E Zusammenfassung der 1. Übung

- Java Überblick
  - Java vs. C/C++
    kein Präprozessor, streng getyped, keine Zeiger Arithmetik, keine Prozeduren, ...
- Grundlagen der OO mit Java
  - Abstraktion and Kapselung
  - Objekte
    - Klassen
    - Methoden und Variablen
    - Konstruktoren
    - Referenzen, Gleichheit und Identität
  - Vererbung

F Überblick über die 2. Übung

- Lösung der 1. Aufgabe
- weitere OO-Konzepte mit Java
  - abstrakte Klassen
  - Interfaces
  - Packages
  - Sichtbarkeitsattribute
  - statische Elemente
  - Konstanten
  - innere Klassen
- JUnit
- Hinweise zur 2. Aufgabe
  - AWT-Events

F.1 Lösung der 1. Aufgabe

- Shape.java
- Circle.java
- WhiteBoard.java

1 Shape.java

```
import java.awt.Graphics;

public interface Shape {
        void move(int x, int y);
        int getX();
        int getY();
        boolean isInside(int x, int y);
        void draw(Graphics g);
}
```

2 Circle.java

```
import java.awt.Graphics;

public class Circle implements Shape {

        int x,y,diameter;

        public Circle ()
                { this(30,20,100); }

        public Circle (int x, int y, int diameter){
                this.x = x;
                this.y = y;
                this.diameter = diameter;
        }
        public void move(int x, int y){
                this.x = x;
                this.y = y;
        }
        public int getX()
                { return x; }

        public int getY()
                { return y; }
                                                //- Fortsetzung folgt -
```


3 WhiteBoard.java

```
import java.awt.Frame;
import java.awt.Graphics;

public class WhiteBoard extends Frame {
        Shape shapes[] = new Shape[100];

        public WhiteBoard (){
                super("WhiteBoard");
                setSize(400,300);
                setVisible(true);
        }

        public void addShape(Shape shape){
                for(int i=0; i < shapes.length; i++) {
                        if (shapes[i] == null) {
                                shapes[i] = shape;
                                break;
                        }
                }
                repaint();
        }


                                                //- Fortsetzung folgt -
```


2 Circle.java (2)

```
        public boolean isInside(int x, int y){
                double radius = diameter/2;
                double centerX = this.x + radius;
                double centerY = this.y + radius;
                double distanceSquared = (centerX-x)*(centerX-x) +
                                         (centerY-y)*(centerY-y);

                return (Math.sqrt(distanceSquared) <= radius);
         }

        public void draw(Graphics g){
                g.drawOval(x,y,diameter,diameter);
         }

}// Circle
```


3 WhiteBoard.java (2)

```
        public void addShape(String classname, String xstr, String ystr) {
                try {
                        Class shapeClass = Class.forName(classname);
                        Shape shape = (Shape) shapeClass.newInstance();
                        int x = Integer.parseInt(xstr);
                        int y = Integer.parseInt(ystr);
                        shape.move(x,y);
                        addShape(shape);
                } catch(Exception e) {
                        System.out.println(e);
                }
        }


        public void paint(Graphics g){
                for(int i=0; i < shapes.length; i++) {
                        if(shapes[i] != null){
                                shapes[i].draw(g);
                        }
                }
        }

                                                //- Fortsetzung folgt -
```

3 WhiteBoard.java (3)

```
        public static void main(String [] args){
                WhiteBoard wb = new WhiteBoard();
                for(int i=0; i<args.length-2; i+=3) {
                        wb.addShape(args[i], args[i+1], args[i+2]);
                }
        }

}
```


G.1 Abstrakte Klassen

- Problem:
  - WhiteBoard mit geometrischen Formen (Kreis, Rechteck, Linie)
  - Zeichenbereich **`Sheet`** kann nur mit **`Shape`** Objekten umgehen
  - **`Sheet`** muss **`draw()`** am **`Shape`** aufrufen
  - **`Shape`** kann keine sinnvolle Implementierung von **`draw()`** bereitstellen
- **`Shape`** ist eine *abstrakte* Klasse

G Weitere OO-Konzepte in Java

- abstrakte Klassen
- Interfaces
- Modularität: Packages & Sichtbarkeitsattribute
- statische Elemente
- Konstanten (**`final`** Methoden, **`final`** Klassen)
- innere Klassen

