Vorlesung Programmieren

09. Ausnahmebehandlung (Exceptions)

PD Dr. rer. nat. Robert Heinrich

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java

Datentypen

Primitive Datentypen
- Wertebereiche
- Konvertierung

Klassen

```
class Ball {
}
```

Zustand
- Konstruktoren

Verhalten
- Kontrollstrukturen
-- Schleifen
-- Verzweigungen
- Rekursion

Methodik

Grundlagen
- Ausführen
- Ein-/Ausgabe

Fehler finden
- Testen
- Debugging

Qualitätsstandards
- Code Conventions
- JavaDoc

Java
- Java Bibliotheken

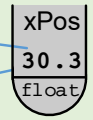
Design-Prinzipien
Best Practices

sind Schablonen für

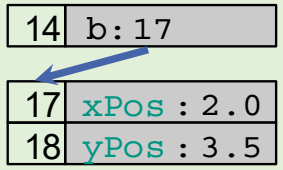
sind Instanzen von

Instanzen

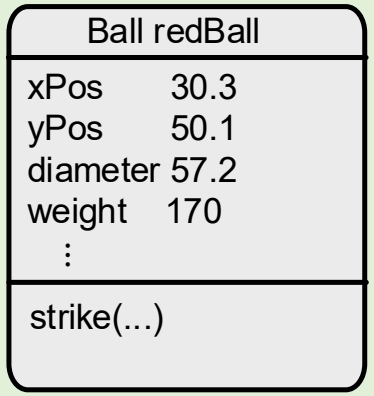
Werte Variablen



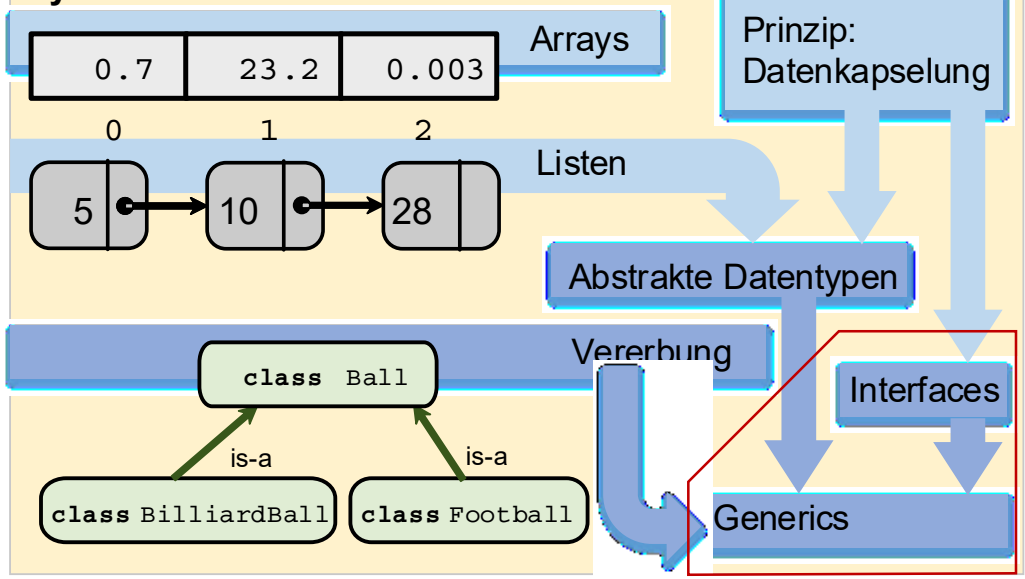
Objektidentität und Referenzen



Objekte



Systeme bauen



Vorlesungsüberblick: Vorläufiger Semesterplan

23.10.2023	Erstsemesterbegrüßung: Einführung
25.10.2023	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
08.11.2023	Typen und Variablen
15.11.2023	Kontrollstrukturen (+Scanner)
22.11.2023	Konstruktoren und Methoden
29.11.2023	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
06.12.2023	Listen und Abstrakte Datentypen
13.12.2023	Vererbung
20.12.2023	Exceptions; Interfaces
10.01.2024	Generics; Rekursion
17.01.2024	Java-API; Objektorientierte Design-Prinzipien
24.01.2024	Best Practices; Finden und Beheben von Fehlern
31.01.2024	Testen und Assertions; LV Evaluation
07.02.2024	Junit; Parsen, Suchen, Sortieren
14.02.2024	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen

Lernziele

Exceptions

- Sie können mithilfe von Exceptions Fehler in Ihrem Programm signalisieren und behandeln.
- Sie können damit sauber Programmlogik und Fehlerbehandlung trennen.
- Sie wissen, dass Fehlererkennung so früh wie möglich und defensiv durchgeführt werden sollte.

