Programmierkurs

Vorlesung 09

M.Sc. Laslo Hunhold

Department Mathematik/Informatik
Abteilung Informatik
Universität zu Köln

18. Januar 2020

Letzte Vorlesung

- Konstanten mit `final static`
- Vertiefung zu Vererbung, Kompatibilität, Polymorphismus und abstrakten Klassen (klassische versus polymorphe Zeiger, Auflösung mit `instanceof`, Verkapselung, Verschachtelung abstrakter Klassen)
- Interfaces
- Rekursion
- Sortieralgorithmen (Sortierproblem, Selection Sort, Mergesort)

Annotationen

- Erlaubt syntaktische Anmerkung des Quelltexts
- Auf eigener Zeile, Beginnt mit `@` gefolgt von einem Schlüsselwort ohne Semikolon, z.B.

```
@Override
```

- Je nach Kontext anwendbar auf Klassen, Methoden, Variablen, Parametern und Packages
- Beispiel: `@Override` vor einer Methodendeklaration gibt bei der Kompilation eine Fehlermeldung zurück, wenn die Methode keine Methode in der Superklasse überschreibt
- Beispiel: `@Deprecated` vor einer Methodendeklaration gibt bei der Kompilation eine Warnung, wenn diese Methode verwendet wird

```
1 class Test {
2         @Deprecated
3         void alte_funktion() { ... }
4 }
```

Export/Import von Packages

Export

- Klassen standardmäßig nur paketweit sichtbar
- Sichtbarkeit über Paket hinaus mit `public` erreichbar („Export“)
- Variablen und Methoden in einer `public` Klasse müssen für Außenzugriff auch `public` sein

Import

- Zugriff auf `public` Klassen, Variablen und Methoden aus anderen Paketen nennt man „Import“
- Zwei Möglichkeiten
  1. Paketname
     - Eindeutiger Zugriff durch Angabe des Paketnamens
     - Beispiel: `Paket.Klasse` greift auf die `public` Klasse `Klasse` im Package `Paket` zu
  2. `import`
     - Schlüsselwort `import` gefolgt von einem Qualifikator
     - Beispiel:

```
import Paket.Klasse;
```

       importiert die `public` Klasse `Klasse` aus dem Package `Paket`
     - Zugriff im Programm nun über `Klasse` (neben `Paket.Klasse`) möglich; `Klasse` ist quasi in den Namensraum des Packages gerückt
     - Kann Probleme machen, wenn zwei gleichnamige Klassen aus zwei Paketen importiert werden
     - „Wildcard“-Platzhalter

```
import Paket.*;
```

       importiert alle `public` Klassen aus dem Package `Paket` (Achtung! Nicht aus „Subpackages“ (nächste Folie))

Programmhierarchie/„Subpackages“

```
/Paket0/Klasse0.java
/Paket0/Unterpaket0/KlasseX.java
/Paket1/Klasse1.java
/Paket2/Klasse2.java
```

- Angenommen unser Java-Programm liegt im Ordner `/`
- Packages `Paket0`, `Paket1`, `Paket2` mit Klassen `Klasse0`, `Klasse1` und `Klasse2` sind Ordner mit dem gleichen Namen in `/`
- Es gibt in Java keine „Subpackages“, sondern nur „Packages“
- Wie entstehen dann „Subpackages“, von denen oft die Rede ist?
- Nehme an `Paket0` hat das „Subpackage“ `Unterpaket0` mit Klasse `KlasseX`; „Subpackage“ wird konventionell `Paket0.Unterpaket0` genannt.
- Hat `Paket0.Unterpaket0` semantisch etwas mit `Paket0` zu tun?
- Nein, aber: Speicherung von `Unterpaket0` als ein Unterordner von `Paket0`; Punkte im Paketnamen werden als Schrägstrich interpretiert
- Kurzum: `Paket0` und `Paket0.Unterpaket0` sind semantisch unabhängige Pakete, aber sind vom Namen her verwandt und werden so in der Ordnerstruktur abgebildet

API/Klassenbibliothek

Motivation

- Wo kommt `System.out.println()` her?
- Welche anderen Funktionen werden bereitgestellt?
- Vorlesung kann nicht alle möglichen Sprachfunktionen behandeln; Verweis auf Dokumentation
- Dokumentation ist zentral für Programmierung

Klassenbibliothek/API

- Sammlung von Packages, die mit der Installation von Java bereitgestellt wird
- API bedeutet „application programming interface“, also Schnittstelle zur Programmierung von Anwendungen
- Dokumentation unter `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`
- `java.lang` („Wrapperklassen“ für primitive Datentypen, Klassen `System` und `Math`, etc.), wird automatisch importiert!
- `java.io` (Ein-/Ausgabe von Daten)
- `java.util` (Hilfsmethoden (z.B. auf Arrays), etc.)
- `javax.swing` (GUI)
- Viele weitere

