Programmierung II

Bernd Müller

Fakultät Informatik
Ostfalia
Hochschule Braunschweig/Wolfenbüttel

Sommersemester 2014

Vorwort

Zur Veranstaltung

- Name: Programmieren
- Fortsetzung von *Grundlagen des Programmieren*
- Paralleler Besuch der Veranstaltung *Algorithmen und Datenstrukturen*

Materialien

- Dieses Skript
- Das Buch *Sprechen Sie Java?* von Hanspeter Mössenböck. Anschaffung dringend empfohlen: Im Skript steht *nicht* alles!
- Die Kapitelnummerierung des Skripts entspricht diesem Buch, die Inhalte – mit Erweiterungen und Änderungen – auch
- Erweiterungskapitel sind mit *A* gekennzeichnet (Addendum)

Materialien (cont'd)

- Alternativ praktisch jedes Buch über Programmierung und Java
- Ein *sehr* ausführliches Java-Buch gibt es auch online: Java ist auch eine Insel. Hat aber über 1300 Seiten und geht weit über ein einführendes Lehrbuch hinaus. Evtl. aber für Sie als Nachschlagewerk geeignet.
- Ebenfalls online sind die Java Tutorials von Sun
- Sehr gute englische Bücher sind *Core Java – Fundamentals* und *Core Java – Advanced Features*
- Ein weiterführendes, aber *sehr* zu empfehlendes Buch ist *Effective Java*

Übungen

- Das Buch von Herrn Mössenböck enthält am Ende eines jeden Kapitels Übungsaufgaben, die Sie unbedingt bearbeiten sollten
- Ich verteile (ca.) 14-tägig Aufgabenblätter, die Sie ebenfalls bearbeiten und die Lösungen abgeben (Details in Vorlesung)
- Es finden betreute Übungen statt (Details in Vorlesung)
- Ihre Abgaben werden mit Punkten versehen, die zu 30% in die Gesamtnote eingehen. 70% der Punkte können durch die Klausur erzielt werden

Generizität

Parametrisierung von Klassen

- Gibt es seit Java 5
- Ziel: Klasse, Interface, Methode soll parametrisiert werden
- Bisher `Object` verwendet

```
class List {
  void add(Object obj) { ... }
  Object remove() { ... }
}
```

Parametrisierung von Klassen (cont'd)

- In eine solche Liste kann man „alles“ reinschreiben
- Beim Lesen muss man casten

```
List list = new List();
list.add("Ein String");
list.add(new Person());
...
Person person = (Person) list.remove();
```

- Problem: man kann keine homogenen Listen erzwingen, also z.B. ausschließlich Strings oder ausschließlich Personen

Generische Typen

Klassen oder Interfaces mit Typparametern

- Typ wird als Parameter der Klasse in spitzen Klammern angegeben

```
class List<T> {
  T[] data = ...;
  void add(T o) { ... }
  T remove() { ...}
}
```

- T ist formaler Parameter, der durch konkreten Typ ersetzt wird

```
List<String> list = new List<String>();
list.add(new Person()); // Fehler
String s = list.remove(); // kein Casten notwendig
```

- Einschränkung: Nur Referenztypen erlaubt

Warum macht man das Ganze?

- Es ist zwar komplizierter, aber
- Compiler kann mehr Fehler erkennen
- Man kennt den Typ (einfacher zu verstehen)
- Man erspart sich das Casten

Keine Einschränkung auf *einen* Typparameter

- Klassen können beliebig viele Typparameter haben
- Beispiel: Map (Abbildung von Schlüsseln auf Werte)

```
class Map<K, V> {
  void put(K key, V value) { ... }
  V remove(K key) { ... }
}
```

- Verwendung:

```
Map<String, Person> persons =
    new Map<String, Person>();
persons.put("Meier", aPerson);
bPerson = persons.remove("Meier");
```

Generische Collections

- Seit Java 5 sind die Collections generisch:
  - `List<E>`
  - `Set<E>`
  - `Map<K, V>`
  - ...
- Beachte Schreibung und Namenswahl

Der Diamond-Operator in Java 7

- Die Typangaben zur Objekterzeugung sind redundant

```
Map<String, Person> persons =
    new Map<String, Person>();
```

- Seit Java 7 kann man diese Typangabe weglassen
- Man spricht vom Diamond-Operator

```
Map<String, Person> persons = new Map<>();
```

- Achtung: Dies ist ein Beispiel. `Map` ist ein tatsächlich existierendes Interface

Rohtypen und Zuweisungskompatibilität

- Generische Typen wurden mit Java 5 eingeführt
- Dabei wurde die Syntax (Sprache) geändert
- Die JVM und das Byte-Code-Format wurden aber nicht geändert (Stichwort Kompatibilität!)
- Generische Klassen werden im Byte-Code also auf nicht-generische Klassen abgebildet
- Die Typinformation wird also gelöscht (Type Erasure)
- Und der Compiler muss zur Compile-Zeit garantieren, dass zur Laufzeit nichts schief gehen kann

Rohtypen und Zuweisungskompatibilität (cont'd)

- Generische Typen werden auf ihren *Rohtyp* abgebildet
- Z.B. `ArrayList<T>` auf `ArrayList`
- Daher ist diese Anweisung zulässig

```
ArrayList rawArrayList = new ArrayList<String>();
```

- Da dabei aber einiges schief gehen kann, erzeugt der Compiler eine Warnung
- Die können Sie mit der Annotation `@SuppressWarnings` ausschalten
- Demo !

Aber Vorsicht . . .

- Sie programmieren jetzt aber wieder gegen eine nicht getypte Liste, so dass Sie alles reinstecken können:

```
ArrayList rawArrayList = new ArrayList<String>();
rawArrayList.add("String");
rawArrayList.add(new Integer(27));
System.out.println(rawArrayList.get(0).getClass());
System.out.println(rawArrayList.get(1).getClass());
```

Generische Typen und Arrays

- Generische Arrays nicht erlaubt
- Zwar ist die folgende Deklaration ok

```
class List<T> {
  T[] data1;
  ...
}
```

- Aber das Erzeugen des Arrays ergibt einen Compiler-Fehler

```
class List<T> {
  T[] data = new T[100]; // Compiler-Fehler
}
```

- Meldung Eclipse: Cannot create a generic array of T
- Meldung javac: generic array creation

Generische Typen und Arrays (cont'd)

- Warum?
- Wie gesehen gibt es die Typinformation zur Laufzeit nicht
- Es würde also ein `Object[100]` Array erzeugt werden
- Da kann man allerdings alles reinschreiben (nicht nur `T`) und die JVM kann es nicht merken
- Man muss dies explizit machen und casten:

```
class List<T> {
  T[] data = (T[]) new Object[100];
}
```

- Man hat dann immer noch dasselbe Problem, aber es ist explizit gemacht und verursacht eine Warnung (unchecked cast)

Generische Typen und Arrays (cont'd)

- Auch Arrays generischer Typen verboten
- Also Raw Type nehmen, casten und Warnung bekommen

```
ArrayList<String>[] array =
    new ArrayList<String>[10];   // Compiler-Fehler
ArrayList<String>[] array =
    (ArrayList<String>[]) new ArrayList[10];
```

Eingeschränkte Typparameter

Motivation eingeschränkter Typparameter

- Will man wissen, welche Methoden ein Typparameter unterstützt (z.B. um eine solche Methode aufzurufen), muss man den Typparameter *einschränken*
- Eine solche Einschränkung/Begrenzung (engl. Bound) wird über das bereits bekannte Schlüsselwort `extends` realisiert

```
public class SortedList<T extends Comparable<T>> {

  private T[] data;
  ...
}
```

Motivation eingeschränkter Typparameter (cont'd)

- `Comparable<T>` ist das „Vergleichbarkeits-Interface“ mit der einzigen Methode `compareTo()`
- Syntaktisch etwas besser lesbar ist die Rohtyp-Variante, erzeugt aber natürlich die entsprechende Warnung

```
public class SortedList<T extends Comparable> {

  private T[] data;
  ...
}
```

Typparameter durch mehrere Typen eingeschränkt

- Man kann den Typparameter durch mehrer Typen einschränken
- Als Trenner wird das & verwendet, da das Komma schon als Parametertrenner verwendet wird

```
public class SortedList
     <T extends Comparable<T> & Serializable> {
     ...
```

- Das `Serializable`-Interface ist ein sogenanntes Marker-Interface ohne Methoden (wird später erläutert) und nicht typparametrisiert

Generizität und Vererbung

Generizität und Vererbung

- Sie werden relativ selten generische Klassen definieren
- Noch viel seltener werden Sie generische Klassen definieren, die von generischen Klassen erben
- Den Abschnitt 15.3 lesen Sie daher bei Interesse im Buch nach
- Es gibt jedoch etwas, was Sie sich merken sollten: Das Substitutionsprinzip für einfache Typen erweitert sich nicht auf deren generische Variante:

```
Number number = new Integer(1); // ok
ArrayList<Number> numbers =
    new ArrayList<Integer>(); // Fehler
```

- Generische Typen sind mit keinem anderen außer dem Rohtyp kompatibel

Wildcard-Typen

Wildcard-Typen

- Die Nichtkompatibilität verschiedener generischer Typen kann über Wildcard-Typen aufgehoben werden: Ein Wildcard-Typ ist zu allen anderen Varianten dieses Typs kompatibel
- Wildcard-Typen werden über das Fragezeichen realisiert

```
ArrayList<?> numbers = new ArrayList<Number>();
             numbers = new ArrayList<Integer>();
```

Eingeschränkte Wildcard-Typen

- Wildcard-Typen können auch eingeschränkt werden

```
ArrayList<? extends Number> numbers;
numbers = new ArrayList<Number>();   // ok
numbers = new ArrayList<Integer>();  // ok
numbers = new ArrayList<Long>();     // ok
```

- Da List-Typ nicht bekannt, add()-Aufruf nicht erlaubt, lesen (löschen) schon

```
numbers.add(new Integer(1));          // Fehler
Number number = numbers.remove(0);    // ok
```

Beispiel eingeschränkte Wildcard-Typen

```
void drawAll(Collection<? extends Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

- Die Iterationsvariable `s` ist ein `Shape`
- Daher kann die Methode `draw()` aufgerufen werden

Generische Methoden

Generische Methoden

- Auch Methoden können mit Typen parametrisiert werden
- Damit Methode für unterschiedliche Typen anzuwenden
- Typparameter vor Rückgabetyp bzw. void

```
<T extends Comparable<T>> T max(T x, T y) {
  if (x.compareTo(y) > 0) {
    return x;
  } else {
    return y;
  }
}

// Dann moeglich:
String s = max("John", "Lisa");
int m = max(5, 3); // Auto-Boxing
```

Weiteres Beispiel

- Mittleres Array-Element

```
public class GenericMethod {

  public static <T> T getMiddle(T[] a) {
    return a[a.length / 2];
  }

  public static void main(String[] args) {
    String[] numbers = { "eins", "zwei", "drei" };
    String middle =
        GenericMethod.getMiddle(numbers);
    ...
  }
}
```

Javas Collections

Status Quo

- Seit Java 1.2 ist im SDK das Collection-Framework als Bibliothek enthalten, die gute und umfangreiche Collections bereit stellt
- Alles im Package `java.util`
- Seit Java 5 sind alle Collections generisch
- Prinzip: Trennung von Schnittstelle und Implementierung
- Zentral: Die Interfaces `Collection` und `Iterator`
- (Im Buch Abschnitt 22.1)

Grundlegender Mechanismus

- Interface Collection mit zwei zentralen Methoden

```
public interface Collection<E> {
   boolean add(E element);
   Iterator<E> iterator();
   ...
}
```

- Damit praktisch alles machbar, weil per Iterator gelöscht werden kann

Interface Iterator

- Machbar mit Hilfe des Iterators

```
public interface Iterator<E> {
  E next();
  boolean hasNext();
  void remove();
}
```

Bearbeiten von Collection-Elementen

```
Collection<MeineKlasse> c = ...;
Iterator<MeineKlasse> iter = c.iterator();
while (iter.hasNext() ) {
  tuWasMit(iter.next());
}
```

- Eleganter mit For-Each, 1. Semester, Sie erinnern sich ;-)

```
for (MeineKlasse elem : c) {
  tuWasMit(elem);
}
```

Weitere wichtige Methoden aller Collections

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

Welche (konkreten) Collections gibt es?

`ArrayList`	Indizierte Sequenz, die dynamisch wächst und schrumpft
`LinkedList`	Geordnete Sequenz, die effizientes Einfügen und Löschen an beliebigen Stellen erlaubt
`ArrayDeque`	Queue, die an beiden Enden Einfügen/Löschen erlaubt
`HashSet`	Nicht geordnete Collection ohne Duplikate
`TreeSet`	Geordnete Menge
`EnumSet`	Menge eines Aufzählungstyps
`LinkedHashSet`	Menge, die Einfügeordnung beibehält
`PriorityQueue`	Collection, die effizientes Löschen des kleinsten Elements erlaubt

Welche (konkreten) Collections gibt es? (cont'd)

`HashMap`	Datenstruktur für Schlüssel/Wert-Beziehungen
`TreeMap`	Eine Map mit sortierten Schlüsseln
`EnumMap`	Eine Map mit Schlüsseln eines Enums
`LinkedHashMap`	Map, die Einfügeordnung beibehält
`WeakHashMap`	Map, deren Elemente garbage-kollektbar sind, falls nicht anderweitig verwendet
`IdentityHashMap`	Map, deren Schlüssel mit == und nicht mit `equals()` verglichen werden
`...`	`...`

Schauen Sie selbst in `java.util` rein !

Listen

Listen

- Interface `List<E>`
- Geordnete Collection, also mit Index
- Möglichkeiten zum allgemeinen Einfügen und Löschen, aber auch Einfügen und Löschen mit Index
- Implementierungen:
  - `ArrayList`, über internes Array
  - `LinkedList`, über verzeigerte Liste (siehe Algo & Datenstr.)

Was ist daran falsch?

