Kapitel 2

Objektorientiertes Programmieren in Java

Was ist objektorientiertes Programmieren?

- Eine präzise Definition ist sehr schwierig!

• Zwei Sorten der Definition

1. Intensional - Interner Ausbau eines Dings
   > Normalerweise sehr präzis.
   > Z.B. Eine Baumdatenstruktur (in Informatik) ist eine Struktur, die entweder ein Blatt ist, oder ein Knoten, der mehrere (Unter-)Bäume verbindet.

2. Extensional - Beschreiben der Eigenschaften eines Dings

   > Oftmals nicht sehr präzise.
   > Z.B. Ein Baum (in der Natur) ist eine Pflanze, die ....

• Was ist OOP? - Eine extensionale Definition

- OO-Denken (eine Sichtweise, ein Paradigma)
- Konkrete Syntax und Semantik
- Bekämpfung der Komplexität
- Wiederverwendbare Software
... ...

Objektorientiertes Denken - sprachunabhängig

• Beispiel:
Wenn ich meiner Mutter Blumen zukommen lassen möchte, kann ich einen Blumenhändler beauftragen.

• *Objekt*: Ich, meine Mutter, der Blumenhändler.

• *Nachricht* (*message*): mein Auftrag.

• *Zusätzliche Info mit der Nachricht*: Name und Adresse meiner Mutter.

• *Nachrichtensender*: Ich.

• *Nachrichtenempfänger*: Blumenhändler.

• *Methode*: wie der Blumenhändler die Blumen liefert. Zahnarzt hat keine Methode für diese Nachricht.

• *Kapselung* (*info hiding*): “wie die Blume geliefert wird” ist mir unbekannt. (Ist auch nicht mein Bier!)

Objektorientiertes Denken - Fortsetzung

• *Klasse*: Menge aller Blumenhändler (oder aller Kunden, etc).

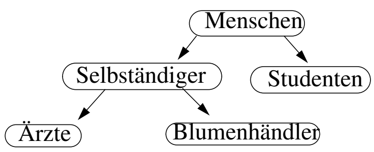
• *Instanzen (Objekte) einer Klasse*: Elemente einer Klasse.

• *Unter-, Oberklasse, Vererbung*.

- Blumenhändler-Klasse - Unterklasse von Selbständiger-Klasse (Blumenhändler sind Selbständiger),

- Selbständiger-Klasse hat eine Kassieren Methode, Blumenhändler-Klasse erbt sie.

• *Klassenhierarchie*.



• *Überschreiben*: Die Kassieren Methode für Ärzte (Abrechnung über Krankenkassen) überschreibt die für Selbständiger (bar, Überweisung, Kreditkarten).

Klassen in Java

- Die Klasse Motorräder.

```
class Motorcycle {

  String make;
  String color;
  boolean engineState;

  void startEngine() {
    if (engineState == true)
      System.out.println(
             "The engine is already on.");
     else {
      engineState = true;
      System.out.println(
                 "The engine is now on.");
     }
  }
}
```

- Variablen für *Attribute*, `make`, `color`, `engineState`.
- *Methoden*: `startEngine()`.
- Genaueres über `System.out.println` später.

Eine `main()` Methode

- Für den Ablauf braucht die `Motorcycle` Klasse noch eine `main()` Methode.

```
public static void main(String args[]) {
  Motorcycle m = new Motorcycle();
  m.make = "Yamaha RZ350";
  m.color = "yellow";
  System.out.println("This motorcycle is a "
               + m.color + " " + m.make);
  System.out.println("Starting engine...");
  m.startEngine();
  System.out.println("... ...");
  System.out.println("Starting engine...");
  m.startEngine();
}
```

- Wenn die `Motorcycle` Klasse übersetzt und interpretiert wird, dann

```
This motorcycle is a yellow Yamaha RZ350
Starting engine...
The engine is now on.
... ...
Starting engine...
The engine is already on.
```


Gucken wir uns das Programm genauer an

- Lokale Variablen - Objekte sind Instanzen einer Klasse:

```
Motorcycle m = new Motorcycle();
```

- Zugriff auf Objektattribute:

```
m.make = "Yamaha RZ350";
m.color = "yellow";
System.out.println("This motorcycle is a "
               + m.color + " " + m.make);
```

- Objektmethoden benutzen.
  (Senden einer Nachricht zu einem Empfänger-Objekt)

```
m.startEngine();
```

