Abstrakte Datentypen

Daten und Operationen
Mögliche Implementierungen
Veränderbare Typen,
Ergebnistypen

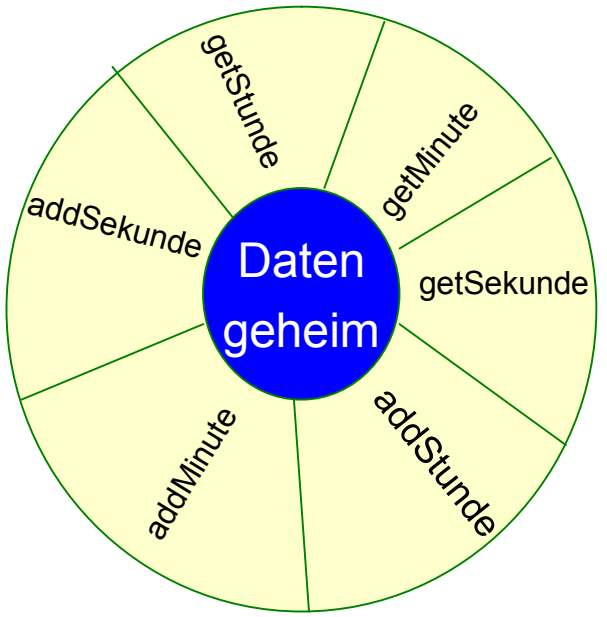
Einheit von Daten und Operationen

- Daten und Operationen auf Daten gehören zusammen
  - Konto:
    - Daten:
      - name, nummer, kontoStand
    - Operationen
      - überweisen, abheben, …
  - Datum
    - Daten:
      - tag, monat, jahr
    - Operationen
      - istSchaltjahr, addiereTage, …
  - Student
    - Daten:
      - name, geburtsDatum, semester
    - Operationen
      - scheinAusstellen, einschreiben, …

- Programmierer sollte dies beherzigen
  - Modellierung
- Programmiersprache muss das unterstützen
  - Klassen, Module, Units

Datenkapselung

- Daten sollen nur kontrolliert verändert werden
  - Kontostand nur durch überweisen, abheben
  - Datum nur durch addDays, addMonth…
  - Erzeugung
    - **Konstruktoren**
  - Ablesung
    - **Selektoren** (***getter***)
  - Veränderung
    - **Mutatoren** (***setter***)
- Methoden bilden Kapsel um die Daten
  - Repräsentation im Inneren versteckt
    - Class Konto{ private int kontoStand; … }
    - Class Datum{ private int msecSeit1970; … }
    - Class Student{
      private Date geburtsDatum;
      private String name, vorname;
  - Repräsentation kann man nachträglich ändern, ohne dass der Anwender etwas merkt



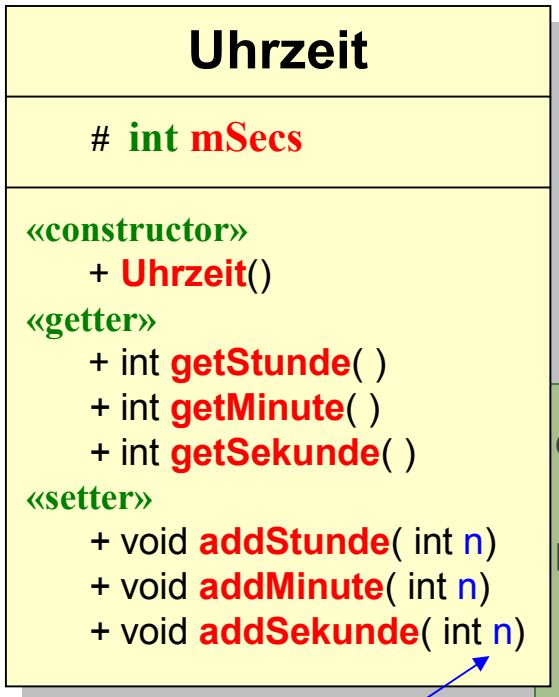
Class UhrZeit

Ob Uhrzeit intern als
mSec seit Mitternacht
oder als Tripel
(h,m,s)
gespeichert wird ist egal

