ETH zürich

Informatik II - Übung 06

Katja Wolff
katja.wolff@inf.ethz.ch

Besprechung Übungsblatt 5

U5

1.) Einfach verkettete Listen
2.) Modifizierung verketteter Listen
3.) Sortieren verketteter Listen
4.) Noch ein wachsender Stack (mit Listen)

U5.A1

```
public static List add(List list, int value){
  return new List(value, list);
}
```

```
public static int size(List list){
  if (list == null) return 0;
  return size(list.next) + 1;
}
```

```
public static int sum(List list){
  if (list == null) return 0;
  return list.value + sum(list.next);
}
```

```
public static List last(List list){
  if (list == null) return null;
  if (list.next == null) return list;
  return last(list.next);
}
```

U5.A1

```
public static List sublist(List list, int index) throws
  IndexOutOfBoundsException
{
  if (list == null || index < 0) throw new IndexOutOfBoundsException();
  if (index == 0) return list;
  return sublist(list.next, index-1);
}
```

```
public static int valueAt(List list, int index) throws
  IndexOutOfBoundsException
{
  if (list == null || index < 0) throw new IndexOutOfBoundsException();
  if (index == 0) return list.value;
  return valueAt(list.next, index-1);
}
```

```
public static int index(List list, int value) throws
  NoSuchElementException
{
  if (list == null) throw new NoSuchElementException();
  if (list.value == value) return 0;
  return 1 + index(list.next, value);
}
```

U5.A2-4

2. Implementierung von `append()`, `concat()`, `insertAt()` und `remove()` mithilfe der Methoden aus A1 a.

3.
```
public static List insertSorted(List list, int value) {
  if (list == null) return new List(value, null);
  if (value < list.value) return new List(value, list);
  list.next = insertSorted(list.next, value);
  return list;
}
```

4. Implementierung des Stacks mithilfe von Listen

Coding style

Coding style

Variablen- & Methodennamen (camelCase):

- beginWithLowerCase,
- areVeryDescriptiveAnd
- upperCaseSeparated

Klassennamen:

- Klassen mit Grossbuchstaben: class MeineKlasse{ … }

Kommentare!

- vereinfacht Korrektur **und** euer Programmverständnis

Coding style

Umständlich

```
public boolean empty() {
  if ((Lists.size(list) == 0)) {
      return true;
  }
  return false;
}
```

Einfacher

```
public boolean empty() {
   return Lists.size(list) == 0;
}
```

```
public boolean empty() {
   return list == 0;
}
```

Noch einfacher (in dieser Übung)

Coding style

```
public static List add(List list, int value) {
  List elem = new List(value, list);
  return elem;
}
```

Umständlich

```
public static List add(List list, int value) {
  return new List(value, list);
}
```

Einfacher

Coding style

```
public static int size(List list) {
  if (list == null) return 0;
  if (list.next == null) return 1;
  return 1 + size(list.next);
}
```

Umständlich

```
public static int size(List list) {
  if (list == null) return 0;
  return 1 + size(list.next);
}
```

