Übersicht

Übersicht

Einführung

- *Bisher:* Funktionen als Mittel zur Abstraktion
  - Dabei spielen Daten eine untergeordnete Rolle
- Objektorientierte Sichtweise
  - legt Fokus auf *Daten*
  - verknüpft Daten und *Operationen* („Funktionen“)
- Im Mittelpunkt einer Anwendung steht dabei
  - weniger **Wie?** – Ausführung als Folge von Anweisungen
  - vielmehr **Was?** – Welche Objekte? Welches Verhalten?
- Zentrale Mittel zur Abstraktion
  - *Objekte* mit Zustand (Daten) und Verhalten (Methoden)
  - *Klassen* von gleichartigen Objekten
  - Relationen zwischen Klassen (Vererbung)
- Java ist für objektorientierte Programmierung entworfen
  - Sprache unterstützt – *und fordert* – objektorientierte Sichtweise

Beispiel: Addition

Was bedeutet $x + y$?

Für ...

- Natürliche oder ganze Zahlen
- Rationale Zahlen (Brüche)
- Komplexe Zahlen (mit Real- und Imaginärteil)
- Vektoren und Matrizen
- Polynome
- ...

Unterschiedliches Verhalten für unterschiedliche *Typen* von Operanden.

Beispiel: OO für rationale Zahlen

- Klassische OOP Sichtweise (z.B. in *Smalltalk*)
- Zwei Objekte p und q (aus „Klasse“ $\mathbb{Q}$)
- Wir wollen $\mathrm{p} \leftarrow \mathrm{p} + \mathrm{q}$ berechnen.

- q sendet Nachricht **add** an p (*message passing*)
- p reagiert auf diese Nachricht und ändert seinen Zustand
  - p „entscheidet“ über richtige Reaktion (*dynamic dispatch*)
  - Diese Entscheidung definiert Verhalten
  - *Hier:* Addition $\mathrm{p} \leftarrow \mathrm{p} + \mathrm{q}$ und anschließendes Kürzen

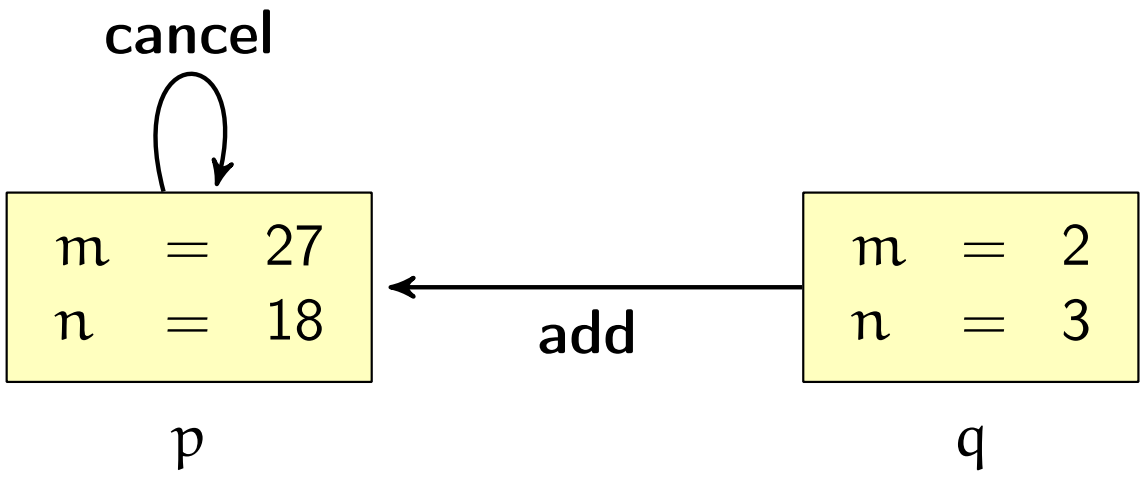
- Zustandsänderung als Reaktion auf Nachrichten

Beispiel: Rationale Zahl als Objekt

- *Objektorientierte Sichtweise*: Bruch als ein *Objekt* mit
  - Zustand: Zähler $\mathfrak{m}$, Nenner $\mathfrak{n}$ (= „Datenstruktur“)
  - Verhalten: Methode zum Addieren und Kürzen
- Zustandsänderung als Reaktion auf Nachrichten
  - *Kapselung* des Zustands im Objekt
  - *Schnittstelle* für Zustandsänderung
  - Zustand muss nicht vollständig nach außen sichtbar sein (Prinzip der Geheimhaltung, *information hiding*)
- Jede Objekt „entscheidet“ selbst über „richtige“ Reaktion

Beispiel: OO für rationale Zahlen $q \in \mathbb{Q}$

- Für $p = \frac{5}{6}, q = \frac{2}{3}$ berechne $p \leftarrow p + q$
- p und q sind Objekte



1. q sendet Nachricht **add** an p
2. p ändert Zustand nach Addition: $p = \frac{5}{6} + \frac{2}{3} = \frac{27}{18}$
3. p sendet Nachricht **cancel** an *sich selbst*
4. p ändert Zustand nach Kürzen: $p = \frac{3}{2}$

Zum Begriff *Objektorientierte Programmierung*

Geprägt von Alan Kay (Smalltalk u.a; Turing Award 2003!):

> *[. . .] OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. [. . .]*

[Quelle]

> *[. . .] The big idea is "messaging" [. . .]*

[Quelle]

> *Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.*

[Quelle]

Übersicht

Objekte

Definition (Objekt)

Ein *Objekt* repräsentiert eine Entität mit Zustand und Verhalten. Jedes Objekt hat eine eindeutige Identität.

- Zustand = Daten $\Rightarrow$ Objekt ist ein Datentyp
  - Beispiel: Zähler m und Nenner n einer rationalen Zahl $\mathrm{q} = \frac{\mathrm{m}}{\mathrm{n}}$
- Dazu kommt Verhalten, z.B. Operationen auf Datentyp
  - Beispiel: Addition $+ : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Identität = Eigenschaft, durch die sich das Objekt von anderen unterscheidet
  - wie etwa Bezeichner $\mathrm{p} \neq \mathrm{q}$, eindeutige Ausweisnummer, Kfz-Kennzeichen,. . . , (Speicher-)Adresse
- *Später:* Zusammenfassen von gleichartigen O. in Klassen: Objekt = Instanz (konkretes Ding) einer (abstrakten) Klasse

Attribute und Methoden

- Zustand eines Objekts = Menge von Attributen
  - Beispiel: Zähler $\mathfrak{m} \in \mathbb{Z}$, Nenner $\mathfrak{n} \in \mathbb{Z}$
  - Auch (Referenzen auf) Objekte als Attribute möglich („Besitz“)
  - In Java *member variables* (im Sinn von „*member of an object*“)
- Senden einer Nachricht = Aufruf einer Methode
  - Methode kann den Zustand des Objekts ändern
  - Beispiel: `add` Methode zur Addition von Brüchen

Definition (Methode)

Eine *Methode* ist eine Funktion im Kontext eines Objekts.