Benutzersicht - Programmierersicht

- Klassendiagramme
  - Signaturen aller Methoden
  - + public
  - - private
  - # protected
- Wichtig für
  - Entwurf
  - Implementierung
  - Dokumentation
- Zusätzliche Spezifikation notwendig
  - $0 \leq$ getMin() ≤ 59
  - { getMin() = A}
    addMinute();
    { getMin() = A+1 }

Klassendiagramm
UML-Notation



Die Namen der formalen Parameter sind irrelevant, daher werden wir sie später oft weglassen

Implementierung
z.B. in Java

```
class Uhrzeit{
 protected int mSecs;
public Uhrzeit(…){
                }
  public int getStunde(){
   …
   }
 …
}
```

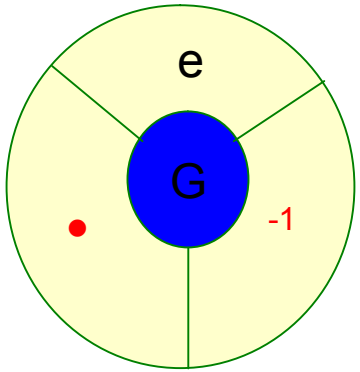
Daten und Operationen in der Mathematik

- Aus der Mathematik kennen wir
  - Ein Gruppe ist eine Menge G mit Operationen
    - $\bullet : G \times G \to G$
    - $^{-1} : G \to G$
    - $e : \to G$

    die gewisse Gleichungen erfüllen ..
  - Eine Boolesche Algebra ist eine Menge B mit Operationen
    - $\wedge, \vee : B \times B \to B$
    - $\neg : B \to B$
    - $0,1 : \to B$

    die gewisse Gleichungen erfüllen
  - Ein Körper ist eine Menge K mit Operationen
    - $+,-,*,^{-1} : K \times K \to K$
    - $0,1 : \to K$

    Wobei gewisse Gleichungen erfüllt sein müssen.
    Die Operation $^{-1}$ ist partiell – d.h. nicht überall definiert.
  - Ein Vektorraum über einem Körper K ist eine Menge V mit Operationen
    - $+, - : V \times V \to V$
    - $0 : \to V$
    - $\cdot : K \times V \to V$



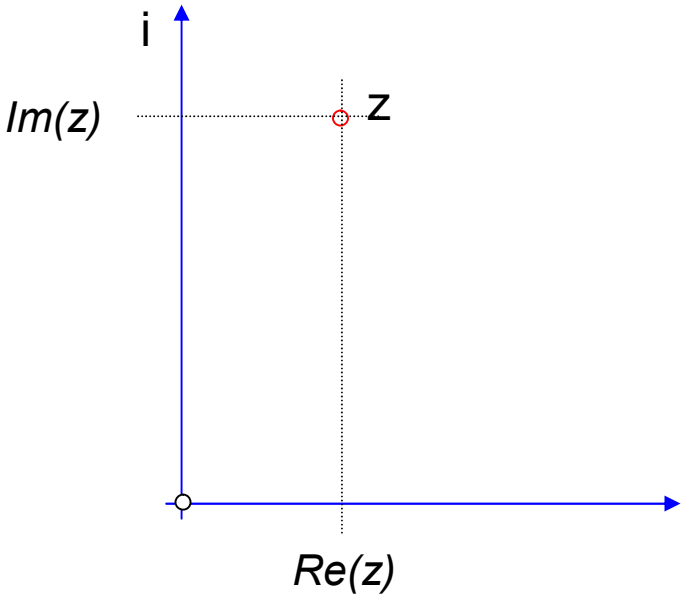
Gruppe

«constructor»
+ **Gruppe**()
«operators»
+ Gruppe **mal**(Gruppe)
+ Gruppe **invers**( )
+ Gruppe **neutral**( )

- Solche mathematischen Gebilde, bestehend aus einer oder mehreren Mengen und Operationen und evtl. noch Gleichungen, nennt man Algebren.