- Empfänger implizit oder als `this` in der Methodendefinition.

```
void startEngine() {
  if (engineState == true)
                    // this.engineState
   ...
   engineState = true;
                    // this.engineState
   ...
```


Überladen von Methoden

- Die Idee
  - Methoden mit demselben Namen aber vers. Argumenttypen.

```
void m(float f) = {...};
void m(double d) = {...};
```

  - Zu Jedem Aufruf der überladenen Methoden muß es eine *einzige* Definition geben, die diesem Aufruf *am besten* paßt

```
o.m(1);    // Aufruf von m(float f)
o.m(1.5f); // Aufruf von m(float f)
o.m(1.5);  // Aufruf von m(double f)
```

- Das Problem von Mehrdeutigkeit

```
void n(double d, int i) {...}
void n(int i, double d) {...}
...
o.n(1,1);  // welche n?
```

- Fehlermeldung zur Übersetzungszeit.

- Mit einer zusätzlichen Definition ist alles in Ordnung!

```
void n(int i, int i) {...}
void n(double d, int i) {...}
void n(int i, double d) {...}
```

Erzeugen von Objekten auf verschiedene Arten.

- Was macht `new Motorcycle()` in

```
Motorcycle m = new Motorcycle();
```

\- Allozieren eines frischen Objektes der `Motorcycle` Klasse mit Standardwerten für *alle* Variablen.

- Eigene Konstruktoren schreiben

\- Immer heißt `Motorcycle`, keine Rückgabetyp, kein `void`, keine `return`-Anweisung. (Überladen von Konstruktoren)

```
Motorcycle (String m, String c) {
   this.make = m; this.color = c;}

Motorcycle (String c) {
   this.make = “BMW”; this.color = c;}
```

\- Dann das Aufrufen

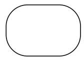
```
Motorcycle m =
new Motorcycle(“black”);
```

- Wenn die `Motorcycle` Klasse überhaupt keine explizite Konstruktorendefinitionen hätte, dann würde ein Konstruktor `Motorcycle() = {}` automatisch eingeführt.

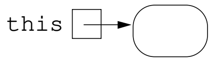
Ablauf eines Initialisierungsprozesses und `this`.

```
Motorcycle m =
new Motorcycle(“black”);
```

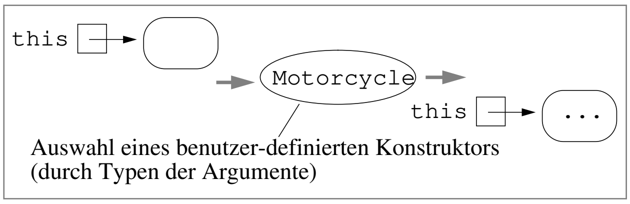
1. Allozieren eines frischen Objektes mit *Standardwerten* für *alle* Instanzvariablen



2. Zuweisung des Objektes an `this`



3. Aufruf des Konstruktors mit `this` als Ein- und Rückgabe



Noch ein Beispiel

- Die Klasse Kreise.

```
public class Circle {
  double x, y, r;    // Mittelpunkt, Radius

  Circle(double r){this.r = r;}
       // ‘this’ notwendig; sonst verdeckt.
       // Konstruktor. Eingabe: Radius.

  Circle(double x, double y, double r){
       this.x = x; this.y = y; this.r = r;
    } // Konstruktor.
       // Eingaben: Mittelpunkt, Radius.

  Circle(Circle c){
       x = c.x; y = c.y; r = c.r;
       // Achtung: ‘this’ nicht notwendig.
    } // Konstruktor. Eingabe: Kreis.

  double circumference() {return 2*3.14*r;}
          // Methode. Umfang

  public static void main(String args[]) {
  Circle c1 = new Circle(2.0);
  Circle c2 = new Circle(c1);
  double circum = c1.circumference();}
}
```


Klassen-Variablen

- Bis jetzt: Variablen für Attribute sind *Instanz*-Variablen.

\- Verschiedene Instanzen können verschiedene Werte für dieselbe Instanz-Variablen tragen.

Z.B. zwei `Motorcycle` Objekte mit vers. Werten für `color`.

- *Klassen*-Variablen (oder `static`-Variablen) sind *globale* Variablen für alle Objekte einer Klasse

\- Alle Instanzen einer Klasse teilen eine Klassen-Variablen.

