Wiederholung

Klassen, Objekte, Nachrichten, Methoden:

Jedes Objekt ist Exemplar (Instanz) einer Klasse.
Diese Klasse definiert das Verhalten des Objektes.
Gewöhnlich haben Klassen mehrere Exemplare.

Jedes Objekt hat seinen eigenen Zustand/seine eigenen Variablen (gilt auch für mehrere Objekte, die zur gleichen Klasse gehören).

Jedes Objekt versteht bestimmte Nachrichten (nämlich diejenigen, für die es Methoden besitzt).
Die Verantwortung für das Interpretieren der Nachricht liegt beim Empfänger-Objekt.

Namenskonvention für Klassen: Anfangsbuchstabe groß.

Wiederholung

Primitive Datentypen (Zahlen, char, boolean):

Variable enthalten Werte.

Andere Datentypen (Arrays, alle Klassen):

Variable enthalten Referenzen.

Referenz = Zeiger auf Objekt
= Verweis auf Objekt
= eindeutiger „Name“ des Objekts.

(wichtig bei Zuweisungen und Gleichheitstests)

Beispiel: Konto

```
public class Konto {
    // Variable, Konstruktoren
    ...

    // Methoden
    public void zahleEin(int betrag, String einzahler)
    ...
    public void zahleAus(int betrag, String empfaenger)
    ...
    public int kontostand()
    ...
    public String kontoauszug()
    ...
}
```


Beispiel: Konto

```
public class Konto {
    // Variable
    int stand;
    String auszug;

    // Konstruktoren
    Konto() {
        stand = 0;
        auszug = "Letzter Kontostand: 0\n";
    }

    // Methoden
    ...
}
```

Beispiel: Konto

```
public class Konto {
    ...

    // Methoden
    public void zahleEin(int betrag, String einzahler) {
        stand += betrag;
        auszug += betrag + " von " + einzahler + "\n";
    }

    public void zahleAus(int betrag, String empfaenger) {
        stand -= betrag;
        auszug += betrag + " an " + empfaenger + "\n";
    }
    ...
}
```


Beispiel: Konto

```
public class Konto {
    ...

    public int kontostand() {
        return stand;
    }

    public String kontoauszug() {
        String ausgabe;

        ausgabe = auszug + "Aktueller Kontostand: " + stand + "\n";
        auszug = "Letzter Kontostand: " + stand + "\n";
        return ausgabe;
    }
}
```


Beispiel: Konto

```
Konto k;
k = new Konto();
k.zahleEin(8000, "Schulzmüller GmbH");
k.zahleAus(2000, "Klaus Meier");
System.out.println(k.kontoauszug());
```

```
Letzter Kontostand: 0
8000 von Schulzmüller GmbH
2000 an Klaus Meier
Aktueller Kontostand: 6000
```

```
k.zahleAus(2500, "Eva Lehmann");
System.out.println(k.kontoauszug());
```

```
Letzter Kontostand: 6000
2500 an Eva Lehmann
Aktueller Kontostand: 3500
```


Beispiel: Zähler

```
public class Zaehler {
    int n;

    Zaehler() {
        n = 1;
    }

    public int wert() {
        return n;
    }

    public void inkr() {
        n++;
    }
}
```

Beispiel: Zähler

```
Zaehler z1, z2;
z1 = new Zaehler();
z2 = new Zaehler();
System.out.println(z1.wert());    //    -> 1
System.out.println(z2.wert());    //    -> 1
z1.inkr();
System.out.println(z1.wert());    //    -> 2
System.out.println(z2.wert());    //    -> 1
z2.inkr();
System.out.println(z1.wert());    //    -> 2
System.out.println(z2.wert());    //    -> 2
z2 = z1;
z2.inkr();
System.out.println(z1.wert());    //    -> 3
System.out.println(z2.wert());    //    -> 3
```


Beispiel: Zähler

Anfangswert bisher im Code der Klasse festgelegt:

```
Zaehler() {
    n = 1;
}
```

oder: dem Konstruktor als Argument übergeben:

```
Zaehler(int start) {
    n = start;
}
```

Gesucht:

Möglichkeit, um festzulegen, daß alle neu erzeugten Zähler mit einem bestimmten Wert anfangen sollen.