Die komplexen Zahlen

- Grundmenge $C = R \times R = \{ (x,y) \mid x,y \in R \}$
  - Statt (x,y) schreibt man auch : x+iy



- Operationen
  - + : $C \times C \to C$
  - * : $C \times C \to C$
  - $Re, Im : C \to R$

- Definition der Operationen:
  - $(x_1,y_1) + (x_2,y_2) = (x_1+x_2, y_1+y_2)$
  - $(x_1,y_1) * (x_2,y_2) = (x_1*x_2 - y_1*y_2 \ , \ x_1*y_2 + y_1*x_2)$
  - Eine reelle Zahl r kann man mit der komplexen Zahlen (r,0) identifizieren $r \equiv (r,0)$. Dabei werden die Operationen erhalten:
    - $r_1 + r_2 \equiv (r_1,0) + (r_2,0) = (r_1 + r_2,0) \equiv r_1 + r_2$
    - $r_1 * r_2 \equiv (r_1,0) * (r_2,0) = (r_1 * r_2,0) \equiv r_1 * r_2$
  - Für die spezielle Zahl i = (0,1) gilt:
    - i * i = (0,1) * (0,1) = (0*0-1*1,0*1+1*0) = (-1,0) ≡ -1
  - Für eine beliebige komplexe Zahl c=(x,y) hat man:
    - c = (x,y) = (x,o) + (0,y) = x*(1,0) + y*(0,1) = x + i*y

Spezifikation

- Grundmenge: Complex
  - $R \times R$
- Operationen:
  - + : $Complex \times Complex \rightarrow Complex$
  - * : $Complex \times Complex \rightarrow Complex$
  - Re : $Complex \rightarrow R$
  - Im : $Complex \rightarrow R$
- Gleichungen:
  - Definierende Gleichungen für add, mult, Re, Im
    - $(x_1,y_1) + (x_2,y_2) = (x_1+x_2,y_1+y_2)$
    - $(x_1,y_1) * (x_2,y_2) = (x_1*x_2 - y_1*y_2 \ , \ x_1*y_2 + y_1*x_2)$
    - $Re(x,y) = x$
    - $Im(x,y) = y$
  - Zusätzliche Gleichungen analog wie für R.
  - Gleichungen können bisher nur als Kommentare angegeben werden

Complex

«constructors»
+ Complex()
+ Complex(float, float)

«ops»
+ Complex sum(Complex,Complex)
+ Complex prod(Complex,Complex)

«getter»
+ float getRealteil(Complex)
+ float getImaginärteil(Complex)
+ String toString(Complex)

Implementierung in Java

- Datentypen modelliert durch Klassen
  - Grundmenge:
    - Kombination von Feldern
  - Operationen
    - Methoden
- Hier:
  - Grundmenge
    - float × float
  - Re
    - getRealteil
  - Im
    - getImaginärteil

```
public class Complex
{
  // Eine komplexe Zahl ist ein Paar von reellen Zahlen
  float x; // Realteil
  float y; // Imaginärteil

  // Konstruktoren
   public Complex(){}

   public Complex(float realteil, float imaginärteil){
        x = realteil;
        y = imaginärteil;
    }
   // Selektoren
   public float getRealteil(){
        return x;
   }
   public float getImaginärteil(){
        return y;
```

