauf reagieren will, schreibt man sie in einen "`try-catch`"-Block:

```
try {
    int n = Integer.parseInt(args[0]);
    System.out.println("n^2 = " + n*n);
} catch(NumberFormatException e) {
    System.out.println("Eingabeformat falsch");
}
```

- Falls bei der Ausführung einer Anweisung eine Exception auftritt, wird sie sofort abgebrochen, und es findet ein Sprung zu einem weiter außen liegenden `catch`-Block statt (mit passendem Exception-Typ).

  Dieses Verhalten ähnelt einer `break`-Anweisung.

try-catch (2)

- Im Beispiel wird der Befehl zum Drucken der Quadratzahl

  ```
  System.out.println("n^2 = " + n*n);
  ```

  also nicht ausgeführt, wenn bei der Umwandlung des Argumentes ein Fehler auftrat.
- Die Variable `n` wäre dann ja auch gar nicht initialisiert.
- Das Bild, das in den Schlüsselworten zum Ausdruck kommt, ist:
  - Eine Exception (Ausnahme) wird "geworfen" (mit dem Befehl "`throw`", siehe Kapitel 18).
  - Irgendwo weiter außen im Programm, nicht unbedingt in der gleichen Methode, wird sie "aufgefangen" (mit einer `catch`-Klausel im `try`-Statement).

try-catch (3)

- Im Beispiel steht die `throw`-Anweisung in der Definition der Methode `parseInt`.
- In der Detail-Beschreibung der Methode `parseInt` ist dokumentiert, dass sie eine Ausnahme vom Typ `NumberFormatException` auslösen kann:

```
public static int parseInt(String s)
                    throws NumberFormatException
```

- Tatsächlich ist `NumberFormatException` eine Klasse:

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

- Beim Werfen der Exception wird ein Objekt dieser Klasse erzeugt, das weitere Informationen über den Fehler enthält.

try-catch (4)

- Der Sprung zu einem "Exception-Handler" (dem passenden `catch`-Block) findet auch über Methodengrenzen hinweg statt ("nicht-lokaler Sprung"):
  - die Ausführung der Methode `parseInt` wird abgebrochen,

    Es ist ganz typisch, dass die Methode, die die Exception auslöst, sie nicht selbst behandelt. Wenn sie das könnte, wären Exceptions das falsche Konstrukt.
  - und es wird in der aufrufenden Methode (unser `main`) nach einem passenden Exception Handler gesucht.

    Ggf. würden auch noch weitere Methoden verlassen, um einen Exception Handler zu finden, aber "`main`" ist schon die äußerste Methode.
  - Hat schließlich auch `main` keinen Exception Handler, wird das Programm mit einer Fehlermeldung beendet.

try-catch (5)

- Im `try`-Block können unterschiedliche Typen von Exceptions auftreten.
- Man kann für jeden Typ eine eigene Ausnahmebehandlung schreiben, indem man nach einem `try`-Block mehrere `catch`-Blöcke hat (Beispiel siehe nächste Folie).
- Die Exception-Typen sind in einer Hierarchie angeordnet (Klassen-Hierarchie), die in Kapitel 18 besprochen wird.
- Die allgemeinste Art normaler Exceptions ist "`Exception`":

```
catch(Exception e) { ... }
```

  Hiermit kann man alles außer echten Fehlern abfangen.

- Da der erste passende Exception-Handler benutzt wird, muss so ein allgemeiner Handler ganz unten stehen.

try-catch (6)

```
import java.util.Scanner;
class TestExcep {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        try {
            String eingabe = scan.nextLine();
            int n = Integer.parseInt(eingabe);
            System.out.println("n^2 = " + n*n);
        } catch(NumberFormatException e) {
            System.out.println(
                "Eingabeformat falsch");
        } catch(Exception e) {
            System.out.println(
                "Anderer Fehler (z.B. EOF)");
        }
        System.out.println("Tschues!");
    }
}
```

try-catch (7)

- Nachdem der ausgewählte `catch`-Block abgearbeitet ist, wird mit Befehlen nach dem `try-catch` fortgesetzt.
- Im Beispiel wird, falls die Eingabe keine Zahl war, zunächst vom ersten Exception Handler Folgendes ausgegeben:

  `Eingabeformat falsch`
- Die anderen Exception Handler werden nicht betreten.

  Obwohl die `NumberFormatException` ein Spezialfall von `Exception` ist.
- Anschließend gibt die Anweisung nach dem `try-catch` aus:

  `Tschues!`
- Hätte es keinen passenden Exception Handler gegeben, wäre sofort die ganze Methode verlassen worden.