Prinzip der Kapselung

Der Zustand eines Objekts kann *nur durch seine Methoden* verändert werden.

Sichtbarkeit von Attributen und Methoden

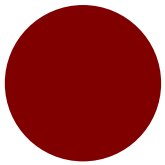
Geheimhaltung

Attribute und Methoden müssen nicht nach außen sichtbar sein!

- Zugriff nur auf eigene Attribute
- Warum *information hiding*?
  - Anwender interessiert Funktionalität – *nicht* Umsetzung
  - Anwender sieht nur Schnittstelle (z.B. wenige Methoden)
  - Implementierung (auch Algorithmus) transparent austauschbar
  - *Programme sind leichter zu warten!*
- Beispiel
  - Attribut `length` – in welcher Einheit speichern?
  - Kein direkter Zugriff auf `length`
  - Stattdessen Methoden `length_in_cm`, `length_in_inch`
  - Einheit und Umrechnung bleiben versteckt
- Oft spezielle Zugriffsmethoden (*accessor*, *getter*)
- Geheimhaltungsprinzip: **So viel wie möglich „verstecken“!**

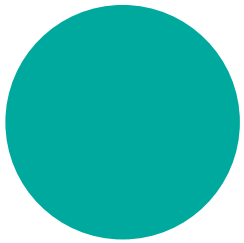
Beispiel: geometrische Figuren – Kreise

- Anwendungsbeispiel: Zeichenprogramm
- Wir beginnen mit **Kreisen**.
- Attribute
  - Mittelpunkt x, y, Radius r
  - Farbe c
- Methoden
  - **position** — Position abfragen*
  - **translate** $(\Delta x, \Delta y)$ — Verschieben
  - **set_color** (c) — Farbe setzen
  - **area** — Fläche berechnen*
  - $\star \Rightarrow$ *Keine Zustandsänderung*



$x = 0 \mid y = 0 \mid r = 1 \mid c = \text{Maroon}$

k_1



$x = 5 \mid y = 0 \mid r = \frac{3}{2} \mid c = \text{Emerald}$

k_2

Klassen von Objekten

Gleichartige Objekte werden in einer *Klasse* zusammengefasst.

Umgekehrt ist eine *Klasse* ein abstraktes Modell von Objekten.

- *Gleichartig* bedeutet i.d.R.
  - gleiche Attribute (gleiche Zustandsmenge)
  - gleiche Methoden (gleiches Verhalten)
- Beispiel: Kreise als Objekte
  - Struktur gleich für k_1 und k_2 (Attribute, Methoden)
  - *Ein* Kreis beschreibt Struktur für *alle* Kreise

Konstruktoren erzeugen Instanzen

- Objekte heißen auch Instanzen von Klassen
  - Die Erzeugung eines Objekts heißt auch *Instantiierung*
  - Dabei wird der initiale Zustand festgelegt (Initialisierung)

Instantiierung durch Konstruktor

Die *Instantiierung* erfolgt i.d.R. durch spezielle Methoden: *Konstruktoren*.

- Verschiedene Konstruktoren für eine Klasse möglich, z.B.
  - Kreis definiert durch Mittelpunkt und Radius
  - Konstruiere Kreis durch 3 Punkte

Klassenmethoden und Klassenattribute

Klassenmethoden (*-attribute*) sind Methoden (Attribute) im Kontext einer Klasse.

- Idealerweise sind auch Klassen selbst Objekte (*Das ist in Java nicht so!*)
- Eigenschaft der Klasse (=Eigenschaft *aller* Instanzen), z.B.
  - Attribut n_F der Klasse Figur zählt Anzahl aller Instanzen.
  - Methode **count_figure** zählt $n_F \leftarrow n_F + 1$.
  - Konstruktoren rufen diese Methode auf.

Beispiel: geometrische Figuren – Rechtecke

- Wir wollen das Programm um **Rechtecke** erweitern.
- Attribute
  - Linke untere Ecke x, y, Breite w, Höhe h
  - Farbe c
- Gleiche Methoden wie für Kreise
- *Für welche Methoden unterscheidet sich die Implementierung?*
- In diesem Fall nur für **area**
  - Kreisfläche $A_k = r^2\pi$
  - Rechtecksfläche $A_r = w \cdot h$
- Andere Methoden bleiben gleich, z.B.
  - **translate**: $(x, y) \to (x + \Delta x, y + \Delta y)$

