Abstrakte Datentypen (ADT)

($\rightarrow$ total, $\not\rightarrow$ partiell)

```
Types ARRAY[X]
Uses  Int, Bool
```

Functions

Konstruktoren (Aufbau des ADT) $\rightarrow$
- `create: Int x Int → Array[X]`
- `put: ARRAY[X] x Int x X ↛ ARRAY[X]`

Selektoren (Zerlegung) $\rightarrow$
- `first: ARRAY[X] → Int`
- `last: ARRAY[X] → Int`
- `get: ARRAY[X] x Int ↛ X`

Prädikat $\rightarrow$
- `empty: ARRAY[X] → Bool`

Axioms $\forall$ `i, j, k: Int; x: X; Q: ARRAY[X]`

Alle Axioms durch Kunjunktion (und) verknüpfen

```
first(create(i, j)) = i
first(put(a, i, x)) = first(a)
last(create(i, j))  = i
last(put(a, i, x))  = last(a)
i == k => get(put(a, i, x), k) = x
i <> k => get(put(a, i, x), k) = get(a, k)
empty(create(i, j))     = true
not empty(put(a, i, x)) = true
```

Preconditions

```
put(a, i, x):  first(a) <= i <= last(a)  \   für
                                          >  partielle
get(a, i):     first(a) <= i <= last(a)  /   Functions
```

zwangsläufiges Überschreiben

Frage: `get(a, i)` ohne vorheriges `put(a, i, x)`?
$\rightsquigarrow$ undef. x

Lösung: Prädikat `defined: ARRAY[X] x Int → Bool`
Preconditions:
`get(a, i): defined(a, i) = true`

Ein OO-Programm ist eine Sammlung von ADTn

Typen in typischen OO-Sprachen (z.B. Java)

1. Primitive Typen
   - `bool, char, int, float, ...`
   - im Quellcode durch Literale präsentiert
2. Klassen
   - Objekte sind nur durch Variablen repräsentiert (Ausnahme: `String` in Java)
3. Interface (nicht in alles Sprachen; nicht in C++)
   - wie Klassen Mengen von Instanzen
   - können aber keine eigenen Instanzen bilden

Java kennt drei Arten von Typkonstrukturen

`[]` (Array), `class, interface`
`[]` und `class` erzeuge Klassen
`interface` erzeugt Interfaces

`final` ...Konstante
`static`...Klassenbezogen

Klassen

- sind Schablonen für das Kreiren von Objekten eines Objekttypes Abstraktion
  Objekte sind *direkte Exemplare (Instanzen)* genau einer Klasse
- sind *statische* Beschreibungen der *Struktur* ihrer Objekte (zunächst)
- sind *neue Programmeinheiten* mit eigenen Namen (Block)
- stellen (öffentliche) Methoden als Schnittstelle für Objete bereit
- stellen die Implementation der Methoden bereit

UML: Unified Modeling Language (Standartisierung OMG)

Klasse	`Kreis`	Klassenname
	`int x, y, r` `Color color`	Attribute

	`getX() getR()`
	`getY()`
	`move(int dx, int dy)`
	`paint(...)`
		Methoden

	`Kreis(xx, yy, rr)`	Konstruktor-Methode

`Kreis: EinKreis`
`x = 200`
`y = 200`
`r = 100`
`color = rot`

Objekt oder Instanz der `Kreis`-Klasse

`: EinKreis`
`x =`
`y =`

irgendein Objekt

`Auto: MeinAuto`		
`tankinhalt = 50`	← Zustand	*Attribute*
`motor = V6`	← Teil von Auto	
`farbe = rot`	← Eigenschaft	
`+ fahren()`	↖	*Objekt-*
`+ tanken()`	← `public`	*kommunikation*
`+ bremsen()`	↙	
`- absControl()`	← `private`	

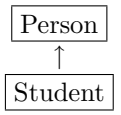
Botschaft:

Objektname | $\underbrace{\text{Methodenname}}_{\text{Selektor}}$ | $\underbrace{\text{(Parameter Objekte, Rückgabeobjekte)}}_{\text{Parameter}}$

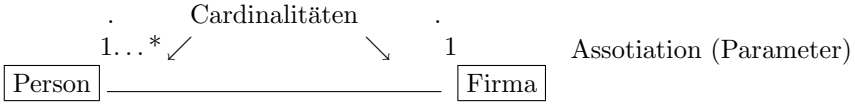
Nachricht $\neq$ Prozeduraufruf

- Adresse des auszuführenden Codes ist an der Aufrufstelle nicht bekannt
- Auch in verteilten Umgebungen einsetzbar
- Bestimmung des auszuführenden Codes erst zur Laufzeit und abhängig vom Empfänger (→ *dynamic binding*)

Beziehungen zwischen Klassen (UML Notation)



Vererbung



Assotiation (Parameter)

Klasse Person; Klassen-Notation

C++

```
class person
{
  public:  person (string nn, int gj);  // Typkonstruktor
           int zeige (void);            // Methode

  private: string name;                 // Private
           int    gebjahr;              // Variablen
}

person::person (string nn, int gj);     // :: <- scope operator
{ name = nn;  gebjahr = gj; }

person::zeige (void)
{ return gebjahr; }
```

Java

```
public class Person
{
  private String name;
  private int    gebjahr;

  Person (String nn, int gj)
    { name = nn; gebjahr = gj; }

  public int zeige ()
    { return gebjahr; }
}
```

Turbo Pascal

```
Type person = Object
    Constructor Create (nn : String; gj : Integer);
    Function    Zeige : Integer;

  Private
    name    : String;
    gebjahr : Integer;
End;

Constructor person.Create (nn : String; gj : Integer);
Begin
  name := nn; gebjahr := gj;
End;

Function person.Zeige : Integer;
Begin
  Zeige := gebjahr;
End;
```

Delphi

```
Type person = Class
  Public
    Constructor Create (nn : String; gj : Integer);
    Function    Zeige : Integer;

  Private
    name    : String;
    gebjahr : Integer;
End;

Constructor person.Create (nn : String; gj : Integer);
Begin
  name := nn; gebjahr := gj;
End;

Function person.Zeige : Integer;
Begin
  Result := gebjahr;  // oder  Zeige := gebjahr;
End;
```