Objektorientierte Programmierung

Bis jetzt kennen wir (fast) nur primitive Datentypen.

Diese entsprechen weitestgehend der Hardware des Rechners (z.B. besitzt ein Rechner Hardware um zwei `floats` zu addieren).

Wir möchten Dinge der realen Welt modellieren, dafür benötigen wir komplexere Datentypen.

Lösung: selbstdefinierte Datentypen

Objektorientierte Programmierung

Angenommen wir möchten eine Adressverwaltung schreiben. Dazu müßten wir zunächst eine Adresse modellieren:

Harald Räcke
Boltzmannstraße 3
85748 Garching

Adresse	
+ Name	: String
+ Strasse	: String
+ Hausnummer	: int
+ Postleitzahl	: int
+ Stadt	: String

Zumindest für diesen Fall ist die Modellierung sehr einfach.

Datentyp ist hier nur eine Komposition (Zusammensetzung) von anderen einfacheren Grundtypen.

Wir visualisieren den Datentyp hier über ein UML-Diagramm. Dies ist eine grafische Modellierungssprache um Software zu spezifizieren. UML ist nicht speziell für Java entwickelt worden; deshalb unterscheidet sich die Syntax leicht.

Objektorientierte Programmierung

Wir werden diesem Grundprinzip, dass man Objekte **nur** über Methoden ändern sollte, nicht immer folgen...

Wie benutzt man den Datentyp?

Geht aus der Ansammlung der Grundtypen nicht hervor. Wenn der Datentyp sehr komplex ist (Atomreaktor), kann man leicht Fehler machen, und einen ungültigen Zustand erzeugen.

Grundidee:
Ändere Variablen des Datentyps nur über Funktionen/Methoden.

Falls diese korrekt implementiert sind, kann man keinen ungültigen Zustand erzeugen.

Daten und Methoden
gehören zusammen
(abstrakter Datentyp)

Objektorientierte Programmierung

Ein (abstrakter) Datentyp besteht aus Daten und einer Menge von Methoden (Schnittstelle) um diese Daten zu manipulieren.

Datenkapselung / Information Hiding

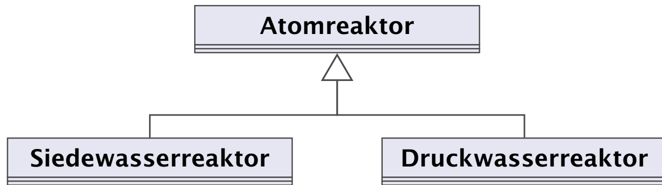
Die Implementierung des Datentyps wird vor dem Benutzer versteckt.

- minimiert Fehler durch unsachgemäßen Zugriff
- Entkopplung von Teilproblemen
  - gut für Implementierung, aber auch
  - Fehlersuche und Wartung
- erlaubt es die Implementierung später anzupassen (↑rapid prototyping)
- erzwingt in der Designphase über das was und nicht über das wie nachzudenken....

Objektorientierte Programmierung

Generalisierung + Vererbung

Identifiziere Ähnlichkeiten zwischen Datentypen und lagere gemeinsame Teile in einen anderen Datentyp aus.



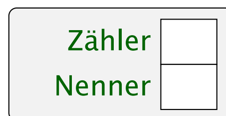
- vermeidet Copy&Paste...
- verringert den Wartungsaufwand...

Objektorientierte Programmierung

Klasse	=	Implementierung eines abstrakten Datentyps
Objekt	=	Instanz/Variable einer Klasse

Beispiel: Rationale Zahlen

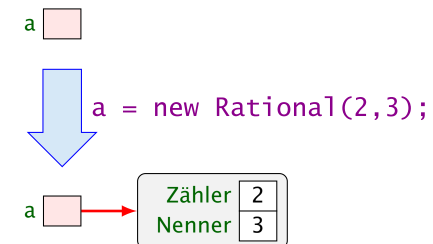
- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $\frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q.
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` erhalten:



- Die Daten eines Objektes heißen Instanz-Variablen oder Attribute.

Beispiel: Rationale Zahlen

- `Rational name;` deklariert eine Variable für Objekte der Klasse `Rational`.
- Das Kommando `new Rational(...)` legt das Objekt an, ruft einen Konstruktor für dieses Objekt auf, und liefert einen Verweis auf das neue Objekt zurück.