$x = 0 \mid y = 0 \mid w = 2 \mid h = \frac{1}{2} \mid c = \text{Melon}$

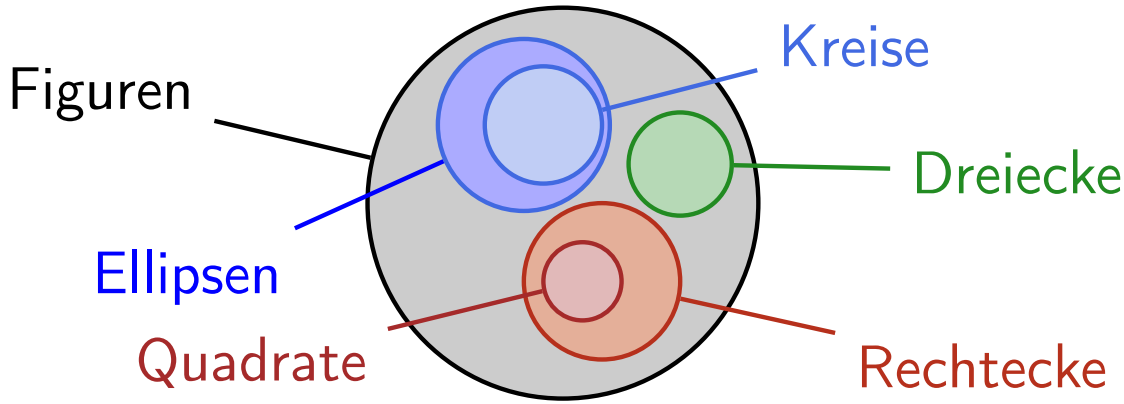
r_1

$x = 6 \mid y = 0 \mid w = 3 \mid h = 1 \mid c = \text{Plum}$

r_2

Nutze Gemeinsamkeiten – Hierarchie von Klassen

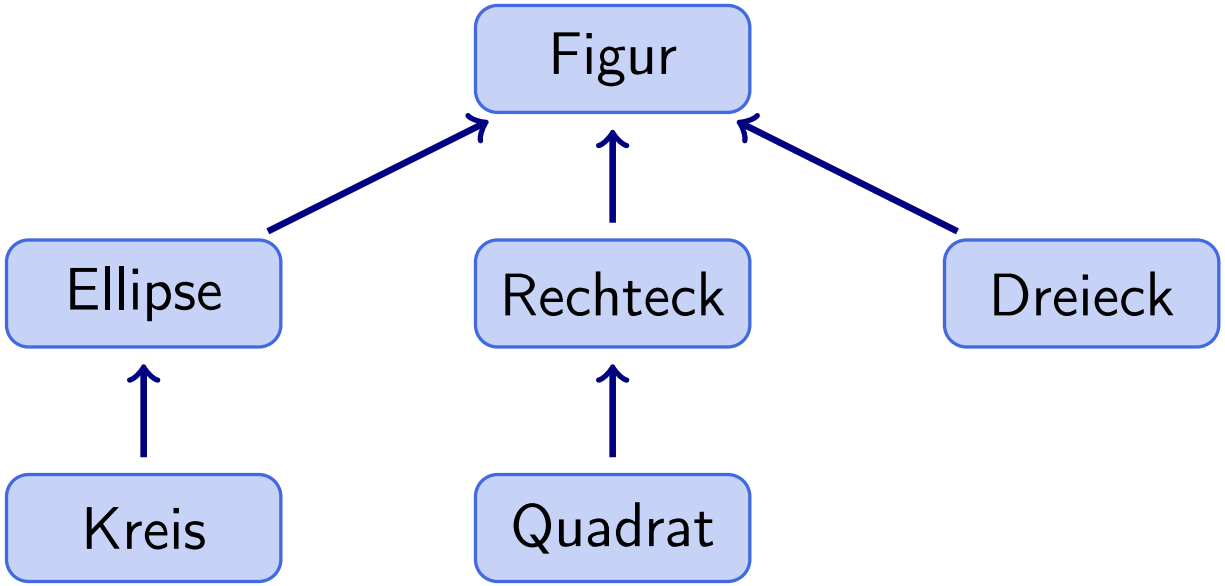
- Klassen von Kreisen, Rechtecken,. . . *oder allgemein* **Figuren**
  - Attribute und Methoden unterscheiden sich zum Teil nicht
  - Semantische Gemeinsamkeit: geometrische Figuren
- Die allgemeinere Klasse **Figuren** enthält die Klassen
  - Kreis
  - Rechteck
  - *und viele mehr, die vielleicht noch implementiert werden*
- **Prinzip**: Hierarchie von Klassen



- Hier Spezialfall: Figur als *abstraktes Konzept*
  - Nicht konkret darstellbar!
  - Aber: Wohldefiniertes Verhalten – Was macht eine Figur aus?

Klassenhierarchie – Vererbung

- Die Klassenhierarchie lässt sich besser als *Baum* darstellen



- Relation $\longrightarrow$
  - *Ist-ein* (*is-a*): „Jeder Kreis *ist eine* Ellipse.“
  - Eltern-Kind Beziehung: Kind erbt Eigenschaften von Elternteil
  - Relation ist *transitiv*: „Jeder Kreis *ist eine* Figur.“
- Beispiel
  - Klasse Rechteck (Kind) erbt von Klasse Figur (Elternteil)
  - Quadrat erbt von Rechteck, damit auch von Figur

Vererbung

- Vererbung (*inheritance*) =
  Vererbung von Eigenschaften einer Klasse
- Für objektorientierte Programmierung bedeutet das
  - Vererbung von Attributen (Zustandsmenge)
  - Vererbung von Methoden (Verhalten)
- Wohldefinierte Schnittstelle zur Interaktion von/mit Objekten
  - z.B. **position**, **translate** für *beliebige* Figur
- Wiederverwendung von Programmcode
  - Nur *Unterschiede* werden neu implementiert!
  - Teils veränderte Methoden, z.B. **area** Methode
- Kind-Klasse kann weiter *spezialisiert* sein
  - Zusätzliche Attribute, zusätzliche Methoden
  - z.B. Methode **triangle_height** berechnet Höhe eines Dreiecks

Begriffe zu Klassen und Vererbung

Abgeleitete Klasse und Basisklasse

Wenn eine Klasse B von Klasse A erbt ($\mathrm{A} \leftarrow \mathrm{B}$), dann sagt man: B ist von A *abgeleitet*. – B ist *Unterklasse* (*subclass*) von A.

Umgekehrt heißt A auch *Basisklasse* (*base class*) von B.

- z.B. Rechteck ist von Figur abgeleitet, und
  Figur ist Basisklasse von Rechteck.

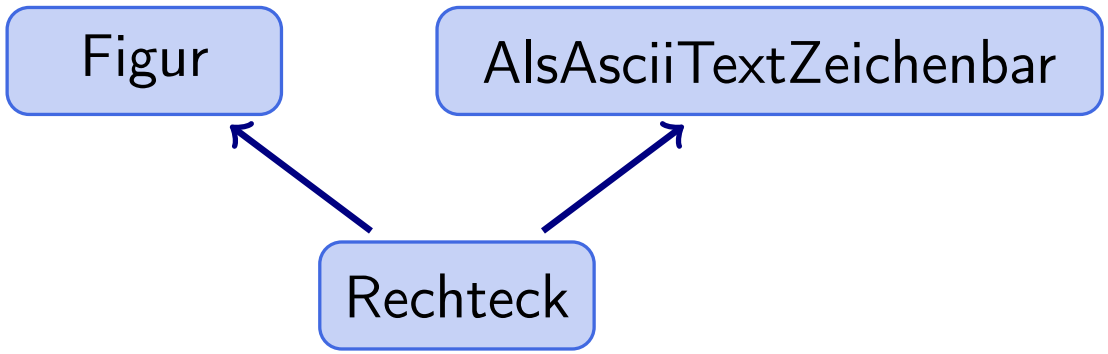
Abstrakte Klasse

Eine Klasse, zu der keine Objekte existieren (oder erzeugt werden können), heißt *abstrakte* Klasse.