Quelle: http://phdcomics.com

Exceptions: Motivation

- Während des Programmablaufs können verschiedene Dinge fehlschlagen:
  - Öffnen einer Datei, die es nicht gibt
  - Zugriff auf ein Array-Element außerhalb der zulässigen Indexgrenzen
  - Division durch Null
  - Zugriff auf eine **`null`**-Referenz
  - ...

- Wie fängt man diese ganzen Fehler ab?
- Wie lässt sich das am besten im Programm umsetzen (ohne der Lesbarkeit des Programms zu sehr zu schaden)?

Fehlerbehandlung im GOTO-Zeitalter

```
loop:
  ...
  if (...) {
    ...
    if (...) {  // Fehler
      goto error;
    }
    ...
    if (...) {  // Schleifenende
      goto loopExit;
    }
  }
  ...
  goto loop;
loopExit:
  goto end;
error:
  ...  // Fehlerbehandlung
end:
  ...
```

- Schleife durch **`goto`**s
- Fehlerbehandlung am Ende
- Schleifenbedingung mitten in Schleife
- Komplizierter Kontrollfluss

Fehlerbehandlung – ohne GOTO, kaum besser

```
boolean quit = false, err = false;
while (!quit) {
  ...
  if (...) {
    ...
    if (...) {  // Fehler
      quit = true; err = true;
    } else {
      ...
      if (...)  // Schleifenende
        quit = true;
    }
  }
  if (!quit) {
    ...
  }
}
if (!err) {
  ...
} else {
  ...  // Fehlerbehandlung
}
```

- Schleifenbedingung mitten in Schleife
- Zahllose `if`-Abfragen
- Komplizierter Kontrollfluss

Lokale Fehlerbehandlung

- Typische (lokale) Fehlerbehandlung:

```
f = openFile("input.txt")
if (f < 0) {
  System.out.println("Datei konnte nicht geöffnet werden.");
  System.out.println("Grund: " + f);
} else {
  ...
}
```

- **Probleme:**
  - Vermischung von Programmlogik und Fehlerbehandlung
  - Fehlerausgaben im Programm verstreut

  ➜ Keine Trennung von Algorithmik und Benutzerinteraktion

Ausnahmen (Exceptions)

Exceptions

- behandeln eine Fehlersituation (eine Ausnahme) im Programmablauf
- werden zur Laufzeit überprüft
- dienen zur Unterbrechung des normalen Kontrollflusses

Typische Verwendung einer Exception:

- Ein Problem tritt auf
- Normales Fortfahren nicht möglich
- Lokale Reaktion darauf nicht sinnvoll / möglich
- Behandlung des Problems **an anderer Stelle** erforderlich

Exceptions in Java (I)

Exceptions in Java sind Objekte

- Enthalten Information zum aufgetretenen Fehler
- Erzeugung mit **new**
- Auslösen mittels **throw**