  Dann wäre auch das `Tschues!` nicht mehr ausgegeben worden.

try-catch (8)

- Falls bei `scan.nextLine()` eine `NoSuchElementException` auftritt (Benutzer hat mit `Ctrl+D` "Dateiende" eingegeben), werden die markierten Befehle ausgeführt:

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    try {
        String eingabe = scan.nextLine();
        int n = Integer.parseInt(eingabe);
        System.out.println("n^2 = " + n*n);
    } catch(NumberFormatException e) {
        System.out.println(
            "Eingabeformat falsch");
    } catch(Exception e) {
        System.out.println(
            "Anderer Fehler (z.B. EOF)");
    }
    System.out.println("Tschues!");
}
```

try-catch (9)

- Falls `nextLine()` klappt, aber bei `parseInt(eingabe)` eine `NumberFormatException` auftritt (Eingabe: “abc”):

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    try {
        String eingabe = scan.nextLine();
        int n = Integer.parseInt(eingabe);
        System.out.println("n^2 = " + n*n);
    } catch(NumberFormatException e) {
        System.out.println(
            "Eingabeformat falsch");
    } catch(Exception e) {
        System.out.println(
            "Anderer Fehler (z.B. EOF)");
    }
    System.out.println("Tschues!");
}
```

try-catch (10)

- Falls alles ohne Fehler durchläuft:

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    try {
        String eingabe = scan.nextLine();
        int n = Integer.parseInt(eingabe);
        System.out.println("n^2 = " + n*n);
    } catch(NumberFormatException e) {
        System.out.println(
            "Eingabeformat falsch");
    } catch(Exception e) {
        System.out.println(
            "Anderer Fehler (z.B. EOF)");
    }
    System.out.println("Tschues!");
}
```

try-catch (11)

- Selbstverständlich wird ein Exception Handler nur betreten, wenn die Exception im zugehörigen `try`-Block auftritt.

  Z.B. wird eine Exception, die in einem Exception Handler auftritt, nicht mehr im gleichen `try-catch`-Statement behandelt, selbst wenn noch ein ganz allgemeiner Exception-Handler folgt (ggf. aber in `finally`).

- Wenn die Exception außerhalb von `try` auftritt, wird die Methode verlassen, und ggf. beim Aufrufer ein Exception Handler ausgeführt, wenn der Methoden-Aufruf dort in einem `try`-Block stand.

  Auf `main` trifft das nicht zu, hier endet die Programmausführung mit einer Fehlermeldung, wenn eine Exception außerhalb eines `try`-Blocks auftritt.
  Das Hauptprogramm `main` steht schon am Ende der Aufruf-Hierarchie.
  Man kann sich das auch so vorstellen, dass ganz außen noch ein Exception Handler ist, der alle Exceptions dadurch "behandelt", dass er eine Fehlermeldung ausgibt und das Programm beendet.

try-catch (12)

Syntaktische Hinweise:

- Nach `try`, `catch` (und `finally`, s.u.) muss jeweils ein Block stehen.

  Auch wenn es sich nur um ein einzelnes Statement handelt, muss man die Klammern `{...}` schreiben.

- In den Klammern vom "`catch(...)`" muss man einen Parameter wie `e` deklarieren, selbst wenn man ihn nicht verwendet.

  Es reicht nicht, nur die Klasse anzugeben.

Methoden für Exceptions (1)

- Beim Auslösen ("Werfen") einer Exception wird ein Objekt erzeugt, das Informationen über die Exception enthält.
- In der `catch`-Klausel deklariert man eine Variable, über die dieses Objekt zugreifbar wird:

```
catch(NumberFormatException e) {
    System.out.println(e.getMessage());
    ...
}
```

- Im Beispiel wird die in der Exception gespeicherte Fehlermeldung ausgedruckt.
- Exceptions sind ein Untertyp von "`Throwable`", deswegen sind alle für diese Klasse deklarierten Methoden für jeden Exception-Typ benutzbar (siehe nächste Folie).

Methoden für Exceptions (2)

- `Throwable` hat u.a. folgende Methoden:

  Siehe [http://docs.oracle.com/javase/6/docs/api/java/lang/Throwable.html]

  - `String getMessage()`
    Liefert einen Fehlermeldungstext ("detail message").
  - `String toString()`
    Name der Exception-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.

Inhalt

finally (1)

- Nach den catch-Klauseln kann man noch einen finally-Block angeben:

```
try {
    ...
} catch(...) {
    ...
} catch(...) {
    ...
} finally {
    ...
}
```

- Es ist auch möglich, finally ohne catch zu verwenden.

`finally` (2)

- Die Anweisungen im `finally`-Block werden auf jeden Fall ausgeführt,
  - sowohl, wenn eine Exception bei der Ausführung des `try`-Blockes aufgetreten ist,

    In diesem Fall nach der Ausführung des passenden Exception Handlers (falls die Klasse in einer der `catch`-Klauseln passte, sonst direkt nach dem `try`-Block).
  - als auch, wenn die Ausführung des `try`-Blocks normal bis zum Ende gekommen ist,
  - sogar, wenn der `try`-Block mit `break` oder `return` verlassen wurde.

finally (3)

- Der Zweck des `finally`-Blocks ist es, eventuelle Abschluss-Arbeiten noch durchführen zu können, bzw. belegte Ressourcen wieder frei zu geben.
- Wenn man z.B. eine Datei eröffnet hat, sollte man die wieder schließen.

  Die Gesamtanzahl der gleichzeitig offenen Dateien pro Prozess ist durch das Betriebssystem begrenzt (z.B. auf 15). Wenn man immer wieder Dateien öffnet, ohne sie zu schließen, kommt man in Schwierigkeiten.
- Es wäre mühsam und fehleranfällig, jeden möglichen Ausführungspfad zu bedenken.

  Deswegen ist es so nützlich, dass `finally` auf jeden Fall ausgeführt wird.

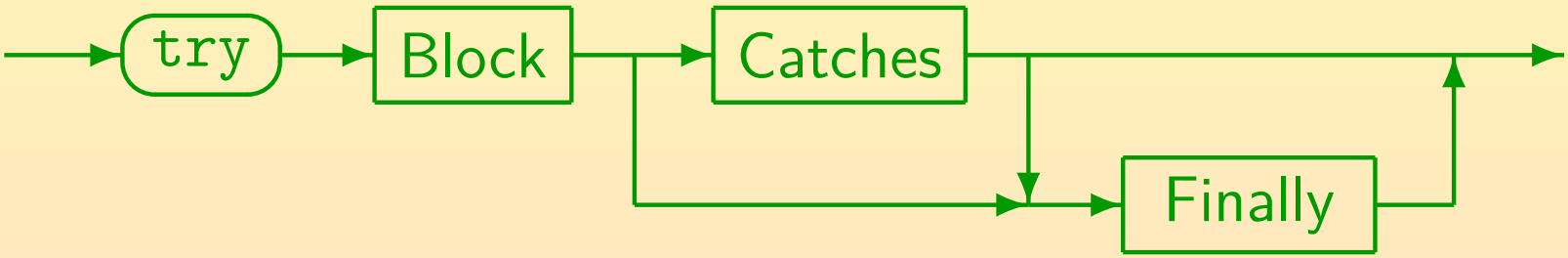
finally (4)

- Beim Betreten des `finally`-Blockes wird gemerkt, welches Ende die bisherige Ausführung hatte, z.B., dass es ein `return` mit einem bestimmten Wert war.
- Wenn der `finally`-Block normal endet, wird dem gemerkten Ende fortgefahren.

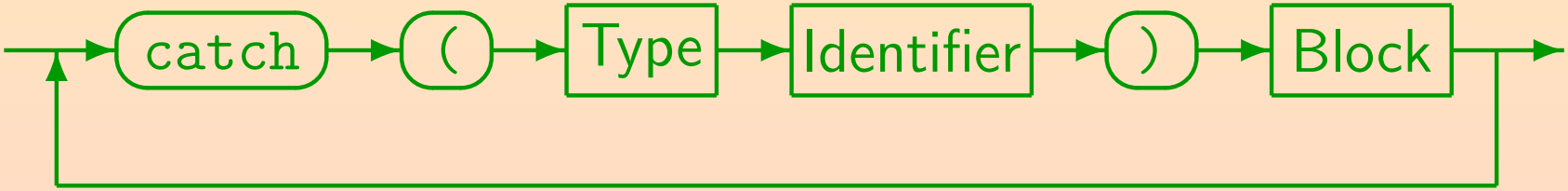