- Eine abstrakte Klasse repräsentiert ein abstraktes Konzept/Modell und kann nicht *instantiiert* werden.
- Im Beispiel ist genau Figur eine abstrakte Klasse

Mehrfachvererbung

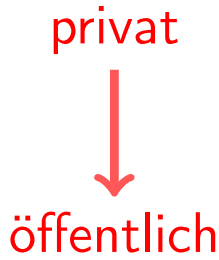
- Mehrfachvererbung *multiple inhertiance* =
  Ableiten von *mehr als einer* Basisklasse
- Erlaubt potentiell beliebig viele Eltern-Klassen
- Kann beim Modellieren von Klassen nötig/hilfreich sein
- Beispiel



- Direkte Unterstützung durch manche Sprachen
  - z.B. C++
  - Java geht einen anderen Weg (*interfaces*)! Später mehr...

Vererbung und Sichtbarkeit

- Attribute und Methoden müssen nicht nach außen sichtbar sein
- Was genau bedeutet *nach außen*? – Sichtbarkeit ...
  - nur für dasselbe Objekt?
  - nur für Instanzen der gleichen Klasse?
  - nur für Instanzen abgeleiteter Klassen?
  - für beliebige andere Objekte?



- Sichtbarkeit von Attributen und Methoden
  - wird pro Klasse definiert und/oder
  - wird mit vererbt
    - „private/öffentliche Erbschaft“
    - geerbte Eigenschaften alle öffentlich/privat
- Zweck: *Geheimhaltung*
  - Grundsätzlich soviel verstecken wie möglich, und nur soviel öffentlich wie nötig

Vererbung und Polymorphie

- Polymorphe *Funktionen*: Unterscheidung nach Signaturen
- Polymorphie in der objektorientierten Programmierung

Definition (Polymorphe Methode)

Eine Methode heißt *polymorph*, wenn sie für verschiedene Klassen die gleiche Signatur hat, aber verschieden definiert ist.

  - Gleiche Schnittstelle, aber unterschiedliches Verhalten
  - Objekt entscheidet, wie es auf Nachricht reagiert
- Bezeichnung Polymorphie im Zusammenhang mit *Vererbung*
  - Basisklasse definiert Methode
  - Abgeleitete Klasse überschreibt Definition (*method overriding*)
  - Im Unterschied zum *overloading* gleiche Schnittstelle!
- Beispiel
  - Berechnung des Flächeninhalts mit **area** Methode
  - Definiert in Klasse Figur, überschrieben z.B. in Kreis, Rechteck

Abstrakte Methoden

- **area** definiert in Klasse Figur — Wie kann das sein?
  - Figur ist eine *abstrakte Klasse*
  - Insbesondere Flächeninhalt ist undefiniert!

Definition (Abstrakte Methode)

Eine Methode heißt *abstrakt*, wenn lediglich ihre Signatur gegeben ist, nicht aber eine Definition.

  - Festlegung einer Schnittstelle
  - Implementierung bleibt *undefiniert*
  - Methode muss für „konkrete“ Unterklassen definiert werden!

- Damit alternative Definition der *abstrakten Klasse*

Definition (Abstrakte Klasse)

Eine Klasse heißt *abstrakt*, wenn sie mindestens eine abstrakte Methode enthält.

  - Es können keine Instanzen der Klasse erzeugt werden!

Zusammenfassung

- Objektorientierung als *Programmierparadigma*
  - Abstraktion: Daten und Operationen
  - Beschreibe Interaktion von Objekten
  - Prinzipien: *Kapselung*, *Geheimhaltung*
- Objekt =

  Identität + Zustand (Attribute) + Verhalten (Methoden)
- Klassen von Objekten
  - Objekte als *Instanzen* von Klassen (durch Konstruktor)
  - *Klassenattribute* und *Klassenmethoden*
- Klassenhierarchien und Vererbung
  - Abgeleitete Klasse und Basisklasse
  - Abstrakte Klasse
  - *Mehrfachvererbung*
- Vererbung und Polymorphie

- Als nächstes: *Umsetzung von OOP in Java*

Übersicht

Objektorientierte Programmierung in Java

- *Umsetzung des OO Paradigma in Java*
- Klassen `class`
  - Attribute und Methoden `public`, `private`, `protected`
  - Vererbung `extends`
- Objekte
  - Konstruktoren `new`
  - Referenzen
  - Selbstreferenz `this`
  - Referenz auf Basisklasse (-objekt) `super`
- Vererbung und Polymorphie
  - Abstrakte Klassen und Methoden `abstract`
- Klassenattribute und Klassenmethoden `static`
  - Java Programme `main()`
- Schnittstellen `implements`, `interface`
  - Konzept von *interfaces* ersetzt Mehrfachvererbung

Definition einer Klasse

- Grundsätzlich jede Definition in einer eigenen *Datei*
  - Java erzwingt saubere Strukturierung in Dateien
  - Dateiname = Klassenname + Endung `.java`
  - Groß- und Kleinschreibung beachten!
- Definition durch `class { ... }`
- **Beispiel:** Datei `Point.java`

```
class Point {
  float x,y;
}
```

  - Klasse `Point`
  - mit Attributen `x`, `y`
- Java kennt auch nested classes.

Definition von Attributen und Sichtbarkeit

- Attribute: *member variables*
  - Definition wie Variablen aber in Klassendefinition
  - Im Gegensatz zu (lokalen) Variablen in Methoden
  - Java erlaubt Initialisierung wie für Variablen (d.h. außerhalb eines Konstruktors)
- *Sichtbarkeit* kann beschränkt werden
  - `private` – Instanzen der *gleichen* Klasse
  - `protected` – Instanzen der gleichen oder *abgeleiteter* Klassen*
  - `public` – explizit öffentlich sichtbar = keine Beschränkung
  - *(leer)* – gleiche Klasse / gleiches *package** (*package-private*)
- **Beispiel**

```
class Point {
  public float x,y;
};
```

- Vorsicht: `public` Attribute verletzen Geheimhaltungsprinzip!