```
List<Integer> list = new ArrayList<Integer>();
list.add(127);
list.add(5912);
list.add(127);
System.out.println(list.size());
list.remove(5912);
System.out.println(list.size());
```

- Merke: Lesen Sie das API-Doc !!!
- `remove()` ist überladen und gibt es für `int` und `Object`
- Im Beispiel wird also das Element an der 5912. Stelle gelöscht
- Richtig wäre gewesen

```
list.remove(new Integer(5912));
```

Womit wir beim Thema wären !

```
List<Integer> list = new ArrayList<Integer>();
Integer i1 = new Integer(17);
Integer i2 = new Integer(17);
System.out.println("i1 == i2: " + (i1 == i2));
System.out.println("i1.equals(i2): "
                   + i1.equals(i2));
list.add(i1);
list.remove(i2);
System.out.println(list.size());
```

- Was ist die Ausgabe?

Beispiele ...

```
List<Integer> list = new LinkedList<Integer>();
list.add(127);
list.add(5912);
list.add(127);
list.add(list.get(1));
System.out.println(list.size());
Integer[] array = list.toArray(new Integer[]{});
for (int i = 0; i < array.length; i++) {
  System.out.print(array[i] + "  ");
}
```

Beispiele ...

```
List<String> list = new ArrayList<String>();
list.add("Dies");
list.add("ist");
list.add("ein");
list.add("Satz");
list.add(3, "laengerer");
StringBuffer buffer = new StringBuffer();
for (String string : list) {
  buffer.append(string + " ");
}
System.out.println(buffer.toString().trim());
```

Beispiele ...

```
List<String> list = new ArrayList<String>();
list.add("Dies");
list.add("ist");
list.add("ein");
list.add("ein");
list.add("Satz");
list.add(4, "laengerer");
list.remove("ein");
StringBuffer buffer = new StringBuffer();
for (String string : list) {
  buffer.append(string + " ");
}
System.out.println(buffer.toString().trim());
```

Mengen

Interface Set

- Abstraktion der mathematischen Menge:
  - Keine Duplikate
  - Keine Ordnung
- Duplikaterkennung über `equals()`, also Vorsicht bei Objektänderungen, die `equals()` beeinflussen ! (nächstes Kapitel)
- Ob `null` erlaubt ist oder nicht, ist implementierungsabhängig

Beispiele

```
Set<String> set = new HashSet<String>();
set.add("eins");
set.add("eins");
set.add("zwei");
set.add("zwei");
System.out.println(set.size()); // 2
```

Implementierungen

- `HashSet`, gerade gesehen
  - Erlaubt `null`
  - Garantiert keine Reihenfolge bei Iteration
- `TreeSet`
  - $log(n)$ Aufwand für `add()`, `remove()`, `contains()`
  - Geordnet nach natürlicher Ordnung oder Comparator

Beispiele

```
Set<String> set = new TreeSet<String>();
set.add("Gerndt"); set.add("Weimar");
set.add("Bikker"); set.add("Kreyssig");
set.add("Schiering"); set.add("Jensen");
set.add("Mengersen"); set.add("Pekrun");
set.add("Mueller"); set.add("Lie");
set.add("Seutter"); set.add("Riegler");
set.add("Gharaei"); set.add("Justen");
set.add("Klages"); set.add("Klawonn");
for (String prof : set) {    // sortierte
  System.out.println(prof); // Ausgabe
}
```

Maps

Interface `Map`

- Speichert einen Wert unter einem Schlüssel ab
- Man muss also Schlüssel kennen, um Wert wiederzufinden
- Beispiele:
  - Personalausweisnummer
  - Kundennummer
  - Versicherungsscheinnummer
  - ...
- Standardimplementierungen:
  - `HashMap` (konstanter Zugriff, da Hash)
  - `TreeMap` (sortiert)

Beispiel Map

```
Map<String, String> plzs =
          new HashMap<String, String>();
plzs.put("38302", "Wolfenbuettel");
plzs.put("38102", "Braunschweig");
plzs.put("30173", "Hannover");
plzs.put("38226", "Salzgitter");

System.out.println(plzs.get("38302"));

for (String plz : plzs.values()) {
  System.out.println(plz);
}
```

Die Klassen Collections und Arrays

Die Klassen Collections und Arrays

- Besitzen Methoden für
  - Sortieren
  - Suchen
  - Kopieren
- Die Sie sich selbst ansehen

Aufgabe

- Wandeln Sie
  - Ein Array in eine Liste
  - Eine Liste in ein Array
  - Eine Liste in eine Menge
  - Eine Menge in eine Liste
- Sortieren Sie
  - Eine Liste
  - Eine Array

Sortieren

Wie konnten die oben verwendeten Methoden sortieren?

- Viele Klassen implementieren das Interface `java.lang.Comparable<T>`
- Die Sortiermethoden setzen dies voraus
- Es besteht aus der einzigen Methode `int compareTo(T other)`
- Diese liefert eine Zahl kleiner 0, 0, oder eine Zahl größer 0 zurück, falls das andere Objekt kleiner, gleich oder größer ist
- Überzeugen Sie sich davon, dass sehr viele SDK-Klassen dieses Interface implementieren

Der Comparator

- Problematisch bei `Comparable` ist, dass eine Klasse nur einmal eine solche Methode implementieren kann
- Was machen, wenn Kunden mal
  - alphabetisch nach Vornamen
  - alphabetisch nach Nachnamen
  - nach Umsatz
  - nach Wohnort
  - ...

  sortiert werden sollen?
- Man verwendet das Interface `java.util.Comparator<T>`

Das Interface Comparator<T>

```
public Interface Comparator<T> {

   int compare(T a, T b);

}
```

- Auch mit kleiner 0, 0, größer 0

Beispiel

```
class StringVergleicher
      implements Comparator<String> {

  @Override
  public int compare(String o1, String o2) {
    return o2.compareTo(o1);
  }

}
```

- Sortiert jetzt rückwärts

Verwendung (Sort-Methode überladen)

```
List<String> strings = new ArrayList<String>();
strings.add("eins");
strings.add("zwei");
strings.add("drei");
strings.add("vier");
strings.add("fuenf");
// alphabetisch sortiert:
Collections.sort(strings);
// anders rum sortiert:
Collections.sort(strings, new StringVergleicher());
```

Weiterführende Konzepte

Motivation

- Es gibt noch eine ganze Reihe von richtigen Spracheigenschaften und nicht wie in den letzten Kapiteln „einfache“ Bibliotheken
- ...

Kopieren und Clonen von Objekten

Kopieren eines Objekts, z.B. durch *Copy-Konstruktor*

- Idee: Konstruktor, der Kopie des Parameters zurück liefert

```
public class Kunde {
  private String vorname;
  private String nachname;
  private Adresse adresse;

  public Kunde(String vorname, String nachname, Adr
    this.vorname = vorname;
    this.nachname = nachname;
    this.adresse = adresse;
  }
  public Kunde(Kunde k) {
    if (k == null) {
      throw new AssertionError();
    } else {
      this.vorname = k.vorname;
      this.nachname = k.nachname;
      this.adresse = k.adresse;
    }
  }
```

Klasse Adresse nach Schema F

```
public class Adresse {

  private String strasse;
  private String ort;

  public Adresse(String strasse, String ort) {
    this.strasse = strasse;
    this.ort = ort;
  }

  ...
}
```

Verwendung

```
Kunde donald = new Kunde("Donald", "Duck",
        new Adresse("Am Teich 1", "Entenhausen"));
System.out.println(donald);
Kunde daisy = new Kunde(donald);
daisy.setVorname("Daisy");
System.out.println(daisy);
// Daisy laesst sich scheiden und zieht aus
daisy.getAdresse().setStrasse("Am Gaensesee 23");
System.out.println(daisy);
System.out.println(donald); // ups!
```

- Donald wohnt jetzt leider auch am Gänsesee 23

Copy-Konstruktor, Version 2

```
public Kunde(Kunde k) {
  if (k == null) {
    throw new AssertionError();
  } else {
    this.vorname = k.vorname;
    this.nachname = k.nachname;
    this.adresse = new Adresse(
                       k.adresse.getStrasse(),
                       k.adresse.getOrt());
  }
}
```

- Es wird ein neues Adressenobjekt angelegt

Copy-Konstruktor, Version 3

```
public Kunde(Kunde k) {
  if (k == null) {
    throw new AssertionError();
  } else {
    this.vorname = k.vorname;
    this.nachname = k.nachname;
    this.adresse = k.adresse.clone();
  }
}
```

- Was ist `clone()`?
- API-Doc

Clone-bare Adresse

```
public class Adresse implements Cloneable {
  private String strasse;
  private String ort;

  public Adresse clone() {
    try {
      return (Adresse) super.clone();
    } catch (CloneNotSupportedException e) {
      // sollte nicht passieren
      return null;
    }
  }
```

- Interface `Cloneable` ohne Methoden
- . . . classes that implement this interface should override `Object.clone()` (which is protected) with a public method. See `Object.clone()` for details on overriding this method.
- Seit Java 5 eigene Klasse statt `Object` erlaubt (covariant)

Korrektes Clonen

- Im Beispiel nur ok, weil Objekt-Properties immutable (`strasse`, `ort` sind Strings)
- Im Regelfall müssen *Sie* den zurückgegebenen Wert explizit kopieren

Objektidentität, Objektgleichheit

== und equals()

- Java legt Objekte auf dem Heap an (für unsere Betrachtungen)
- Zwei Objekte sind identisch (es gibt nur eins), wenn ihre Referenzen auf dasselbe Objekt verweisen
- Dies wird mit dem ==-Operator geprüft
- Die `equals()`-Methode der Klasse `Object` prüft, ob zwei Objekte gleich sind
- Sie macht dies durch Gleichheitstest der Referenzen
- Wenn Sie etwas anderes haben wollen, müssen Sie das selbst tun !

Gleichheit

- Häufig: Zwei Objekte sind gleich, wenn sie denselben Zustand haben
- Zustand: Wert der Properties
- Wenn Kunden nur aus Vor- und Nachname bestehen, sind zwei Kunden mit selbem Vor- und Nachnamen derselbe Kunde
- Das müssen Sie programmieren

Die equals()-Methode

```
public class Kunde {
  private String vorname;
  private String nachname;
  ...
  public boolean equals(Object obj) {
    // true, fall identisch
    if (this == obj) return true;
    // false, falls es das andere nicht gibt
    if (obj == null) return false;
    // false, falls andere Klasse
    if (getClass() != obj.getClass()) return false;
    // jetzt wissen wir, dass es ein Kunde ist
    Kunde other = (Kunde) obj;
    // identische Properties
    return vorname.equals(other.vorname)
        && nachname.equals(other.nachname);
  }
```

Die `equals()`-Methode (cont'd)

- Was bedeutet Gleichheit?
- Durch Nachdenken kommt man (mindestens) zu den Eigenschaften
  - reflexiv
  - symmetrisch
  - transitiv
- Neben diesen mathematisch motivierten Eigenschaften gibt es noch programmiertechnische, siehe API-Doc

Die `hashCode()`-Methode

- Ganz wichtig: Wenn Sie `equals()` überschreiben, müssen Sie auch `hashCode()` überschreiben !
- Wenn zwei Objekte gleich sind, müssen Sie denselben Hash-Code haben, siehe API-Doc, auch für weitere Bedingungen
- Sie können Eclipse `equals()` und `hashCode()` erstellen lassen.
- Das kann aber auch daneben gehen, z.B. Lazy-Loading bei OR-Mappern

Mengen

- Aus `java.util.Set`:

  A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

Beispiel Mengen

- Kunde mit `equals()` von oben

```
Kunde k1 = new Kunde("Bernd", "Mueller");
Kunde k2 = new Kunde("Bernd", "Mueller");
Kunde k3 = k2;

Set<Kunde> kunden = new HashSet<Kunde>();
kunden.add(k1);
kunden.add(k1);
kunden.add(k2);
kunden.add(k3);
kunden.size() // ???
```

Innere Klassen

Innere Klassen

- Klasse, die innerhalb einer anderen Klasse definiert ist
- Warum könnte man dies wollen?
  - Kann auf Daten der umschließenden Klasse zugreifen, auch wenn diese `private` sind
  - Innere Klassen können von anderen Klassen desselben Package versteckt werden
  - *Anonyme* innere Klassen können für Call-Backs sehr praktisch mit wenig zusätzlichem Code erstellt werden

Beispiel einer einfachen inneren Klasse

```
public class Outer {
  private String outer;
  private void foo() {
    System.out.println("foo() in Outer");
  }
  private class Inner {
    public void foo() {
      System.out.println("foo() in Inner");
      Outer.this.foo();
      System.out.println(Outer.this.outer);
    }
  }
}
```

- `Inner` hat Zugriff auf `Outer`, obwohl `private`
- `Outer.this` ist das Objekt, mit dem Instanz von `Inner` erzeugt wurde

Beispiel einer einfachen inneren Klasse (cont'd)

```
public class Outer {

  public static void main(String[] args) {
    Outer o = new Outer();
    //Outer.Inner i = new Outer.Inner(); // Fehler
    Outer.Inner i = o.new Inner();
    i.foo();
    Inner j = o.new Inner();
    j.foo();
  }
```

- Inner-Instanz kann nur für eine Outer-Instanz erzeugt werden

Lokale innere Klassen

- Begrenzung der Sichtbarkeit eines Klassennamens auf eine Methode

```
public class LocalInnerClass {
  public void foo() {
    class MyListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
          ...
      }
    }
    ActionListener myListener = new MyListener();
    JButton button = new JButton();
    button.addActionListener(myListener);
  }
}
```

Anonyme innere Klassen

- Wenn man den Namen nur einmal benötigt, kann man ihn auch ganz weg lassen

```
public void foo() {

  ActionListener myListener = new ActionListener() {

    public void actionPerformed(ActionEvent e) {
      // ...
    }
  };
  JButton button = new JButton();
  button.addActionListener(myListener);
}
```

- Wird *sehr* häufig in der Swing-Programmierung verwendet

Static innere Klassen

- Falls Instanzen der inneren Klasse keine Referenz auf das Objekt der äußeren Klasse haben muss, können Sie innere Klassen `static` deklarieren
- Man kann sie dann z.B. auch in static Methoden erzeugen