Operationen als statische Methoden

- In der Darstellung der Operation
  - $+ : Complex \times Complex \rightarrow Complex$
  - sind beide Argumente gleichberechtigt.
  - dies modellieren wir, wenn wir add als Klassenmethode (static) implementieren:

```
/** Komplexe Addition */
public static Complex sum(Complex z1, Complex z2){
     return new Complex(z1.getRealteil()+z2.getRealteil(),
                        z1.getImaginärteil()+z2.getImaginärteil());
}

/** Komplexe Multiplikation */
public static Complex prod(Complex z1, Complex z2){
     float x1 = z1.getRealteil();
     float y1 = z1.getImaginärteil();
     float x2 = z2.getRealteil();
     float y2 = z2.getImaginärteil();
     return new Complex(x1*x2-y1*y2,x1*y2+y1*x2);
}

//Darstellung
public String toString(){
  if (y==0) return ""+x;
  else return ""+x+" + i("+y+")";
}
```

Ein Testaufruf

- Statische Methoden
  - wie sum, prod
- sind Methoden der Klasse.

- Ein Aufruf verlangt als Adressat der Methode den Namen der Klasse:
  - Complex.sum(z1,z2)

- Bei dem Aufruf aus der definierenden Klasse, kann der Adressat entfallen.
  - prod(z1,z2)

```
public static void test(float x1, float y1, float x2, float y2){
    Complex z1 = new Complex(x1,y1);
    Complex z2 = new Complex(x2,y2);

    System.out.print("Summe ist ");
    System.out.println(Complex.sum(z1,z2));
    System.out.print("Produkt ist ");
    System.out.println(prod(z1,z2));
}
```

BlueJ: Terminal Window

Options

```
Summe ist 0.0 + i(2.0)
Produkt ist -1.0
```

BlueJ: Method Call

void test(float x1, float y1, float x2, float y2)

Complex.test (0 , float x1
1 , float y1
0 , float x2
1) float y2

Ok Cancel

Objektorientierte Modellierung

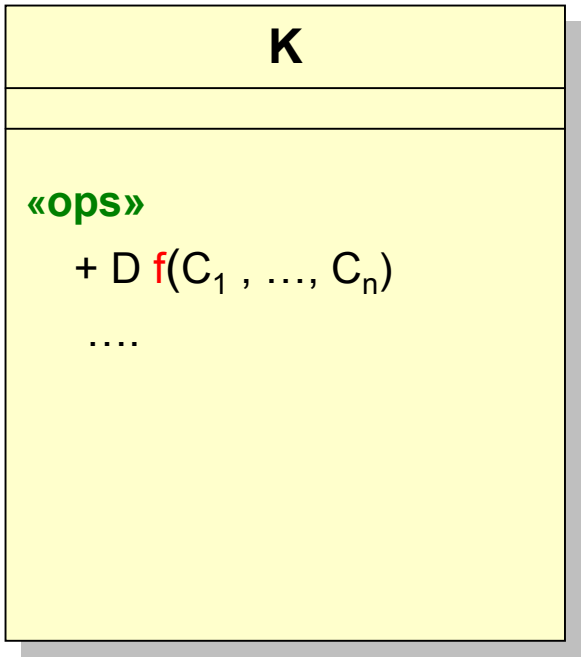
- Bei einer Objektmethode ist das Objekt, nicht die Klasse, der Adressat der Methode.
- Eine Operation vom Typ

  $f : \boldsymbol{K} \times C_1 \times C_2 \times \ldots \times C_n \rightarrow D$

  wird *in der Klasse* **K** durch eine Java-Methode der folgenden Signatur implementiert:

  D f (C$_1$ x$_1$, C$_2$ x$_2$, …, C$_n$)

- Das fehlende Argument der Operation ist implizit
  - this
- Operationen liefern neuen Wert
- Ein Aufruf ist ein Expression und liefert einen Wert vom Typ D.



Komplexe Zahlen - OO

- Die Operationen
  - + : $Complex \times Complex \rightarrow Complex$
  - * : $Complex \times Complex \rightarrow Complex$

  werden als Objekt-Methoden mit der Signatur
  - Complex plus(Complex z)
  - Complex mal(Complex z)

  implementiert.