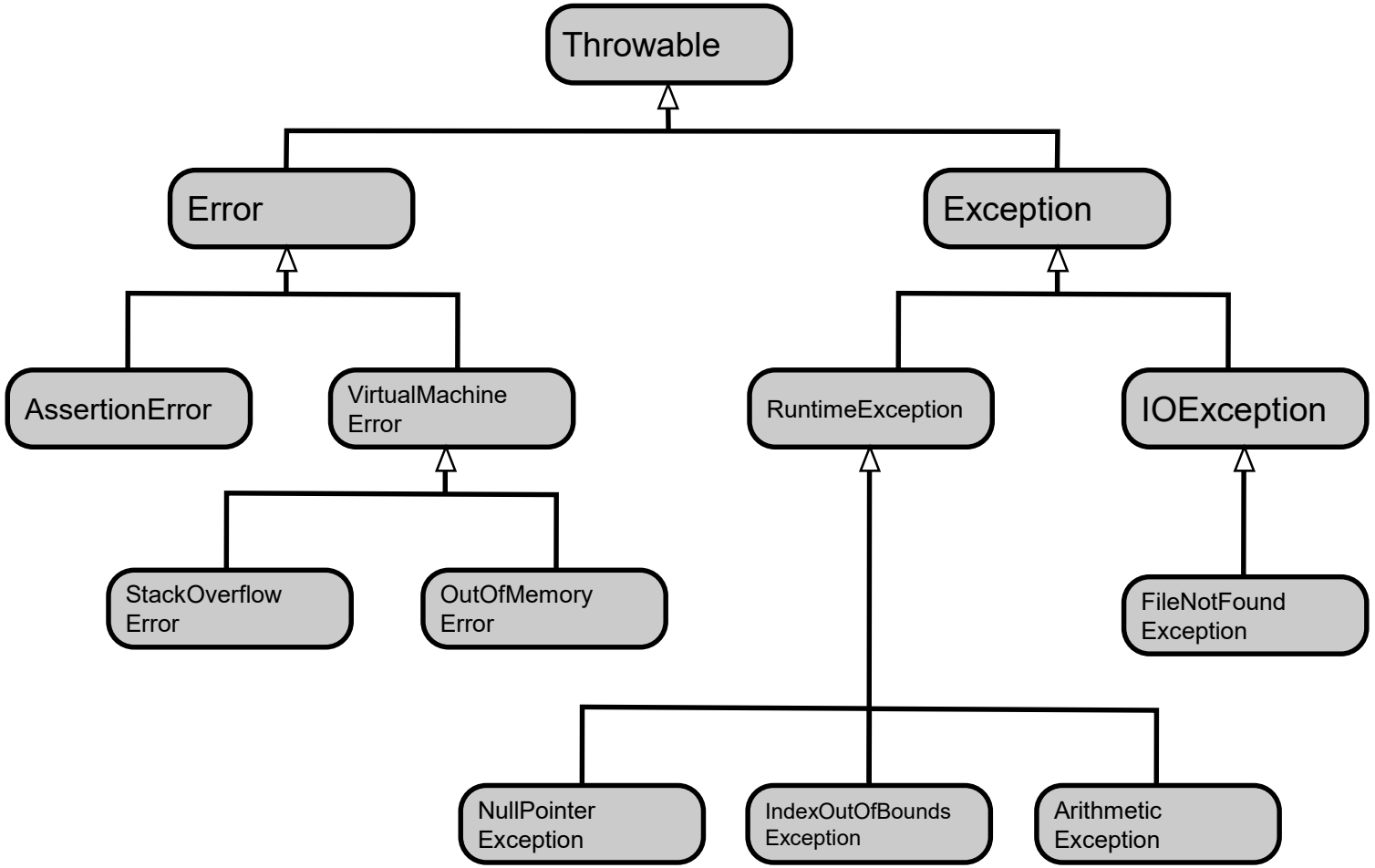
Beispiel

```
...
public void setMonth(int month) {
  if (month < 1 || month > 12) {
    throw new IllegalArgumentException("wrong month: " + month);
  }
  this.month = month;
}
...
```

Exceptions in Java (II)

- Ausnahmen sind von der Klasse **`Exception`** abgeleitet
  - Java stellt bereits Exception-Klassen zur Verfügung
  - Auch eigene Exception-Klassen sind möglich
- Eigene Exceptions:
  - Von der Exception-Klasse abgeleitet
  - Mindestens Default-Konstruktor
  - Konstruktor mit einem String-Parameter
  - ➜ Zusätzliche Information zu den Fehlerumständen
- Wichtige Methoden der Exception-Klasse:
  - `String getMessage()` — Nachricht der Ausnahme
  - `void printStackTrace()` — Ausgabe des Aufrufstacks
  - `StackTraceElement[] getStackTrace()`
- Außerdem: `Error` mit Unterklassen
  - Für schwerwiegende Fehler, keine sinnvolle Behandlung möglich