```
public class StaticInnerClass {

  public static void main(String[] args) {
    Inner inner = new Inner();
  }

  private static class Inner {
    ...
  }

}
```

Rekursion

Motivation

- Eine Methode, die sich selbst aufruft, nennt man *rekursiv*
- Eine Methode, die sich selbst indirekt aufruft, nennt man *indirekt rekursiv*
- Rekursion *kann* sehr viel einfacher und eleganter sein als Iteration
- Aus der Theoretischen Informatik weiß man, dass es für jeden iterativen Algorithmus einen semantisch gleichwertigen rekursiven Algorithmus gibt und umgekehrt
- Aus der Bekanntheit der Existenz hat man ihn allerdings noch lange nicht vorliegen ;-)

Beispiel Fakultät

- Iterative Definition:

$$n! = 1 * 2 * 3 * \ldots * (n-1) * n$$

- Rekursive Definition:

$$n! = \begin{cases} 1, \text{ für n} = 1 \\ \text{(n-1)! * n, für n} > 1 \end{cases}$$

Iterativ:

```
public static int fakultaet(int n) {
  int fakultaet = 1;
  for (int i = 1; i <= n; i++) {
    fakultaet *= i;
  }
  return fakultaet;
}
```

Und rekursiv:

```
public static int fakultaet(int n) {
  if (n == 1) {
    return 1;
  } else {
    return fakultaet(n -1) * n;
  }
}
```

Aufrufhiearchie und Stack

Demo!

Allgemeines Muster rekursiver Methoden

```
if (Problem klein genug)
        führe nichtrekursiven Zweig aus
else
        führe rekursiven Zweig mit kleinerem Problem aus
```

Schrittweise Verfeinerung

Generelles Problem der Software-Erstellung

- Kunde: „Ich hätte gerne ein Programm zur Unterstützung aller meiner betrieblichen Prozesse“
- Aufgaben:
  - Anforderungen erheben
  - Anforderungen umsetzen
- Ohne auf die Software-Entwicklungs- und Management-Prozesse einzugehen (das machen wir in der Veranstaltung Software-Technik) bleibt das Problem, eine Aufgabe (Menge von Anforderungen) in Software umzusetzen
- Dies ist in einem Schritt nicht möglich
- Außer manchmal bei unseren kleinen Beispielchen ;-)

Schrittweise Verfeinerung

- Gesamtaufgabe in Teilaufgaben zerlegen
- Jede Teilaufgabe wiederum in Teilaufgaben zerlegen, bis diese so einfach, dass man die Aufgabe direkt implementieren kann
- Jede Teilaufgabe wird zu einer Methode
- Zum Schluss Methodenaufrufe in richtiger Reihenfolge zusammensetzen, um Gesamtaufgabe zu lösen

Geschichte

- Die Methode der *Schrittweise Verfeinerung* geht auf Niklaus Wirth zurück und gehört zu den *Top-Down*-Ansätzen
- Wirth schrieb 1971 den Aufsatz *Program Development by Stepwise Refinement*, den Sie hier als PDF oder hier als HTML lesen können
- Niklaus Wirth gehört zu den großen Informatikern
- Er definierte unter anderem die Sprachen Pascal, Modula und Oberon, wofür er 1984 den Turing Award erhielt

Vorgehen bei der schrittweisen Verfeinerung

1. Zerlege die Gesamtaufgabe in Teilaufgaben
   - Nicht zu detaillierte Teilaufgaben
   - Beispiel Geldautomat: führe Benutzerdialog, lies Kontostand, gib Geld aus
   - Jede Teilaufgabe spezifiziert durch Methodenschnittstelle
2. Implementiere die Gesamtaufgabe mittels der Teilaufgaben
   - Annahme: Methoden für Teilaufgaben existieren
   - In richtiger Reihenfolge aufrufen, evtl. zusätzlich Abfagen und Schleifen
3. Implementiere die Teilaufgaben
   - Falls Teilaufgabe einfach genug: implementieren
   - Falls Teilaufgabe komplex: gehe zu Schritt 1 und zerlege in kleinere Teilaufgaben

Was gewinnen wir dabei ?

- Komplexität (etwas zu verstehen, zu programmieren, zu ...) nicht linear zur Größe, sondern überproportional
- Wenn wir die Größe eines Problems verkleinern, verkleinern wir die Komplexität wesentlich
- Das hilft uns, die Problemlösung überhaupt umzusetzen zu können

Beispiel Häufigkeitszählung

- Häufigkeitszählung von Wörtern in einer Datei
- Im Buch: Entwicklung einer Lösung, die einige (wenige) Klassen/Methoden des SDKs verwenden
- Sie arbeiten diese Lösung durch
- Wir entwickeln hier eine alternative Lösung, die andere Klassen/Methoden verwendet und *deutlich* kürzer ist
- Schrittweise Verfeinerung
  - Lese Wörter aus Datei
  - Zähle Wörter
  - Gib Worthäufigkeiten aus
- Wir lassen Exceptions (Kapitel 19) im Folgenden weg

```
public class WordCount {

  private Map<String, Integer> counter =
    new Hashtable<>();

  public void countWordsInFile(String filename) {
    String[] words = readWordsFromFile(filename);
    countWords(words);
    print();
  }
  ...
```

```
private String[] readWordsFromFile(String filename) {
  File file = new File(filename);
  Scanner scanner = new Scanner(file)
                          .useDelimiter("\\Z");
  String content = scanner.next();
  return content.split(" "); // Demo RegExp
}
```

oder (Demo Refactor Inline):

```
  private String[] readWordsFromFile(String filename)
    return new Scanner(new File(filename))
      .useDelimiter("\\Z").next().split(" ");
  }
```

```
private void countWords(String[] words) {
  for (String word : words) {
    if (counter.get(word) == null) {
      counter.put(word, 1);
    } else {
      counter.put(word, counter.get(word) + 1);
    }
  }
}
```

```
private void print() {
  for (String key : counter.keySet()) {
    System.out.println("Anzahl '" + key + "': "
                       + counter.get(key));
  }
}
```

Pakete

Motivation

- Anwendungen bestehen aus hunderten/tausenden von Klassen
- Klasse ungeeignet als einziges Strukturierungsmittel
- Klassen (und andere Dateien) werden zu Paketen (Packages) zusammengefasst
- Zusätzlich Definition von Sichtbarkeitsbereichen von Klassen
- Beispiele für Pakete im SDK
  - `java.lang`: Standardklassen
  - `java.io`: Ein-/Ausgabe
  - `java.util`: Kalender, Collections, Timer, ...
  - `javax.swing`: Oberflächenelemente
- Demo !

Anlegen von Paketen

Deklaration

- Erste Nichtkommentarzeile in Datei:
  `package meinpaket;`
- Name frei wählbar (aber Regeln, die später noch kommen)
- Alle Klassen der Datei sind in diesem Paket
- Sichtbarkeit: Alles, was zum Paket gehört, lokal im Paket und in anderen Paketen nicht sichtbar
- Fehlt die Package-Deklaration sind Klassen im Default-Paket (wie bisherigen Beispiele)
- Default-Paket verwenden Sie ab sofort nie mehr !

Export und Import von Namen

Regeln

- Regel 1: Was zu einem Paket gehört, ist außerhalb des Pakets unsichtbar
- Alle Klassen eines Pakets können aber gegenseitig auf ihre Variablen und Methoden zugreifen (solange nicht `private`)
- Resultat:
  - Namenskollisionen nur im eigenen Paket möglich. Auf fremde Pakete muss keine Rücksicht genommen werden (kann auch gar nicht)
  - Andere Klassen können die eigenen Klassen nicht sehen und damit nichts böses mit ihnen tun (versehentliches Überschreiben von Werten, ...)

Regeln (cont'd)

- Regel 2: Namen können von einem Paket exportiert werden
- Ein Name wird exportiert, indem er mit `public` versehen wird
- Exportiert werden können Klassen, Konstruktoren, Variablen, Methoden
- Exportierte Variable oder Methode in anderer Klasse nur sichtbar, wenn auch Klasse exportiert
- Demo !

Regeln (cont'd)

- Regel 3: Exportierte Klassen können in anderen Paketen importiert werden
- Entweder voll qualifizierten Namen verwenden oder
- Import durch `import`-Deklaration
- Zwei Arten:
  - Expliziter Import einer Klasse
  - Import aller Klassen eines Pakets (heute eher unüblich)
- Demo !

Statischer Import

- Statische Variablen und Methoden so importieren, dass sie nicht qualifiziert verwendet werden können
- Geht nur für Static Variablen und Methoden
- Beispiel:

```
import static java.lang.Math.PI;

public class Import {

  public static void main(String[] args) {
    System.out.println(PI);
  }

}
```

Pakete und Verzeichnisse

Regeln für Dateien und Verzeichnisse

- Eine `public` Klasse `Hugo` in Datei `Hugo.java` / `Hugo.class`
- Alle Klassen des Pakets `p` im Verzeichnis `p`
- Eine Datei darf beliebig viele Klassen enthalten, aber nur eine `public`
- Demo !

Pakethierarchie

- Pakete können hierarchisch aufgebaut sein
- Die Ebenen werden durch einen Punkt getrennt
- Beispiel:
  - `java.util`
  - `java.util.jar`
  - `java.util.concurrent`
  - `java.util.concurrent.atomic`

Weltweit eindeutige Paketnamen

- Software-Entwicklungsfirmen müssen unabhängig voneinander arbeiten können
- Daher Paketname als umgekehrter Internet-Domainname
- Beispiele
  - `org.omg` im API-Doc
  - `org.w3c` im API-Doc
  - `com.sun.tools` im Compiler
  - `sun.misc` System-Classloader (Ausnahmen bestätigen die Regel)

Information Hiding

Information Hiding / Geheimnisprinzip

- „*Verstecke die Implementierung komplexer Daten und erlaube den Zugriff ausschließlich über Methoden*“
- Achtung: Unterschied zum Buch !
- Warum Information Hiding ?
  - Interner Zugriff bedingt Kenntnis der komplexen Datenstrukturen. Das ist kompliziert und fehleranfällig
  - Direkt zugegriffene Datenstrukturen sind nur schwer änderbar, weil andere Programmteile dann nicht mehr funktionieren
- Also: Am besten gegen ein Interface programmieren
- Wir machen aber zuerst die allgemeinen Regeln für Sichtbarkeit

Sichtbarkeitsregeln

`public`	Name in allen Paketen sichtbar, die importieren und im deklarierenden Paket
`protected`	Name in allen Unterklassen sowie Klassen des deklarierenden Pakets sichtbar
Default (kein Schlüsselwort)	Name in allen Klassen des deklarierenden Pakets sichtbar
`private`	Name nur in deklarierender Klasse sichtbar

Sichtbarkeitsregeln zum Merken ;-)

Zugriff durch:	public	protected	package	private
definierende Klasse	ja	ja	ja	ja
Klasse desselben Package	ja	ja	ja	nein
Unterklasse eines anderen Package	ja	ja	nein	nein
Nicht-Unterklasse eines anderen Package	ja	nein	nein	nein

Abstrakte Datentypen und abstrakte Datenstrukturen

Abstract Data Type nach Sommerville

- *Software Engineering* von Sommerville ist das Standardwerk für Software Engineering

  „As well as using meaningful type names, the operations which are associated with a type should be packaged with the declaration to create an abstract data type. the abstract data type hides information about these operations and about the type representation. Operation implementation and the type representation may be changed without changing those parts of the program which use the abstract type. “

... without changing those parts of the program ...

- Wie kann das gehen?
- Indem man gegen eine Schnittstelle und nicht gegen eine Implementierung programmiert
- Also: Interface und implementierende Klasse

Beispiel: Komplexe Zahlen

```
public interface Complex {

  Complex plus(Complex b);
  Complex times(Complex b);

  double getRealPart();
  double getImaginaryPart();

}
```

Implementierung 1

```
public class ComplexImpl1 implements Complex {

  private final double re;
  private final double im;

  public ComplexImpl1(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public Complex plus(Complex b) {
    return new ComplexImpl1(
    re + b.getRealPart(),
    im + b.getImaginaryPart());
  }
```

Implementierung 1, Teil 2

```
    public Complex times(Complex b) {
      return new ComplexImpl1(
        re * b.getRealPart() - im * b.getImaginaryPart
        re * b.getImaginaryPart() + im * b.getRealPart
    }

    public String toString() {
      return "Complex(" + re + ", " + im + ")";
    }
    public double getRealPart() {
      return re;
    }
    public double getImaginaryPart() {
      return im;
    }
}
```

Implementierung 2

```
/**
 * Komplexe Zahlen mit Polarkoordinaten
 */
public class ComplexImpl2 implements Complex {

  private double r;
  private double theta;

  private ComplexImpl2() {
    r = 0.0;
    theta = 0.0;
  }

  public ComplexImpl2(double re, double im) {
    r = Math.sqrt(re * re + im * im);
    theta = Math.atan2(im, re);
  }
```

Implementierung 2, Teil 2

```
public Complex plus(Complex b) {
  return new ComplexImpl2(
      getRealPart() + b.getRealPart(),
      getImaginaryPart() + b.getImaginaryPart());
}
```

```
public Complex times(Complex b) {
  ComplexImpl2 product = new ComplexImpl2();
  product.r = r * ((ComplexImpl2) b).r;
  product.theta = theta + ((ComplexImpl2) b).theta;
  return product;
}
```

Implementierung 2, Teil 3

```
    public String toString() {
      return "Complex(" + getRealPart() + ", "
                        + getImaginaryPart() + ")";
      }

      public double getRealPart() {
        return r * Math.cos(theta);
      }

      public double getImaginaryPart() {
        return r * Math.sin(theta);
      }
  }
```

Verwendung