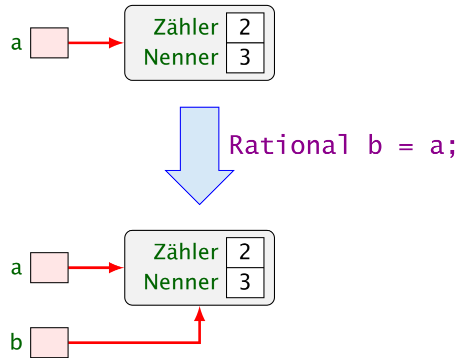


- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objektes initialisieren kann.

Erinnerung: Referenzen

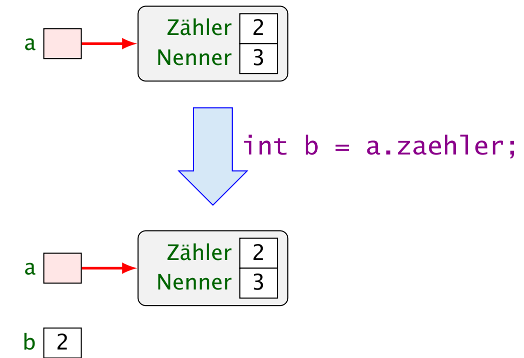
Der Wert der Rational-Variablen ist eine Referenz/Verweis auf einen Speicherbereich.

Rational b = a; kopiert den Verweis aus a in die Variable b:



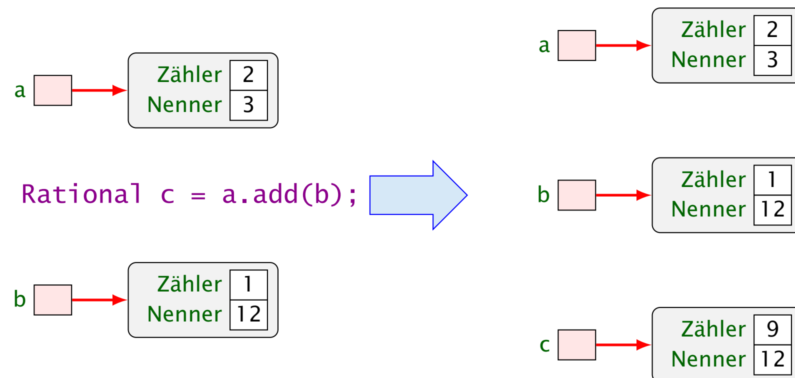
Beispiel: Rationale Zahlen

a.zaehler liefert den Wert des Attributs zaehler des Objektes auf das a verweist:

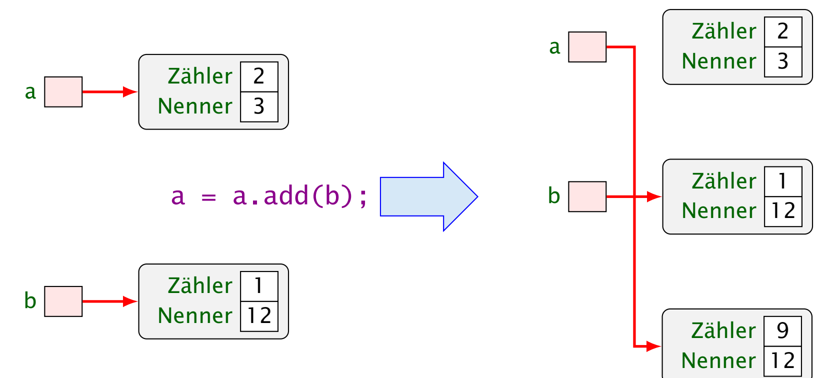


Beispiel: Rationale Zahlen

a.add(b) ruft die Operation add für a mit dem zusätzlichen aktuellen Parameter b auf:



Beispiel: Rationale Zahlen



Die Operationen auf Objekten einer Klasse heißen auch Methoden, genauer: Objekt-Methoden.

Zusammenfassung

Eine Klassendeklaration besteht folglich aus:

- Attributen für die verschiedenen Wertkombinationen der Objekte;
- Konstruktoren zur Initialisierung der Objekte;
- Methoden, d.h. Operationen auf Objekten.

Diese Elemente heißen auch Members der Klasse.

