Overview

- Debriefing Exercise 5
- Briefing Exercise 6
U05 Some Hints

Variables & Methods

- `beginWithLowerCase`,
- `areVeryDescriptiveAnd`
- `upperCaseSeparated`
- `aMethodWhichHasAVeryLongName()`
- `Classes with capital letters: class MyClass{ … }`
Object Oriented Programming
Class Person

- **Attributes**
  - Name
  - Age
  - Address
  - PhoneNumber

- **Accessors**
  - getName
  - getAge
  - getAddress
  - getPhoneNumber

- **Mutators**
  - setAddress(newAddress)
  - setPhoneNumber(newPhoneNumber)

Getter and Setter Methods
How do we implement class “Person”?

```java
public class Person {
    private String m_name;
    private int m_age;
    private String m_address;
    private String m_phone;

    public Person(String name, int age,
                   String address, String phone) {
        m_name = name; m_age = age;
        m_address = address; m_phone = phone;
    }

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

    public String getName() { return m_name; }
    public int getAge() { return m_age; }
    public String getAddress() { return m_address; }
    public String getPhoneNumber() { return m_phone; }
}
.....
```
What about students?

<table>
<thead>
<tr>
<th>Student</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Age</td>
<td>Age</td>
</tr>
<tr>
<td>Address</td>
<td>Address</td>
</tr>
<tr>
<td>PhoneNumber</td>
<td>PhoneNumber</td>
</tr>
</tbody>
</table>

**Legi**

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

- `toString()`  
- `getName()`   
- `getAge()`     
- `getAddress()` 
- `getPhoneNumber()`  
- `setAddress(newAddress)`
- `setPhoneNumber(newPhoneNumber)`
Student class

- defines a constructor
- calls the basis class constructor through the usage of `super`

```java
public class Student extends Person {

    private String m_legi;

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

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

    public String getLegi() { return m_legi; }
}
```
Inheritance

- Student extends Person

- Student can:
  - Add new fields: m_legi
  - Add new methods: getLegi()
  - Override existing methods: toString()

- Student cannot:
  - Remove fields
  - Remove methods
Why inheritance?

- Better design
- Code reuse
- Code «maintenance»
- Abstraction of the real world
Inheritance

Everything is an object!

Object

Animal
(Silently extends Object)

Mammal
(extends Animal)

Dog
(extends Mammal)

Cat
(extends Mammal)

upcasting

downcasting

by Sinipull for codecall.net
Upcasting

Cat c = new Cat();
System.out.println(c);
Mammal m = c; // upcasting
System.out.println(m);

/*
This printed:
Cat@a90653
Cat@a90653
*/

- Cat is still exactly the same Cat after upcasting, it didn't change to a Mammal, it's just being labeled Mammal right now. This is allowed, because Cat is a Mammal.

- Upcasting is done automatically, no need to do it manually
I'm still a Cat after upcasting, but compiler treats me as an Object

That means I can't do anything, that's specific for Animals... or Cats.

by Sinipull for codecall.net
Downcasting

Cat c1 = new Cat();
Animal a = c1; //automatic upcasting to Animal
Cat c2 = (Cat) a; //manual downcasting back to a Cat

- Downcasting must be done manually!
- Why?
  - Multiple child classes
Downcasting

I can’t, because you don’t know if i’m a Cat, you must downcast me before i can do it.

by Sinipull for codecall.net
Static & Dynamic Casting

Person

Student

Employee

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 dps = (Student) ps  
Ok (casting from base class to derived class)

Employee deps = (Employee) ps  
Runtime error (ps points to object of class Student)
Static & Dynamic Casting

Person

Student

Employee

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
Object class in Java

- Is a superclass for all other classes defined in Java's class libraries, as well as for user-defined Java classes.
- This does not include primitive 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:

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

- is equivalent to:

```java
public class MyClass extends Object {
    ......
}
```
Object class in Java

Object

Person

Student

Employee

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

- 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
  - Eclipse provides help!
Interfaces

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.