```
Complex a = new ComplexImpl1(3, 2);
Complex b = new ComplexImpl1(5, 5);
Complex c = a.plus(b);
System.out.println(c);
Complex d = a.times(b);
System.out.println(d);
```

- `ComplexImpl1` durch `ComplexImpl2` austauschbar *ohne weitere Änderung* !
- Javas Collections basieren vollständig auf diesem Prinzip (und viele andere Interfaces/Klassen auch)

Ausnahmebehandlung

Motivation

- Programme müssen mit Fehlern umgehen können
- Beispiele:
  - „Hugo“ oder 127 im Eingabefeld für Geburtstag
  - Zu öffnende Datei existiert nicht
  - Web-Server nicht erreichbar
  - ...

Fehler-Codes

Fehler-Codes

- Wurde früher gemacht (in C)
- Problem: Code-Struktur geht kaputt
- Noch größeres Problem: Keiner macht's

Konzepte der Ausnahmebehandlung

Was muss eine gute Fehlerbehandlung können?

- Beliebige Methode in Ruferkette solle Fehler behandeln können
- Meldung eines Fehlers nicht über Rückgabewert, um Fehlerbehandlung vom fehlerfreien Programmablauf zu trennen
- Sicherstellen, dass jeder mögliche Fehler behandelt wird
- Gemeldeter Fehler darf nicht ignoriert werden

Arten von Ausnahmen in Java

Realisierung in Java

- Geschützter Block:
  Anweisungsfolge kann gegen Auftreten von Fehlern geschützt werden (Schlüsselwort `try`)
- Exception-Handler:
  Geschützter Block hat einen oder mehrere Exception-Handler. Dieser ist auch nur eine einfache Anweisungsfolge
- Exception:
  Fehler wird durch Ausnahme signalisiert. Beim Auftreten wird nach passendem Exception-Handler gesucht und dieser ausgeführt. Anschließend Fortsetzung nach geschütztem Block

Syntax in Java

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

- Optional mehrere `catch` und `finally` möglich (später mehr)

Es gibt Ausnahmen verschiedener Art/Schwere

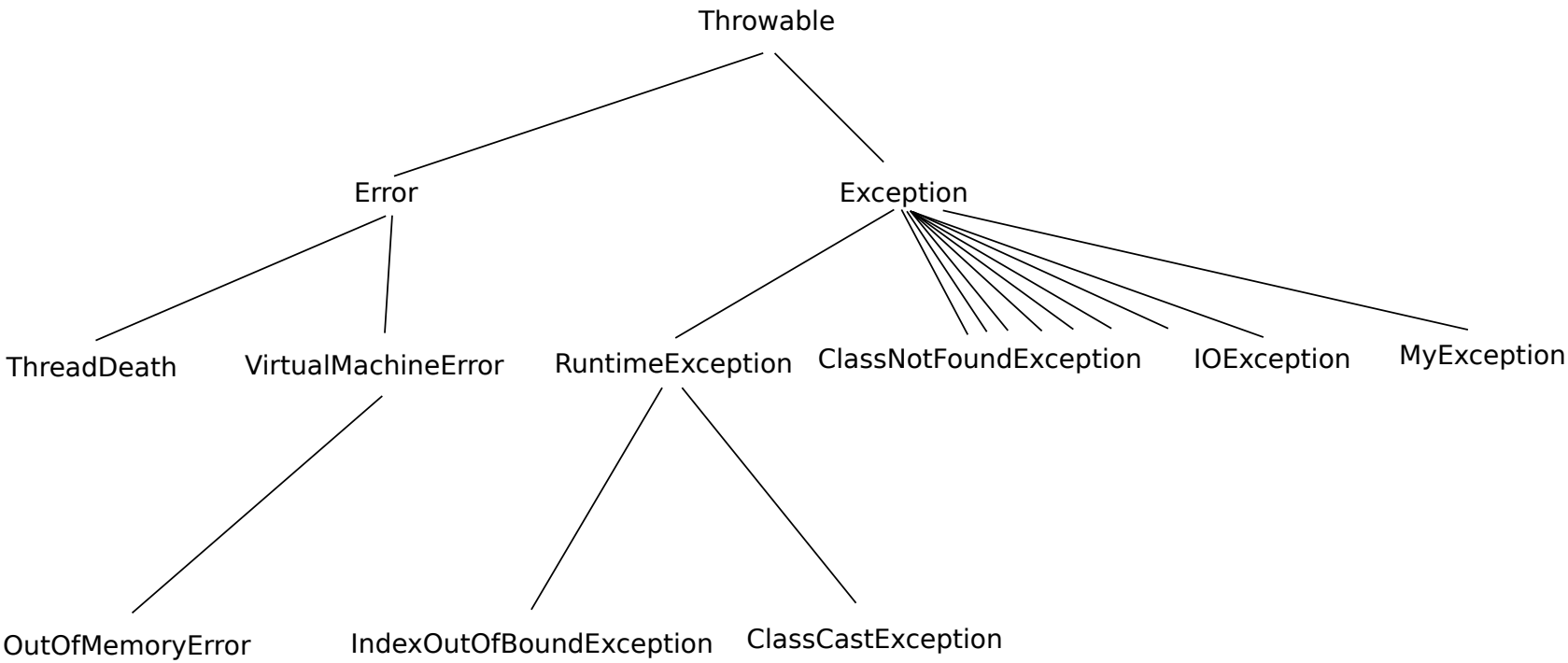
- Es wird unterschieden:
  - Checked Exceptions
  - Unchecked Exceptions
- Checked Exceptions durch Methodensignatur deklariert (`throws`-Klausel)
- Unchecked Exceptions ganz furchtbare Dinge, auf die man in der Regel nicht reagieren kann/soll (Klassen `RuntimeException` und `Error`, später mehr)

Forderung nach Behandlung

- Wir hatten gefordert, dass *jeder* mögliche Fehler behandelt wird
- Checked Exceptions müssen Sie behandeln, sonst lässt sich das Programm nicht compilieren
- Unchecked Exceptions müssen Sie nicht behandeln ! ?
- Nicht praktikabel, da Sie sonst *jedes* Statement überprüfen müssten
- Tritt eine Unchecked Exception auf, die nicht abgefangen wird, bricht Programm mit Fehlermeldung ab

Die Exception-Hierarchie

- Exceptions sind Objekte. Ihre Klassen bilden eine Vererbungshierarchie



Eigene Exceptions erben von Exception

```
public class Exception {
  public Exception();
  public Exception(String message);
  ...
  public String toString();
  void printStackTrace();
  ...
  }
```

- Verwendung:

```
catch(Exception e) {
  Log.error(e.getMessage());
  ...
  e.printStrackTrace();
  ...
}
```

Eigene Exception-Klassen

- Erben von `Exception`
- Ersten beiden Konstruktoren überschreiben

Ausnahmebehandler

besser

Exception-Handler

Exception-Handler

- `catch`-Klausel der `try-catch-finally`-Anweisung
- Parametrisiert mit Exception-Typ
- Mehrere `catch`-Klauseln möglich
- Die der Reihe nach ausprobiert werden
- Erste passende wird genommen, Rest ignoriert
- Falls Exceptions in Vererbungshierarchie also zuerst die Unterklassen

Beispiel Exception-Handler

```
try {
   ...
} catch(Exception1 e) {
   ...
} catch(Exception2 e) {
   ...
} catch(Exception3 e) {
   ...
} catch(Exception4 e) {
   ...
}
```

Auslösen einer Ausnahme

Werfen einer Exception

- Mit der `throw`-Anweisung
- Parameter ist Exception-Objekt
- Also typischerweise:
  `throw new Exception("da lief was falsch");`

Was passiert?

- Programmausführung wird abgebrochen
- In allen umgebenden `try`-Blöcken der laufenden Methode wird (von innen nach außen) nach passender `catch`-Klausel gesucht
- Wird passende `catch`-Klausel gefunden, wird deren Rumpf mit Parameterübergabe der Exception ausgeführt
- Nach Abarbeiten der `catch`-Klausel Fortsetzung nach `try`
- Wenn keine passende `catch`-Klausel gefunden wird, propagiert Exception bis ganz nach oben (main()) und Programm bricht ab
- Demo !

finally-Klausel

Und wer räumt den ganzen Mist auf ?

- Wenn Sie im `try` eine Datei öffnen und danach eine Exception auftritt, wo wird die Datei wieder geschlossen?
- Antwort: Im `finally`-Block

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

- Wird immer ausgeführt
- Vorsicht bei Methoden, die Exceptions werfen können
- Tipp: kein `return` verwenden, außer Sie wissen was Sie tun ;-)
- Demo !

Spezifikation von Ausnahmen im Methodenkopf

Moderne Anwendungen sind geschichtet

- Fehlerbehandlung häufig nicht an Stelle des Auftretens
- Ihre Methoden können Exceptions weiterreichen
- Dazu `throws`-Klausel im Methodenkopf (bietet Eclipse an)
- Falls Sie eigene Exceptions werfen, müssen Sie dies natürlich auch tun (werden Sie im Labor)

Automatisches Ressourcenmanagement

Ressourcen sollten immer wieder freigegeben werden

- Dateien, Netzwerkverbindundungen, etc. nach Benutzung immer wieder freigeben
- Typisches Pattern vor Java 7:

```
FileInputStream s = null;
try {
   fis = new FileInputStream("myfile.txt");
   ...
} catch (...) {
   ...
} finally {
   if (fis != null) fis.close();
}
```

- Sehen Sie Probleme ?

Automatic Resource Management (ARM)

- Neue Syntax in Java 7 mit runden Klammern im `try`
- Ressource(n) werden darin deklariert und geöffnet
- Alle geöffneten Ressourcen werden am Ende des `try` garantiert geschlossen, unabhängig vom Auftreten von Exceptions
- Damit das funktioniert, müssen die Ressourcen das Interface `AutoClosable` implementieren
- Die IO-Klassen des SDK tun dies seit Java 7

ARM-Verwendung

```
public void copyFile(String src, String dest)
                throws IOException {
  try (BufferedReader in =
         new BufferedReader(new FileReader(src));
       BufferedWriter out =
         new BufferedWriter(new FileWriter(dest))){

         String line;
         while ((line = in.readLine()) != null) {
           out.write(line);
           out.write('\n');
         }
   } // kein Bedarf fuer 'finally'
}
```

To check or not to check?

Debatte, ob Checked Exception gut oder schlecht sind

- Unchecked Exceptions — The Controversy
- Checked or Unchecked Exceptions?
- Java theory and practice: The exceptions debate

Ein-/Ausgabe

Motivation

- Kein Programm kommt ohne die Kommunikation mit Menschen, anderen Programmen oder Aktoren/Sensoren aus
- Unsere bisherigen Java-Programme haben mit Menschen kommuniziert und zwar über die *Streams* `System.out` und `System.in`
- Streams sind eine UNIX-Erfindung und einfach als Folge von Bytes vorstellbar
- ...

Streams

InputStream, OutputStream

- Die Klassen `InputStream` und `OutputStream` sind Javas Oberklassen für das Lesen und Schreiben von Streams
- Sie sind im Package `java.io` enthalten
- Alle anderen I/O-relevanten Klassen sind ebenfalls in diesem Package enthalten, so dass wir in Zukunft auf die Nennung verzichten
- New I/O ist im Package `java.nio` enthalten, machen wir aber (erstmal) nicht

Byteweises lesen und schreiben

- Da die Streams byte-weise arbeiten, gibt es entsprechende Methoden
- `int read()`
- `int read(byte[] b)`
- `void write(int b)`
- `void write(byte[] b)`
- Wie wir im ersten Semester gesehen haben, verwendet Java Unicode zur Codierung von Zeichen.
- Da Unicode mehrere Bytes verwendet, ist die byte-weise Verarbeitung nicht sonderlich geschickt
- Streams haben aber trotzdem ihre Daseinsberechtigung als niedrigstes Abstraktionsniveau für I/O

Lesen und Schreiben von Text-Dateien

Wir beginnen mit dem Schreiben

- Klasse `PrintWriter` einfachste Möglichkeit hierfür
- Alle Print-Methoden der Klasse `PrintStream` (`System.out`)
- Verwendet Property `line.separator`, also ok für Unix (`\n`) und Windows (`\r\n`)
- Beispiel:

```
PrintWriter out = new PrintWriter(dateiname);
out.println("Erste Zeile");
out.println("Zweite Zeile");
out.println("Dritte Zeile");
out.close();
```

- Exception-Handling fehlt in Darstellung

Und jetzt das Lesen einfacher Textdateien

```
BufferedReader in=
     new BufferedReader(new FileReader(dateiname));
String line;
while (((line = in.readLine()) != null)) {
  System.out.println("gelesen: " + line);
}
```

- `FileReader`: Lesen von Zeichen aus Textdateien mit Default-Character-Encoding
- `BufferedReader`: Methoden für Lesen einer Zeile

Grundlage: Streams

- Statt:

```
BufferedReader in=
    new BufferedReader(new FileReader(dateiname));
```

- Können wir auch schreiben:

```
BufferedReader in=
    new BufferedReader(
        new InputStreamReader(
            new FileInputStream(dateiname)));
```

- Was haben wir gewonnen?

Allgemeine Methode für Streams

```
void readStream(InputStream inputStream) {
  try {
    BufferedReader in= new BufferedReader(
                  new InputStreamReader(inputStream))
    String line;
    while (((line = in.readLine()) != null)) {
      System.out.println("gelesen: " + line);
    }
  } catch (
  ...
```

- Ist komplizierter, aber ...

Alles sind Streams

- Eine Datei
- Ein URL
- Die Tastatur
- ...