Implementierung

```
1  public class Rational {
2      // Attribute:
3      private int zaehler, nenner;
4      // Konstruktoren:
5      public Rational(int x, int y) {
6          zaehler = x;
7          nenner  = y;
8      }
9      public Rational(int x) {
10         zaehler = x;
11         nenner  = 1;
12     }
```


Implementierung

```
13     // Objekt-Methoden:
14     public Rational add(Rational r) {
15         int x = zaehler * r.nenner + r.zaehler * nenner;
16         int y = nenner * r.nenner;
17         return new Rational(x,y);
18     }
19     public boolean isEqual(Rational r) {
20         return zaehler * r.nenner == r.zaehler * nenner;
21     }
22     public String toString() {
23         if (nenner == 1) return "" + zaehler;
24         if (nenner  > 0) return zaehler +"/"+ nenner;
25         return (-zaehler) +"/"+ (-nenner);
26     }
27     public static Rational[] intToRationalArray(int[] a) {
28         Rational[] b = new Rational[a.length];
29         for(int i=0; i < a.length; ++i)
30             b[i] = new Rational(a[i]);
31         return b;
32     }
```

Implementierung

```
33     // Jetzt kommt das Hauptprogramm
34     public static void main(String[] args) {
35         Rational a = new Rational(1,2);
36         Rational b = new Rational(3,4);
37         Rational c = a.add(b);
38
39         System.out.println(c.toString());
40     } // end of main()
41 } // end of class Rational
```

Implementierung

Bemerkungen:

- Jede Klasse sollte in einer separaten Datei des entsprechenden Namens stehen.
- Die Schlüsselworte `public` bzw. `private` klassifizieren, für wen die enstprechenden Members sichtbar, d.h. zugänglich sind.
- `private` heißt: nur für Members der gleichen Klasse sichtbar.
- `public` heißt: innerhalb des gesamten Programms sichtbar.
- Nicht klassifizierte Members sind nur innerhalb des aktuellen ↑Package sichtbar.

Implementierung

Falls kein Konstruktor definiert wird, stellt Java einen Default-Konstruktor zur Verfügung, welcher keine Argumente entgegennimmt.

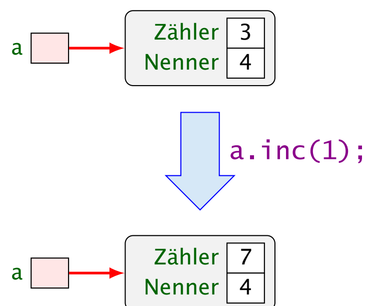
Bemerkungen:

- Konstruktoren haben den gleichen Namen wie die Klasse.
- Es kann mehrere geben, sofern sie sich im Typ ihrer Argumente unterscheiden.
- Konstruktoren haben keine Rückgabewerte und darum auch keinen Rückgabetyp.
- Methoden haben dagegen stets einen Rückgabe-Typ, evt. `void`.

```
1 public void inc(int b) {
2     zaehler = zaehler + b * nenner;
3 }
```


Implementierung

Die Objekt-Methode `inc()` modifiziert das Objekt, für das sie aufgerufen wird.



Eine Klasse, deren Objekte nach der Initialisierung nicht verändert werden können, ist immutable. Mit dem Hinzufügen der Operation `inc` wird die Klasse `Rational` mutable. Es ist eine sehr wichtige Designentscheidung ob man eine Klasse als mutable oder immutable implementiert.

Implementierung

- Die Objektmethode `isEqual()` ist nötig, da der Operator `==` bei Objekten die Identität der Objekte testet, d.h. die Gleichheit der Referenz!!!
- Die Objektmethode `toString()` liefert eine `String`-Darstellung des Objekts.
- Sie wird implizit aufgerufen, wenn das Objekt als Argument für die Konkatenation `+` auftaucht.
- Innerhalb einer Objektmethode/eines Konstruktors kann auf die Attribute des Objektes direkt zugegriffen werden.
- `private`-Klassifizierung bezieht sich auf die Klasse nicht das Objekt: die Attribute aller `Rational`-Objekte sind für `add` sichtbar!!!

`isEqual` ist auch nötig, da Brüche mit unterschiedlichen Werten für Zähler und Nenner trotzdem gleich sind. Normalerweise sollte man für den Gleichheitstest eine Methode `equals` definieren, da diese Methode von verschiedenen Java-Klassen vorausgesetzt wird. Für eine vernünftige Implementierung dieser Methode benötigen wir aber weitere Konzepte...

