Objekte

Ziele der Verwendung von Objekten

für große, komplexe Programme nötig:

- viele Algorithmen zu Einheit integrieren
- inkrementelle Softwareentwicklung, einfache Wartbarkeit
- viele unterschiedliche Strukturierungsmöglichkeiten

Objekte helfen dabei durch:

- menschliche Fähigkeiten im Umgang mit Objekten
- Trennung zwischen Innen- und Außenansicht
- Verständnis als abstrakte Maschinen, verteilte Kontrolle

Faktorisierung

- Faktorisierung = Art der Zerlegung des Ganzen in Teile
- gute Fatorisierung $\Rightarrow$ einfach änderbar
- Faktorisierungsmöglichkeiten:
  - seiteneffektfreie Funktionen (Ausdrücke, ~~Zuweisungen~~)
  - Prozeduren mit Seiteneffekten (wie auf Arrays)
  - Prozeduren und Variablen zu Objekten zusammenfügen

Paradigmen und Programmzustände

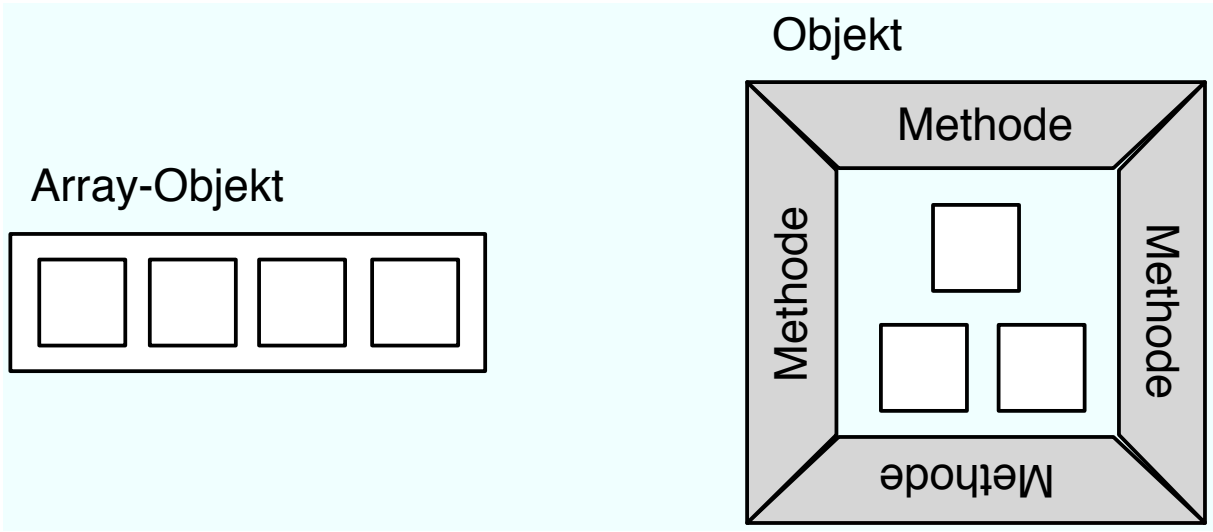
funktional: kein Programmzustand sichtbar