Throwables (Exceptions/Errors)

Ausnahmen („Exceptions“)

- Zeigen Abweichungen vom vorgesehenen Programmablauf an
- Beispiel 1

```
Integer.parseInt("1.0");
```

  - "1.0" repräsentiert eine Fließkommazahl, keinen Integer
  - Fehlermeldung und Programmabbruch:

```
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "1.0"
```

- Beispiel 2

```
((String)(null)).charAt(0);
```

  - „Nullzeigerdereferenzierung“
  - Fehlermeldung und Programmabbruch:

```
Exception in thread "main" java.lang.NullPointerException
```

Fehler („Errors“)

- Zeigen Fehler mit der Ausführungsumgebung (Virtuelle Maschine) an
- Beispiel

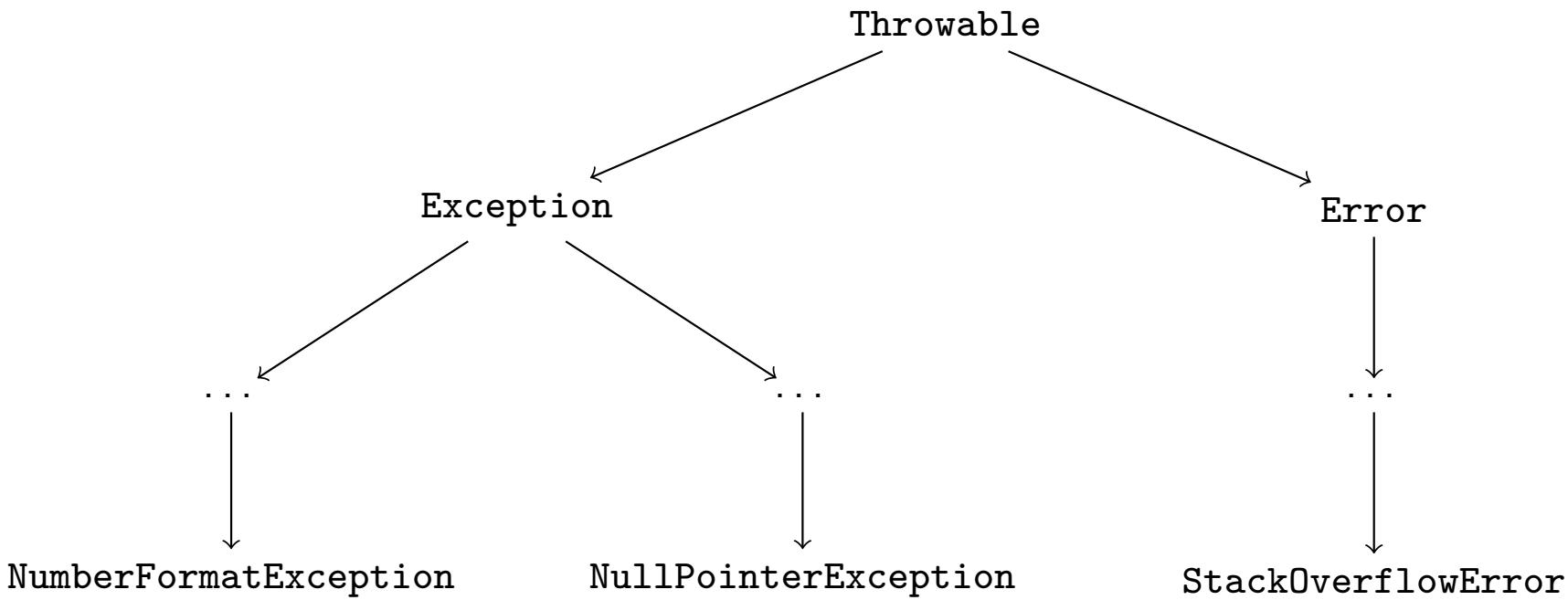
```
public class Main {
        static void hoelle() {
                hoelle();
        }

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

  - Stapelüberlauf
  - Fehlermeldung und Programmabbruch:

```
Exception in thread "main" java.lang.StackOverflowError
```

Throwable

- `java.lang` enthält die Klasse `Throwable`
- Instanzen von `Throwable` und aller von ihm (auch indirekt) erbende Klassen können „geworfen“ werden (siehe vorherige Beispiele)
- Fehlermeldung und Programmabbruch, wenn eine „geworfene“ Instanz nicht „gefangen“ wird (siehe vorherige Beispiele)



Abfangen von Throwables (`try...catch`)