\- Wofür? Z.B. Anzahl der erzeugten Motorräder.

```
class Motorcycle {
  static int total_num; // Klassen-Variable
  ...
  Motorcycle (String m, String c) {
     this.make = m; this.color = c;
     total_num++; // total_num=total_num+1;
  }
  Motorcycle (String c) {
     this.make = “BMW”; this.color = c;
     total_num++; // total_num=total_num+1;
  }
}
```

Klassen-Variablen - Fortsetzung

- Zugriff auf Klassen-Variablen außerhalb der Klasse
  `Klassenname.Klassenvariable`

  Z.B. `Motorcycle.total_num`

- Das Print-Kommando
  `System.out.println(...)`

- `System` eine Klasse.

- `out` eine Klassen-Variable in der `System` Klasse,
  Die Variable hat die Klasse `PrintStream.`

- `System.out` hält eine Instanz der `PrintStream` Klasse

- `println` ist eine Methode in der `PrintStream` Klasse

- Wie werden Klassen-Variablen initialisiert? - Später.

- Achtung: Es gibt keine üblichen globalen Variablen in Java!

Klassen-Methoden

- Bis jetzt: alle Methoden sind *Instanz*-Methoden.

- *Konzeptuell* gehört eine Instanz-Methode zu einer Instanz,

```
Motorcycle m = new Motorcycle("red");
m.startEngine();

Circle c1 = new Circle(2.0);
double circum = c1.circumference();
```

..., obwohl die *Implementierungen* von Instanz-Methoden normalerweise mit der von Klassen zusammen sind.

- *Klassen*-Methoden sind Methoden, die auch *konzeptuell* zu einer Klasse (anstatt zu einer Instanz) gehört.

- Form der Aufrufe außerhalb der Klasse
  `Klassenname.Methodenname(...)`

  Z.B. `Math.sqrt()` in

```
double distance(Circle c) {
   double dx = x - c.x;
   double dy = y - c.y;
   return (Math.sqrt(dx * dx + dy * dy))
 }
```


Klassen-Methoden - Fortsetzung

- Deklarationen von *Klassen*-Methoden mit `static`

  Z.B.

```
static void m() = {...};

public static void main(...) {...}
```

- *Klassen*-Methoden wofür?
  Es gibt viele Situationen, wobei das Erzeugen einer Instanz nicht ganz natürlich ist.

- Z.B.
  1. Anfang der Ausführung (mit `main`)
  2. Mathematische Funktionen
  3. Alle Methoden in der `System` Klasse sind Klassen-Methoden.

  ... ...

- Eine Klassen-Methode für Kreise

```
static Circle bigger(Circle c, Circle d) {
   if (c.r > d.r) return c;
   else return d;
 }
```


Verweissemantik

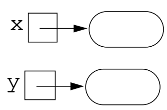
- Welche Werte in Java werden "per Verweis" behandelt?

- Alle Werte eines *nicht-einfachen* Typs.

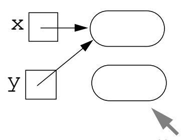
- Die *nicht-einfachen* Typen sind `Array` und alle Klassen.

- Beispiel

```
Circle x = new Circle(2.0);
Circle y = new Circle(3.0);
```




```
y = x;
```



Müll - Kann nie mehr benutzt werden

Wertsemantik

- Welche Werte werden “nach dem Wert” behandelt?

- Alle Werte eines *einfachen* Typs.

- Die *einfachen* Typen in Java sind
```
boolean,
char,
byte, short, int, long,
```
-- Integer mit vers. Längen

`float, double` --Fließkommatypen mit vers. Längen

- Beispiel

```
int x = 2;
int y = 3;
```
x $\boxed{2}$

y $\boxed{3}$

```
y = x;
```
x $\boxed{2}$

y $\boxed{2}$

Arbeiten mit Objekten

- Objekte vergleichen

- Die Operatoren == und != für Objekte: Ist dasselbe Objekt?
```
A x = new A();   // A ist eine Klasse
A y = new A();
A z = x;
... x==y ... // liefert false
... x==z ... // liefert true
```

- Keine benutzer-definierten == und != (nicht wie in C++)

- Es gibt vordefinierten Methode
```
public equals(...){...}
```
in der Standardumgebung. Vorsicht, sorgfältig Testen!

- Ratschlag: (1) eigene Vergleich-Methode schreiben,
  (2) die `equals()` Methode überschreiben.