- Das erste Argument der Operation ist Empfänger der Methode.
  - this

- Operationen liefern neue komplexe Zahl

- Ein Aufruf ist ein Expression vom Typ Complex:

Komplexe Zahlen

«construktors»
+ Complex()
+ Complex(float, float)
«Ops»
+ Complex plus(Complex)
+ Complex mal(Complex)
«getter»
....

```
/** Komplexe Addition - als Objektmethode*/
public Complex plus(Complex z2){
      return new Complex(x + z2.getRealteil(),
                         y + z2.getImaginärteil());
}
public void testeObjektmethode(){
      Complex i = new Complex(0,1);
      Complex drei = new Complex(3,0);
      System.out.println(drei.plus(i.mal(i)));
 }
```

Complex als veränderbarer Typ

- Die Operationen
  - + : $Complex \times Complex \rightarrow Complex$
  - * : $Complex \times Complex \rightarrow Complex$

  werden jetzt als Anweisungen implementiert
  - Berechne Ergebnis
  - Speichere es im Empfänger
- Operationen verändern das Objekt this
  - kein *new*

Complex

«Konstruktoren»
…
«Operatoren»
+ void add(Complex)
+ void mult(Complex)
«Selektoren»
…

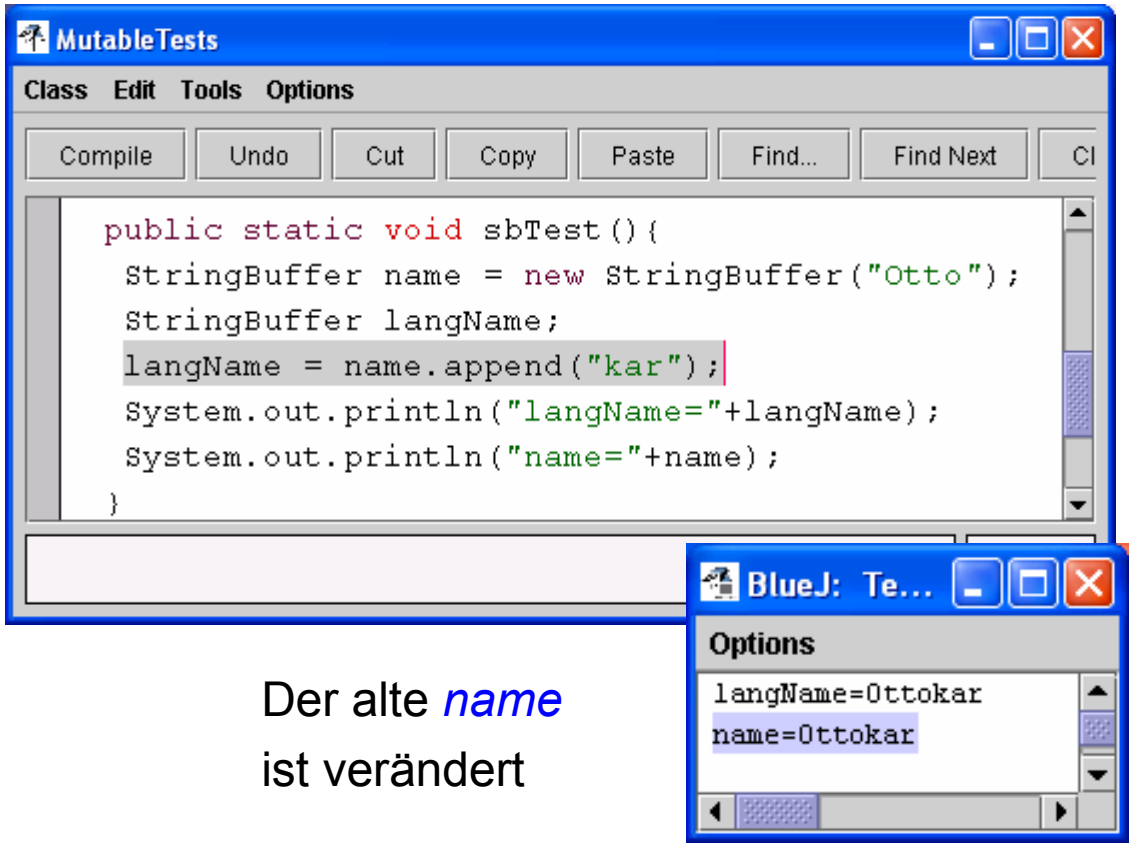
z *= i ;