Zugriff auf Attribute

- Zugriff auf Attribute mittels `an_object.an_atribute` (Punkt)

```
Point p=...;   // some initialization
float x=p.x;   // assume public attribute x
```

- Zugriff "*von außen*" ist ggf. beschränkt z.B. durch `private`
- Zugriff *innerhalb der Klasse* kann über `this` erfolgen

```
{ // assume block in method definition
  this.x+=delta_x;
}
```

- Die Selbstreferenz `this` bezeichnet das Objekt, in dessen Kontext die "aktuelle" Methode ausgeführt wird.
- Aufruf von Methoden erfolgt ähnlich (Punkt, `this`)

Sichtbarkeit von Klassen und Methoden

- Sichtbarkeit von Methoden und Klassen kann beschränkt werden.
- Mit `private`, `protected` (`public` oder „leer“)
  - Klassen – *Wer darf Klasse verwenden?*
  - Methoden – *Wer darf Methode aufrufen?*
- Wirkung analog zu Attributen

```
public class Point {
  public float x,y;
};
```

Definition von Methoden

- Methoden werden ähnlich wie Funktionen definiert.
  (*Tatsächlich kennt Java nur Methoden und keine Funktionen.*)
- *Sichtbarkeit* von Methoden genau so wie für Attribute
- Methoden können *überladen* werden, d.h.
  gleicher Name, unterschiedliche Signatur

```
class Point {
  public float x,y;

  public void translate(float x,float y) {
    this.x+=x; // self-reference
    this.y+=y; // ... in a minute
  }
  public void translate(Point p) {
    translate(p.x,p.y);
  }
}
```

Vererbung

- Ableiten von *einer* Basisklasse mit `extends`
  - Definition der neuen Klasse als Unterklasse
  - Unterklasse *erweitert* Basisklasse im Sinn von Spezialisierung

```
class Figure {                              // base class
  protected Point p;

  Point position() { return p; }
  void translate(float x,float y) {
    p.translate(x,y);
} }
```

```
class Rectangle extends Figure { ... }
```

```
class Square extends Rectangle { ... }
```

Was ist hier schlecht gelöst?

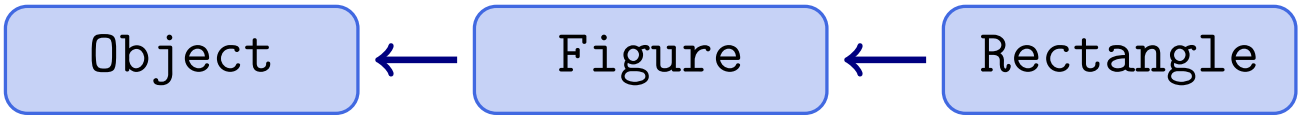
```
class Figure {                              // base class
  protected Point p;

  Point position() { return p; }

  void translate(float x,float y) {
    p.translate(x,y);
  }
  // ...
}
```

Vererbung: `class Object`

- *Jede* Java Klasse ist implizit von der Klasse `Object` abgeleitet
  - *implizit* = ohne weiteres Zutun (ohne explizites `extends`)
  - Gemeinsame Basisklasse `java.lang.Object`
  - Gemeinsame Basis von Methoden, die für jedes Objekt definiert sind



- Auszug aus Methoden von `Object`
  - `String toString()` — liefert lesbare Darstellung
  - `boolean equals(Object other)` — teste Gleichheit
  - `protected Object clone()` — liefert Kopie des Objekts
  - `protected void finalize()` — Aufruf vor Zerstörung
  - `Class getClass()` — liefert eigene Klasse (Reflexion)
  - `int hashCode()` — *nächstes Semester*
  - diverse Methoden zur Synchronisation

Instantiierung von Klassen: Konstruktoren

- Instantiierung = Erzeugung von Objekten
- Instantiierung erfolgt durch spezielle Methoden: Konstruktoren
  - Name des Konstruktors = Name der Klasse
  - Konstruktor übernimmt Initialisierung ($\Rightarrow$ definierter Zustand)
  - Verschiedene Konstruktoren pro Klasse möglich
- **Beispiel**

```
class Point {
    float x,y;

    public Point() { x=y=0; }
    public Point(float x,float y) {
      this.x=x;
      this.y=y;
    }
}
```

Instantiierung von Klassen

- Aufruf des Konstruktors durch `new`
- `new Class(...)`
  - erzeugt eine neue Instanz der Klasse `Class`
  - ruft dafür entsprechenden *Konstruktor* (Signatur!) auf
  - liefert eine *Referenz* auf das erzeugte Objekt
- **Beispiel**

```
Point x=new Point(1.0f,2.0f);
Point y=new Point();

Point z; // defines only reference,
         //  but no instance!
```

- Falls `new` nicht verwendet wird, (hier z.B. `z`)
  - wird kein Objekt instantiiert, stattdessen
  - wird nur eine *Referenz* mit Wert `null` definiert.
- `null` bedeutet, das zugehörige Objekt ist *undefiniert*!

Objekte und Referenzen

- Java arbeitet immer mit Referenzen auf Objekte!
  - $\Rightarrow$ Zuweisung (=) und Vergleich (==) von Referenzen!
  - Referenz = *Identität* von Objekten
  - Ausnahme: Primitive Datentypen. Die sind *keine* Objekte!
- Auch Objekte als Attribute möglich („Besitz“ des Objekts)
  - z.B. Position `p` in Klasse `Figure`
  - Auch das ist eine Referenz!
- „Konstante“ für *undefinierte* Referenz: `null`
- Noch ein Spezialfall: *Selbstreferenz* `this`
  - `this` = Referenz auf *dieses, das „eigene“* Objekt
  - Nur innerhalb von Methoden definiert
  - z.B. um eindeutig auf ein Attribut zu verweisen

```
public Point(float x,float y) {
  this.x=x;       // argument names shadow
  this.y=y;       // ... attributes
}
```

Nochmal: Zugriff auf Attribute und Methoden

- Zugriff auf Attribute und Methoden mit . (Punkt) Operator
- **Syntax:** `object.attribute` bzw. `object.method(...)`
- Beispiel

```
Point p=new Point(1.0f,2.0f);
Point q=new Point(p.x,p.y);

p.translate(new Point(1.0f,0.0f));

float dx=q.x-p.x;
```