- Objekte kopieren - Eigene Methoden, oder `clone()`-Methode (Details später).

- Die Klasse eines Objektes feststellen, Instanz-Relation testen
```
Object o = new A();
String cn = o.getClass().getName();//cn="A"
boolean isInstance = o instanceof A; // True
```


Unterklassen und Vererbung

- Die Unterklasse `GraphicCircle`

```
class GraphicCircle extends Circle{
  Color outline, fill;
  void draw(DrawWindow dw) {
    dw.drawCircle(x,y,outline,fill)
  }
  ...
}
```

- `Color` und `DrawWindow` sind vordefinierten Klassen.

- Die `GraphicCircle` Klasse erbt alle Variablen und Methoden der `Circle` Klasse.

- Erzeugen von Objekten, Zugriff auf Variablen, Aufruf von Methoden:

```
GraphicCircle gc = new GraphicCircle();
double x = gc.x;
double circum = gc.circumference();
   // gc erbt die Variable x und
   // die Methode circumference()
Circle c = gc;
   // gc kann als Circle betractet werden.
```


`final`-Klassen und die `Object`-Klasse

- `final` Klassen sind Klassen, die nicht erweitert werden können

```
public final class A {
 ...
}

public class B extends A {    // Illegal
 ...
}
```

- Anwendung:
  (1) Ungewollte Erweiterungen verhindern.
  (2) Compiler-Optimierungen ermöglichen.

- Die `Object` Klasse: implizite Oberklasse von allen Klassen

- Keine mehrfache Vererbung für Klassen.

```
class Circle{ ... }

class Graphic{ ... }

class GraphicCircle
   extends Graphic, Circle{ ... } // Illegal
```

Einfache Typen konvertieren (casting)

- Von einem kürzeren Typ zu einem längeren - oft automatisch.

- Einfache Beispiele: `int zu long`, `int` zu `float`, `float` zu `double`, ...

- In Methoden

```
double static m(double d) { return d; }
... ...
m(1);
```

- Ausnahmen: Z.B. keine Konvertierung von `boolean` zu anderen einfachen Typen.

- Von einem längeren Typ zu einem kürzeren - normalerweise nicht automatisch.

Beispiele:
```
(int) 2.0
(int) (x/y)
```


Klassen konvertieren

- Nur zwischen Unter- und Oberklasse.

- Von Unterklasse zu Oberklasse, Konvertieren oft unnötig.
- Beispiel (von `GraphicCircle` zu `Circle`)

```
public static void m(Circle c) {... }
... ...
m(new GraphicCircle);
```

- Expliztes Konvertieren, von Unterklasse zu Oberklasse, und umgekehrt:

- Von Unter- zu Oberklasse - Keine Auswirkung.

```
GraphicCircle gc = new GraphicCircle;
Circle c1 = gc; // Konvertieren unnötig
Circle c2 = (Circle) gc;
      // Genau wie ohne (Circle)!
```

- Betonung: *Kein* neues Objekt wird erzeugt
  Nur der Verweis wird übergeben.

- Von Oberklasse zu Unterklasse - Ausnahme kann auftauchen.

```
.. c1.outline ..                // Illegal
.. ((GraphicCircl) c1).outline .. // Legal
```


Das Verstecken von Variablen und die Schlüsselwörter `super` und `this`

```
class A { int x; }

class B extends A { /* Keine x hier; */ }

class C extends B {
  int x;  // Versteckt x in A

  void test() {
  x = 1; // Die erste x
         // ab der C Klasse nach oben.
  this.x = 2;        // ... ab der C Klasse
  super.x = 3;       // ... ab der B Klasse
  ((B) this).x = 4;  // ... ab der B Klasse
  ((A) this).x = 5;  // ... ab der A Klasse
  // super.super.x;  ist Illegal
  }
}

class D extends C {
  public static void main(...) {
   (new C).test(); // (new C) beeinflußt
  }                // nicht die Entscheidung
}                  // oben.
```


Überschreiben von Methoden und dynamisches Binden

- Überschreiben eine Methode: nur mit gleichem Namen, Argumenttypen und *Rückgabetyp*

```
class Circle {
...
void draw() {...}
}

class GraphicCircle extends Circle {
...
void draw() {...}
}

Circle c = ...;
c.draw(); // welche draw()?
```

- Dynamisches Binden (zur Laufzeit)
- `draw()` für `GraphicCircle`, wenn `c` ein Objekt der `GraphicCircle` Klasse hält;
  `draw()` für `Circle`, wenn `c` ein Objekt der `Circle` Klasse hält.