z += z;

```
/** Komplexe Addition - als Anweisung*/
   public void add(Complex z2){
         x = x + z2.getRealteil();
         y = y + z2.getImaginärteil();
   }
/** Komplexe Multiplikation - Anweisung */
   public void mult(Complex z2){
         float x2 = z2.getRealteil();
         float y2 = z2.getImaginärteil();
         float xNeu=x*x2-y*y2;
            y = x*y2+y*x2;
            x = xNeu;
   }
/* ein Test */
public static Complex wasKommtRaus(){
  Complex z = new Complex(1,1);
  Complex i = new Complex(0,1);
  z.mult(i);
  z.add(z);
  return(z);
}
```

Veränderung von Objekten

- Ein StringBuffer ist eine veränderbare (mutable) Variante von String
- Umwandlungen mit Konstruktoren
  - StringBuffer(String s)
  - String(StringBuffer b)
- Methoden verändern this
  - StringBuffer **append**(StringBuffer sb);
  - StringBuffer **insert**(int offset, String str)



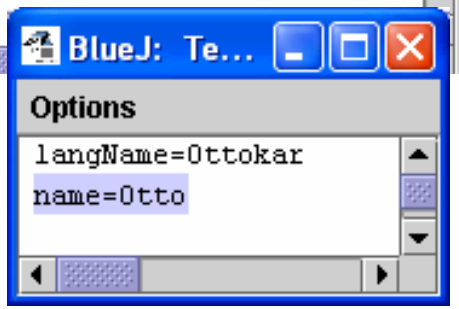
Der alte *name* ist verändert

Veränderbare Typen - Ergebnistypen

- Veränderbare Typen (engl.: mutable Types)
  - Methoden verändern ihr Objekt
    - Beispiel: Konto, Telefonbuch, Arrays
- Ergebnistypen
  - Können nicht verändert werden
  - Methoden liefern immer ein neues Objekt. Altes Objekt bleibt unverändert
    - Beispiel: String, int,

```
Konto meins = new Konto("HP",1000);
Konto deins =  new Konto(„Du",200);
deins.überweise(meins,500);
System.out.println(meins.getKontoStand());
```

```
public static void stringTest(){
 String name = "Otto";
 String langName;
 langName = name.concat("kar");
 System.out.println("langName="+langName);
 System.out.println("name="+name);
}
```



Der alte *name* unverändert

Veränderbare - und Ergebnistypen

- Veränderbare Typen erkennt man an der Signatur der Operationen:
- Aufgabe: Implementiere
  - $f : K \times C_1 \ldots C_n \rightarrow K$
- In einer Klasse K als *Mutator*
  - void f(C_1 x_1,…,C_n x_n)
  - K fehlt als Argument, Ergebnistyp void.
  - Aufrufe sind Anweisungen, sie verändern ihren Aufrufer
    - x.f(a_1,…,a_n)
    - x wird modifiziert
- In einem *Ergebnis- oder Werttyp* K (engl.: immutable) als:
  - K f(C_1 x_1,…,C_n x_n)
  - Aufrufe sind Expressions, sie liefern ein komplett neues Objekt
    - **K y = x.f(a_1,…,a_n)** ;
    - **x** bleibt unverändert