  Das `return` findet also noch statt, der `finally`-Block schiebt sich nur dazwischen.
- Der `finally`-Block kann aber auch selbst ein `return` durchführen oder eine Exception auslösen, dann wird das gemerkte Ende überschrieben.

Syntax

- TryStatement:

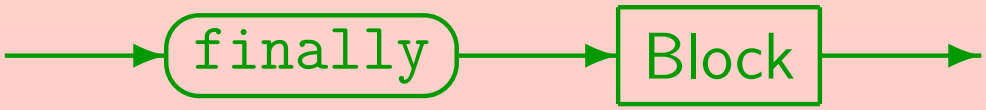


- Catches:



Dies ist etwas vereinfacht, vor dem Typ wären noch "VariableModifiers" möglich (z.B. `final`).

- Finally:



Inhalt

Arten von Exceptions (1)

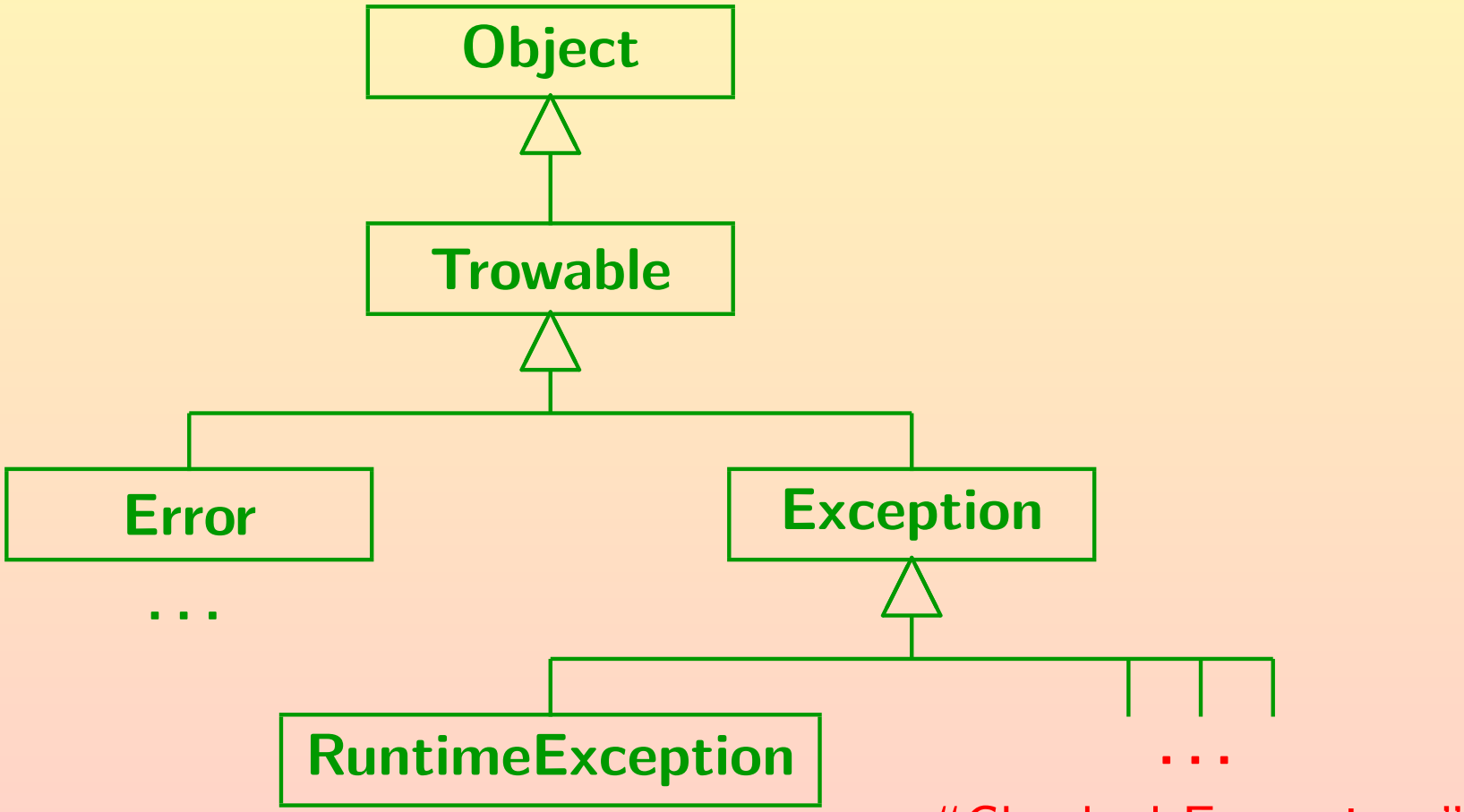
- Java unterscheidet drei Kategorien von Exceptions:
  - Exceptions, bei denen die Situation ziemlich hoffnungslos ist, z.B. `VirtualMachineError`.

    Die übergeordnete Klasse für solche schweren Fehler ist `Error`. Man kann versuchen, diese Situation mit `catch` abzufangen, aber es ist nicht klar, ob das noch Sinn macht oder wirklich funktioniert.
  - Exceptions, die man abfangen kann, aber nicht muss, z.B. `ArrayIndexOutOfBoundsException`.

    Dies sind alles Spezialfälle von `RuntimeException`. Man kann schlecht bei jedem Array-Zugriff ein `try` vorsehen, aber schon für einen größeren Teil des Programms.
  - Exceptions, die man behandeln muss, sogenannte "Checked Exceptions". Hierzu gehört z.B. `IOException`.

    Dies sind alle, die in der Klassenhierarchie unterhalb von `Exception` stehen, mit Ausnahme der Teilmenge der `RuntimeException`.

Arten von Exceptions (2)



Zwang zur Behandlung von Exceptions (1)

- Für "Checked Exceptions" gilt die sogenannte "catch or throw" Regel: Wenn man sie nicht behandelt, muss man im Methodenkopf deklarieren, dass man sie liefern kann:

```
import java.io.IOException;

class ExcepDecl {
    public static void main(String[] args)
        throws IOException
    {
        ...
```

- Für `main` macht das keinen besonderen Sinn, weil es keinen "Aufrufer" mehr gibt.
- Für andere Methoden ist aber wichtig, mit welchen Exceptions man rechnen muss.

Zwang zur Behandlung von Exceptions (2)

- Wenn man eine Methode aufruft, die eine "Checked Exception" auslösen kann, und sie nicht behandelt oder deklariert, erhält man folgende Fehlermeldung vom Compiler:

```
DateiLesen.java:27:
error: unreported exception IOException;
       must be caught or declared to be thrown
       in.close();
                  ^
```

- Man kann dann ein `try-catch` darum bauen, oder die Exception im Methodenkopf mit "`throws`" deklarieren.

  In realen Programmen sollte man die Exception behandeln (spätestens in `main`): Wenn das Programm später mit einer Exception abgebrochen wird, und der Benutzer bekommt nur die Java-Fehlermeldung, wird er Sie für einen schlechten Programmierer halten. Zu Recht.

Diskussion: NumberFormatException (1)

- Man kann sich fragen, warum `NumberFormatException` eine `RuntimeException` ist, so dass sie nicht unbedingt behandelt werden muss (keine "checked exception").
- Manchmal gibt es Fälle, die nicht ganz eindeutig sind.
- Offensichtlich sind die Java-Entwickler davon ausgegangen, dass der Programmierer vor dem Aufruf von `parseInt` die Zeichenkette testen sollte.
- Deswegen haben sie das Auftreten dieser Exception als Fehler angesehen, und nicht als vorhersehbaren Ausnahmefall, den man behandeln muss.

  `NumberFormatException` fällt unter `IllegalArgumentException`, die ihrerseits `RuntimeException` sind. Eine Methode mit ungültigem Argument aufzurufen, klingt nach Fehler.

Diskussion: NumberFormatException (2)

- Man kann es aber auch anders handhaben:
  - Man weiss ja, dass `parseInt` eine `NumberFormatException` liefern wird, wenn die Eingabe-Zeichekette keine Zahl ist.
  - Also braucht man die Zeichenkette nicht vorher zu prüfen, sondern kann die Exception abfangen.
  - Wenn man das vergisst, weist der Java-Compiler aber nicht darauf hin.

    Die Methoden der Klasse `java.text.NumberFormat` erzeugen dagegen eine `ParseException`, die unter die "checked exceptions" fällt.
- Es ist sehr wichtig, Eingaben immer zu prüfen: Egal, was der Benutzer eingibt, er sollte immer eine verständliche Fehlermeldung bekommen.

  Nicht einfach einen Programm-Abbruch mit Java Exception.

Ausblick

- Exceptions (Ausnahmen) sind eines der Java-Konstrukte, die es schwierig machen, den Stoff in eine logische Reihenfolge ohne Vorwärts-Referenzen zu bringen.
  - Einerseits braucht man Exceptions schon früh, weil z.B. Ein-/Ausgabemethoden recht stark davon abhängen.
  - Andererseits ist ein volles Verständnis von Exceptions nur möglich, nachdem man schon Subklassen kennt.
- Hier wurde versucht, das notwendige Wissen zur Benutzung von Methoden, die Exceptions auslösen können, zu vermitteln.
- Später wird die Definition eigener Exceptions und das Auslösen von Exceptions mittels `throw` beschrieben.