- `final` Methoden - Überschreiben unmöglich!

Überladen und Überschreiben

- Das Problem:

- Es gibt keine einzige Definition, die einem Aufruf am besten paßt.

```
class A { void m(int i) {...} }

class B extends A { void m(double d) {...} }


B b = new B();
b.m(1);   // m(int i) in A oder
          // m(double i) in B
          // passt besser?
```

- Fehlermeldung.

- Mit einer zusätzlichen Überschreibung ist alles in Ordnung!

```
...
class B extends A {
   void m(int d) {...}     \\ Neu eingefuehrt!
   void m(double d) {...}
}
...
```


Der überschriebene Methodenaufruf via `super`

```
class A {
   int x;
   int m(){...}
 }
class B extends A {
   int x;
   int m(){
      x = super.x;
      // Zugriff auf versteckte x
      // der statischen Oberklasse A
      x = super.m();
      // Zugriff auf überschriebene m()
      // der statischen Oberklasse A
     }
}
class C extends B {
  public static void main(...) {(new C).m();}
}
```

- Ähnlich wie im Falle vom Verstecken von Variablen `super` in `super.x` und `super.m()` bedeutet Suche nach der ersten Variable bzw Methode ab der `A` Klasse nach oben, egal, von welcher Klasse `this` ist.
- `this` und Klassenkonvertieren haben keine Auswirkung.

Konstruktoren für Unterklassen

- Direkte Definition:

```
GraphicCircle(double x, double y,
        double r,Color outline, Color fill) {
  this.x = x; this.y = y; this.r = r;
  this.outline = outline; this.fill = fill;
}
```

- Wiederverwendung eines Konstruktors der Oberklasse:

```
GraphicCircle(double x, double y,
        double r,Color outline, Color fill) {
super(x,y,r);
     // Aufruf des entsprechenden
     // Konstruktors der Oberklasse
this.outline = outline; this.fill = fill;
}
```

- Achtung: Aufruf eines Konstruktors der Oberklasse *nur als erste Anweisung*.

- Hier ist die Rolle von `super` analog aber unterschiedlich zu ihren Rollen früher: `super` bedeutet den Konstruktor *in* der direkten Oberklasse, aber nicht Suche *ab* der direkten Oberklasse.

Konstruktoren verketten

- Der Ablauf eines Initialisierungsprozesses (nochmals)

  1. Allozieren eines frischen Objektes mit *Standardwerten* für *alle* Instanzvariablen und Zuweisen des Objektes an `this`.

  2. Starten des Rumpfs des Konstruktors
  (Wenn nicht explizit deklariert, wird ein Konstruktor der Form
  `Klassenname(){ };`
  implizit eingeführt.)
  (a) Wenn die erste Zeile nicht `super(...);` ist, dann wird `super()` implizit eingeführt. (Es gibt eine Ausnahme! Details später.)
  (b) Aufruf des entsprechenden Konstruktors der Oberklasse.
  (c) Setzen aller Initialisierungswerte für Instanzvariablen der jetzigen Klasse.
  (d) Ausführen des Restrumpfs.

- Beispiel

```
class A { int a; }
           // Implizit "A(){ super(); };"
class B extends A { int b; B(){}; }
           // Implizit "B(){ super(); }
```

- Aufruf eines Konstruktor der Oberklasse immer explizit, wenn Oberklasse explizite Konstruktoren enthält.

“Abstrakte Klassen”

- Was ist eine *konkrete* Klasse?

- Eine konkrete Klasse beschreibt eine Menge von Objekten.
- Alle Klassen bis jetzt sind konkrete Klassen.

- Was ist eine *abstrakte* Klasse?

- Eine abstrakte Klasse kann eine *Schnittstelle* mehrerer konkreten Klassen beschreiben.

```
abstract class Shape {
  abstract double circumference();
    // Abstrakte Methode hat keinen Rumpf
  abstract double area();           }

class Circle extends Shape {
  ...
  Circle(double x, double y, double r){... }
  double area() {...}
  double circumference() {...}      }

class Rectangle extends Shape {
  ...
  Rectangle(double x, double y, double w,
                           double h) {... }
  double area() {...}
  double circumference() {...}      }
```


“Abstrakte Klassen” - Fortsetzung 1

- Wozu braucht man eine abstrakte Klasse (bzw eine Schnittstelle)?