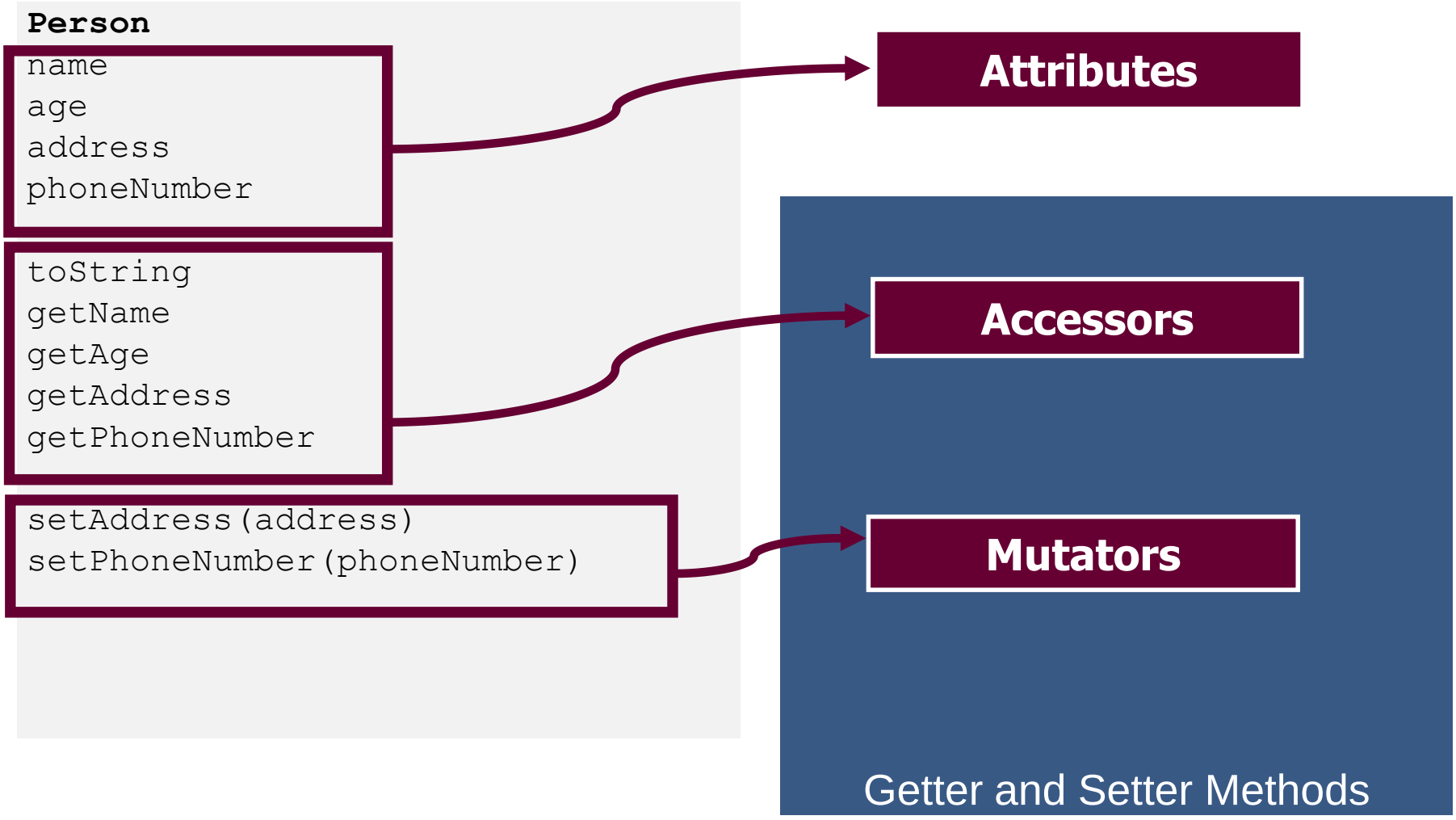
Einfacher

Objektorientierte Programmierung

Outline

- Person – Beispiel einer Klasse
- Inheritance
- Type compatibility
- Polymorphismus
- `instanceof`
- Visibility Regeln
- Constructor and `super()`
- Final Methoden und Klassen
- abstrakte Klassen und Methoden
- Interfaces
- abstrakte Klassen vs. interfaces

Class „Person“



Class „Person“: Implementation

```
public class Person {
   private String name;
   private int age;
   private String address;
   private String phone;

   public Person(String name, int age,
                 String address, String phone) {
       this.name = name; this.age = age;
       this.address = address; this.phone = phone;
   }

   public String toString() {
       return getName() + " is " + getAge() +
                          "old and lives in " + getAddress();
   }

   public String getName() { return name; }
   public int getAge() { return age; }
   public String getAddress() { return address; }
   public String getPhoneNumber() { return phone; }

   .....
```

Class „Student“

Person
name
age
address
phoneNumber

toString()
getName()
getAge()
getAddress()
getPhoneNumber()

setAddress(newAddress)
setPhoneNumber(newPhoneNumber)

Student
name
age
address
phoneNumber
legi

toString()
getName()
getAge()
getAddress()
getPhoneNumber()
getLegi()

setAddress(newAddress)
setPhoneNumber(newPhoneNumber)

Class „Student“ erbt von „Person“

```
public class Student extends Person {

    private String legi;

    public Student(String name, int age,
                   String address, String phone, String legi){
        super(name, age, address, phone);
        this.legi = legi;
    }

    public String toString() {
        return getName() + " is " + getAge() + "old, lives in " +
               getAddress() + " and has legi-nr.: " + getLegi();
    }

    public String getLegi() { return legi; }
}
```