```
1        try {
2                ...
3        } catch (ThrowableErbe0 e) {
4                ...
5        } catch (ThrowableErbe1 e) {
6                ...
7        }
8        ...
9        } finally {
10               ...
11       }
```

- Im `try`-Block steht der Code, der „ausprobiert“ und von dem eventuell etwas abgefangen werden soll
- `catch` ist optional (wenn `finally` vorkommt) und kann mehrmals vorkommen, jeweils für jeden Abfangobjekt-Typ; es wird bei Erfüllung nur ein `catch` ausgeführt
- Ein `catch` „fängt“ nicht nur Instanzen des angegebenen Typs ab, sondern auch alle davon (auch indirekt) erbende Klassen (Zeiger e ist damit unter Umständen polymorph) (Ausblick: Multi-Catch)
- Der `finally`-Block ist optional und wird am Ende immer ausgeführt, egal ob etwas geworfen wurde oder nicht (z.B. Cleanup)

Beispiel

„import java.util.Scanner;“ im Header

```
1  int[] a = { 1, 2, 3 };
2
3  try {
4          Scanner s = new Scanner(System.in);
5          int c = s.nextInt();
6          int b = a[3];
7  } catch (IndexOutOfBoundsException e) {
8          System.out.println("Fehler: Index zu hoch");
9  } catch (Exception e) {
10         System.out.println("Fehler: Allgemeine Exception");
11 } finally {
12         System.out.println("Wird immer ausgefuehrt");
13 }
14
15 System.out.println("Rest des Programms");
```

- Mögliche Testeingaben „5“ und „5.0“
- `InputMismatchException` ist Urenkel der Klasse `Exception` und wird deshalb im zweiten `catch` gefangen

Werfen von Throwables (`throw`)

```
1  class Main {
2          static void werfer() {
3                  NullPointerException e = new NullPointerException();
4                  throw e;
5                  /* oder "throw new NullPointerException();" */
6                  /* funktioniert auch mit Errors (z.B. StackOverflowError) */
7          }
8
9          public static void main(String[] args) {
10                 werfer();
11         }
12 }
```

```
1  class Main {
2          static void werfer() {
3                  throw new InterruptedException();
4                  /* Fehler: "Unhandled exception type" */
5          }
6
7          public static void main(String[] args) {
8                  werfer();
9                  /* Fehler: "Unhandled exception type" */
10         }
11 }
```

Warum dürfen wir ohne weiteres `NullPointerException` werfen, jedoch `InterruptedException` nicht?

Runtime Exceptions versus Checked Exceptions

- Zum vorherigen Beispiel: `InterruptedException` muß explizit im Methodenkopf angegeben (Schlüsselwort `throws` gefolgt von kommaseparierter Liste von Exceptions)

```
1 static void werfer() throws InterruptedException {
2         throw new InterruptedException();
3 }
```

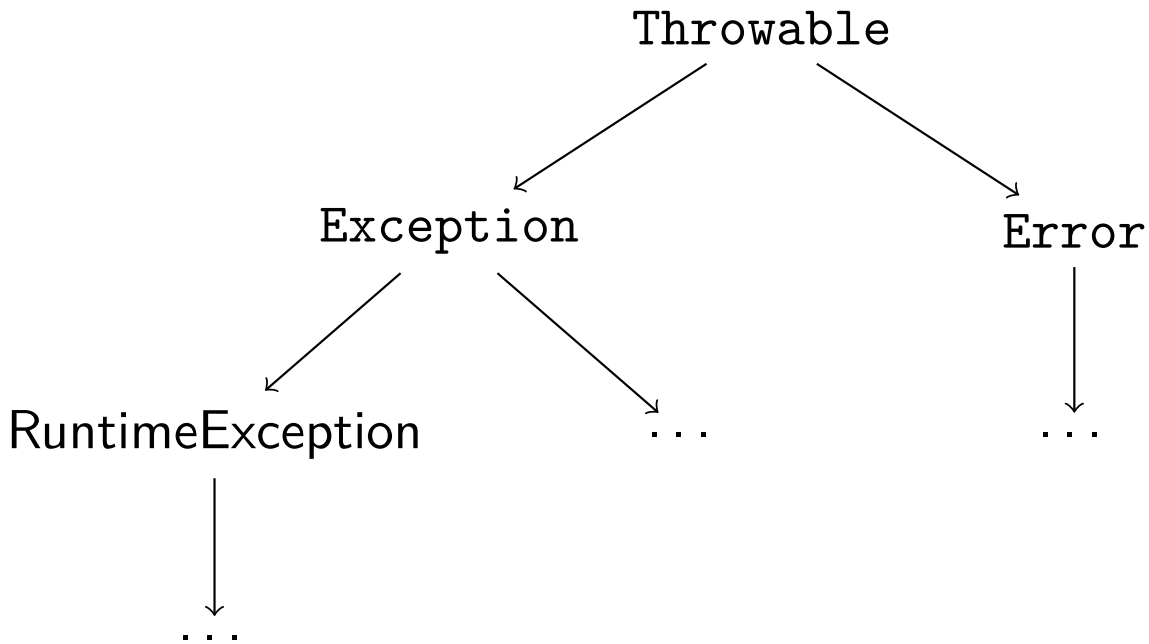
und beim Methodenaufruf (explizit oder implizit) gefangen werden