prozedural: nur globaler Programmzustand verständlich

objektorientiert: jedes einzelne Objekt verständlich

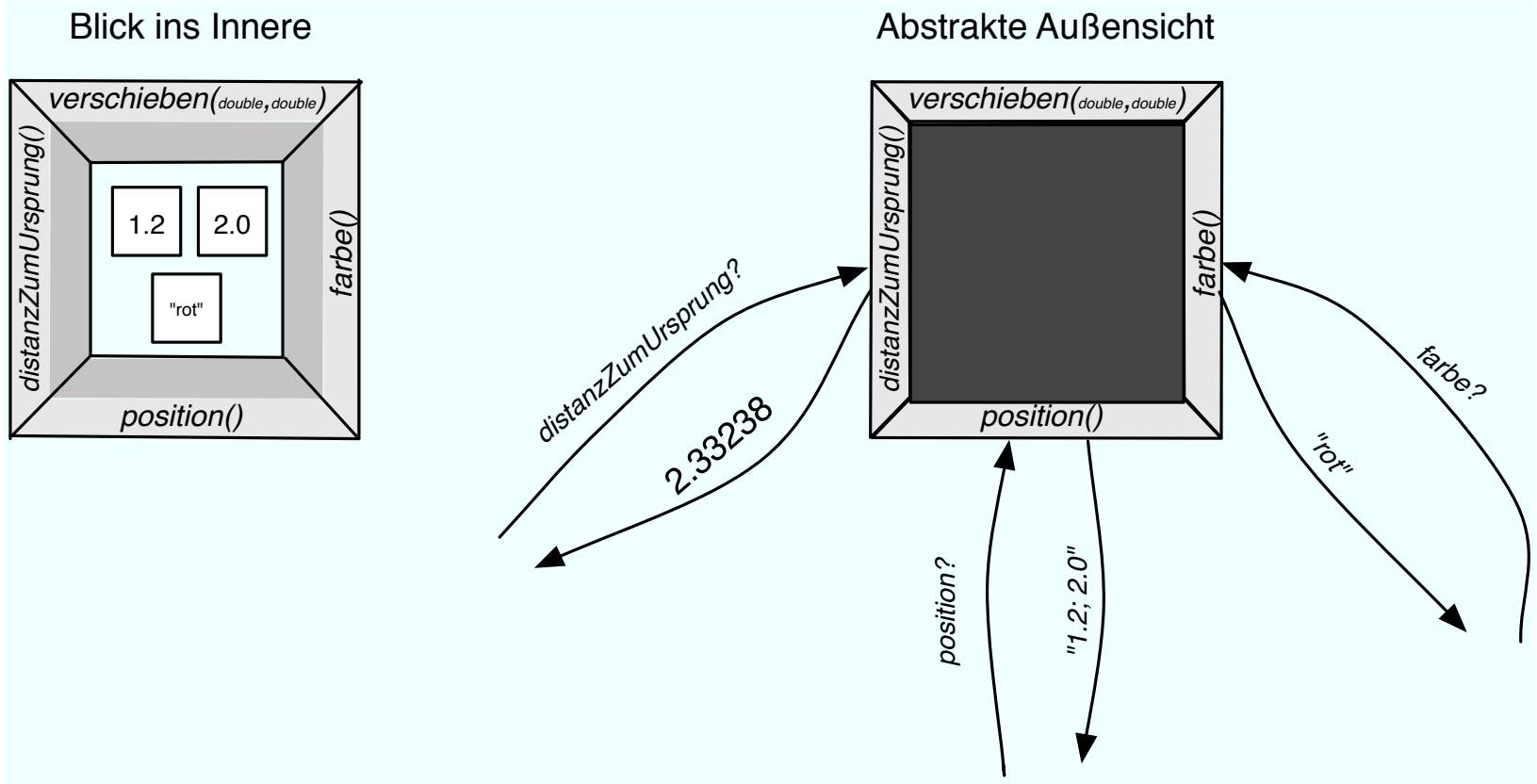
- objektorientiert ideal für große, komplexe Programme
- alles andere besser für kleine, einfache Programme

Kapselung



Kapselung = Objektvariablen und Methoden als Einheit

Datenabstraktion

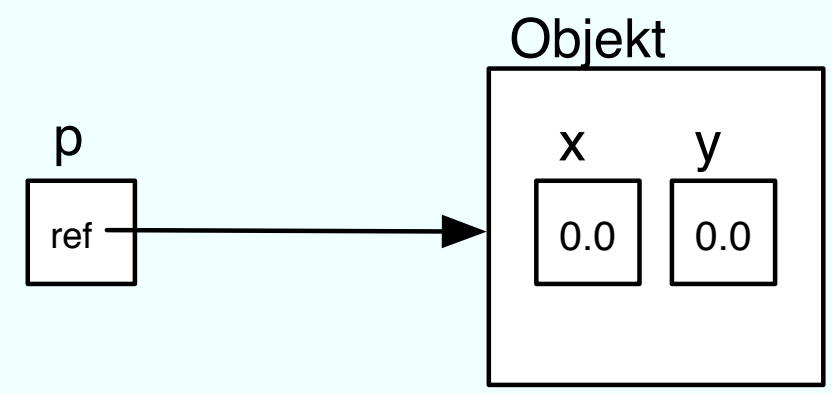


Datenabstraktion = Kapselung + Data Hiding

Klasse mit Objektvariablen

```
class Punkt {
    double x;  // x-Koordinate
    double y;  // y-Koordinate
}
```

```
    Punkt p;
    p = new Punkt();
    p.x = 0.0;
    p.y = 0.0;
```



```
class Punkt {
    double x = 0.0, y = 0.0;

    void verschiebe(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }
    double distanzZumUrsprung() {
        return Math.sqrt(x*x + y*y);
    }
}

class PunktTester {
    public static void main(String[] args) {
        Punkt p = new Punkt();
        p.verschiebe(1.0,1.5);
        System.out.println(p.distanzZumUrsprung());
    }
}
```