Auszug aus der Exception-Hierarchie



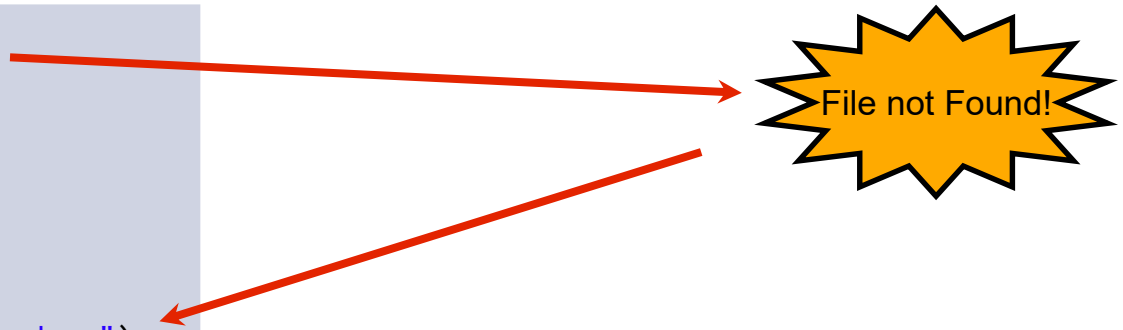
Ausnahmebehandlung in Java

- Fehlerbehandlung in Java mittels **`try-catch`**-Blöcken:

```
try {
...  // hier könnte eine Exception auftreten
} catch (ExceptionType1 e) {
...  // Fehlerbehandlung für Ausnahmen vom Typ ExceptionType1
} catch (ExceptionType2 e) {
...  // Fehlerbehandlung für Ausnahmen vom Typ ExceptionType2
}
```

Beispiel

```
try {
  FileReader fr = new FileReader("test.txt");
  int nextChar = fr.read();
  while (nextChar != -1) {
    ... // verarbeite Zeichen
    nextChar = fr.read();
  }
} catch (FileNotFoundException e) {
  System.out.println("Datei test.txt nicht gefunden.");
} catch (IOException e) {
  System.out.println("Ein-/Ausgabefehler in read().");
}
```



- Geworfene Exception löst Sprung zum **`catch`**-Block aus

Kontrollfluss bei Exceptions

- Bei einer ausgelösten Exception wird der Programmfluss unterbrochen
  - Suche im Aufrufstack nach umgebenden **`try-catch`**-Blöcken
  - Erster **`catch`**-Block passt, für den gilt: Deklarierter Typ ist Obertyp des Ausnahmetyps
  - Nach Behandlung (Abarbeitung des passenden **`catch`**-Blocks) Fortsetzung direkt nach dem Ende des **`try-catch`**-Blocks
- Beispiel:

```
void foo() throws IOException {
  ...
  if (...) {
  throw new IOException();
  }
  ...
}

void bar() throws IOException {
  ...
  foo();
  ...
}
```

```
void baz() {
  try {
  ...
  bar();
  ...
  } catch (IOException e) {
  ... // Fehlerbehandlung
  }
}
```

Ausnahmebehandlung

- **Exception Handler (= `catch`-Block)**
  - Behandlung einer Ausnahme an einer Stelle irgendwo im Aufrufstack
  - Getrennt vom normalen Programmcode
- ***Catch or Specify***
  - Jede ausgelöste Exception muss
    - **behandelt** (→ Exception Handler) oder
    - **deklariert** (→ **`throws`**-Klausel)

  werden.

Deklarieren von Ausnahmen

Nicht behandelte Exceptions müssen im Methodenkopf deklariert werden

```
String readFile(String fileName) throws IOException, FileNotFoundException {
  FileReader fr = new FileReader(fileName);
  String content = "";
  int nextChar = fr.read();
  while (nextChar != -1) {
    ... // Abarbeitung der gesamten Datei
  }
  return content;
}
```

Bedeutung:

- Aufrufer der Methode muss sich um Behandlung der Exception kümmern
- **`throws`**-Klausel ist Teil der Methoden-Deklaration (zusätzlich zu Name und Typen der Parameter)

Nicht deklarationspflichtig:

- `RuntimeException` und
- `Error` (jeweils mit Unterklassen)

Anmerkung: Da `IOException` Oberklasse von `FileNotFoundException` ist, müsste letztere nicht deklariert werden (→ trotzdem machen zwecks Dokumentation)

Error und Exception

`Error`:

- Ernsthafte Probleme, die nicht vernünftig behandelt werden können, z.B.
  - illegaler Bytecode
  - JVM-Fehler, ...

`RuntimeException`:

- Programmierfehler, die von der JVM erst zur Laufzeit festgestellt werden können, und meist kein sinnvolles Fortsetzen des Programms erlauben, z.B.:
  - Division durch Null (`ArithmeticException`)
  - Zugriff auf ein **`null`**-Objekt (`NullPointerException`)
  - Zugriff auf unzulässiges Arrayelement (`IndexOutOfBoundsException`)

Sonstige Exceptions:

- Behandelbare Programmfehler
  - Datei nicht vorhanden, Festplatte voll, Fehler beim Parsen, ...

Behandeln von Ausnahmen

Welche Ausnahme-Typen sollten behandelt werden?