Student
- erweitert ein bestehendes Konzept, die Klasse Person
- enthält ein zusätzliches Feld: legi mit Getter: getLegi()

Class „Student“ erbt von „Person“

```
public class Student extends Person {

    private String legi;

    public Student(String name, int age,
                   String address, String phone, String legi){
        super(name, age, address, phone);
        this.legi = legi;
    }

    public String toString() {
        return getName() + " is " + getAge() + "old, lives in " +
               getAddress() + " and has legi-nr.: " + getLegi();
    }

    public String getLegi() { return legi; }
}
```

Student
- definiert ein Konstruktor
- ruft Basisklassenkonstruktor mit **`super`** auf

Class „Student“ erbt von „Person“

```
public class Student extends Person {

    private String legi;

    public Student(String name, int age,
                   String address, String phone, String legi){
        super(name, age, address, phone);
        this.legi = legi;
    }

    public String toString() {
        return getName() + " is " + getAge() + "old, lives in " +
               getAddress() + " and has legi-nr.: " + getLegi();
    }

    public String getLegi() { return legi; }
}
```

Student
- definiert die Methode toString() neu.

Inheritance: `Student extends Person`

Student can:

- Add new fields
  - legi
- Add new methods
  - getLegi()
- Override existing methods
  - toString()

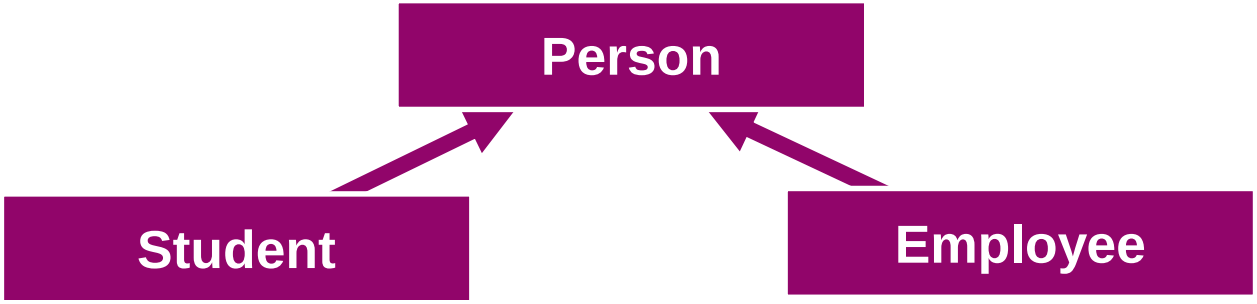
Student cannot

- Remove fields
- Remove methods
- Override private methods

Why inheritance?

- Better design
- Code reuse
- Code „maintenance”
- Abstraction of the real world

Static and dynamic casts

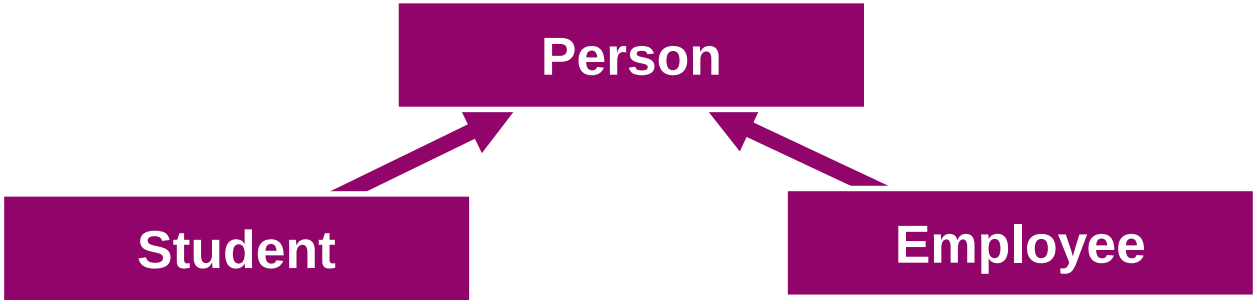



```
Person p = new Person(...);
Student s = new Student(...);
Employee e = new Employee(...);
```

```
Person ps = s
```
→ ok
```
Person pe = e
```
→ ok
```
Student sp = p
```
→ compile error

```
Student sps = ps
```
→ compile error
```
Student dsps = (Student) ps
```
→ ok
```
Employee deps = (Employee) ps
```
→ runtime error

instanceof




```
Person p = new Person(...);
Student s = new Student(...);
Employee e = new Employee(...);
```

`p instanceof Person`
→ true
`p instanceof Student`
→ false

`s instanceof Person`
→ true
`s instanceof Student`
→ true

The Object class in Java

- Object class in Java
  - Superclass for all other classes in Java
  - This does not include base types (char, int, float, etc.): They are not classes!