Sichtbarkeit

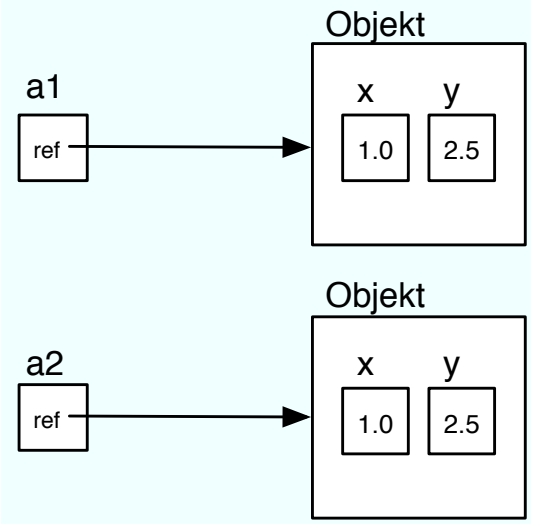
```
public class Punkt {
    private double x = 0.0;
    private double y = 0.0;

    public void verschiebe(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }

    public double distanzZumUrsprung() {
        return Math.sqrt(x*x + y*y);
    }
}
```

Sichtbarkeit auf Klasse, nicht Objekt!

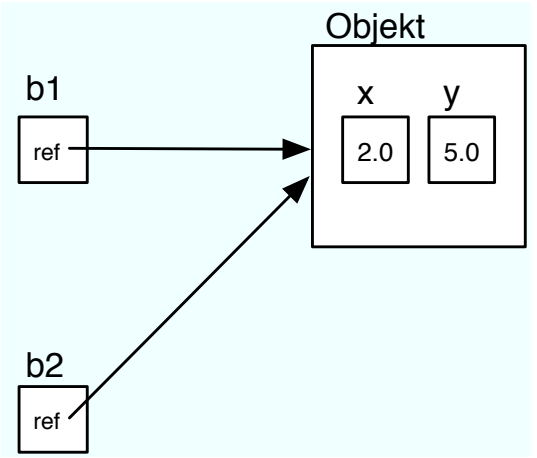
```
// zu Punkt hinzugefügt ...

private double entfernung(Punkt p) {
    double dx = x - p.x;
    double dy = y - p.y;
    return Math.sqrt(dx*dx + dy*dy);
}

public static double pfadlaenge(Punkt[] ps) {
    double sum = 0.0;
    for (int i=1; i < ps.length; i++)
        sum += ps[i-1].entfernung(ps[i]);
    return sum;
}
```

Gleichheit versus Identität

```
Punkt a1 = new Punkt();
Punkt a2 = new Punkt();

a1.verschiebe(1.0,2.5);
a2.verschiebe(1.0,2.5);
```




```
Punkt b1 = new Punkt();
Punkt b2 = b1;

b1.verschiebe(1.0,2.5);
b2.verschiebe(1.0,2.5);
```



Objekteigenschaften

Gleichheit: gleicher Zustand, kann sich jederzeit ändern

Vergleich: `s.equals(t)` (nur für Referenztypen)

Identität: unveränderlich, entsteht bei Objekterzeugung

Vergleich: `s == t` (nicht für String-Vergleich)

bei elementaren Typen entspricht Gleichheit der Identität

Verhalten: unveränderlich, nur von Klasse abhängig

abstrakt, kein dynamischer Vergleich möglich

Initialisierung mittels Konstruktor

```
public class Punkt {
    private double x;
    private double y;

    public Punkt(double initX, double initY) {
        x = initX;
        y = initY;
    }

    ...
}


    Punkt p = new Punkt(1.3, 2.7);
```

Beispiele für weitere Konstruktoren

```
public Punkt() {} // Default-Konstruktor, wenn
                  // kein Konstruktor definiert
```

```
public Punkt() {
    x = 0.0;
    y = 0.0;
}
```

```
public Punkt(Punkt p) {
    x = p.x;
    y = p.y;
}
```

```
public Punkt() {
    this(0.0,0.0);
}
```

```
public Punkt(Punkt p) {
    this(p.x,p.y);
}
```