```
reader.readStream(new FileInputStream(DATEINAME));

URL url = new URL("http://www.ostfalia.de");
reader.readStream(url.openStream());

reader.readStream(System.in)
```

Serialisierung von Objekten

Motivation

- Java unterstützt mit *Remote Method Invocation* (RMI) verteilte Anwendungen
- Java-EE verwendet dies
- Objekte müssen also zwischen JVM ausgetauscht werden können
- Idee: Objekt serialisieren und ab übers Kabel
- Geht natürlich auch in/aus Datei ;-)

Serialisierung in Datei

```
ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("kunde.dat"));
Kunde kunde = new Kunde();
kunde.setVorname("Bernd");
kunde.setNachname("Mueller");
oos.writeObject(kunde);
oos.close();
```

- `Kunde` ist einfache Klasse mit den beiden Properties `vorname` und `nachname`
- `Kunde` muss das `Serializable`-Interface implementieren. Dies ist ein Marker-Interface ohne Methoden

De-Serialisieren aus Datei

```
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("kunde.dat"));
Kunde kunde = (Kunde) ois.readObject();
System.out.println(kunde.getVorname());
System.out.println(kunde.getNachname());
```

- Einfach, oder ?

Welche Probleme kann es geben?

- Serialisierung eines Objekts mit Version 1.2 der Klasse
- Deserialisierung mit Version 1.5 der Klasse
- Mechanismus zur Verifikation identischer Klassen benötigt
- Realisierung mit einer Klassenversionsnummer: `serialVersionUID`
- Wird implizit intern verwendet
- Kann explizit gesetzt werden
- Compiler meldet Warnung falls nicht gemacht
- Doku im API-Doc von `Serializable`
- Weitere Infos im Abschnitt Serializable-Interface
- Demo !

Datei-Management

Management

- Mit Streams werden Dateiinhalte be/verarbeitet
- Man benötigt jedoch auch Methoden für
  - Verzeichnis oder Datei erzeugen oder löschen
  - Prüfen, wann Datei zum letzten mal modifiziert wurde
  - Datei umbenennen
  - ...
- Die Klasse `java.io.File` ist eine Abstraktion eines Datei- oder Verzeichnisnamens
- Für eine portable Verwendung verwenden Sie das Property `File.separator`

Dateiinformationen

```
File file =
  new File("datei.die.es.wahrscheinlich.nicht.gibt");
System.out.println(file.getAbsolutePath());
System.out.println(file.exists());
```

Datei erzeugen und löschen

```
File file =
  new File("datei.die.es.wahrscheinlich.nicht.gibt");
file.createNewFile();
file.delete();
```

Verzeichnis erzeugen und löschen

```
File file = new File(
  "verzeichnis.das.es.wahrscheinlich.nicht.gibt");
file.mkdir();
file.delete();
```

Dateisysteminformationen

```
File file = new File("bla");
file.createNewFile();
System.out.println(file.getTotalSpace());
System.out.println(file.getUsableSpace());
System.out.println(file.getFreeSpace());
```

Und was es sonst noch so gibt ...

Lesen Sie bitte das API-Doc

Aufgabe

- Die Main-Methode bekommt einen String übergeben, der eine Datei oder ein Verzeichnis relativ angibt
- Erzeugen Sie die Datei oder das Verzeichnis
- Enthält der String einen Punkt, so ist es eine Datei, andernfalls ein Verzeichnis
- Beispiel: `"dir1/dir2/dir3/datei.txt"`

Aufgabe

- Die Main-Methode bekommt einen String übergeben, der den absoluten Pfad eines Verzeichnisses auf Ihrem Rechner angibt
- Geben Sie die Anzahl der Dateien in diesem Verzeichnis aus

New I/O

New I/O

- Mit Java 1.4 bekam Java eine zusätzliche „*new I/O*“
- Features: Nichtblockierend, memory-mapped Dateien, Datei-Locking,
- NIO befindet sich im Package `java.nio`
- Mit Java 7 wurde sogar NIO.2 eingeführt

Dateilesen mit Einzeiler (Java 7)

```
import java.nio.file.Paths;
import java.nio.file.Files;

String content = new String(
    Files.readAllBytes(Paths.get("Dateiname")));
```

Über Verzeichnis iterieren (Java 7)

```
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

  DirectoryStream<Path> directoryStream =
      Files.newDirectoryStream(Paths.get("."));
  for (Path path : directoryStream) {
    System.out.println(path);
  }
```

Das Serializable-Interface

Keine Schnittstelle im herkömmlichen Sinn

- Das `Serializable`-Interface ist keine Schnittstelle im herkömmlichen Sinn, da methodenlos
- Um Objekte mit `writeObject()` und `readObject()` verwenden zu können, muss Klasse serialisierbar sein
- Durch „`implements Serializable`“ wird dem Laufzeitsystem angezeigt, dass Klasseninstanzen besonders zu behandeln sind
- Jedem Objekt wird eine Seriennummer zugewiesen
- Wenn das Objekt mehrfach geschrieben wird, wird nur beim ersten Mal tatsächlich das Objekt, danach die Seriennummer geschrieben
- Beim Einlesen wird dann Seriennummer korrekt aufgelöst, d.h. auf dasselbe Objekt gezeigt und nicht fälschlicherweise Duplikate angelegt
- Also: Wenn zwei Objekte auf dasselbe Objekt referenzieren, ist dies nach Schreiben/Lesen immer noch so

Verwendung

- Alle Instanzvariablen einer serialisierbaren Klasse müssen serialisierbar sein
- Nicht-Serialisierbarkeit aus Sicherheitsgründen !
  - Was nicht serialisierbar ist, kann die VM nicht verlassen
- Was kann nicht serialisierbar sein?
  - Streams (Austausch zwischen VMs ?)
  - Socket (dito)
  - ...

Properties

Properties

- Die Klasse `java.util.Properties` repräsentiert eine persistente Menge von Properties
- Ein Property ist dabei ein Schlüssel/Wert-Paar
- Daher ist die Klasse `java.util.Properties` auch von `Hashtable` abgeleitet, aber mit Strings als Schlüssel und Wert
- Es gibt Methoden, um Properties
  - Zu setzen: `put()`, `setProperty()`
  - Zu lesen: `get()`, `getProperty()` mit Default-Wert
  - Und in Dateien zu schreiben und aus Dateien zu lesen
  - Geerbte Methoden besser nicht, da `Object`

Property erzeugen und in Datei schreiben

```
Properties properties = new Properties();
properties.setProperty("Ein-Schluessel", "Ein Wert");

properties.getProperty("Ein-Schluessel");
properties.getProperty("Tippfehler",
                       "Der Default-Wert");

PrintWriter printWriter = new PrintWriter(PROP_FILE);
properties.store(printWriter, "--Kommentar--");
printWriter.close();
```

Properties von Datei lesen und ausgeben

```
Properties properties = new Properties();
BufferedReader reader =
    new BufferedReader(new FileReader(PROP_FILE));
properties.load(reader);
Enumeration<?> propertyNames =
    properties.propertyNames();
while (propertyNames.hasMoreElements()) {
  String key = (String) propertyNames.nextElement();
  System.out.println("Key: " + key + ", Value: "
                     + properties.getProperty(key));
}
```

System-Properties

- Die Klasse `System` enthält Methoden und Fields des Systems (Java/VM)
- Mit `System.getProperties()` kann man sich die *System-Properties* geben lassen
- Mit `System.getProperty()` kann man sich ein bestimmtes System-Property geben lassen

Property-Dateien als Ressourcen

- Java-Programme werden als Menge von Jars gepackt
- Eine Graphik (gif, png, jpeg) oder eine Property-Datei können dann nicht als Datei zugegriffen werden
- Lösung: Eine Resource:

```
ClassLoader loader =
    Thread.currentThread().getContextClassLoader();
InputStream is =
    loader.getResourceAsStream(PROPERTIES_FILE);
Properties properties = new Properties();
properties.load(is);
```

Aufgabe

- Bestimmen Sie das Home-Verzeichnis des Benutzers, der den java-Befehl ausführt
- Bestimmen Sie Ihr Betriebssystem
- Bestimmen Sie die Versionsnummer des Byte-Codes

Threads

Motivation

- Thread: Programmstück, das (logisch) parallel zu anderen Programmstücken läuft
- Beispiel GUI:
  - Thread 1: Benutzerinteraktion (Maus, Tastatur)
  - Thread 2: Berechnungsvorgang
  - Thread 3: Fortschrittsbalken für diesen Vorgang
  - ...
- Auf Einprozessorsystem?
  - Analog Betriebssystem in Zeitscheiben (quasiparallel)
  - Mit möglichst häufigen Wechseln

Gründe

- GUI reaktiver
  - Lang laufende Aktivitäten in eigenem Thread
  - Thread für Benutzerinteraktion davon unabhängig
- Nutzen von Mehrprozessorsystemen
  - Betriebssystemabhängig, VM-abhängig
  - Thread prinzipiell auf jedem Prozessor/Kern möglich
- Einfachere Modellierung
  - Simulation mit Aktivitäten zwischen Objekten
  - Erzeuger/Verbraucher-Probleme
- Asynchrone Aktivitäten, Hintergrundaktivitäten (GC, z.T.)
  - Server-Anwendungen
  - GC (nicht stop the world)

Demos

Das Gebet !

Durcheinander Schreiben

Primzahlberechnung für bestimmte Zeit

...

Erzeugen von Threads

Threads

- Threads sind einfache Klasse: `java.lang.Thread`
- Können auf zwei verschiedene Arten erzeugt werden:
  - Unterklasse erzeugen
  - Interface `java.lang.Runnable` implementieren
- Methode `Thread.run()` die wichtige:

  If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method dones nothing and returns.
  Subclasses of Thread should override this method.

Alternative Unterklasse

```
public class MyThread extends Thread {

  public void run() {
    for (;;) {
      System.out.println("Hallo vom MyThread-Thread")
    }
  }

  public static void main(String args[]) {
    (new MyThread()).start();
  }

}
```

Alternative Runnable

```
public class MyRunnable implements Runnable {

  public void run() {
    for (;;) {
      System.out.println("Hallo vom MyRunnable-Thread
    }
  }

  public static void main(String args[]) {
    (new Thread(new MyRunnable())).start();
  }

}
```

Beispiel Durcheinander Schreiben

```
public class CharPrinter extends Thread {

  private static final int MAX = 1000;
  private char ch;

  public CharPrinter(char ch) {
    this.ch = ch;
  }

  public void run() {
    for (int i = 0; i < MAX; i++) {
      System.out.print(ch);
      try {
        sleep((int) (Math.random() * 1000));
      } catch (InterruptedException e) {
        return;
      }
    }
  }
  ...
```

Beispiel Durcheinander Schreiben (cont'd)

```
    ...

    public static void main(String[] args) {
      CharPrinter printer1 = new CharPrinter('x');
      CharPrinter printer2 = new CharPrinter('o');
      printer1.start();
      printer2.start();
      System.out.println("Start");
    }
}
```

- nochmals Demo mit Alternativen

Die Hintergründe

- Threads laufen unabhängig voneinander
- Das ist einserseits gut so, um obige Vorteile zu bekommen
- Das ist gefährlich, wenn sie „zusammen“ etwas machen (sollen)
- `main()` läuft in eigenem Thread
- Wer erzeugt den?
  - Das Laufzeitsystem (API-Doc Thread)
- Wann terminiert das Programm ?
  - Wenn der letzte Thread terminiert
  - Wenn System.exit(n) aufgerufen und nicht von Security-Manager abgebrochen wurde

Merke

- Threads sind normale Klassen/Objekte
- Instanzen haben ihre eigenen Instanzvariablen
- Methoden (`run()` und alle anderen) haben selbstverständlich auch ihre eigenen lokalen Variablen
- Zugriff auf Variablen anderer Objekte kann zu Problemen führen
- Anschauliches Beispiel:
  - Ein Thread sortiert großes Array
  - Anderer Thread iteriert über dieses Array

Synchronisation

Problem

```
public class Account {

  private BigDecimal balance;

  void deposit(BigDecimal amount) {
    balance = balance.add(amount);
  }
  void withdraw(BigDecimal amount) {
   balance = balance.subtract(amount);
  }
```

- Kritischer Bereich
- Zugriff nicht atomar
- Unwahrscheinlich, aber möglich, dass es zu falschen Berechnungen kommt

Lösung

- Man muss verhindern, dass kritische Bereiche von mehreren Threads gleichzeitig verwendet werden
- Geht über einen Lock (Sperre)
  - Lock anfordern
  - Wenn bekommen, dann Aktion ausführen
  - Lock freigeben
- Jedes Java-Objekt kann gelockt werden
- Geht über `synchronized` (Anweisung und Modifier)

Lösung Alternative 1

- Neue Variable `lock` als *Sperrvariable*

```
Object lock = new Object();
...
synchronized (lock) {
  balance = balance.add(amount);
}
...
synchronized (lock) {
  balance = balance.subtract(amount);
}
```

- Thread 1 betritt kritischen Bereich und lockt
- Thread 1 wird unterbrochen
- Thread 2 kommt dran
- Kann den Lock aber nicht bekommen und gibt Kontrolle wieder ab
- Thread 1 kommt dran, beendet und gibt Lock frei
- Thread 2 kommt dran
- Bekommt Lock, beendet, gibt Lock frei
- Dies ist ein Beispiel! Man kann *nie* sagen, wie es tatsächlich sein wird.
- Was bedeutet das für Ihr Testen?
- Man spricht bei einem solchen Konstrukt auch von einem *Monitor*

Lösung Alternative 2

- Methode vor konkurrierendem Zugriff schützen

```
synchronized void deposit(BigDecimal amount) {
  balance = balance.add(amount);
}

synchronized void withdraw(BigDecimal amount) {
 balance = balance.subtract(amount);
}
```

wait and notify

Deadlocks

- Es kann vorkommen, dass zwei Threads jeweils einen Lock haben und darauf warten, den Lock des anderen zu bekommen
- Keiner kann weitermachen, also auch nicht den eigenen Lock wieder freigeben, da er ja auf den anderen Lock wartet
- Man spricht von einem *Deadlock*

Deadlock schematisch

```
Object resource1 = new Object();
Object resource2 = new Object();
...
Thread t1 = new Thread(new Runable() {
  public void run() {
    synchronized(resource1) {
      synchronized(resource2) { tuWas();}
    }
  }
});
Thread t2 = new Thread(new Runable() {
  public void run() {
    synchronized(resource2) {
      synchronized(resource1) { tuWas();}
    }
  }
});
t1.start();
t2.start();
```

Lösung

- Lösung: Alle Locks temporär freigeben und warten, bis man sie erneut bekommen kann
- Dazu muss ein solcher wartender Thread benachrichtigt werden
- Dafür die beiden Methoden `wait()` und `notify()`, bzw. `notifyAll()`
- Da alle Objekte einen Lock und alle Objekte eine Liste von wartenden Threads haben, sind dies Methoden der Klasse `Object`, nicht `Thread`

Beispiel