- Man kann schreiben

```
Shape[] shapes = new Shape[3];
              // Array von Shapes erzeugen
shapes[0] = new Circle(1.0, 1.0, 2.0);
shapes[1] = new Circle(1.0, 2.0, 3.0);
shapes[2] = new Rectangle(1.0, 2.0, 3.0);

double total_area = 0;
for(int i = 0; i < shapes.length; i++)
   total_area += shapes[i].area();
              // Jedes shapes[i] hat
              // garantiert eine area()
```

- Analyse

(1) `Shape[] shapes`
deklariert Variable `shapes` vom Typ Array von `Shape`.

(2) `new Shape[3]`
erzeugt ein Array von `Shape` mit 3 Elementen (Standardwerte)

(3) `shapes.length` liefert die Länge des Arrays

(4) `for`-Schleife:
`for (Init; Fortsetzen?; Weiterzählen) Rumpf;`

(5) Abkürzung: `a += b;` steht für `a = a + b;`

“Abstrakte Klassen” - Fortsetzung 2

- Eine abstrakte Klasse ist eine Klasse, die in der Form
  `abstract class ...`
  deklariert wird.

- Jede Klasse, die eine abstrakte Methode deklariert, oder erbt, ohne sie mit einem Rumpf zu überschreiben, muß eine abstrakte Klasse sein.

- Eine abstrakte Klasse kann keine Instanzen haben.

- Eine abstrakte Klasse kann normale Methoden enthalten und muß nicht abstrakte Methoden enthalten. Z.B.
  (Eine abstrakte Klasse muß nicht nur eine *Schnittstelle* beschreiben.)

- Eine abstrakte Klasse kann als eine Unterklasse einer konkreten Klasse deklariert werden.

```
class Circle {...}
abstract class GraphCirc extends Circle {
  abstract draw();              }
```

- Eine Klasse kann eine konkrete Klasse sein, wenn sie nur konkrete Methoden deklariert und erbt, und wenn sie alle geerbten abstrakten Methoden mit einem Rumpf überschreibt.

Das Konzept “Interface”

- Was ist ein Interface?

- Ein Interface ist eine *Schnittstelle* mehrerer konkreten Klassen.
- Ein Interface kann als eine abstrakte Klasse betrachtet werden.

- Wozu braucht man Interface, wenn man schon abstrakte Klassen hat?

- Es gibt keine mehrfache Vererbung für Klassen in Java.

- Was ist das Problem mit mehrfacher Vererbung?

Mehrfache Vererbung: Vor- und Nachteile

- Der Hauptvorteil: konzeptionell einfach und natürlich.

- Einige Nachteile:

  Rautenstruktur der Klassenhierarchie in einer anderen Sprache

```
                class A { int a; A(){...} }

class B ext A {               class C ext A {
  int b; B(){...}}              int c; C(){...}}

          class D ext B, C {int d; D(){...}}
```

(1) Feste Reihenfolge der Aufrufe der Konstruktoren `B()` und `C()` in `D` ist unnatürlich!

(2) Der Aufruf `super()` in `D(){...}` ist komplizierter.

(3) Entscheiden, wieviele Kopien von `a` in einem Objekt von `D`!

(4) Implementierungsprobleme (keine Details hier).

... ...

Wieviele Kopien von `a` in einem Objekt von `D`

- Wofür solche Entscheidung?

```
        class Person { String name; }

class Student              class Angestellte
     ext Person{                 ext Person{
String fach;}               int gehalt; }

     class StudentAngestellte
       ext Student, Angestellte {}
```

- Jedes Objekt von `StudentAngestellte` soll *eine* Kopie von `name` haben.

```
        class Buffer { String buffer; }

class InBuffer               class OutBuffer
      ext Buffer{                  ext Buffer{
void input(){...};}          void output(){...};}

     class InOutBuffer
       ext InBuffer, OutBuffer {}
```

- Jedes Objekt von `InOutBuffer` soll *zwei* Kopien von `buffer` haben, eine für `InBuffer`, eine für `OutBuffer`.

Das Konzept “Interface”

- Ein Interface normalerweise enthält nur

(1) abstrakte Methoden,
 (Das Schlüsselwort `abstract` kann weggelassen werden.)