G.1 Abstrakte Klassen

- ...werden verwendet um gemeinsame Eigenschaften von Klassen herauszuarbeiten und zusammenzufassen.
- ...können **nicht** zur Erzeugung von Objekten verwendet werden.
- ...enthalten Methoden ohne Implementierung (*abstrakte Methoden*)
- Abstrakte Klassen und Methoden werden mit dem Schlüsselwort **`abstract`** deklariert:

```
abstract class Shape {
    public abstract void draw();
}
```

- Wenn eine konkrete Klasse von einer abstrakten Klasse abgeleitet wird, so müssen alle abstrakten Methoden implementiert sein:

```
class Circle extends Shape {
    public void draw() { ... }
}
```

G.2 Interfaces

- Java trennt das Klassenkonzept vom Typkonzept
- Interfaces repräsentieren Typen
- Interfaces enthalten
  - Methodennamen und -signaturen
  - Konstanten (**`static final`**)
- alle Methoden sind (implizit) abstrakt
- Klassen können zu Interfaces kompatibel sein (Schlüsselwort **`implements`**)
- Diese Klassen müssen alle Methoden des Interfaces implementieren.
- Definition einer Schnittstelle mit dem Schlüsselwort **`interface`**

1 Beispiel

1. Definition einer Schnittstelle:

```
public interface Printable {
    public void print();
}
```

2. Definition einer Klasse, welche die Schnittstelle implementiert:

```
public class Account implements Printable {
    ...
    public void print() {
        System.out.println("balance="+balance());
    }
}
public class Person implements Printable { ... }
```


1 Beispiel (2)

3. Verwendung der Interfaces:

```
public class PrintQueue {
    public void add(Printable p ) { ... }
}
....
PrintQueue queue = new PrintQueue();
Printable p=new Person(...);
queue.add(p);
Account account = new Account(...);
queue.add(account);
```


2 Vererbung von Interfaces

- Interfaces können mehrere andere Interfaces *erben*, Klassen können mehrere Interfaces *implementieren*.
- Beispiel:

```
interface Streamable extends FileIO, Printable {
    // zusätzliche Methoden
}

class Test implements Streamable, TestInterface {
    ...
}

class Test1 extends Test {
    ...
}
// Test1 ist zu FileIO, Printable, Streamable,
// TestInterface und zur Klasse Test kompatibel
```

3 Abstrakte Klassen vs. Interfaces

- Abstrakte Klassen können eine partielle Implementierung einer Abstraktion bereitstellen.
- Abstrakte Klassen können Instanzvariablen enthalten.
- Ein abstrakte Klasse sollte verwendet werden, wenn nur ein Teil der Implementierung offen bleiben soll.
- Ein Interface ist gut geeignet um bestimmte Eigenschaften zu repräsentieren. (Printable, Clonable,...)

G.3 Packages

- Klassen lassen sich in Pakete (packages) zusammenfassen.
- Paket = Programm-Modul
  - mit eindeutigem Namen (z.B. **`java.lang`** oder **`java.awt.image`**)
  - enthält eine oder mehreren Klassen
- Pakete partitionieren den Namensraum.

1 Schlüsselwort package

- Packages werden mit **`package`** deklariert:

```
package test;
public class TestClass ...
```

- **`package`** muss die erste Anweisung in einer Datei sein.
- Hierarchien von Packages sind möglich:

```
package test.unittest1;
```


2 Schlüsselwort: import

- Klassen anderer Packages können mit **`import`** verwendet werden:

```
import java.util.*; // use all classes from package java.util
import java.io.File;// use class File from package java.io
```

- Bei der Suche von Klassen wird der Package-Name als Verzeichnis verwendet.
- Beispiel:
  - Package **`bank`** mit Klasse **`Customer`**
  - kann verwendet werden mit **`import bank.Customer`**
  - Bytecode-Datei wird gesucht als **`bank/Customer.class`**
- Package **`java.lang.*`** wird automatisch importiert.
- Zugriff auf Klassen ohne import: durch Verwendung des vollständigen Klassennamens, inklusive Package.

2 Schlüsselwort: import - Beispiel

Datei `editor/shapes/X.java`

```
package editor.shapes;
public class X {
    public void test() {
        System.out.println("X");
    }
}
```

Datei `editor/filters/Y.java`

```
package editor.filters;
import editor.shapes.*;
class Y {
    X x;
    void test1() { x.test();}
}
```

Datei `editor/filters/Z.java`

```
package editor.filters;
class Z {
    editor.shapes.X x;
    void test1() { x.test();}
}
```