```
public class Warteschlange {
  LinkedList<E> schlange = new LinkedList<E>();
  public synchronized void push(E o) {
    schlange.add(o);
    this.notifyAll();
  }
  public synchronized E pop() {
    while(schlange.size() == 0) {
      try { this.wait(); }
      catch (InterruptedException ignore) {}
    }
    return schlange.remove(o);
  }
}
```

- Weiteres Beispiel im Buch: nicht überziehbares Konto

Merke

- `wait()` immer in Schleife mit Wartebedingung
  - Beim Aufwachen kann nicht garantiert werden, dass Bedingung erfüllt, daher nochmals testen und evtl wieder schlafen legen
- `wait()` und `notify()` nur aus synchronisierter Methode aufrufbar, sonst Exception
- `notifyAll()` weckt alle, die auf diesen Monitor warten
- `wait()` kann Exception (Strg-C) auslösen, daher in try/catch

Und was Sie auch noch wissen sollten

- Seit Java 5 neues Package `java.util.concurrent` für Thread-Programmierung
- Sub-Package `java.util.concurrent.atomic` enthält Datentypen für Thread-Programmierung ohne Locks, da die Zugriffe garantiert atomar sind
- Sub-Package `java.util.concurrent.locks` Framework für Locking und Warten, das weit über `wait()` und `notify()` hinaus geht
- Es gibt viele Bibliotheksklassen, die den konkurrierende Zugriff erlauben, viele andere (die meisten) nicht. Lesen Sie das API-Doc !!!
- Beispiel von vorhin mit der Schlange:
  - Interface `java.util.concurrent.BlockingQueue`
  - Mit Implementierung `java.util.concurrent.ArrayBlockingQueue`
- Machen wir z.T. im nächsten Kapitel

Und was noch ...

- Threads haben Prioritäten (`getPriority()`, `setPriority()`)
- Scheduling mit Timer (`java.util.Timer`, `java.util.TimerTask`)
- Executor Interface und Service (sicheres Starten von Threads)
- Sie sehen, wir haben nur an der Oberfläche gekratzt ;-)

Aufgaben

- Implementieren Sie die Aufgaben aus dem Buch:
- Konkurrierende I/O
- Monitore
- Speisende Philosophen (Dining Philosophers, Philosophenproblem)

Nebenläufigkeit und Parallelität

Aktualität von Java bzgl. Nebenläufigkeit und Parallelität

- Bei der Einführung von Java im Jahr 1995 (Konzeptphase einige Jahre früher) waren Einprozessorsysteme die Regel
- Es kam darauf an, zu verhindern, dass ein Thread wartet (z.B. auf langsame I/O) und das ganze System steht
- Heute gibt es viele Prozessoren/Kerne und es kommt darauf an, die Arbeit auf möglichst viele Prozessoren/Kerne zu verteilen
- Und zwar möglichst einfach für den Entwickler ;-)
- Dies ist eine völlig andere Aufgabe !
- Die im Kapitel 20 vorgestellte Vorgehensweise ist mehr oder weniger veraltet und sollte in der Regel nicht mehr verwendet werden
- Es gibt eine Reihe alternativer Neuerungen, die wir uns jetzt z.T. anschauen

Motivation Atomic-Variablen

- Multi-Threaded Programmierung ist *sehr* schwierig. Wir schauen uns hier nur etwas um
- Eine Alternative zu `synchronized` sind Volatile-Variablen, also Variablen mit dem `volatile` Schlüsselwort
- Dies basiert auf dem Java-Memory-Modell, das definiert, wann Variablen, die von einem Thread manipuliert werden, von anderen Threads gesehen werden
- Eine weitere Alternative sind Atomic-Variablen, die mit Java 5 eingeführt worden sind

Atomic-Variablen

- Aus API-Doc: „*Package java.util.concurrent.atomic: A small toolkit of classes that support lock-free thread-safe programming on single variables.* “
- Werfen Sie einen Blick rein

Beispiel Atomic-Variable

```
public class MyCounter1 {

  private static int counter = 0;

  public static int getCount() {
    return counter++;
  }

}
```

- Ist der Counter-Zugriff thread-sicher?

Beispiel Atomic-Variable (cont'd)

```
public class MyCounter2 {

  private static int counter = 0;

  public static synchronized int getCount() {
    return counter++;
  }

}
```

- Klasse wird gelockt
- Falls nicht static, wird Instanz gelockt
- Ist aber old-school und besser mit Atomic-Variable zu machen

Beispiel Atomic-Variable (cont'd)

```
public class MyCounter3 {

  private static AtomicInteger
    counter = new AtomicInteger(0);

  public static int getCount() {
    return counter.getAndIncrement();
  }

}
```

Callables and Futures

Motivation

- Ein `Runnable` (Abschnitt 20.1) kapselt eine Task, die asynchron läuft
- Also so etwas wie: Asynchrone Methode ohne Parameter und ohne Rückgabewert
- Ein `Callable` (Package `java.util.concurrent`) ist einem `Runnable` ähnlich, hat aber einen Rückgabewert
- Da die Methode asynchron läuft, ist das Ergebnis „irgend wann“ mal da
- Es wird in ein `Future` (Package `java.util.concurrent`) gepackt
- Methode `get()` liefert das Ergebnis, blockiert aber, bis es berechnet ist
- Methode `isDone()` prüft, ob Berechnung fertig
- Damit Beispiel ...

Fibonacci mit Callable und Future

```
public class CallableFibonacci
             implements Callable<Integer> {

  private int n;

  public CallableFibonacci(int n) {
    this.n = n;
  }

  @Override
  public Integer call() throws Exception {
    return fibonacci(n);
  }
  private int fibonacci(int i) { ... }
}
```

Und die Verwendung

```
Callable<Integer> fibo = new CallableFibonacci(45);
FutureTask<Integer> task = new FutureTask<>(fibo);
Thread thread = new Thread(task);
thread.start(); // laeuft in anderem Thread
for(;;) {
   System.out.println("In Schleife");
   Thread.sleep(100);
   if (task.isDone()) {
      break;
   }
}
System.out.println("Ergebnis: " + task.get());
```

- FutureTask Wrapper, um Callable in Future und Runnable zu wandeln

Executors und Thread-Pools

Motivation

- Erzeugen eines Threads ist relativ teuer (z.B. Interaktion mit Betriebssystem)
- Falls viele kurzlaufende Threads zu erzeugen sind, nutzt man besser einen Thread-Pool
- Ein Thread-Pool enthält eine Menge von (nicht tätigen) Threads, die nur darauf warten, loszulaufen
- Man übergibt den Pool ein `Runnable`/`Callable` und einer der Threads ruft die `run()`/`call()`-Methode auf
- Wenn die `run()`/`call()`-Methode fertig ist, stirbt der Thread nicht, sondern wartet auf die nächste Aufgabe
- Wie bekommt man einen solchen Pool ?

Zentraler Einstiegspunkt: Fabrikmethoden von `Executors`

`newCached-ThreadPool()`	Neue Threads bei Bedarf erzeugt. Idle Threads 60 Sekunden aufbewahrt. Daher sinnvoll für viele kurzlebige asynchrone Tasks
`newFixed-ThreadPool()`	Feste Anzahl Threads. Idle Threads sterben nicht
`newSingle-ThreadExecutor()`	Einzelner Thread, der Tasks sequentiell abarbeitet
`newScheduled-ThreadPool()`	Feste Anzahl. Geplante Ausführung
`newSingleThread-ScheduledExecutor()`	Einzelner Thread für geplante Ausführung. Ersatz für Timer

ExecutorService

- Die Fabrikmethoden liefern eine Instanz von `ExecutorService`, bzw. des Sub-Interfaces `ScheduledExecutorService`
- Damit können Task an den Thread-Pool in Auftrag gegeben werden
- Dazu verschiedene Methoden (die Sie sich selbst anschauen):
  - `execute()`
  - `submit()`
  - `invokeAll()`
  - `invokeAny()`

Beispiel mit Runnable

```
public class RunnableFibonacci implements Runnable {
  private int n;
  public RunnableFibonacci(int n) {
    this.n = n;
  }

  @Override
  public void run() {
    System.out.println("Fibonacci von " + n + " ist "
                       + fibonacci(n));
  }

  private int fibonacci(int i) { ... }
}
```

Verwendung und Demo

```
ExecutorService service =
    Executors.newFixedThreadPool(2);
service.execute(new RunnableFibonacci(43));
service.execute(new RunnableFibonacci(44));
service.execute(new RunnableFibonacci(45));
service.execute(new RunnableFibonacci(46));
service.shutdown();
```

- Geht auch mit `Callable`

Das Fork-Join-Framework

Fork-Join-Framework, neu in Java 7

- Die gezeigten Neuerungen in Java 5 und 6 erlauben die einfache Verwaltung nebenläufiger Prozesse
- Das Fork-Join-Framework in Java 7 erlaubt die einfache Realisierung von Divide-and-Conquer, bzw. Map/Reduce-Algorithmen
- Da es keine/wenige Java-7-Bücher gibt, hilft ein Blick in Oracles Java Tutorial
-

Das Prinzip (aus Tutorial)

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

- Erkennen Sie die Ähnlichkeit zur Rekursion in Kapitel 16 ?

Verwendung

- Zentral: die Klasse `ForkJoinPool`
- Realisiert einen „*work stealing*“Algorithmus: Alle Threads des Pools versuchen Sub-Tasks zu finden und auszuführen, die von anderen Tasks erzeugt wurden
- Wenn ein Worker-Thread also nichts zu tun hat, versucht er anderen Threads, die gerade beschäftigt sind, ihre Tasks zu stehlen
- Außerdem werden über den `ForkJoinPool` die auszuführenden Tasks eingestellt
- Der Default-Konstruktor erzeugt einen Pool, der soviel Parallelismus vorsieht, wie Prozessoren/Kerne vorhanden sind

Beispiel: Summe von / bis

```
public class Sum extends RecursiveTask<Long> {

  private static final int BOUND = 100;

  private int from;
  private int to;

  public Sum(int from, int to) {
    this.from = from;
    this.to = to;
  }
  ...
```

Beispiel (cont'd)

```
@Override
protected Long compute() {
  if (to - from > BOUND) {
    Sum sum1 = new Sum(from, (to + from) / 2);
    sum1.fork();
    Sum sum2 = new Sum((to + from) / 2 + 1, to);
    return sum2.compute() + sum1.join();
  } else {
    long sum = 0;
    for (int i = from; i <= to; i++) {
      sum += i;
    }
    return sum;
  }
}
```

Beispiel (cont'd)

```
public static void main(String[] args) {
  ForkJoinPool pool = new ForkJoinPool(4);
  long sum = pool.invoke(new Sum(1, 900000000));
  System.out.println("Summe: " + sum);
}
```

Selbststudium

Referenzen für's Selbststudium

- The Java Memory Model
- 55 New Things in Java 7 — Concurrency
- Java Concurrent Animated

Annotationen

Motivation

- Bei großen Anwendungen oder Anwendungen, die mit der Java-Enterprise-Edition erstellt werden, gibt es in der Regel viel zu konfigurieren, bzw. Meta-Informationen hinzuzufügen
- Dies wurde in der Vergangenheit häufig über XML-Dateien realisiert
- Und führte zur sogenannten „ XML-Hölle“(XML hell)
- Um dies für Java-EE 5 zu verhindern, wurden in Java-SE 5 Annotationen eingeführt
- Obwohl die Verwendung quantitativ vor allem in EE stattfindet, ist die Verwendung auch in SE sinnvoll
- Annotationen repräsentieren Meta-*Informationen*, sind also erst einmal für die Programmausführung nicht relevant
- Sie werden von Werkzeugen, Frameworks, . . . genutzt

Verwendung

Verwendung

- Für Annotationen wurde die Java-Syntax in der Version 5 erweitert
- Sie verwendet nun den Klammeraffen zur Kennzeichnung von Annotationen
- In Java 5 wurden die folgenden Annotationen eingeführt
  - `@Deprecated`
  - `@Override`
  - `@SuppressWarnings`
- In Java 7 kam `@SafeVarargs` hinzu
- Wir haben `@SuppressWarnings` bereits im Kapitel über Generics verwendet, `@Override` haben Sie bereits in den Übungen kennengelernt
- Diese Standard-Annotationen befinden sich im Package `java.lang`

Beispiele

- Wenn eine Programmelement nicht mehr verwendet werden soll, so kann es mit `@Deprecated` annotiert werden
- Die Verwendung ist bei allen Programmelementen zulässig: Packages, Klassen, Interfaces, Annotationen, Enums, Methoden, Instanzvariablen, lokalen Variablen, Parametern, Konstruktoren
- Die häufigste Verwendung ist sicher bei Klassen und Methoden:

```
@Deprecated
public class Veraltet {
  @Deprecated // implizit, da Klasse annotiert
  public void method(int i) {
    ...
  }
}
```

Verwendung (cont'd)

- Wenn Sie an anderer Stelle die Klasse/Methode/... verwenden, zeigt der Compiler eine Warnung an
- Diese kann man mit der Annotation `@SuppressWarnings` unterdrücken
- Die Annotation `@SuppressWarnings` verlangt Parameter. Im Falle veralteter Elemente das Literal `"deprecation"`
- Im Kapitel Generics hatten wir `"rawtypes"` und `"unchecked"` verwendet

Selbstdefinierte Annotationen

Annotationen für Annotationen

- Annotationen sind syntaktisch spezielle Interfaces mit dem Klammeraffen vor dem Schlüsselwort `interface`
- Um Annotationen zu definieren, werden Annotation verwendet
- Dies befinden sich im Package `java.lang.annotation` und sind `@Documented`, `@Inherited`, `@Retension` und `@Target`, wobei die beiden letztgenannten die wichtigen sind
- Mit `@Retension` wird angegeben, ob die Annotation nur zur Compile-Zeit oder auch zur Laufzeit vorhanden ist
- Mit `@Target` wird das Ziel der Annotation definiert, also z.B. Klasse, Methode, Variable, ...
- Parameter der Annotation müssen als Methoden deklariert werden, da Interfaces keine Variablen haben

Beispiel für Annotation

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value =
    {ElementType.TYPE, ElementType.METHOD})
public @interface Version {

  int number();

}

@Version(number = 2)
class MyClass {
  ...
}
```

Schreibersparnis

- Wenn eine Annotation nur den einzigen Parameter `value` hat, darf dieser bei der Verwendung weggelassen werden
- Obiges Beispiel kann man also auch so schreiben:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Version {

  int number();