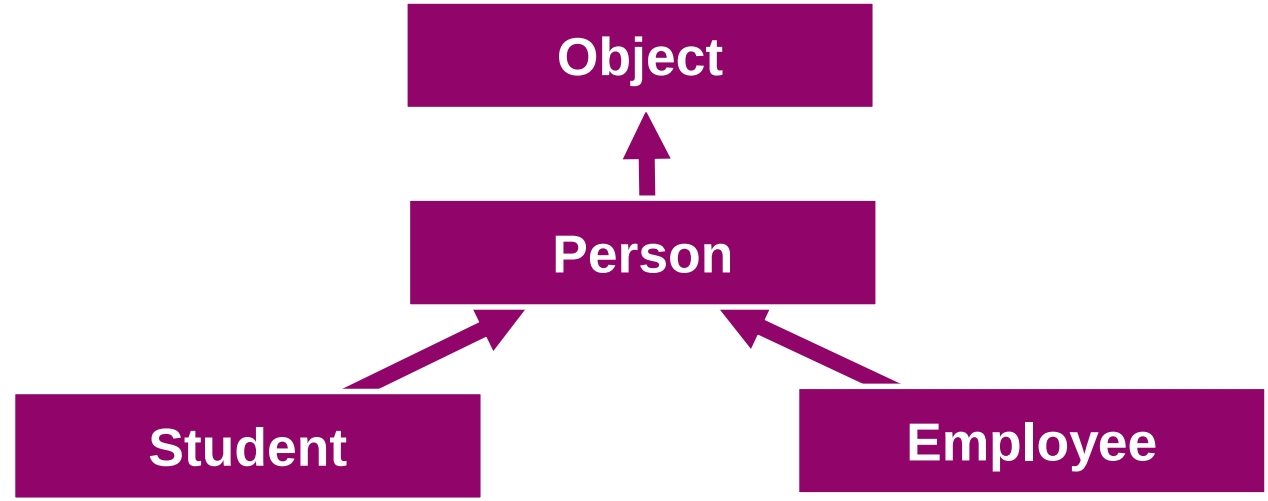
- When a class is defined in Java, the inheritance from the `Object` class is implicit, therefore:

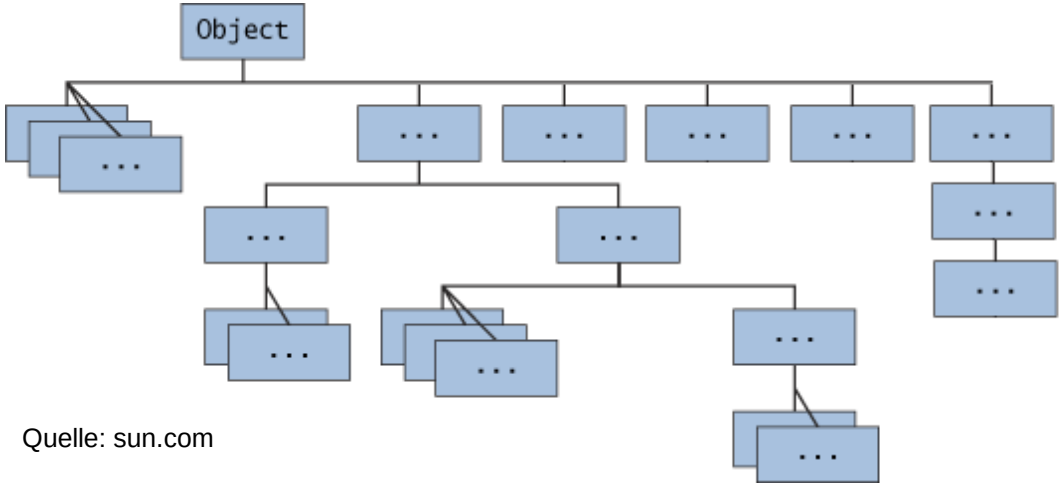
```
public class MyClass {
        ......
}
```

- is equivalent to:

```
public class MyClass extends Object {
        ......
}
```

The Object class in Java





Quelle: sun.com

Visibility rules

- private members
  - Private members in the base class are not accessible to the derived class, and also not to anyone else
- protected members
  - Protected members are visible to methods in a derived class and also methods in classes in the same package, but not to anyone outside
- public members
  - Everyone

Final Methods and Classes

- A derived class
  - Can accept the base class methods
  - Or can override the base class methods

  - Javadoc: *@Override*

- A method declared as final in the base class cannot be overridden by any derived class
- A final class cannot be extended!
  - E.g. Integer, Character,...