3 Packages und der CLASSPATH

- Klassen werden mit Hilfe der Umgebungsvariable CLASSPATH gesucht.
- Beispiel:
  - Package **`bank`** mit Klasse **`Customer`**
  - wird verwenden mit **`import bank.Customer;`**
  - Compiler und Interpreter suchen die Bytecode-Datei: **`bank/Customer.class`**
  - **`CLASSPATH`** enthält **`/proj/test:/tmp`**
  - gesucht wird nach:
    - **`/proj/test/bank/Customer.class`**
    - **`/tmp/bank/Customer.class`**
  - beim kompilieren kann man das Zielverzeichnis angeben:
    - **`javac -d /proj/test Customer.java`**

4 Standard Java Pakete

- **`java.lang`**: fundamentale Java Klassen (Thread, String,...)
- **`java.io`**: Ein- / Ausgabe Unterstützung (Files, Streams,...)
- **`java.net`**: Netzwerk Unterstützung (Sockets, URLs, ...)
- **`java.awt`**: GUI Unterstützung (Abstract Windowing Toolkit)
- **`java.applet`**: Applet Unterstützung
- **`java.util`**: Hilfsklassen (Random) und Datenstrukturen (Vector, Stack)
- **`java.rmi`**: Remote Method Invocation
- **`java.beans`**: Unterstützung für Java Beans
- **`java.sql`**: Unterstützung zum Zugriff auf Datenbanken
- **`java.security`**: kryptographische Unterstützung

4 Standard Java Pakete

- Nähere Informationen unter:

`http://www4/Services/Doc/Java/jdk-1.4/docs/api/index.html`

G.4 Sichtbarkeitsattribute

- *Kapselung* ist eines der Grundprinzipien objektorientierter Programmierung.
- Kapselung wird zum verstecken unnötiger Information verwendet (*information hiding*).

1 Sichtbarkeitsattribute - Klassen

- Eine Klasse kann öffentlich (**`public`**) oder nicht öffentlich sein.

```
public class X { ... }

class X { ... }
```

- **`public`** Klassen sind außerhalb des Package verfügbar.
- Klassen ohne **`public`**-Deklaration (private Klassen) sind nur innerhalb desselben Package sichtbar.
- Eine public-Klasse muss in einer eigenen Datei deklariert werden.
  Dateiname := Klassenname + ".java"
  Beispiel: Klasse **`X`** muss in der Datei **`X.java`** definiert werden.

2 Sichtbarkeitsattribute - Klassenelemente

- Sichtbarkeitsattribute für Methoden und Variablen:
  - public, default, protected, private
- Wirkung:
  - **`public`**: global sichtbar
  - default: innerhalb des gleichen Packages sichtbar
  - **`protected`**: innerhalb des gleichen Packages und in Unterklassen sichtbar.
  - **`private`**: nur innerhalb der gleichen Klasse sichtbar
- Sichtbarkeitsattribute müssen bei *jeder* Methode bzw. Instanzvariable extra angegeben werden.

2 Sichtbarkeitsattribute - Klassenelemente (2)

- Übersicht:

	Sichtbarkeitsattribute			
sichtbar in	**`public`**	**`protected`**	default	**`private`**
gleiche Klasse	ja	ja	ja	ja
gleiches Package	ja	ja	ja	nein
Unterklassen	ja	ja	nein	nein
andere Packages	ja	nein	nein	nein

3 Kapselung

- ohne Kapselung:

```
class Person {
    public String name;// "name" can be modified/read globally
}
```

- besser:

```
class Person {
    private String name;     // only Person can access "name"
    public String getName() {           // access method
        return name;
    }
}
```


2 Klassenkonstruktor

- Klassen haben einen Zustand (**`static`** Variablen) und müssen deshalb durch einen Klassenkonstruktor initialisiert werden.
- Beispiel:

```
class Test {
    static int counter;
    static {
        counter = 9;
    }
}
```


G.5 statische Elemente

1 Klassenvariablen und Klassenmethoden

- Klassen können Variablen und Methoden enthalten: statische Variablen und Methoden
- Diese können ohne Objekt der Klasse genutzt werden:

```
class Test {
    private static int counter = 0;
    public Test() { counter++; }
    public static int howMany() { return counter; }
}
Test t = new Test();
System.out.println("No. of Test objects: " + Test.howMany());
```

- weiteres Beispiel:
  - **`System.out`** ist eine statische Variable der Klasse **`System`**.