- Zugriff auf Methoden/Attribute der *Basisklasse*: `super`
  - Zugriff nach Überschreiben (gleiche Namen) noch möglich
  - **Syntax:** `super.attribute` bzw. `super.method(...)`
  - Ähnlich wie `this` Referenz auf *Instanz der Basisklasse*

Konstruktoren und Vererbung

- `super(...)` ruft Konstruktor der Basisklasse auf
- Nur im Konstruktor der abgeleiteten Klasse definiert

```
class Square extends Rectangle {
  public Square(float a) { super(a,a); }
}
```

- Ähnlich: `this(...)` ruft anderen K. der *gleichen* Klasse auf

```
class Rectangle extends Figure {
  public Rectangle(float w, float h) { ... }
  public Rectangle(float aspect) {
     this(aspect, 1.0f); }
}
```

Vererbung und Polymorphie

- Überschreiben von Methoden
  - Methode der abgeleiteten Klasse mit
  - gleicher Signatur
- **Beispiel**

```
class Figure {
  public float area() { return -1.0f; } // undef
}
class Rectangle extends Figure {
  private float w,h;
  Rectangle(float w,float h) { this.w=w; this.h=h; }
  public float area() { return w*h; }
}
class Circle extends Figure {
  private float r;
  Circle(float r) { this.r=r; }
  public float area() { return r*r*3.14159265; }
}
```

Vererbung und Polymorphie – Beispiel

- Überschreiben von Methoden

```
class Rectangle extends Figure {
  public float area() { return w*h; }
  ...
class Circle extends Figure {
  public float area() { return r*r*3.14159265; }
  ...
```

- **Beispiel:** Aufruf von Methoden

```
Rectangle r=new Rectangle(2,3);
Circle    c=new Circle(1);

float a=r.area();          // ⇒ a = 6
a=c.area();                // ⇒ a = 3.14159265

Figure f=null;
f=r; a=f.area();           // ⇒ a = 6
f=c; a=f.area();           // ⇒ a = 3.14159265
```

Vererbung und Polymorphie – Selbsttest

```
class A {
  public void m() { System.out.println("A"); }
}
class B extends A {
  public void m() { System.out.println("B"); }
}
class C extends B {
  public void m() { System.out.println("C"); }
}
```

```
A a=new A(); B b=new B(); C c=new C();   A x;

x=a; x.m();
x=b; x.m();
x=c; x.m();
b=c; b.m();
b=a; b.m();
```

Vererbung und Polymorphie – Selbsttest

```
A a=new A(); B b=new B(); C c=new C();   A x;

x=a; x.m();
x=b; x.m();
x=c; x.m();
b=c; b.m();
b=a; b.m();
```

```
A
B
C
C
compile error: incompatible types, found: A, required: B
```

Überschreibe Methoden von Basisklasse `Object`

```
class Point {
  String toString() {
    return "(" + x + ","+ y + ")";
  }
  boolean equals(Object other) {
    if (other instanceof Point) {
      Point q = (Point) other;
      return q.x == this.x && q.y == this.y;
    }
    return false;
  }
  Object clone() {
    return new Point(this.x,this.y);
  }
}
```

Abstrakte Methoden und Abstrakte Klassen

- `Figure` soll eine *abstrakte Klasse* sein, d.h.
  - nicht instantiierbar
  - mindestens eine Methode ist *abstrakt* (nur Signatur)
- Das wollen wir vermeiden:

```
class Figure {
  public float area() { return -1.0f; }
}
```

  - Definition der Methode ist nicht sinnvoll! Nur Platzhalter.
- Stattdessen definieren wir eine abstrakte Methode

```
abstract class Figure {
  public abstract float area();
}
```

  - *Abstrakte Methode* `area()` ist nicht definiert und ...
  - ...muss von einer abgeleiteten Klasse definiert werden.
  - `area()` macht `Figure` zu einer *abstrakten Klasse*

Klassenattribute und Klassenmethoden

- Attribute/Methoden im Kontext einer Klasse
  - Eigenschaft der Klasse – nicht eines einzelnen Objekts
  - $\Rightarrow$ „global“, d.h. gleich für *alle* Objekte
- Definition als `static` Attribute bzw. Methode
- **Beispiel:** Anzahl Instanzen von Figuren

```
class Figure {
  protected static int mCount=0;

  public Figure() { ++mCount; }
  public static int count() { return mCount; }
}
```

- Klassen*attribut* `mCount` (Zähler)
- Konstruktor `Figure()` erhöht Zähler
- Klassen*methode* `count()` liefert aktuellen Wert
- Zählt alle Instanzen von `Figure`, also auch `Rectangle`, ...

Funktionen in Java = Klassenmethoden

- Java kennt keine Funktionen — Java kennt nur Methoden
  - Java bleibt dem OO Paradigma hier treu
  - C++ beispielsweise tut das nicht
- Deshalb: Funktionen $\approx$ Klassenmethoden
  - Für uns im wesentlichen ein syntaktischer Unterschied
  - Wir benötigten ein Gerüst, z.B. zur Definition von `gcd(m,n)`
  - Klasse `GCD` nur als „Namensraum“, keine Instanzen

```
class GCD {
  final int UNDEF=0;

  public static int gcd(int m,int n) {
    if (m==0 && n==0) return UNDEF;
    else if (m==0)    return n;
    else if (m>n)     return gcd(n,m);
    else              return gcd(m,n-m);
  }
}
```

Die Klassenmethode `main()`

- Wir kennen bereits eine Klassenmethode: main()
- `public static void main(String[] args)`
  - Einsprungpunkt in ein Java-*Programm*
  - `args` sind Kommandozeilenargumente an `java`
  - Beispiel: `java Foo bar 42` → `args={"bar","42"}`
- Ein Java-Programm besteht aus
  - einer Menge von Klassen mit
  - *genau einer* Klasse, die die Klassenmethode `main()` definiert.

```
class Hello {

  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Schnittstellen / *interfaces*

- *interface* bezeichnet in Java eine Art abstrakter Klasse
  - Alle Methoden sind implizit `public` und `abstract` definiert
  - Keine Attribute! – Konstanten (`static final`) sind erlaubt.
- Wozu Schnittstellen?
  - Java erlaubt *keine Mehrfachvererbung*, aber ...
  - ...eine Klasse kann beliebig viele *interfaces* implementieren.
  - *Schnittstellen simulieren Mehrfachvererbung.*
- **Beispiel**

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