- `Error` und Unterklassen: **Nein** (per Definition nicht sinnvoll)
- `Exception`-Klasse direkt: **Niemals!** (viel zu allgemein)
- `RuntimeException` und Unterklassen: Im Allgemeinen: **Nein**
  - Programmierfehler beheben, nicht behandeln!
  - (Ausnahme: `NumberFormatException`)
- Unterklassen von `Exception`: **Ja!**

Werfen von Ausnahmen

Welche Ausnahme-Typen dürfen ausgelöst werden?

- `Error`: **Ja**, ggf. mit eigener Unterklasse
- `Exception`: Nur als eigene Unterklasse
- `RuntimeException`: **Ja**, mit eigener oder semantisch passender Unterklasse

Beispiel

```
if (month < 1 || month > 12) {
  throw new IllegalArgumentException("ungültiger Monat");

  switch (month) {
    case 1:  ...; break;
    case 2:  ...; break;
    ...
    case 12: ...;
  }
}
```

Eigene Exceptions

Definition eigener Exceptions:

- Ableiten einer eigenen Unterklasse von
  - `Exception` oder
  - `RuntimeException`
- Beispiel für Shapes: `OverlappingShapesException`
- Implementierung mindestens der zwei Standard-Konstruktoren
- Definition einer eigenen, sinnvollen Exception-Hierarchie bei Bedarf
- Verwendung der vorhandenen Exceptions nur für dafür vorgesehene Zwecke (siehe Java-API)
  - `IllegalArgumentException`
  - `IllegalStateException`
  - `UnsupportedOperationException`
  - `NullPointerException`

Verwendung von Exceptions

- **Exceptions sollen:**
  - zur Vereinfachung dienen
  - die absolute Ausnahme darstellen
  - mittels `@throws` im Javadoc-Kommentar beschrieben werden
    - alle deklarierten Exceptions
    - auch alle nicht deklarierten RuntimeExceptions
  - **nicht** den normalen Kontrollfluss steuern!

- **Sehr schlecht**:

```
try {
  while (character != array[i]) { i++; }
} catch (ArrayIndexOutBoundsException e) {
  System.out.println("Element nicht gefunden.");
}
```

Faustregeln (I)

- **Verboten**: `try`-Block um das ganze Programm

```
try {
  ...  // hier steht das gesamte Programm
  ...
  ...
  ...
  ...
  ...
} catch (SomeException e) {
...
} catch (OtherException e) {
...
} catch (ThirdException e) {
...
}
```

- **Ausnahmen: Keine**

Faustregeln (II)

- **Verboten**: Leerer `catch`-Block

```
try {
   ...
} catch (SomeException e) { }
```

- **Ausnahmen: Keine**

Faustregeln (III)

- **Verboten**: Explizites Fangen des Typs `Exception`

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

- **Ausnahmen: Keine**

Faustregeln (IV)

- **Verboten:** Explizites Fangen des Typs `Throwable`

```
try {
  ...
} catch (Throwable e) {
  ...
}
```

- **Ausnahmen: Absolut keine!**

Frühe Fehlererkennung („fail fast“)

Möglichst frühe Erkennung eines Fehlers

- Kleinerer Suchbereich beim Debugging

Abbruch/Fehlermeldung bei inkonsistentem Programmzustand

Häufig auftretende Fälle:

- `null`-Werte in Attributen
- negative Zahlen (z.B. für Größenangaben)

Defensive Programmierung

- Anwender einer Klasse versuchen, diese (absichtlich oder unabsichtlich) zu missbrauchen
- Verhinderung durch Implementierung

Zusätzliche Abfragen im Programmcode!

Schlechtes Beispiel: `java.util.Properties`

```
// A Properties object maps Strings to Strings
public class Properties extends Hashtable {
  public Object put(Object key, Object value);

  // Throws ClassCastException if this Properties
  // contains any key or value that is not a String
  public void store(OutputStream out, String header);
}
```

- `put`-Methode verlangt keine `String`-Argumente, `Objects` genügen
- Fehler (`ClassCastException`) wird aber erst bei `store` gemeldet

Zusammenfassung Exceptions

Ausnahmen (Exceptions)

- werden ausgelöst (→ **`throw`**) und behandelt (→ **`try/catch`**) oder deklariert (→ **`throws`**).
- sollen die Ausnahme bleiben.
- trennen sauber Programmlogik und Fehlerbehandlung.

Fehlererkennung

- so früh wie möglich
- defensiv
- mittels **`assert`** (Foliensatz 13 zu Testen) oder **`if`** + Exceptions

Literaturhinweis - Weiterlesen

- Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger „Grundkurs Programmieren in Java“, 7. Auflage, 2014 (mit Java 8), Hansa-Verlag
  - „Exceptions and Errors“