G.6 Konstanten

- Variablen können konstant (*final*) sein:
  - Sie müssen entweder bei der Deklaration (JDK 1.0) oder später (JDK 1.1: *blank finals*) initialisiert werden.
  - Dieser initiale Wert kann nicht verändert werden.
- Beispiel:

```
class Test {
    public static final int x=5; // konstante Klassenvariable
    private final int t=10;      // konstante Instanzvariable
}
```

G.6 Konstanten (2)

- Seit Java 1.1 können die Parameter von Methoden und lokale Variablen konstant sein.
- Beispiel:

```
class Test {
    String name;
    void setName(final String name) {
       final int i = 42;
       this.name = name;
    }
}
```


1 Final Methoden

- Wenn Methoden als **`final`** deklariert werden, können sie nicht überschrieben oder verdeckt werden
  - Sicherheit
  - Effizienz (inlining ist möglich)

```
class Test {
    final void hello() {...}
}

class Test2 extends Test {
    void hello() { ... } // Fehler!! hello ist final in Test
}
```


2 Final Klassen

- Wenn Klassen als **`final`** deklariert werden, kann man keine Klassen davon ableiten.

Beispiel:

```
public final class Test {
    ...
}

public class Test2 extends Test {//Fehler!! Test ist final
    ...
}
```


G.7 Innere Klassen

- **local inner class**: nur von der umschliessenden Klasse nutzbar
- **inner class with method scope**: nur innerhalb der Methode nutzbar
- **anonymous inner class**: nur bei der Definition nutzbar
- **static inner class**: global nutzbar

1 Lokale Innere Klassen

- Innere Klassen können auf Instanzvariablen der umschliessenden Klasse zugreifen:

```
class Test {
  private String array[] = { "hans", "otto", "max"};
  class Inner {
    String method(int i) { return "Name:+" + array[i]; }
  }
}
```

- Sichtbarkeitsattribute wie für Methoden und Instanzvariablen (**`private`**, default, **`protected`**, **`public`**)

```
class Test {
  private String array[] = { "hans", "otto", "max"};
  private class MyEnum implements Enumeration {
    private int counter = 0;
    public Object nextElement() { return array[counter++]; }
    public boolean hasMoreElements() { return counter<array.length; }
  }
  public Enumeration enumerate() { return new MyEnum(); }
}
```


2 Innere Klassen innerhalb einer Methode

- Klasse, die nur innerhalb einer Methode gebraucht wird:

```
class Test {
  private String array[] = { "hans", "otto", "max"};
  public Enumeration enumerate() {
    class MyEnum implements Enumeration {
      private int counter = 0;
      public Object nextElement() { return array[counter++]; }
      public boolean hasMoreElements() { return counter<array.length; }
    }
    return new MyEnum();
  }
}
```


2 Innere Klassen innerhalb einer Methode (2)

- Kann auf konstante (**`final`**) Parameter und konstante (**`final`**) lokale Variablen der umschliessenden Methode zugreifen
- Zugriff auf **`this`** der umschliessenden Klasse **`X`** mit **`X.this`**:

```
class Test {
  public void test(final String msg) {
    class Inner {
     public void output(String hello) { System.out.println(hello+msg);}
    }
    ...
  }
}
```


3 Anonyme Klassen

- Der Klassenname **`MyEnum`** im vorherigen Beispiel enthält keinerlei Information ➜ Einsatz einer anonymen inneren Klasse:

```
class Test {
  private String array[] = { "hans", "otto", "max"};

  Enumeration enumerate() {
    return new Enumeration() {
      int counter = 0;
      public boolean hasMoreElements() {
        return counter < array.length; }
      public Object nextElement() {
        return array[counter++]; }
    };
  }
}
```

Ende des return Statements

4 Statische Innere Klassen

- Statische innere Klassen können nur auf statische Variablen und Methoden der umschliessenden Klasse zugreifen:

```
class Test {
  private static String array[] = { "hans", "otto", "max"};

  static class E implements Enumeration() {
    int count = 0;
    public boolean hasMoreElements() {
      return count < array.length; }
    public Object nextElement() {
      return array[count++]; }
  }
}
...
Enumeration e = new Test.E();
System.out.println(e.nextElement());
```


H Testen mit JUnit

- Vorteile der Kombination aus Unit-Tests und Test-First-Ansatz:
  - verändert den Programmcode der Applikation nicht
  - gewährleistet definierte Funktion auch nach großen Modifikationen des Programmcodes
  - weniger debugging am kompletten System
  - kleinerer Programmcode
  - reduziert den Wartungsaufwand

H Testen mit JUnit

- Test-First-Ansatz
  - Tests werden erstellt bevor der eigentliche Produktionscode implementiert wird
  - sobald ein Test erfolgreich aufgerufen werden kann ist der zugehörige Produktionscode ausreichend
  - Entwicklung verläuft in Mikro-Iterationen von 10-15 min
- Unit-Test
  - testet eine Methode einer Klasse
  - überprüft das Verhalten einer Methode unter Rand- und Normalbedingungen
  - ist jederzeit wiederholbar