Abstract classes

- Abstract method
  - Is a method that all derived classes must implement
- Abstract class
  - A class that has at least one abstract method
- If a class derived from an abstract class fails to override an abstract method, the compiler will detect an error

Interface

- The interface in Java is the ultimate abstract class
- *A class can implement many interfaces*
- A class implements an interface if it provides definitions for all the methods „declared“ in the interface
- So, both abstract classes and interface provide a specification of what subclasses must do

Abstract classes vs. Interfaces

Abstract class

- An abstract class can provide complete code, default code, and/or just stubs that have to be overridden
- May declare methods as protected abstract
- A class may extend only one abstract class

Interface

- An interface cannot provide any code, much less default code
- All methods declared are implicitly public abstract
- A class may implement several interfaces

- http://java.sun.com/docs/books/tutorial/java/landl/index.html

Example: Interface IStack

```
public interface IStack {
        int size();
        void push(Object obj);
        Object pop();
        Object peek();
        boolean empty();
}
```

```
public class MyStack implements IStack {
  private int size;

  public int size() {
    return size;
  }

  public void push(Object obj) {
    ...
  }
  ...

}
```

Example: Abstract class BaseStack

```
public abstract class BaseStack implements IStack {
        public abstract int size();
        public abstract void push(Object obj);
        public abstract Object pop();
        public Object peek()
                { Object top = pop(); push(top); return top; }
        public boolean empty() { return size() == 0; }
}
```

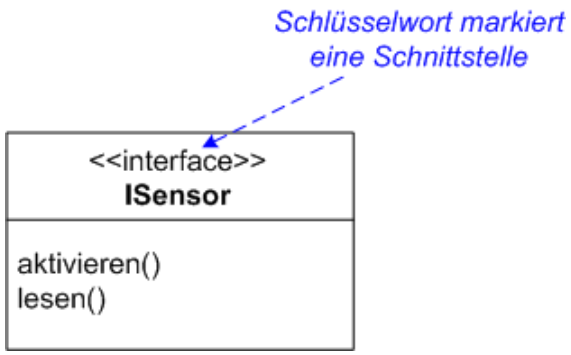
```
public class MyStack extends BaseStack {
 private GenericList first;

 public Object peek() {
   return first.value;
 }

 ...

}
```

