Wiederholung

Klassenhierarchie:

- `class Unter extends Ober {...}`
- Die Unterklasse `Unter` erweitert die Funktionalität ihrer Oberklasse `Ober`.
- Objekte der Klasse `Unter` können anstelle von Objekten der Klasse `Ober` verwendet werden:
  - Sie verstehen alle Nachrichten, die `Ober`-Objekte verstehen.
  - Sie können Variablen des Typs `Ober` zugewiesen werden.
  - Sie können anstelle von `Ober`-Objekten als Parameter übergeben werden.
- `Object`: Oberklasse aller anderen Klassen in Java.

Wiederholung

Typische Anwendung:

- Zusammenfassen gemeinsamer Teile der *Implementierung* verschiedener Klassen:
- `Component` implementiert `Rectangle getBounds()`
- Unterklassen `Button`, `Window`, `Scrollbar`, `TextField` erben die Implementierung.

Wiederholung

Typische Anwendung:

- Zusammenfassen gemeinsamer Teile der *Schnittstelle* verschiedener Klassen:
- `Menge` deklariert `enthaelt(), fuegeEin()`
- Unterklassen `UngeordnetesArray`, `GeordnetesArray` besitzen jeweils eigene Implementierung dieser Methoden.

Wiederholung

Vorteil:

- Man kann so Methoden implementieren, die als Argument sowohl ein `UngeordnetesArray` als auch auf ein `GeordnetesArray` akzeptieren:

```
int f(Menge m) {
    ...
}
```

Abstrakte Klassen

Wenn Objekte nur von den Unterklassen angelegt werden (sollen), dann ist die Implementierung der Methoden in der Oberklasse beliebig (da sie ohnehin in den Unterklassen überschrieben werden).

Dies sollte man im Programmtext irgendwie kennzeichnen.
Dazu dient das Schlüsselwort `abstract` (vor Klassen und Methoden).

Abstrakte Klassen

Abstrakte Klassen:

haben keine „eigenen“ Objekte:
Objekte können nur von Unterklassen gebildet werden.

Abstrakte Methoden:

haben nur Schnittstelle (Funktionskopf), keine Implementierung;

dürfen nur in abstrakten Klassen vorkommen;

müssen in konkreten Unterklassen implementiert werden.

Abstrakte Klassen

```
abstract class AbstrakteKlasse {

    abstract int abstrakteMethode();
    // Diese Methode wird hier *nur* deklariert.
    // Unterklassen, die nicht selbst abstract sind,
    // *müssen* diese Methode implementieren.

    int konkreteMethode() {
        ...
    }

    ...
}
```


Abstrakte Klassen

```
class KonkreteKlasse extends AbstrakteKlasse {

    int abstrakteMethode() {
    // Da diese Klasse nicht abstract ist, muß hier
    // die Methode implementiert werden.
        ...
    }
}
```

Abstrakte Klassen

```
int test (AbstrakteKlasse a) {
  return a.abstrakteMethode();
}
// funktioniert: Jedes Objekt, das in a stecken kann, muß zu
// einer *konkreten* Unterklasse von AbstrakteKlasse gehören;
// in dieser muß abstrakteMethode implementiert sein.


AbstrakteKlasse a1;
a1 = new KonkreteKlasse();
n = xyz.test(a1);
// funktioniert

a1 = new AbstrakteKlasse();
// funktioniert nicht!
```


Einfache und mehrfache Vererbung

In Java: Jede Klasse (außer `Object`) hat genau eine direkte Oberklasse

```
class B extends A {...}
```

$\leadsto$ Klassenhierarchie hat Baumstruktur (einfache Vererbung).

Einfache und mehrfache Vererbung

Alternative (in anderen Programmiersprachen):

Klassen können von mehreren Oberklassen erben:

```
class Quadrat extends Rechteck extends Rhombus {...}
```

$\leadsto$ mehrfache Vererbung.

Einfache und mehrfache Vererbung

Anwendung: Sortieren nach laufender Nummer (Personalausweisnummer, Matrikelnummer, Mitgliedsnummer, Inventarnummer, . . .)

```
public class Nummer {
    int n;
    static void sortiere(Nummer[] array) {...}
}

public class Person {
    String vorname;
    String familienname;
}
```

Einfache und mehrfache Vererbung

```
public class Student extends Nummer extends Person {
    // KEIN JAVA!
    String hauptfach;
    String nebenfach;
}

Student[] studenten;
...
Nummer.sortiere(studenten);
```


Einfache und mehrfache Vererbung

Probleme der mehrfachen Vererbung:

- In verschiedenen Oberklassen können Variable mit dem gleichen Namen deklariert sein.
- In verschiedenen Oberklassen können Methoden mit dem gleichen Namen deklariert sein:
  - eine konkrete Methode, andere Methoden abstrakt: ok.
  - mehrere gleiche Implementierungen: ok
  - mehrere verschiedene Implementierungen:
    - welche Methode soll aufgerufen werden?
- Darum: in Java nicht erlaubt.