Die Innenansicht – „this“

```
Punkt p = new Punkt(this);
p.verschiebe(2.0,3.0);
verschiebe(4.0,5.0);  // this.verschiebe(4.0,5.0);
```

```
public Punkt(double x, double y) {
    this.init(x,y);
}
private void init(double x, double y) {
    this.x = x;
    this.y = y;
}
```

```
public class Punkt {
    ...
    public void verschiebe(double x, double y) {...}
    public static double pfadlaenge(Punkt[] ps) {...}
    ...
}

public class Test {
    public static final int ANZAHL = 10;
    public static void main(String[] args) {
        Punkt[] punkte = new Punkt[ANZAHL];
        for (int i = 0; i < Test.ANZAHL; i++) {
            punkte[i] = new Punkt();
            punkte[i].verschiebe((double)i, 3.0*i);
        }
        double l = Punkt.pfadlaenge(punkte);
        System.out.println(l);
    }
}
```

Konstanten und Aufzählungen

```
    public static final double PI = 3.14;
```

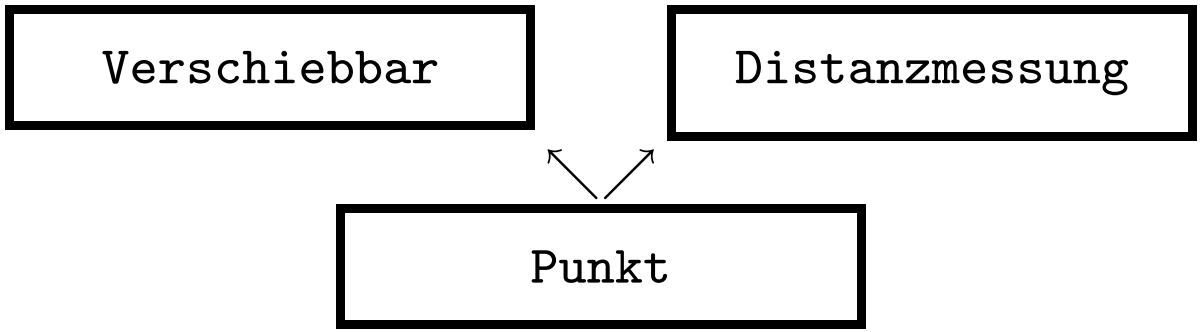
```
public class EnumExample {
    public enum Tag { MO, DI, MI, DO, FR, SA, SO }
    public static void main(String[] args) {
        System.out.println(Tag.valueOf("MO"));
        for (Tag t: Tag.values()) {
            System.out.println(t + " = " + t.ordinal());
        }
    }
}
```

Schnittstellenbeschreibungen

```
public interface Verschiebbar {
    // verschiebe Objekt um deltaX Einheiten nach rechts
    // und um deltaY Einheiten nach oben (wenn positiv)
    // bzw. nach links und unten (wenn negativ)
    void verschiebe(double deltaX, double deltaY);
}

public interface Distanzmessung {
    // gib Distanz zum Ursprung zurück
    double distanzZumUrsprung();
}
```

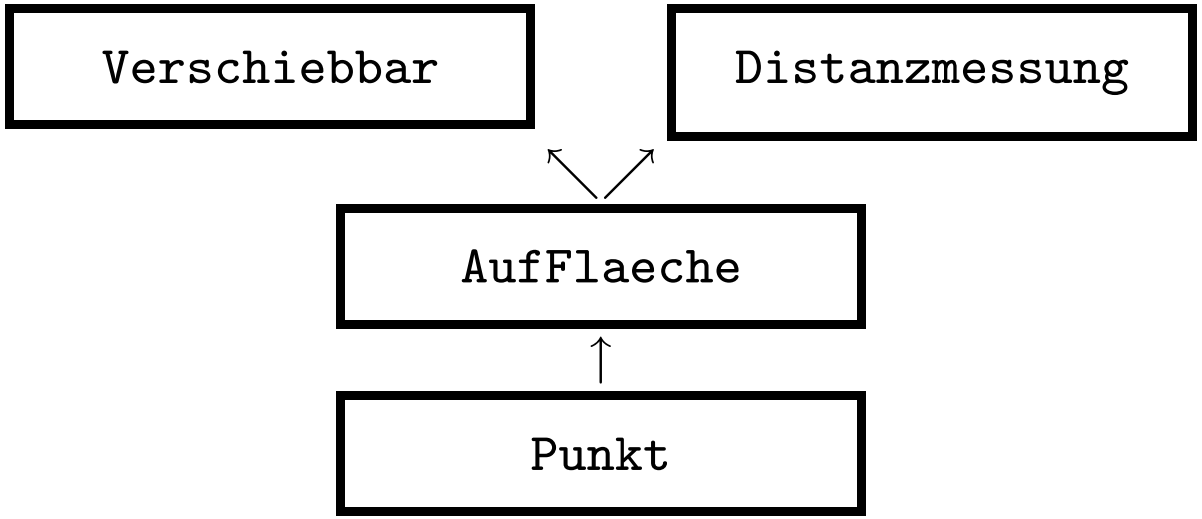
Implementierung von Schnittstellen

```
public class Punkt implements Verschiebbar,Distanzmessung {
    private double x, y;
    ...
    public void verschiebe(double dx, double dy) {...}
    public double distanzZumUrsprung() {...}
    public double entfernung(Punkt p) {...}
}
```