UML-Diagramm

Eine graphische Visualiserung der Klasse `Rational`, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:

Rational
– zaehler : int – nenner : int
+ add (y : Rational) : Rational + isEqual (y : Rational) : boolean + toString () : String

Diskussion und Ausblick

- Solche Diagramme werden von der UML, d.h. der Unified Modelling Language, bereitgestellt, um Software-Systeme zu entwerfen (↑Software Engineering)
- Für einzelne Klassen lohnt sich ein solches Diagramm nicht wirklich.
- Besteht ein System aber aus sehr vielen Klassen, kann man damit die Beziehungen zwischen den Klassen verdeutlichen.

Achtung:
UML wurde nicht speziell für Java entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht oder nur schlecht modellieren.

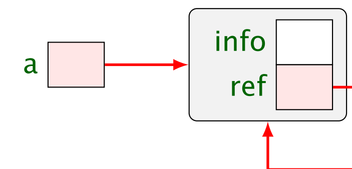
10.1 Selbstreferenzen

```
1  public class Cyclic {
2      private int info;
3      private Cyclic ref;
4      // Konstruktor
5      public Cyclic() {
6          info = 17;
7          ref = this;
8      }
9          ...
10 } // end of class Cyclic
```

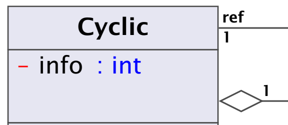
Innerhalb eines Members kann man mit Hilfe von `this` auf das aktuelle Objekt selbst zugreifen!

10.1 Selbstreferenzen

Für `Cyclic a = new Cyclic();` ergibt das



Modellierung einer Selbstreferenz



Die Rautenverbindung heißt auch Aggregation

Das Klassendiagramm vermerkt, dass jedes Objekt der Klasse `Cyclic` einen Verweis mit dem Namen ref auf ein weiteres Objekt der Klasse `Cyclic` enhält.

Ausserdem, dass jedes `Cyclic`-Objekt in genau einem anderen `Cyclic`-Objekt die Rolle ref übernimmt.

Die this-Referenz

Woher kommt die Referenz `this`?

- Einem Aufruf einer Objektmethode (z.B. `a.inc()`) oder eines Konstruktors wird implizit ein versteckter Parameter übergeben, der auf das Objekt (hier `a`) zeigt.
- Die Signatur von `inc(int x)` ist eigentlich:

```
void inc(Rational this, int x);
```

- Zugriffe auf Objektattribute innerhalb einer Objektmethode werden mithilfe dieser Referenz aufgelöst, d.h.:

```
zaehler = zaehler + b * nenner;
```

in der Methode `inc()` ist eigentlich

```
this.zaehler = this.zaehler + b * this.nenner;
```


10.2 Klassenattribute

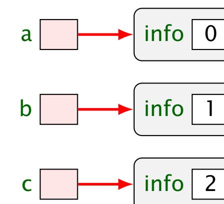
- Objektattribute werden für jedes Objekt neu angelegt,
- Klassenattribute einmal für die gesamte Klasse,
- Klassenattribute erhalten die Qualifizierung `static`

```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8         ...
9 } // end of class Count
```


10.2 Klassenattribute

count 3

```
Count b = new Count();
```



10.2 Klassenattribute

- Das Klassenattribut count zählt hier die Anzahl der bereits erzeugten Objekte.
- Das Objektattribut info enthält für jedes Objekt eine eindeutige Nummer.
- Außerhalb der Klasse Class kann man auf die öffentliche Klassenvariable name mit Hilfe von Class.name zugreifen.

- Funktionen und Prozeduren der Klasse ohne das implizite this-Argument heißen Klassenmethoden und werden auch durch das Schlüsselwort static kenntlich gemacht.

Man kann auf Klassenattribute und Klassenmethoden zugreifen ohne überhaupt je ein Objekt der Klasse zu instantiieren.

Beispiel

In Rational definieren wir:

```
public static Rational[] intToRationalArray(int[] a) {
    Rational[] b = new Rational[a.length];
    for(int i=0; i < a.length; ++i)
        b[i] = new Rational(a[i]);
    return b;
}
```

- Die Funktion erzeugt für ein Feld von int's ein entsprechendes Feld von Rational-Objekten.
- Außerhalb der Klasse Class kann die öffentliche Klassenmethode meth() mit Hilfe von Class.meth(...) aufgerufen werden.