Interfaces

Ausweg: Interface

- Interface $\approx$ „völlig abstrakte Klasse“.
- nur Methodendeklarationen (ohne Implementierung) und Konstanten.
- Syntaktisch: `class` $\rightsquigarrow$ `interface`,
  `extends` $\rightsquigarrow$ `implements`.

Interfaces

```
public interface Numeriert {
    int nummer();
    // implizit: public
}

public class NumHilfsfkt {
    public static void sortiere(Numeriert[] array) {
        ...
    }
}
```

Interfaces

```
public class Student extends Person implements Numeriert {
    int matrikelnr;
    String hauptfach;
    String nebenfach;
    ...
    public int nummer() {      // ist in Numeriert deklariert
        return matrikelnr;     // und muß hier implementiert
    }                          // werden.
}

Student[] studenten;
...
NumHilfsfkt.sortiere(studenten)
```


final

`final` verhindert, daß sich etwas ändert:

`final` vor Variablendeklaration:

Der Wert der Variablen ist unveränderlich (Konstante).

`final` vor Methodendeklaration:

Die Methode kann in Unterklassen nicht überschrieben werden.

`final` vor Klassendeklaration:

Von der Klasse können keine Unterklassen gebildet werden.

this

Beispiel: Doppeltverkettete Liste

```
public class Element {
    // Ein Element hat Vorgänger und Nachfolger; der Vorgänger des
    // Nachfolgers (und umgekehrt) ist wieder das Element selbst.
    Element vor;
    Element nach;

    public void setzeNachfolger(Element e) {
        nach = e;
        e.vor = this;
        // this: das Objekt, "in dem wir uns gerade befinden", d.h.,
        // dem die Nachricht setzeNachfolger() geschickt wurde.
    }
    ...
}
```


this

Beispiel: Doppeltverkettete Liste

```
public class Element {
    // Ein Element hat Vorgänger und Nachfolger; der Vorgänger des
    // Nachfolgers (und umgekehrt) ist wieder das Element selbst.
    Element vor;
    Element nach;

    public void setzeNachfolger(Element nach) {
        this.nach = nach;
        nach.vor = this;
        // this: das Objekt, "in dem wir uns gerade befinden", d.h.,
        // dem die Nachricht setzeNachfolger() geschickt wurde.
    }
    ...
}
```

Packages

Klassenhierarchie:

strukturelle Ähnlichkeit
(gemeinsame Teile der Schnittstelle und Implementierung).

Packages:

thematischer Zusammenhang, gemeinsames Anwendungsgebiet.

Beispiel: Package `java.awt` (Abstract Window Toolkit)
enthält u.a. die Klassen
`Color` (RGB-Farbwerte),
`Point` (2-dimensionale Punkte (in einem Fenster)),
`TextField`.

Packages

	Oberklasse:	Oberklasse:
Package: allgemeines	Fluessigkeit	Behaelter
	Unterklassen:	Unterklassen:
Package: ernaehrung	Orangensaft	Trinkglas
Package: strassenverkehr.kfz	Benzin	Kanister

Packages

Packages = Ansammlungen von (thematisch) verwandten Klassen und Interfaces

- strukturieren Programm:
  Klassen sind für Programmierer leichter auffindbar.
- regeln Zugriffsschutz:
  für Klassen im selben Package gelten geringere Einschränkungen als für Klassen in verschiedenen Packages;
- liefern separate Namensräume:
  Klassen in verschiedenen Packages dürfen die gleichen Namen besitzen: `ernaehrung.Oel` und `strassenverkehr.kfz.Oel` sind verschiedene Unterklassen von `allgemeines.Fluessigkeit`.

Packages

```
package strassenverkehr.kfz;
// Der Inhalt dieser Datei gehört zum Package
// strassenverkehr.kfz

public class Kanister extends allgemeines.Behaelter {
    ...
}
```

Packages

```
package strassenverkehr.kfz;
import allgemeines.Behaelter;
// die Klasse Behaelter aus dem Package allgemeines
// kann unter ihrem kurzen Namen angesprochen werden

public class Kanister extends Behaelter {
    ...
}
```


Packages

```
package strassenverkehr.kfz;
import allgemeines.*;
// alle Klassen aus dem Package allgemeines können
// unter ihrem kurzen Namen angesprochen werden

public class Kanister extends Behaelter {
    ...
}
```


Packages

Klassenname ↔ Dateiname

Packagename ↔ Directoryname

Die Klasse `Kanister` im Package `strassenverkehr.kfz` steht in der Datei `Kanister` im Directory `strassenverkehr/kfz`.

Falls in einer Datei kein Package angegeben wird, gehören die Klassen darin zum sogenannten „Default-Package“.

Zugriffsrechte

protected, public, etc.:

spezifizieren, von wo aus auf eine Variable/Methode/Klasse zugegriffen werden kann:

	aus der Klasse selbst	aus dem gleichen Package	aus Unter-klassen	von überall
`private`	x			
„package“/„friendly“	x	x		
`protected`	x	x	x	
`public`	x	x	x	x