But....
Abstract class vs. interface

```
Animal
  hunger
  isAwake
  eat()
  getHunger()
  sleep()
  wakeUp()
  isAwake()
  makeNoise()
  roam()

Feline
  livesLeft
  getLivesLeft()
  roam()

Cat
  eat()
  makeNoise()

Lion
  makeNoise()
  sit()
  stand()

Canine
  roam()

YipYip
  sit()
  stand()

Dog
  makeNoise()
  sit()
  stand()

Wolf
  makeNoise()
```
## Abstract class vs. Interface

<table>
<thead>
<tr>
<th>Abstract class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An abstract class can provide complete code, default code, and/or just stubs that have to be overridden</td>
<td>- An interface cannot provide any code, much less default code</td>
</tr>
<tr>
<td>- May declare methods as protected abstract</td>
<td>- All methods declared are implicitly public abstract</td>
</tr>
<tr>
<td>- A class may extend <strong>only one</strong> abstract class</td>
<td>- A class may implement several interfaces</td>
</tr>
</tbody>
</table>

Example interface

```
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

```java
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; }
}
```

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

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

    ...
}
```
Overview

- Debriefing Exercise 5
- Briefing Exercise 6
U06

- Q1: Classes, Interfaces and Casts
- Q2: Interfaces and their implementation
- Q3: Polymorphism
- Q4: ChunkedStack (optional)
Hints

A keyword represents an interface

```
<<interface>>
ISensor
aktivieren()
lesen()
```

Notation for the dependance of the instantiation of the interface. Thermal sensor instantiates the iSensor interface

```
<<interface>>
ISensor
aktivieren()
lesen()
```
```
Wärmesensor
aktivieren()
lesen()
```

Keyword

Property

Section with attributes (detailed representation)

```
<<gui>>
Window
{abstract}
+ size: Area = (100, 100)
# visibility: Boolean = true
+ display()
+ hide()
```

Section with operations (detailed representation)

```
Person
name: String
vorneame: String
```
```
Privatkunde
kundennummer: Integer
```

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

Private client as a specialized Person
A factory method builds an object which implements a certain interface, but the inner functionality of the object is hidden.

→ Programmer 1 implements different lists which implement the `IList` interface.

→ Programmer 2 uses lists but doesn’t want to know about the functionality. When Programmer 1 writes a new implementation, Programmer2 has to rewrite all lines of `new ListA()` as `new ListB()`.

→ Programmer 1 puts a factory method at disposal and Programmer 2 can always call for example `Factory.giveMeNewList()` and gets an object from the newest implementation of the `IList` interface.
Factory Method

Get me a Car

Car Factory

Car V1 Factory

Car VII Factory

Car GenX Factory

Car Prototype

Car V1

Car VII

Car GenX

ScriptS.GE
List of integers

class List {
    int value;
    List next;

    public List(int v, List e) {
        value = v;
        next = e;
    }
}
class GenericList {
    Object value;
    GenericList next;

    public GenericList(Object v, GenericList e) {
        value = v;
        next = e;
    }
}
U06.A03 a and c

- Methods are not static anymore!
  - Ex5.Q1: toString, add, size
  - Can be easily passed

- Ex5.Q3: sort
  - Similar (same idea)
  - Minimal interfacing. This way, it can work with generic objects

- Interface Comparable

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

- You can cast as Comparable without checking its type
U06.A04 Stacks again!

- Optional, so not trivial

- Self-test: Whoever can do it will have no problem during the exam.

- Combines the efficiency of arrays to the effortless growth of lists

- Implement according to the interface

- Performance analysis
Have Fun!
ArrayList and Generics

- Each group consists of multiple students:
  \[\text{ArrayList}\langle\text{Student}\rangle\ \text{group}\]
- There are multiple groups of students:
  \[\text{ArrayList}\langle\text{ArrayList}\langle\text{Student}\rangle\rangle\ \text{groups};\]

a) Implement factory method
b) Implement filterRaw (without generics: ArrayList)
c) Implement filterGeneric (using Generics: ArrayList\langle\text{Student}\rangle)