Beispiel: Zähler

```
public class Zaehler {
    int n;
    static int start = 1;

    Zaehler() { n = start; }

    public static void setzeStartwert (int m) {
        start = m;
    }

    public int wert() { return n; }

    public void inkr() { n++; }
}
```


Beispiel: Zähler

```
Zaehler z1, z2, z3;

Zaehler.setzeStartwert(100);
z1 = new Zaehler();
z2 = new Zaehler();
Zaehler.setzeStartwert(200);
z3 = new Zaehler();

System.out.println(z1.wert()); // -> 100
System.out.println(z2.wert()); // -> 100
System.out.println(z3.wert()); // -> 200
z1.inkr();
System.out.println(z1.wert()); // -> 101
System.out.println(z2.wert()); // -> 100
System.out.println(z3.wert()); // -> 200
```

static

static-Variable und static-Funktionen:

- existieren einmal pro Klasse (nicht einmal pro Objekt),
- können nicht nur über ein Objekt, sondern auch über die Klasse angesprochen werden:
  - `Zaehler.start`, `z1.start` und `z2.start` bezeichnen dieselbe Variable,
  - `Zaehler.setzeStartwert(...)`, `z1.setzeStartwert(...)` und `z2.setzeStartwert(...)` sind äquivalent.

static

Beachte:

- static-Funktionen können keine Variable oder Funktionen ansprechen, die nicht selbst static sind:
  - in `setzeStartwert` (static) kann `start` (static) ein Wert zugewiesen werden, aber nicht der Variablen `n` (nicht static).
  - Es wäre nämlich nicht klar, welches `n` (in welchem Objekt!) gemeint ist.

public static void main

Die Methode `public static void main(String[] argv)` hat eine besondere Funktion:

```
public class Test {

    public static void main(String[] argv) {
        Zaehler z1;

        z1 = new Zaehler();
        System.out.println(z1.wert());
        z1.inkr();
        System.out.println(z1.wert());
    }
}
```


public static void main

In einem Shellfenster (z.B. xterm):

```
einfo99@x1 [~]: javac Zaehler.java
einfo99@x1 [~]: javac Test.java
einfo99@x1 [~]: java Test
1
2
```

public static void main

Mit Argumentübergabe:

```
public class Test {
    static public void main(String[] argv) {

        System.out.println("erstes Argument: " + argv[0]);
        System.out.println("zweites Argument: " + argv[1]);
        System.out.println("Anzahl der Arg.: " + argv.length);
    }
}
```

```
einfo99@x1 [~]: javac Test.java
einfo99@x1 [~]: java Test Merkur Venus Erde Mars
erstes Argument: Merkur
zweites Argument: Venus
Anzahl der Arg.: 4
```


Klassenhierarchie

Oberklassen und Unterklassen:

Idee: Oberklasse stellt Funktionalität zur Verfügung,
Unterklasse erweitert diese:

in der Regel neue Methoden und Konstruktoren,
meist auch neue Variable.

Klassenhierarchie

Beispiele:

Oberklasse: Konto (ohne Kontoauszug)
Unterklasse: Konto mit Kontoauszug

Oberklasse: Component (versteht `getBounds()`)
Unterklasse TextField (versteht `getColumns()`)
Unterklasse Scrollbar (versteht `setOrientation()`)

Oberklasse Hund (versteht `bellen()`, `beissen(Lebewesen l)`)
Unterklasse Windhund (versteht `sprinten()`)
Unterklasse Schosshuendchen (versteht `maennchenMachen()`)

Klassenhierarchie

Syntax:

```
public class Windhund extends Hund {
  // neue Variable, Konstruktoren, Methoden
}
```

Klassenhierarchie

Objekte der Unterklasse

verstehen alle Nachrichten, die Objekte der Oberklasse verstehen:

```
Windhund wh;
wh = new Windhund();
wh.bellen();
wh.sprinten();
```


Klassenhierarchie

Objekte der Unterklasse

können anstelle von Objekten der Oberklasse verwendet werden.

Sie können Variablen der Oberklasse zugewiesen werden:

```
Hund h;
h = wh;
h.bellen();
```

Sie können statt Objekten der Oberklasse einer Funktion als Parameter übergeben werden:

```
Tierarzt ta;
ta.impfen(wh);     // impfen(Hund h) {...}
```


Klassenhierarchie

Weitere Möglichkeit:

Oberklasse stellt Funktionalität zur Verfügung,
Unterklasse *ändert* Methoden der Oberklasse („override“)
(meist: Erweiterung der Funktionalität der existierenden Methoden)

„harmlos“: Konto mit Kontoauszug
weniger „harmlos“: Konto ohne Überziehungsmöglichkeit

Klassenhierarchie

```
public class Konto {
// Konto ohne Kontoauszug

    // Variable
    protected int stand;
        // protected: auf stand kann auch aus einer
        //              Unterklasse zugegriffen werden.

    // Konstruktoren
    Konto() {
        stand = 0;
    }

    ...
```