}
```

Annotationen zur Laufzeit

- Falls Annotationen zur Laufzeit existieren, können sie über Reflection zugegriffen werden
- Das Programm kann dann also bestimmte Dinge tun oder lassen, je nachdem, ob die Annotation verwendet wurde oder nicht
- Dies wird vor allem seit Version 5 von Java-EE gemacht, wie eingangs motiviert
- Ein schönes Beispiel ist Java Persistence API (JPA), das praktisch nur aus Annotationen besteht

Graphische Oberflächen

Grundlagen: MVC und Events

Model-View-Controller Architektur (oder Pattern)

- Allgemein akzeptierter und verbreiteter Standard für graphische Oberflächen
- Basiert auf der Dreiteilung einer Anwendung mit GUI
  - **Model:** Die Daten, also das Geschäftsmodell. Kunde, Rechnung, Bestellung, ...
  - **View:** Präsentation des Modells und Entgegennahme von Benutzerinteraktionen
  - **Controller:** Steuerung. Nimmt Benutzeraktionen von View entgegen und agiert entsprechend. Ändert Daten im Modell aber nicht direkt (ist ja objektorientiert gekapselt), sondern kann nur Modell-Methoden aufrufen. Änderungen im Modell müssen an die View kommuniziert werden, damit diese sie anzeigen kann
- Unbedingt notwendig: *Lose Kopplung*

Event-gesteuerte Programmierung

- Programmierstil mit *Signal-and-Response*-Ansatz zur Programmierung und nicht wie bisher *Methodenaufruf*
- Die Signale werden *Events* (Ereignisse) genannt
- Diese werden *gefeuert* (to fire an event) oder ausgelöst
- Objekte, die an Events interessiert sind, heißen *Listener*
- Beispiel:
  - Das Klicken einer Schaltfläche ist ein Event
  - Listener reagieren darauf

Event-gesteuerte Programmierung (cont'd)

- Man erreicht eine *lose Kopplung*
- Objekte, die Events feuern, und Listener, die auf diese Events reagieren, nicht programmatisch verbunden
- Es gibt keine Code-Abschnitt, der die Event- oder Listener-Reihenfolge definiert
- Das wird durch die Events gesteuert !
- Der Programmablauf wird durch das nächste ausgelöste Event bestimmt
- Ungewöhnlich für Sie: Sie schreiben Methoden, die Sie nie aufrufen

Java und GUIs

Historie

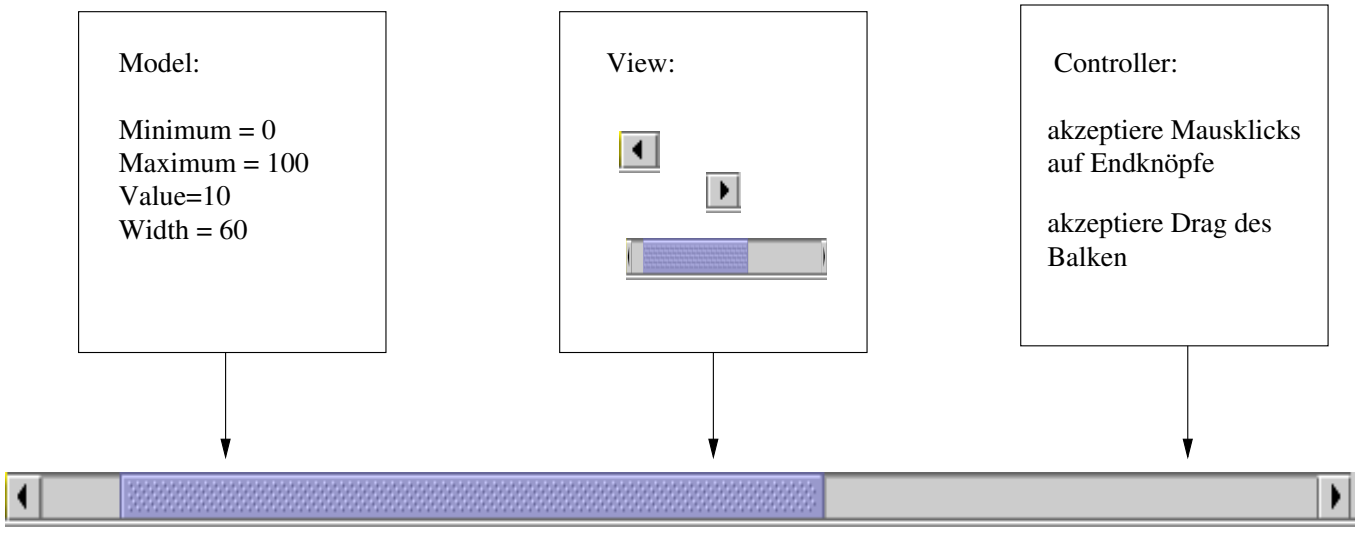
- Java 1.0: Abstract Window Toolkit (AWT)
- Package `java.awt` und Sub-Packages
- Verwendet natives GUI des Betriebssystems
- Es zeigte sich, dass qualitativ hochwertige, portable Graphikbibliotheken, die native UI-Elemente verwenden, sehr schwierig und aufwändig zu realisieren sind
- 1996 erstellte Netscape die Internet Foundation Classes (IFC)
- Idee: leeres Fenster und malen im Fenster nativ vom Betriebssystem, den Rest macht die Bibliothek selbst
- Ergebnis: Sieht überall gleich aus
- Sun kooperierte mit Netscape und erstellte UI-Bibliothek mit Namen *Swing*
- Seit Java 1.2 im SDK enthalten. Package `javax.swing` und Sub-Packages

Historie (cont'd)

- Swing kein (vollständiger) Ersatz für AWT
- Teile, z.B. `java.awt.event` werden weiter verwendet
- Swing-Demo in älteren SDKs enthalten, in neueren separat : `<install>/demo/jfc/SwingSet2`
- Seit 6.10 SwingSet3 als JNLP
- IBM mit Performanz-Verhalten von Swing sehr unzufrieden
- Für Eclipse Standard Widget Toolkit (SWT) gemacht
- Daher Eclipse *nicht* pure Java (verschiedene Downloads)
- Für Rich Internet Applications: JavaFX
- Konkurrenz zu Flash und Silverlight
- Kein Ersatz für Swing (verwendet z.T. Swing)
- Einschätzung: wird sich durchsetzen und Swing ablösen

Programmieren mit Swing
— die Anfänge —

MVC angepasst



- Jede Swing-Komponente hat ein Model und ein UI-Delegate
  - Model verwaltet Komponentenzustand
  - UI-Delegate weiß, wie Komponente darzustellen ist und auf Events zu reagieren ist

Events

- Alle im Package `java.awt.event`
  - `ActionEvent`
  - `FocusEvent`
  - `KeyEvent`
  - `MouseEvent`
  - `WindowEvent`
  - `...`
- Listener nach Muster: `<Art>Event` mit `<Art>Listener`
  - `ActionListener`
  - `FocusListener`
  - `KeyListener`
  - `MouseListener`
  - `WindowListener`
  - `...`
- Später Beispiele

Unser erstes Fenster

```
class MyFrame extends JFrame {

  static final int WIDTH = 300;
  static final int HEIGTH = 200;

  public MyFrame() {
    setSize(WIDTH, HEIGTH);
  }
}
```

- `JFrame` Top-level Fenster mit Titel und Rahmen
- `setSize()` offensichtlich geerbte Methode

Und das Erzeugen

```
public static void main(String[] args) {
  EventQueue.invokeLater(new Runnable() {
    public void run() {
      JFrame frame = new MyFrame();
      frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
    }
  });
}
```

- Muss im Event-Dispatcher-Thread aufgerufen werden für korrektes Funktionieren
- JVM beenden nach Fenster schließen
- Fenster anzeigen (sonst existent aber unsichtbar)

Verbesserungsvorschlag

```
class MyFrame extends JFrame {

  static final int WIDTH = 300;
  static final int HEIGTH = 200;

  public MyFrame(String title) {
    super(title);
    setSize(WIDTH, HEIGTH);
  }
}
```

- und dann natürlich Aufruf mit

```
JFrame frame = new MyFrame("Fenster-Titel");
```

Und so sieht's aus

Eine Anwendung, die etwas (sinnvolles?) tut

```
class MyFrame extends JFrame {

  public MyFrame(String title) {
    super(title);
    JButton button = new JButton("Drueck mich");
    add(button);
    button.addActionListener(new KnopfListener());
  }
}
```

- JButton feuert ActionEvents und besitzt Methode addActionListener(), um Listener zu registrieren

```
class KnopfListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Ich wurde gedrueckt");
  }
}
```

- Besagte Methoden, die Sie nie aufrufen

Container und Layout-Manager

Container

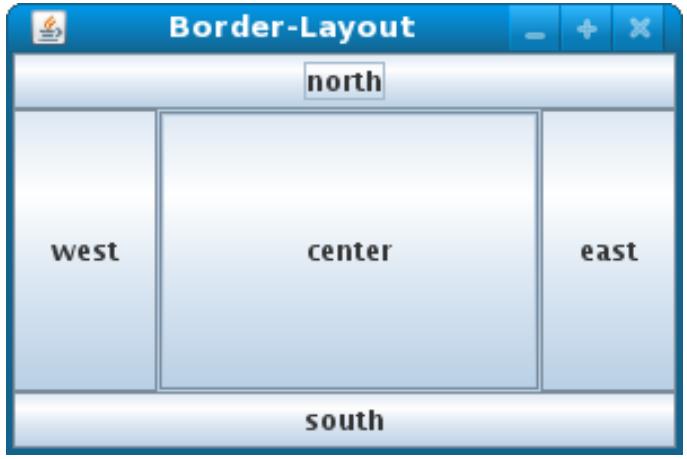
- In ein `JFrame` kann man mehrere UI-Elemente reinstecken
- Ein `JFrame` ist ein `Container` (`java.awt`)
- Es gibt
  - Top-Level Container (`JDialog`, `JFrame`, `JWindow`, `JApplet`, `JInternalFrame`) implementieren Interface `RootPaneContainer`
  - Innere Container (`JPanel`, `JList`, `JTree`, `JTable`, ...)
- Besitzen überladene `add()`-Methoden
- Diese fügen neue Komponenten als Söhne der Hiarchie hinzu, ohne jedoch zu sagen, wie diese anzuordnen sind (nebeneinander, übereinander, ...)

Layout-Manager

- `BorderLayout`
  Fünf Regionen (NORTH, WEST, CENTER, EAST, SOUTH)
- `FlowLayout`
  Einfachstes Layout
  Alle nacheinander, von links nach rechts
- `GridLayout`
  Tabelle mit Zeilen und Spalten
- `GridBagLayout`
  kompliziertester Layout-Manager, da alles machbar

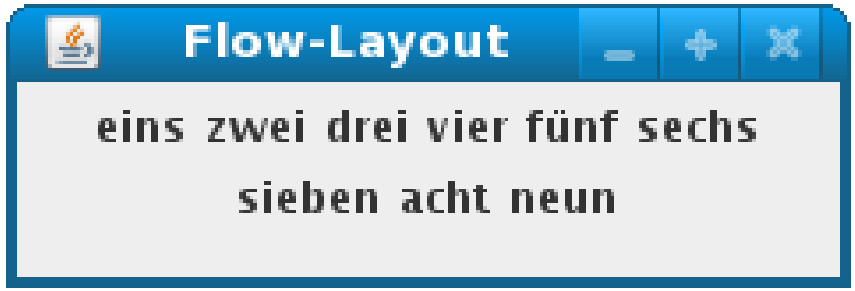
BorderLayout

```
class MyFrame extends JFrame {
  public MyFrame() {
    setTitle("Border-Layout");
    setLayout(new BorderLayout());
    add(new JButton("north"), BorderLayout.NORTH);
    add(new JButton("west"), BorderLayout.WEST);
    add(new JButton("center"), BorderLayout.CENTER);
    add(new JButton("east"), BorderLayout.EAST);
    add(new JButton("south"), BorderLayout.SOUTH);
  }
```



FlowLayout

```
class MyFrame extends JFrame {
  public MyFrame() {
    setTitle("Flow-Layout");
    setLayout(new FlowLayout());
    add(new JLabel("eins"));
    add(new JLabel("zwei"));
    add(new JLabel("drei"));
    add(new JLabel("vier"));
    add(new JLabel("fuenf"));
    add(new JLabel("sechs"));
    add(new JLabel("sieben"));
    add(new JLabel("acht"));
    add(new JLabel("neun"));
  }
```



GridLayout

```
class MyFrame extends JFrame {
  public MyFrame() {
    setTitle("Flow-Layout");
    setLayout(new GridLayout(3, 3));
    add(new JLabel("eins"));
    add(new JLabel("zwei"));
    add(new JLabel("drei"));
    add(new JLabel("vier"));
    add(new JLabel("fuenf"));
    add(new JLabel("sechs"));
    add(new JLabel("sieben"));
    add(new
    add(new
    add(new
  }
```

Stammdatenverwaltung mit GridBagLayout

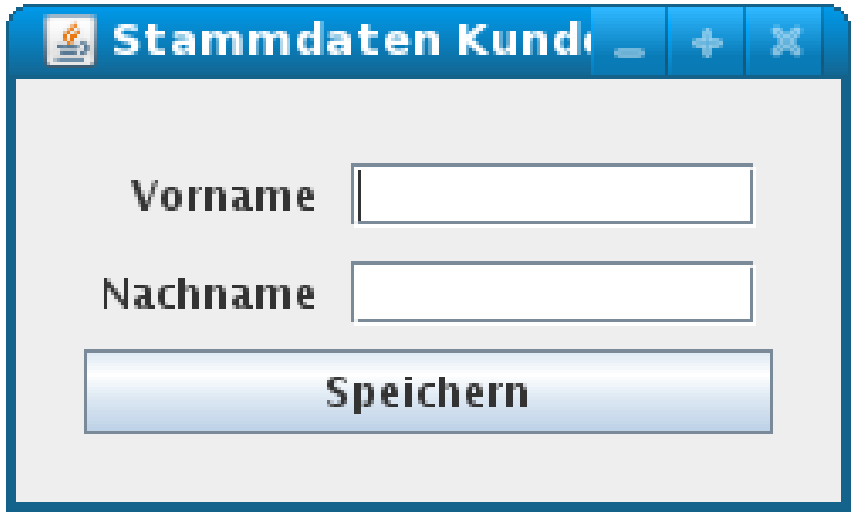
```
private JPanel getStammdatenPanel() {
  JPanel panel = new JPanel();
  JLabel vornameLabel = new JLabel("Vorname");
  JLabel nachnameLabel = new JLabel("Nachname");
  panel.setLayout(new GridBagLayout());
  GridBagConstraints c = new GridBagConstraints();
  c.insets = new Insets(5, 5, 5, 5);
  c.anchor = GridBagConstraints.EAST;
  c.gridwidth = GridBagConstraints.RELATIVE;
  c.fill = GridBagConstraints.NONE;
  c.weightx = 0.0;
  panel.add(vornameLabel, c);
  c.gridwidth = GridBagConstraints.REMAINDER;
  c.fill = GridBagConstraints.HORIZONTAL;
  c.weightx = 1.0;
  panel.add(vornameText, c);
  c.gridwidth = GridBagConstraints.RELATIVE;
  c.fill = GridBagConstraints.NONE;
  c.weightx = 0.0;
  panel.add(nachnameLabel, c);
  c.gridwidth = GridBagConstraints.REMAINDER;
  c.fill = GridBagConstraints.HORIZONTAL;
  c.weightx = 1.0;
  panel.add(nachnameText, c);
  return panel;
}
```

Stammdatenverwaltung mit GridBagLayout (cont'd)