H.1 TestCase

- Sammelt alle Unit-Tests zu einer Klasse.
- Erbt von **`junit.framework.TestCase`**.
- Verfügt per Konvention über den Klassennamen **`<Name>Test.java`**.
- Beispiel:

```
import junit.framework.*;

public class MoneyTest extends TestCase {

    public MoneyTest (String name){super(name);}

    public void testAmount(){
        Money money = new Money(3.00);
        assertTrue(3.00 == money.getAmount());
    }
}
```

H.1 TestCase

- Erzeugen der Testbasis (*Fixture*) mit **`setUp()`**:

```
import junit.framework.*;

public class MoneyTest extends TestCase {
    private Money money;

    public MoneyTest (String name){super(name);}

    protected void setUp(){money = new Money(3.00);}

    public void testAmount(){
        assertTrue(3.00 == money.getAmount());
    }
}
```

- Aufräumen der Testbasis mit **`tearDown()`**.

H.2 Asserts

- **`fail()`** kann verwendet werden wenn es keine passende **`assert()`** Methode gibt
- Beispiel:

```
public void testIndexOutOfBoundsException() {
    Vector v= new Vector(10)

    try{
        Object o= v.elementAt(v.size());
    fail("Should raise ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {}
}
```


H.2 Asserts

- **`assert()`**-Methoden überprüfen Testbedingungen:
  - **`assertTrue()`** stellt fest, ob eine Bedingung wahr ist
  - **`assertEquals()`** verifiziert, ob zwei Objekte gleich sind
  - **`assertSame()`** verifiziert, ob zwei Referenzen auf das gleiche Objekt verweisen
  - **`assertNull()`**
  - **`assertNotNull()`**
- Beispiel:

```
assertEquals("rounded amount",2,money.getAmount(),0.002);
```

- Ausgabe im Fehlerfall:

```
junit.framework.AssertionFailedError:
      rounded amount expected:<2.0> but was:<1.995>
at MoneyTest.testRounding(MoneyTest.java:16)
```


H.3 Testsuiten

- Aufruf eines einzelnen Tests:

```
TestResult result = (new MoneyTest("testAmount")).run();
```

- Sammlung der Testfälle eines TestCase:

```
public class MoneyTest extends TestCase {

    public static Test suite(){
        TestSuite suite = new TestSuite();
        suite.addTest( new MoneyTest("testAmount");
        suite.addTest( new MoneyTest("testSimpleAdd");
        return suite;
    }
    ....
}
```

H.3 Testsuiten

- Sammlung der Testfälle mehrere TestCases:

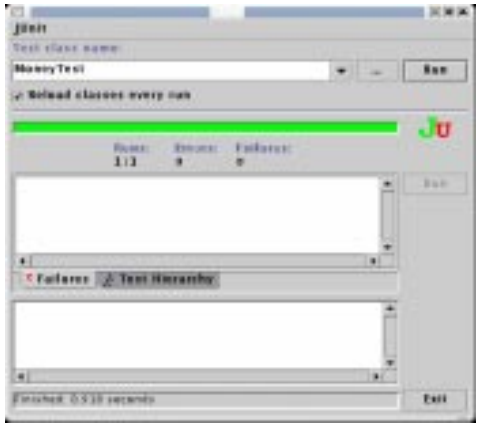
```
public class AllTests extends TestCase {

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(MoneyTest.suite());
        suite.addTest(CustomerTest.suite());
        return suite;
    }

    public static void main(String args[]){
        junit.textui.TestRunner.run(suite());
    }
```

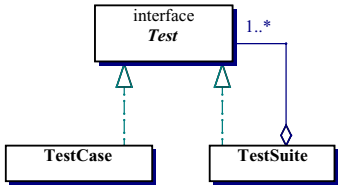

H.4 TestRunner

- batch-TestRunner : junit.textui.TestRunner
- swing-TestRunner: junit.swingui.TestRunner



H.3 Testsuiten

- UML-Darstellung der Abhängigkeiten



H.5 AWT Events

- Reaktion auf Mausklicks: MouseListener:

```
import java.awt.Frame;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;

class MyFrame extends Frame{
    public MyFrame() {
        ....
        this.addMouseListener(
            new MouseAdapter(){
                public void mousePressed(MouseEvent e) {
                    ...
                }
            }
        );
        ....
```

- Reaktion auf Mausbewegungen: MouseMotionListener