(2) Konstanten mit Namen.
 Ein Name ist eine `static` und `final` Variable.

```
final static PI = 3.14159;
```

- Eine Interface-Definition

```
interface Drawable {
  void setColor(Color c);
  void setPosition(double x, double y);
  void draw(DrawWindow dw);          }
```

- Eine Klasse kann ein Interface implementieren.

```
class DrawableCircle extends Circle
                     implements Drawable {
  ...
  void setColor(Color c){...}
  void setPosition(double x, double y){...}
  void draw(DrawWindow dw){...}          }
```

- Die Klasse muß jede Methode des Interface (oder seiner Ober-Interfaces) implementieren.

Weitere Nutzung von “Interface”

- Objekte eines Interfaces:
- Keine Objekte *direkt* aus einem Interface.
 Nur Objekte aus einer Klasse, die ein Interface *implementiert*.

```
 class DrawableRectangle extends Rectangle
                 implements Drawable {...}

Drawable[] drawables = new Drawable[3];
                        // Array erzeugen

 // Erzeugen von Objekten von Drawable
 drawables[0] = new DrawableCircle(...);
 drawables[1] = new DrawableCircle(...);
 drawables[2] = new DrawableRectangle(...);

 double total_area = 0;
 for(int i = 0; i < shapes.length; i++)
    drawables[i].draw(draw_win);
               // Ann. draw_win ist da.
```

- Ein Interface kann ein anderes Interface erben (Unter-Interface und Ober-Interface)

```
 interface DrawingObject extends Drawable {
  ... }
```

Unterschiede zwischen “Interface” und “abstrakten Klassen”

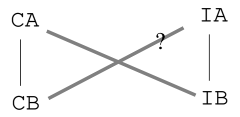
- Mehrfache Vererbung

```
interface Rotateable { ... }

class DrawableCircle extends Circle
      implements Drawable, Rotateable {
   ...
 }

interface DrawingObject
   extends Drawable, Rotateable {
  ...
 }
```

- Ein Interface kann *nur Interfaces* erben, daher nicht die Superklasse `Object`.

Kombination von Interface- und Klassenhierarchie

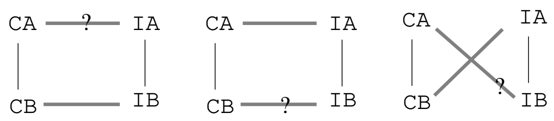
```
interface IA { ... }

interface IB extends IA { ... }

class CA implements IB { ... }

class CB extends CA { ... }
```



- Ist die folgende Anweisung legal?

```
IA ia = new CB();
```

- Was ist noch erlaubt oder nicht erlaubt?



- Ist dies nicht unnötig kompliziert?

- Ist “Interface” die korrekte Lösung?

Pakete (“Packages”)

- Java-Klassen sind in Paketen organisiert.

- Ein Paket kann Klassen und Pakete enthalen.

- “Pakete organisieren Klassen” analog zu “Verzeichnisse organisieren Dateien”.

- Funktionen eines Paketes:

- Definition eines Namensraums - Klassen in ververschiedenen Paketen können den gleichen Namen haben.

- Definition eines Sichtbarkeitskreises. (Gleich mehr).

- Die `package`-Anweisung
  `package packageName`

- Nur als die *erste* Anweisung einer .java-Datei.

- Alle Klassen der .java-Datei sind im Paket `packageName`.

- Es gibt ein Default-Paket - ohne `package`-Anweisung

Nutzung von Paketen

- Die `import`-Anweisung, z.B. in den Formen
  `import packageName1.ClassName`
  `import packageName2.*`

- Dann kann man die Abkürzung `ClassName` schreiben, wenn `packageName2` nicht `ClassName` enthält.

- Konvention in Java-Bib:
  Paket - kleiner Buchstabe am Anfang
  Klasse - großer Buchstabe am Anfang.

- Nur am Anfang oder direkt nach der `package`-Anweisung.

- Physikalische Organisation von Klassen
- Ein Verzeichnis für jeden Paket-Namen.
- .class-Dateien der Klassen des Paketes muß in das Verzeichnis.

- Beispiel

(a) Die Datei `TestClass.java`

```
    package testPackage;
    public class TestClass {...}
```

Die Übersetzung ist `TestClass.class`

(c) “`mkdir testPackage`”, wenn `testPackage` nicht da.