Beziehungen zwischen Schnittstellen

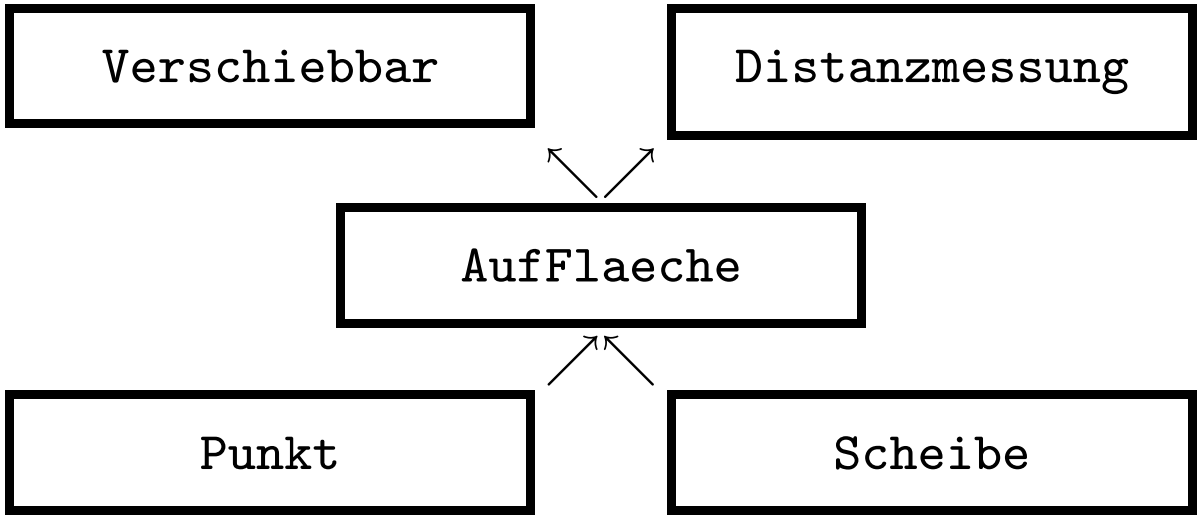
```
public interface AufFlaeche extends Verschiebbar,
                                    Distanzmessung {
    double entfernung(Punkt p);  // Entfernung zu p
}
public class Punkt implements AufFlaeche {...}
```



```
public class Scheibe implements AufFlaeche {
    private double x, y, r;

    public Scheibe(double ix, double iy, double radius) {
        x = ix; y = iy;
        r = Math.abs(radius); // darf nicht negativ sein
    }
    public void verschiebe(double dx, double dy) {
        x = x + dx; y = y + dy;
    }
    public double distanzZumUrsprung() {
        return this.entfernung(new Punkt(0.0, 0.0));
    }
    public double entfernung(Punkt p) {
        Punkt q = new Punkt(x,y);
        double d = q.entfernung(p);
        return Math.max(d - r, 0.0);
    }
}
```

Mehrere Implementierungen

Typdiagramm (Schnittstelle entspricht Typ):



```
public class FlaechenTester {
    private static void bewege(Verschiebbar v) {
        v.verschiebe(1.5,2.5);
    }
    private static void gibDistanzAus(Distanzmessung d) {
        System.out.println(d.distanzZumUrsprung());
    }
    public static void teste(AufFlaeche f) {
        for (int i = 0; i < 5; i++) {
            bewege(f);
            gibDistanzAus(f);
        }
    }
    public static void main(String[] args) {
        teste(new Punkt(0.0, 0.0));
        teste(new Scheibe(0.0, 0.0, 2.1));
    }
}
```