Klassenhierarchie

```
    ...

    // Methoden
    public void zahleEin(int betrag, String einzahler) {
        stand += betrag;
    }

    public void zahleAus(int betrag, String empfaenger) {
        stand -= betrag;
    }

    public int kontostand() {
        return stand;
    }
}
```


Klassenhierarchie

```
public class KontoMitAuszug extends Konto {
    // Variable
    protected String auszug;

    // Konstruktoren
    KontoMitAuszug() {
        auszug = "Letzter Kontostand: 0\n";
        // Die Anweisungen des Konstruktors der Oberklasse
        // werden automatisch mitausgeführt.
    }

    ...
```


Klassenhierarchie

```
    ...

    // Methoden
    public void zahleEin(int betrag, String einzahler) {
        // überschreibt Methode der Oberklasse.
        stand += betrag;
        auszug += betrag + " von " + einzahler + "\n";
    }

    public void zahleAus(int betrag, String empfaenger) {
        // überschreibt Methode der Oberklasse.
        stand -= betrag;
        auszug += betrag + " an " + empfaenger + "\n";
    }
    ...
```


Klassenhierarchie

```
    ...

    // public int kontostand() wird von der Oberklasse geerbt.

    public String kontoauszug() {
        // zusätzliche Methode der Unterklasse.
        String ausgabe;

        ausgabe = auszug + "Aktueller Kontostand: " + stand + "\n";
        auszug = "Letzter Kontostand: " + stand + "\n";
        return ausgabe;
    }
}
```

Klassenhierarchie

Alternative Formulierung für `zahleEin`:

```
...
// Methoden

public void zahleEin(int betrag, String einzahler) {
    // überschreibt Methode der Oberklasse.

    super.zahleEin(betrag, einzahler);
        // super: überschriebene Methode der
        //        Oberklasse aufrufen.

    auszug += betrag + " von " + einzahler + "\n";
}
...
```


Klassenhierarchie

Weitere Anwendungen von Ober- und Unterklassen:

Zusammenfassen gemeinsamer Teile der Implementierung verschiedener Klassen:

Oberklasse: Component
Unterklassen: Button, Window, Scrollbar, TextField, ...
gemeinsam z.B.: `Rectangle getBounds()`

Oberklasse: Applet
Unterklassen: benutzerdefiniert
gemeinsam z.B.: `void resize(int width, int height)`

Klassenhierarchie

Weitere Anwendungen von Ober- und Unterklassen:

Zusammenfassen gemeinsamer Teile der Schnittstelle verschiedener Klassen:

Oberklasse: Menge
Methoden: `enthaelt(), fuegeEin()`
Unterklassen:
UngeordnetesArray, GeordnetesArray, Hashtabelle, ...
(jede mit eigener Implementierung dieser Methoden)

Klassenhierarchie

Vorteil:

Je nach Anwendung Auswahl zwischen verschiedenen Implementierungen möglich.

Zur Änderung der Implementierung einer bestimmten Menge, reicht der Austausch einer einzigen Anweisung aus:

```
Menge m;

m = new UngeordnetesArray(1000);
// oder: m = new GeordnetesArray(1000);
// oder: m = new Hashtabelle(1000);

m.fuegeEin(23);
```

Klassenhierarchie

`Object`: Oberklasse aller Java-Klassen

Mit anderen Worten:

```
class Xyz
```

bedeutet das gleiche wie

```
class Xyz extends Object
```


Klassenhierarchie

Anwendungsbeispiel für `Object`:

`Vector`: in Java vordefiniert,
(ähnlich wie Arrays, aber ohne festgelegte Größe)

Methoden:

```
void setElementAt(Object o, int n)
Object elementAt(int n)
```


Klassenhierarchie

```
Vector v;
Object o;
String s;
Zaehler z;

v = new Vector();
v.setSize(50);

// void setElementAt(Object o, int n)
v.setElementAt("EInfo", 2);
v.setElementAt(z, 43);
```


Klassenhierarchie

```
o = v.elementAt(2);
// ok

s = v.elementAt(2);
// Compiler: illegale Zuweisung (Object an String-Variable)

s = (String)v.elementAt(2);
// Compiler: ok
// Cast: Überprüfe zur Laufzeit, ob das Objekt
//       wirklich zur Klasse String gehört
// Überprüfung zur Laufzeit: ok

s = (String)v.elementAt(43);
// Compiler: ok,
// Überprüfung zur Laufzeit: Fehler
```