(b) “`mv TestPachage.class testPackage`”

Sichtbarkeitsmodifikatoren

• Für Klassen:

- `public`-Klassen: *Überall* sichtbar.

- Nicht `public` Klassen (auch "package classes" genannt): Nur *innerhalb des Paketes* sichtbar.

• Für Instanzmethoden und Instanzvariablen:

- `public`-Methoden und Variablen: *Überall* sichtbar.

- Ohne Sichtbarkeitsmodifikator (ohne `public`, `protected`, `private`, auch package-methods/variables genannt): Nur *innerhalb des Paketes* sichtbar.

- `protected`-Methoden und Variablen:
Grob - Nur *in den Unterklassen oder innerhalb des Paketes* sichtbar.
Genauer - siehe die nächste Folie

- `private`-Methoden und Variablen:
Nur *innerhalb ihrer Klasse sichtbar.*

• Klassen des gleichen Paketes werden als *Freunde* betrachtet.

• Sichtbarkeit von Klassenmethoden und Klassenvariablen - analog - Details ausgelassen.

Sichtbarkeit von `protected`-Methoden und Variablen

• Zwei Aspekte der Sichtbarkeit von `protected`-Methoden und Variablen:

- Erben: Unterklassen erben alle `protected`-Methoden und Variablen der Oberklasse.

- Zugriff auf Instanzen:
(a) Jede Klasse kann auf jede `protected`-Methode/Variable in einer Klasse in *demselben* Paket zugreifen.
(b) Eine Klasse `B` kann auf eine `protected`-Methoden/Variablen der Oberklasse `A` nur über eine Referenz der Klasse `B` oder einer Unterklasse `C` von `B` zugreifen.

```
class A { protected int i; } // In Paket AA

public class B extends A {}  // In Paket BB

class C extends A {          // In Paket CC
C c = new C(); ... c.i ...;  // Legal
D d = new D(); ... d.i ...;  // Legal
A a = new C(); ... a.i ...;  // Illegal
           ... ((C) a).i ...  // Legal
B b = new B(); ... b.i ...;  // Illegal
           ... ((A) b).i ... // Illegal
}
public class D extends C {}  // In Paket CC
```


Mischung (oder Weglassung) von Modifikatoren

• Die allgemeinen Regeln:

- Mischung (oder Weglassung) von Modifikatoren ist möglich.

- Reihenfolge der Modifikatoren ist beliebig.

- Nicht jede Mischung oder Weglassung ist erlaubt.

- Manche Mischungen sind äquivalent.

- Die Bedeutung einer Modifikator kann die einer anderen implizieren.

- Die Bedeutung einer Mischung ist nicht unbedingt die Summe der Einschränkungen.

Einige Mischungen von Modifikatoren

• `private protected`-Methoden und Variablen:

Grob - Nur *in den Unterklassen (aber nicht in den Nicht-Unterklassen des gleichen Paketes)* sichtbar.
Genauer - analog zur letzten Folie

• Weglassen des `public`-Modifikators für Konstruktoren. Konstruktoren haben den (bzw keinen) `public`-Modifikator wenn die Klasse (bzw keinen) hat.

• `final` Modifikator
- `final` Klassen - Unterklassen unmöglich.
- `final` Methoden - Überschreiben unmöglich.
- `final static` Instanzvariable - Name für Konstante.

• `private` Methoden sind effektiv `final`.

• `final abstract` Methoden machen keinen Sinn. (Fehlermeldung.)

• `private abstract` Methoden machen keinen Sinn. (Fehlermeldung.)

Einige Mischungen von Modifikatoren - Fortsetzung

- `final protected` ist effektiv äquivalent zu `private`.

- Weil alle Methoden in Interface sind `abstract`!

- Wie kann man “read-only”, “write-only” Instanzvariablen haben?

- `private`-Variablen sind nur über Methoden derselben Klasse zu zugreifen. Programmieren von “read-only”, “write-only” in diesen Methoden!

- Keine `static`-Methoden in Interface.

- Weil alle Methoden in Interface sind `abstract`!

- `public`-Oberklasse und package-Unterklasse.

- Es macht Sinn!

- Package-Oberklasse und `public`-Unterklasse.

- Überall sichtbar der Oberklasse durch Unterklasse.
- `final`-Package-Oberklasse ist für außerhalb unsichtbar.

... ...