```
1 try {
2         werfer();
3 } catch (InterruptedException e /* oder z.B. Exception e */) { ... }
```

Diese Angaben kann man auch im `NullPointerException`-Fall machen, aber warum wird man dazu nicht gezwungen?

- Java unterscheidet zwischen Exceptions, die angegeben und abgefangen werden müssen („Checked Exceptions“), und welche, die davon befreit sind („Runtime Exceptions“), mit der `Exception`-Subklasse `RuntimeException`
- Erbt eine Klasse von `RuntimeException`, ist sie eine Runtime Exception (und „befreit“), ansonsten ist sie eine Checked Exception
- Eine nichtgefangene Runtime Exception wird einfach weitergereicht

Throwable-Hierarchie



Elegante Fehlerbehandlung mit Throwables

- `Throwable` enthält unter anderem die Methoden `toString()` und `getMessage()`, erbende Klassen noch mehr

```
1 try {
2         throw new Throwable("Fehlermeldung");
3 } catch (Throwable t) {
4         System.out.println(t.getMessage());
5 }
```

```
1 try {
2         ((String)(null)).charAt(0);
3 } catch (Exception e) {
4         System.out.println(e.toString());
5 }
```

- Weiterreichung von Checked Exceptions

```
1 try {
2         throw new Exception();
3 } catch (Exception e) {
4         throw e;
5 }
```

- Definition eigener Throwables als Subklassen anderer

Dateien lesen und schreiben

Klasse `File`

- Definiert in `java.io`
- Repräsentiert Dateien und Ordner, eventuell nichtexistente
- Konstruktor u.a. mit Signatur `File(String p)`
- `p` ist der Pfad zur Datei/zum Ordner; Zum Beispiel „/home/joe/testdatei“ (Un*x) oder „C:/Users/Joe/testdatei“ (Windows)
- Nach Instanziierung stehen einige Methoden zur Verfügung (`.exists()`, `.createNewFile()`, `.length()`, `.isDirectory()`, `.mkdir()`, `.delete()`, etc.)

„`import java.io.File;`“ im Header

```
1  File f = new File("/home/joe/testdatei");
2
3  if (f.exists()) {
4          System.out.println("Datei existiert, " + f.length() + "B");
5  } else {
6          System.out.println("Datei existiert nicht");
7  }
```

Klasse `FileReader`/`FileWriter`

- Definiert in `java.io`
- Dient dem Lesen bzw. Schreiben von Dateien, Zeichen für Zeichen
- Konstruktor u.a. mit Signatur `FileReader(File f)` bzw. `FileWriter(File f)`
- Nach Instanziierung steht u.a. die Methode `.read()` bzw. `.write()` zur Verfügung, die jeweils ein Zeichen liest bzw. schreibt; ersteres gibt -1 aus, wenn das Ende der Datei erreicht ist

Beispiel

„`import java.io.*;`“ im Header (oder explizit mit `File`, `FileReader` und `FileWriter`

```
 1 try {
 2         File f_in = new File("/home/joe/testdatei"),
 3              f_out = new File("/home/joe/testdatei.caps");
 4
 5         f_out.delete();
 6         f_out.createNewFile();
 7
 8         int i;
 9         FileReader f_in_r = new FileReader(f_in);
10         FileWriter f_out_w = new FileWriter(f_out);
11         while ((i = f_in_r.read()) != -1) {
12                 f_out_w.write(java.lang.Character.toUpperCase(i));
13         }
14         f_in_r.close();
15         f_out_w.close();
16 } catch (IOException e) {
17         System.out.println("I/O-Fehler: " + e.getMessage());
18 }
```

Abbildung: iPhone OS 1 „Home“-Bildschirm (2007) (Quelle: Apple Inc.)