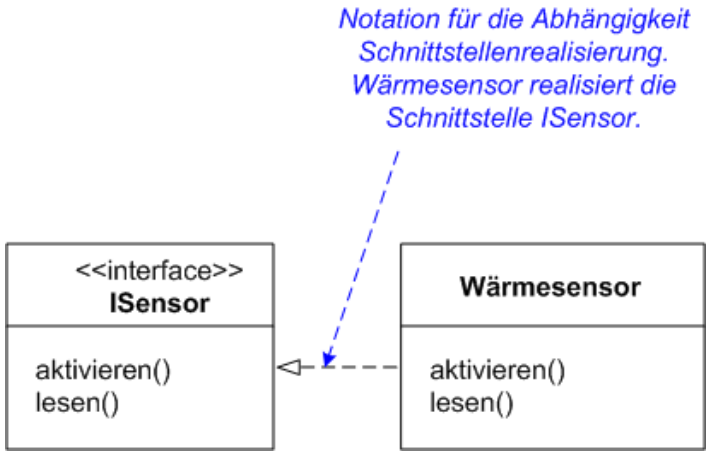
Hinweise zu Blatt 06

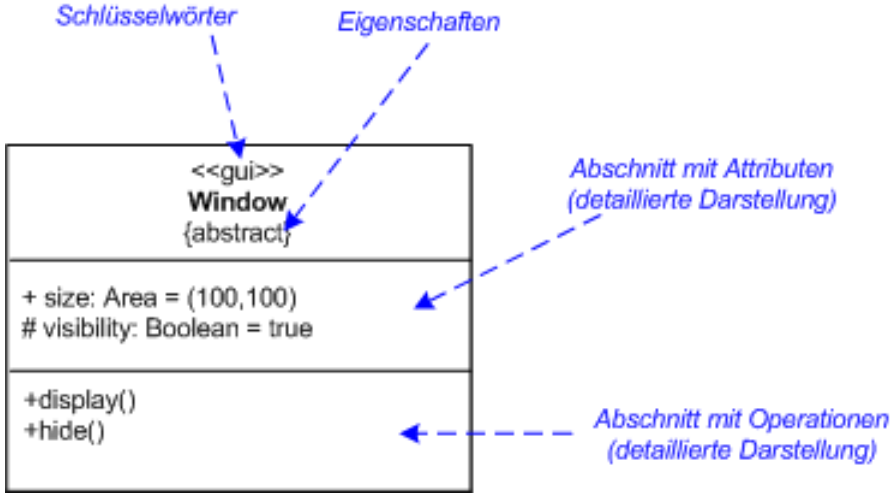
Überblick

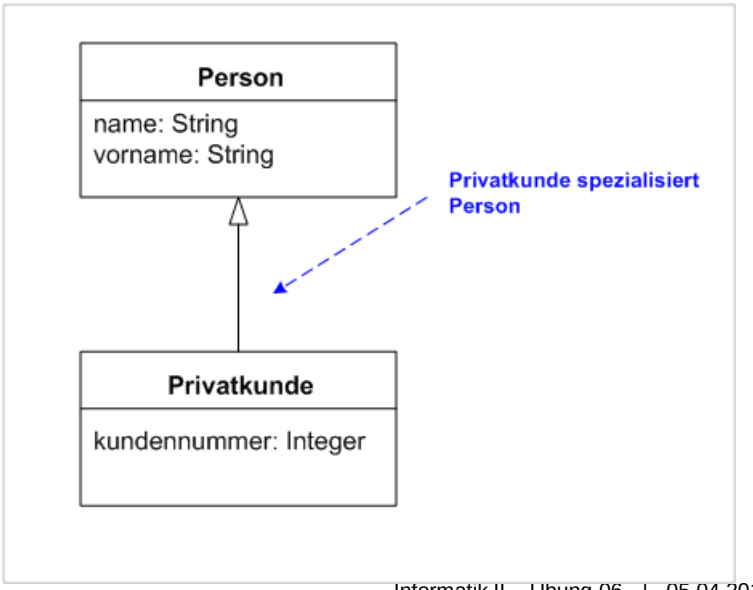
- A1: Klassen, Schnittstellen
- A2: Schnittstellen und Implementierungen
- A3: Polymorphie
- A4: Wieder ein Stack (Advanced, freiwillig)

Tipps zu U6.A1: UML









http://de.wikipedia.org/wiki/Klassendiagramm

U6.A2: Fabrikmethode

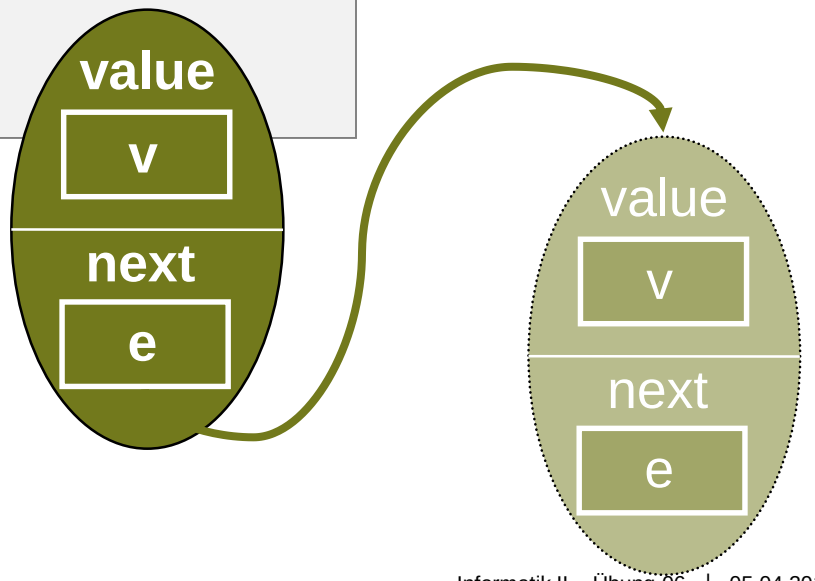
- Die **Fabrikmethode** erstellt ein Objekt, das ein bestimmtes Interface implementiert - die innere Funktionsweise des Objektes ist jedoch versteckt.
  - *Entwickler 1* implementiert verschiedene Listen, die alle das Interface *IList* verwirklichen
  - *Entwickler 2* benutzt Listen, die Funktionsweise ist ihm aber verborgen. **Problem:** Wenn *Entwickler 1* eine neue Implementierung schreibt, muss *Entwickler 2* alle Stellen `new ListA()` auf `new ListB()` umschreiben.
  - **Lösung:** *Entwickler 1* stellt eine Fabrikmethode zur Verfügung, die *Entwickler 2* zur Erzeugung von Listen nutzt: `Factory.giveMeNewList()` – Als Ergebnis erhält er eine Implementierung des Interfaces *IList*.