```
class Rectangle
 extends Figure implements Printable {
   ...
  void print() { ... }
}
```

Anwendung von Schnittstellen

- Beispiel aus der Standardbibliothek

```
interface Comparable {
  int compareTo(Object other);
}
```

  Tatsächlich *generic* Schnittstelle `Comparable<T>`

- Klassen können beliebig viele Schnittstellen implementieren

```
class Rectangle extends Figure
      implements Comparable, Printable { ... }
```

- Schnittstellen können beliebig viele Schnittstellen erweitern

```
interface Fruit { ... }
interface Vegetable { ... }
interface Tomatoe extends Fruit, Vegetable { ... }
```

- Erweiterte Funktionalität ab Java 8 (z.B. Default Methods)

Zusammenfassung: OOP in Java

- Klassen — `class`
  - Attribute und Methoden — `public`, `private`, `protected`
  - Vererbung — `extends`
- Objekte
  - Konstruktoren — `new`
  - Referenzen
  - Selbstreferenz — `this`
  - Referenz auf Basisklasse (-objekt) — `super`
- Vererbung und Polymorphie
  - Abstrakte Klassen und Methoden — `abstract`
- Klassenattribute und Klassenmethoden — `static`
  - Java Programme — `main()`
- Schnittstellen — `implements`, `interface`
  - Konzept von *interfaces* ersetzt Mehrfachvererbung

- Als nächstes abschließend *noch einige Details zu Java*

Übersicht

Übersicht

- Wir kennen Java schon ganz gut, insbesondere OOP in Java
- Abschließend ein Überblick über Verschiedenes, was
  - nicht (direkt) mit OOP zu tun hat
  - in der Praxis hilfreich ist

- Attribute, Methoden und Klassen unveränderlich machen
- Sonderrolle der primitive Datentypen und *wrapper*
- Strukturierung von Programmen mit *packages*
- Java Klassenbibliotheken
- Eigenständige Programme in Java: `.jar` Archive
- Speicherverwaltung: *garbage collector*
- Konventionen für Quellcode
- Dokumentation von Quellcode

- *Es gibt natürlich noch mehr...*

`final` — Unveränderliches

- `final` legt fest, dass etwas *unveränderbar* ist
- Variablen/Attribute ⇒ keine Änderung nach Initialisierung
  - z.B. `public final int x;`
- Methoden ⇒ Überschreiben nicht möglich
  - z.B. `public final void method();`
- Klassen ⇒ Ableiten nicht möglich
  - z.B. `final class Square {...};`
- Spezialfall: Konstanten, z.B.
  - `public static final double PI=3.14159265358979;`
  - Konvention: Großschreibung für Bezeichner

- Warum `final`?
  - Hinweise, die den Code einfacher lesbar machen
  - Automatische Überprüfung durch den Compiler
  - ⇒ Vermeiden von Fehlerquellen

Anwendung: *immutable objects*

- Ein Objekt heißt unveränderbar (*immutable*), wenn sein Zustand nicht mehr verändert werden kann, nachdem es konstruiert wurde:
- Alle Attribute sind `final`!
- Falls Attribute den Besitz von Objekten bedingen, sind diese Objekte ebenfalls unveränderbar!
- Beispiel: Klasse `String` im Unterschied zu `StringBuilder`
- Vorteile
  - Vermeide Fehler: keine „unerwünschten“ Zustandsänderungen z.B. durch mit *call-by-value* übergebene Referenzen
  - Unproblematisch in parallelen Algorithmen
  - Compiler darf u.U. Annahmen treffen (Code Optimierung)

wrapper-Klassen für primitive Datentypen

- Primitive Datentypen (z.B. `int`) sind keine Klassen
- Konsequenz: Ungleichbehandlung
  - Übergabe *per value*
  - Objekte werden dagegen *immer* als Referenz übergeben
  - Keine Methoden *(dafür: Operatoren)*
- Warum ist das so?
  - Effizienz! – Abbildung direkt auf Prozessorregister
  - Java geht hier einen Kompromiss ein
- Teil des Kompromisses: *wrapper*-Klassen `Integer`, `Double`, ...
  - „Umschließen“ (*wrap*) primitiven Datentyp mit Klasse
  - Kann wie jedes Java `Object` behandelt werden
  - Methoden zur Konvertierung, z.B. auch
    ```
    Integer.parseInt(String text)
    ```
  - Abgeleitet von `java.lang.Number`

Java *packages*

- *package* = Mittel zur Strukturierung von Quelltexten
  - Logische Strukturierung in zusammengehörige Klassen
  - Oft Trennung von Anwendungslogik und Programmierschnittstelle
  - Eigener Namensraum pro *package*
  - Zugriffsbeschränkungen z.B. `private class Foo {...}`
- Hierarchische Gliederung
  - z.B. `einfinf.demos`
  - Abbildung von Namensraum auf Verzeichnisstruktur, z.B. `./einfinf/demos/`
  - Wurzel(n) definiert durch Umgebungsvariable `CLASSPATH`, z.B. `bash> export CLASSPATH=/home/roessl/projects`
- *package* erzeugen: `package` am Anfang der `.java` Datei
  - z.B. `package einfinf.demos;`
- *package* verwenden: `import`
  - z.B. `import einfinf.demos;`
  - *wildcard* möglich z.B. `import einfinf.*;`

Java Standardbibliotheken

- Java beinhaltet bereits eine umfangreiche Klassenbibliothek
- Organisation in *packages*, z.B.
- `java.lang` – *Grundfunktionalität*
  - z.B. `Object`, `String`, `Number`, `Integer`, `Comparable`, ...
- `java.io` – *Ein-/Ausgabe*
  - z.B. `OutputStream`, `PrintStream`, ...
- `java.lang.system` – *Rund um Java-/Betriebssystem*
  - z.B. `System.in`, `System.out`, `System.err`
    (Instanzen von `java.io.PrintStream`*)*
- `java.math` – *beliebige Genauigkeit*
  - `BigInteger`, `BigDecimal`
- `java.util` – *verschiedene Datenstrukturen*
- `javax.swing` – *graphische Benutzeroberflächen*
- u.v.m. ...

Java Programme

- Java Programm =
  - Menge von Klassen
  - Genau eine Klasse mit `public static void main(...)`
- D.h. Java Programm =
  - Menge von `.class` Dateien
  - Ggf. Organisation in Unterverzeichnissen bzw *packages*
- Das Programm `jar` erstellt daraus ein `.jar` *Archiv*
  - Archiv = Sammlung von `.class` Dateien
  - Ähnlich `.tar` oder `.zip` Archiven
  - Der `java` Interpreter kann `.jar` Archive auch direkt ausführen!
  - `.jar` Archiv als Programmdatei
- **Beispiel**