```
public class KundenStammdaten extends JFrame {

  JTextField vornameText = new JTextField(20);
  JTextField nachnameText = new JTextField(20);

  public KundenStammdaten() {
    super("Stammdaten Kunden");
    JPanel panel = new JPanel();
    panel.setBorder(BorderFactory.createEmptyBorder(20

    BorderLayout layout = new BorderLayout(20, 20);
    panel.setLayout(layout);
    panel.add(getStammdatenPanel(), BorderLayout.NORTH
    JButton button = new JButton("Speichern");
    button.addActionListener(new SpeichernListener())
    panel.add(button, BorderLayout.SOUTH);
    add(panel);
  }
```

Stammdatenverwaltung mit GridBagLayout (cont'd)

```
class SpeichernListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    String vorname = vornameText.getText().trim();
    String nachname = nachnameText.getText().trim();
    // jetzt in Datenbank schreiben oder:
    System.out.println("speichern von " + vorname
                       + " " + nachname);
  }
}
```



Java im Netzwerk

Client/Server

- Netzwerke mittlerweile Standard
- Es gibt keine Stand-Alone-Anwendungen mehr
- Typische Architektur: Client/Server (es gibt andere)
- Java von Beginn an auf Netzwerk ausgelegt
- Package für Netz (`java.net`) und weitere

Grundlagen Netzwerke

- Protokoll:
  - Datenaustausch paarweise: Zwei Rechner nehmen Kontakt auf und tauschen Daten aus
  - Damit Kommunikation funktioniert, müssen sich beide auf *Protokoll* einigen
  - Bekannt: ISO/OSI 7 Schichten
  - Praxis: TCP/IP
- TCP/IP:
  - Anwendungsschicht: FTP, HTTP, SMTP, ...
  - Transportschicht: TCP und UDP
  - Netzwerkschicht: IP
  - Physikalische Schicht: Ethernet, FDDI, ...

Grundlagen Netzwerke (cont'd)

- IP-Adressen:
  - Kommunikation benötigt eindeutige Identifizierung
  - IP-Adresse besteht aus 4 Bytes (derzeit, IPV6 steht vor der Tür)
  - Domain/Host-Namen werden auf IPs abgebildet (DNS)
- Beispiel:

```
InetAddress adresse =
InetAddress.getByName("www.ostfalia.de");
System.out.println("IP:    " +
                   adresse.getHostAddress());
System.out.println("Name:  " +
                   adresse.getHostName());
```

Grundlagen Netzwerke (cont'd)

- Meist nur eine (oder wenige) physikalische Verbindungen pro Rechner
- Meist mehrere Anwendungen pro Rechner mit Kommunikationsbedarf
- Also: Adressierung innerhalb eines Rechners mit Port-Nummern
- Ganze Zahlen im Bereich 0 bis 65535
- Well known Ports 0 - 1023
- Registered Ports 1024 - 49151
- Dynamic / private Ports 49152 - 65535
- Blick in `/etc/services`

Sockets

Grundlagen

- Sockets sind keine Java-Erfindung
- Sockets gehen auf eine Abstraktion in Berkley Unix zurück, die die Unix-Stream-Idee auf Netzwerke ausweitet
- Unix-I/O ist immer Lesen und Schreiben von Streams von Bytes
- keine Unterscheidung: normale Datei, Tastatur, Maus, Display . . . Netzwerkverbindung
- Sockets kapseln low-level Details des Netzwerks, etwa Medientypen, Paketgrößen, erneute Übertragung verlorengegangener Pakete, Netzwerkadressierung . . .
- Sockets sind Grundlage des Internets und auf allen Rechnern verfügbar

Grundlagen II

Ein Socket ist eine Verbindung zwischen zwei Rechnern mit sieben Basisoperationen:

- Verbinden mit einem entfernten Rechner
- Daten senden
- Daten empfangen
- eine Verbindung schließen
- Binden an einen Port
- auf eingehende Daten horchen
- Verbindungen von entfernten Rechnern auf dem gebundenen Port akzeptieren

Grundlagen III

Grundsätzliches Vorgehen

- `java.net.Socket` verwenden, um Verbindung zwischen zwei bekannten Prozessen herzustellen
- beide Prozesse müssen eine Instanz von Socket erzeugen, über die der Datenaustausch erfolgt
- `java.net.SocketServer` verwaltet initiale Verbindung zwischen Client und Server auf Server-Seite.
- `SocketServer` erzeugt bei ersten Client-Anfrage einen `Socket`, so dass die Socket-Socket-Verbindung erstellt werden kann

Java-freier Client-Socket

Ein einfacher Socket kann schon mit Betriebssystem-Bordmitteln getestet werden:

```
bernd@work:~> telnet www.fh-wolfenbuettel.de 80
Trying 141.41.1.249...
Connected to www.fh-wolfenbuettel.de.
Escape character is '^]'.
```

- wenn „`Connectet to ...`“ erscheint, ist Socket-Verbindung hergestellt
- jetzt müsste man HTTP können !!
  (GET / HTTP/1.0, gefolgt von *zwei* Newlines)

. . . und in Java

Das geht natürlich auch in Java

```
public class HostTest {

  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: HostTest <host> <port>");
      System.exit(1);
    }
    try {
      Socket socket = new Socket(args[0],
                                 Integer.parseInt(args[1]));
      System.out.println("Dienst ist da");
    } catch (UnknownHostException e) {
      System.out.println(
          "Die IP kann nicht ermittelt werden");
    } catch (IOException e) {
      System.out.println("Kein Dienst");
    }
  }
}
```

Erläuterung

- Konstruktor `Socket(String, int)` erwartet Host und Port
- es wird lediglich die Verbindung aufgebaut, sonst passiert nichts
- es können die entsprechenden Exceptions auftreten
- Aufruf:
  `java HostTest www.ostfalia.de 80`
- das Programm testet lediglich, ob ein TCP-Dienst auf Rechner/Port vorhanden ist
- es kann daher einfach zu einem Port-Scanner ausgebaut werden

Portscanner

```
public class PortScanner {

  private static String host;
  private static final int ROOT_DIENSTE = 1024;
  private static final int ALLG_DIENSTE = 65536;

  public static void main(String[] args) {
    if (args.length > 0) {
      host = args[0];
    }
    for (int i = 1; i < ALLG_DIENSTE; i++) {
    try {
      Socket s = new Socket(host, i);
      System.out.println("Dienst auf port " + i +
          " von Host " + host + " gefunden");
    } catch (UnknownHostException e) {
      System.err.println(e);
      break;
    } catch (IOException e) {
       // gibt hier keinen Dienst, nichts tun
    }
  }
```

Mögliche Ausgabe

- Ausgabe:

```
Dienst auf port 22 von Host localhost gefunden
Dienst auf port 25 von Host localhost gefunden
Dienst auf port 111 von Host localhost gefunden
Dienst auf port 631 von Host localhost gefunden
```

Warnung

Achtung!
Führen Sie dieses Programm nur gegen Ihren eigenen Rechner aus.

Ein Server-Admin könnte dies als Angriff werten

Socket-Server

Prinzip

- auch Socket-Server in Java leicht zu realisieren
- Klasse `java.net.ServerSocket`
- prinzipielles Vorgehen:
  - Erzeugen einer `ServerSocket`-Instanz (lauscht auf einem Port)
  - Aufruf der accept()-Methode, die einen Socket zurück gibt

```
private static final int PORT = 9999;

public static void main(String[] args) {
  ServerSocket socket;
  Socket csocket;
  try {
    socket = new ServerSocket(PORT);
    while ((csocket = socket.accept()) != null) {
      process(csocket);
    }
  }
  catch (IOException e) {
    System.err.println(e);
  }
}
private static void process(Socket csocket) throws IOExcepti
  System.out.println("Anfrage von Client "
                     + csocket.getInetAddress());
  csocket.close();
}
```

Aufgabe

Testen Sie den Server per `telnet`

Socket-Antworten

- Sockets können auch Daten vom Client lesen und Daten an den Client zurück schicken
- dazu wird der von der `accept()`-Methode zurückgegebene Socket verwendet
- Input-Stream für die Parameter, Output-Stream für die Antwort
- dann Reader und Writer erzeugen
- Beispiel: Echo-Protokoll (Port 7)

Echo-Server

```
public class EchoServer {
  // eigentlich 7, aber da darf nur der Admin
  private static final int ECHOPORT = 9999;
  private ServerSocket socket;

  public EchoServer(int port) {
    try {
      socket = new ServerSocket(port);
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
  public static void main(String[] args) {
    new EchoServer(ECHOPORT).serve();
  }
}
```

Echo-Server (cont'd)

```
private void serve() {
  Socket iosocket = null;
  BufferedReader br = null;
  PrintWriter pw = null;
  while (true) {
    try {
      iosocket = socket.accept();
      System.out.println("Request von "
          + iosocket.getInetAddress().getHostName());
      br = new BufferedReader(new InputStreamReader(iosocke
          .getInputStream(), "ISO-8859-1"));
      pw = new PrintWriter(new OutputStreamWriter(iosocket
          .getOutputStream(), "ISO-8859-1"));
      String line;
      while ((line = br.readLine()) != null) {
        System.out.println("Gelesen: " + line);
        pw.print(line + "\r\n"); pw.flush();
      }
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
```

Aufgabe

Testen Sie den Echo-Server

Objektorientierte Sockets

- objektorientierte Sockets ???
- Java kann Objekte serialisieren, wenn sie das `Serializable`-Interface implementieren
- Idee:
  - ObjectOutputStream auf der einen Seite
  - ObjectInputStream auf der anderen Seite
- Hatten wir schon im Abschnitt Serialisierung von Objekten

Reflection

Einordnung

- Unter *Reflective Programming* oder `Meta-Programming` versteht man Programme, die
  - Aussagen über sich selbst machen und/oder
  - Sich selbst verändern
- Das erste geht in Java sehr einfach, das zweite geht (machen wir aber nicht)
- Dazu Package `java.lang.reflect`
- Während Objekte Instanzen von `java.lang.Object` sind, sind Klassen (Interfaces, Annotationen, Enums) Instanzen von `java.lang.Class`

Erste Verwendung von Class

```
public class ReflectionIntro {
  public static void main(String[] args) {
    S.o.println(new ReflectionIntro().getClass());
    S.o.println(new ReflectionIntro()
                  .getClass().getName());
    S.o.println(new ReflectionIntro().getClass()
                  .getCanonicalName());
    S.o.println(int[].class);
    S.o.println("Huhu".getClass().getCanonicalName())
    S.o.println(new Date().getClass().getCanonicalName
  }
}
```

```
class de.pdbm.reflection.ReflectionIntro
de.pdbm.reflection.ReflectionIntro
de.pdbm.reflection.ReflectionIntro
class [I
java.lang.String
java.util.Date
```

Methoden der Klasse `Class`

- Konstruktoren
- Öffentliche Methoden (eigene und geerbte)
- Öffentliche Variablen (eigene und geerbte)
- Annotationen
- Interfaces
- Oberklasse
- Package
- ...

Beispiel Methoden und Variablen

```
for (Method method : this.getClass().getMethods())
  System.out.println("Methode " + method);
}

for (Field field : this.getClass().getFields()) {
  System.out.println("Field " + field);
}
```

Beispiel eigene Methoden und Variablen

```
System.out.println("Klasse: " + c.getName());
System.out.println("Oberklasse: "
                        + c.getSuperclass().getName());
Field[] fields = c.getDeclaredFields();
System.out.println("Attribute: ");
for (int i = 0; i < fields.length; i++) {
  System.out.println(fields[i].getType()
        + " "  + fields[i].getName());
}
Method[] methods = c.getDeclaredMethods();
System.out.println("Methoden: ");
for (int i = 0; i < methods.length; i++) {
  System.out.println(methods[i].getReturnType()
          + " "  + methods[i].getName());
}
```

Man kann Methoden sogar aufrufen

- Man benötigt Objekt
- Methode
- Und Parameter

```
Class<String> c = (Class<String>)
                    Class.forName("java.lang.String");
Method m = c.getMethod("concat",
                       new Class[] { String.class });
Object ergebnis = m.invoke("huhu",
                           new Object[] { " haha" });
System.out.println("Ergebnis: " + ergebnis);

// falls Objekt explizit erzeugt werden soll:
String string = c.newInstance();
```

Literatur I

- Hanspeter Mössenböck.
  Sprechen Sie Java?
  Dpunkt, 4 Auflage, 2011.
- Cay S. Horstmann and Gary Cornell.
  Core Java, Volume I — Fundamentals.
  Prentice Hall, 8. Auflage, 2008.
- Cay S. Horstmann and Gary Cornell.
  Core Java, Volume II — Advanced Features.
  Prentice Hall, 8. Auflage, 2008.

Literatur II

- Joshua Bloach.
  Effective Java.
  Addison-Wesley, 2. Auflage, 2008.
- Al Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson.
  The Elements of Java Style.
  Cambridge University Press, 2000.