U6.A2: Generische Listen

- Übungsserie 5
  - Elemente der Liste: Ganzzahlen
    - *int*
- Übungsserie 6
  - Elemente der Liste: generische Objekte
    - *Object*
- Selbst eine Utility-Klasse erstellen: *ListUtils*
  - `implements IListUtils`: handhabt generische Listen
    - Vergleiche mit den Utility-Klassen aus A1 & A3 von Übung 5
    - Diesmal wird die Utility Klasse instantiiert (nicht mehr static)

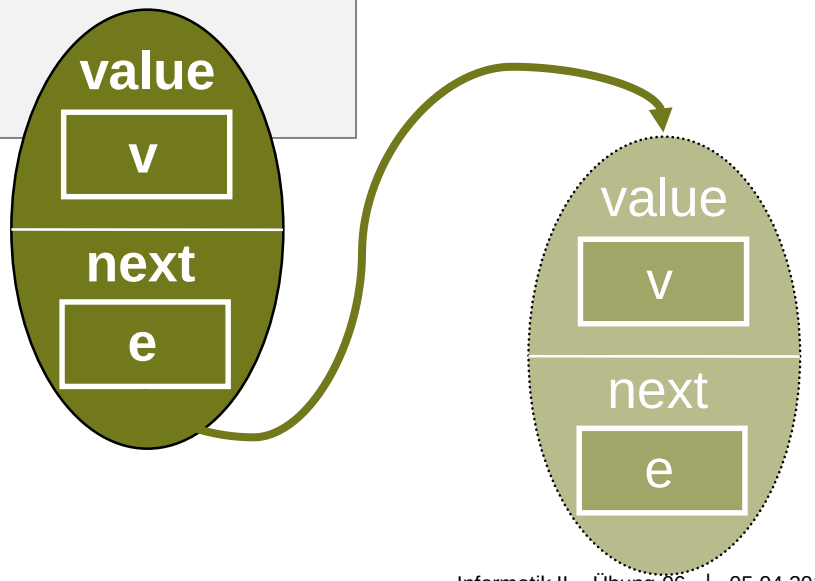
U6.A3: Generische Listen

```
class List {
    int value;
    List next;

    public List(int v, List e){
        value = v;
        next = e;
    }
}
```



U6.A3: Generische Listen

```
class GenericList {
    Object value;
    GenericList next;

    public GenericList(Object v, GenericList e){
        value = v;
        next = e;
    }
}
```



Tipps zu U6.A3

- Methoden sind nicht mehr static!
- U5.A1: `toString, add, size`
  - können fast übernommen werden
- U5.A3: `sort`
  - Ähnliche Idee
  - Minimale Anpassung, da es sich statt `int` um generische `Object`-Elemente handelt.
  - Wie sortiere ich etwas, das ich nicht kenne?
  - **Lösung:** Interface Comparable, cast nach `Comparable` möglich

```
public interface Comparable
{
   boolean smallerThan(Comparable rhs);
}
```

Tipps zu U6.A4

- Schwere Aufgabe
- Selbsttest: Wer das kann, hat keine Programmierprobleme in der Prüfung ☺
- Vereint die Effizienz von Arrays mit dem aufwandslosen Wachsen von Listen
- Implementierung des Interface *Istack*
  - → Es kann `u6a2.StackFactory.create()` benutzt werden

…viel Spass!