```
bash> ls
Hello.java Hello.class
bash> jar -cf hello.jar *.class   # Archiv erstellen
bash> jar -tf hello.jar           # Inhalt auflisten
Hello.class
bash> java -jar hello.jar         # Ausfuehren
Hello world!
```

Speicherverwaltung

- Was heißt *Speicherverwaltung*?
  - Anforderung von Speicher beim Erzeugen eines Objekts
  - Speicher muss auch wieder freigegeben werden, am besten sobald das Objekt nicht mehr benötigt wird
- Java kümmert sich um Speicherverwaltung
  - Im Normalfall transparent für Benutzer
  - Große Arbeitserleichterung z.B. im Gegensatz zu C/C++! Beseitigung einer gefährlichen Fehlerquelle!
- Java verwendet dazu einen *garbage collector* (GC)
  - GC erkennt, ob ein Objekt gebraucht (=referenziert) wird.
  - GC löscht unnötige Objekte von Zeit zu Zeit.
  - Explizites Löschen *jetzt(!)* mit `System.gc();`
- Aufruf von `Object.finalize()` *unmittelbar* vor Löschen
  - Benachrichtigung z.B., um Ressourcen freizugeben
  - Dazu Methode `finalize()` überschreiben
  - Vorsicht: `finalize()` ist *kein* Destruktor (C++)

Konventionen

- Eine Programmiersprache definiert *Syntax* und *Semantik*
- Code kann syntaktisch korrekt aber schwer zu lesen sein
  - Korrekte Syntax = Lesbarkeit *für Compiler*
  - Daraus folgt nicht notwendig lesbar *für Menschen*!
  - Analogie: *Schönschreiben*
- Warum ist Lesbarkeit von Code wichtig?
  - In der Regel Arbeit im Team $\Rightarrow$ einheitliche Standards
  - Wechselnde Autoren (aus verschiedenen Ländern/Kulturen?!)
  - Programme müssen i.d.R. gepflegt/gewartet/erweitert werden
  - Und das auch noch nach Jahren!
  - $\approx$ 80% Aufwand (=Kosten!) für Pflege und Wartung von existierendem Code
  - *Lesbarkeit als ein Hauptkriterium für Erfolg!*
- Lesbarkeit von Code erfordert Einhalten von Konventionen
  - Keine festen Regeln. Oft firmeninterne Regeln.
  - Allgemein anerkannte *good practice* für Programmiersprachen
  - z.B. Google Java Style

Dokumentation

- Kommentare helfen, Code zu verstehen
- Gute Kommentare brauchen Erfahrung und Übung
  - Nicht zu viel — nicht Code nacherzählen
  - Nicht zu wenig — erkläre Idee genau
- DRY Prinzip — Extrahiere Dokumentation aus Kommentaren
  - z.B. online Handbuch in HTML oder PDF aus LaTeX
- Für Java übernimmt das `javadoc`
  - Interpretiert bestimmte *tags* in Kommentaren
  - Erstellt daraus online Dokumentation in HTML
  - Offizielle Homepage mit Dokumentation zu `javadoc`
- `Javadoc` benötigt
  - Kommentare für möglichst *alle* Klassen/Methoden
  - Bestimmtes Format der Kommentare `/**...*/`
  - *tags* wie z.B. `@param`, `@return`, aber auch `@todo`, `@see`
  - Optional (HTML-)*tags* zur Formatierung von Text
- Doxygen als Alternative
  - mehr Sprachen (u.a. C++), mehr Optionen

Beispiel: javadoc

```
/** Compute square root of a real number
    using Heron's algorithm.
    @author Christian Roessl
 */
public class Heron {
  /** relative error bound
   */
  private double relerr;
  ...
  /** Compute square root of x up to an approximation
      error of x*relerr.
      @param x real number >=0 (exit program if x<0)
      @return square root of x
   */
  public double sqrt(double x) { ... }
}
```

- Beispiele für javadoc Ausgabe: Standardbibliothek
  z.B. Integer

Zusammenfassung: Mehr zu Java

- `final` verbietet Veränderungen von
  - Attributen (nur Initialisierung) $\Rightarrow$ Konstanten
  - Methoden (kein Überschreiben)
  - Klassen (kein Ableiten)
- *wrapper* Klassen umschließen primitive Datentypen
  - z.B. `Integer` für `int` ($\Rightarrow$ Objekte)
- *packages* strukturieren Code
  - Zuordnen mit `package`
  - Verwendung mit `import`
- Java beinhaltet eine umfangreiche *Klassenbibliothek*
  - Siehe Dokumentation! z.B. online
- Eigenständige Java *Programme* als `.jar` Archive
- Speicherverwaltung durch *garbage collector*
  - `finalize()` Methode
- Beachte *code conventions*
- Dokumentation mit `javadoc`

Zusammenfassung: Objektorientierte Programmierung

- Objektorientierung als Programmierparadigma
  - Modelliere Interaktion zwischen Objekten
  - Fasse gleichartige Objekte in Klassen zusammen
  - Klassenhierarchie durch Vererbung
  - Polymorphie durch Überschreiben von Methoden
- Viele Programmiersprachen unterstützen OOP
  - Aber auch ohne explizite Unterstützung möglich! (z.B. in C)
  - Oft nicht „reine Lehre“ sondern Mischform
- Objektorientierung in Java
  - Java unterstützt (und erzwingt) OOP
  - z.B. Funktionen nur als Klassenmethoden
  - Besonderheiten: Referenzen, Schnittstellen
- Zusätzliche Mittel
  - *packages* zur Strukturierung
  - Klassenbibliothek
  - Werkzeuge wie `jar` oder `javadoc`