2. Objektorientierte Sprachen

1. Überblick über objektorientierte Sprachen
2. Smalltalk - Sprache
3. Smalltalk - Klassenbibliothek

Klassifikation objektorientierter Sprachen(1)

nach den angebotenen Konzepten:

- *Objektbasierte* Sprachen
  unterstützen Objekte
- *Klassenbasierte* Sprachen
  objektbasiert + Klassen
- *Objektorientierte* Sprachen
  klassenbasiert + Vererbung

"Klassenbasiert" heißt hier, daß benutzerdefinierte Klassen genau gleich behandelt werden können wie vordefinierte Typen (Deklaration, Funktionsresultate, Initialisierung...)

(Einteilung nach Wegner, 88)

Klassifikation objektorientierter Sprachen(2)

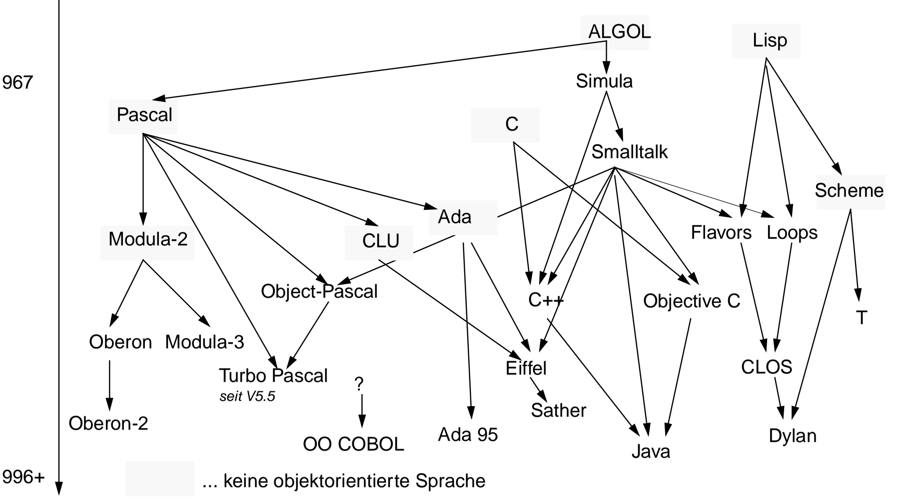
1. objektbasierte Sprachen: Ada, Actors, Modula-2
2. klassenbasierte Sprachen: CLU
3. objektorientierte Sprachen:
   - **C++, Objective-C, Java**: C-basiert
   - **Flavors, Loops, CLOS, Dylan**: Lisp-basiert
   - **Simula, Object-Pascal, Turbo Pascal**

hybride Sprachen

   - **Smalltalk**
   - **Eiffel, Trellis/Owl, Sather**
   - **Oberon, Oberon-2, Modula-3**

voll oo Sprachen

Geschichte der objektorientierten Sprachen



Terminologie

Sprache	Operation	Signatur	Instvar.
C++, Java	member function (virtual function)	head	member
Smalltalk	method	selector	instance variable
Eiffel	routine, function	feature	attribute, field

Terminologie 2

Sprache	Oberklasse	Unterklasse
C++, Java	base class	derived class
Smalltalk, Eiffel, ...	superclass	subclass

Sprache	Self	abstrakte Operation
Smalltalk	self	abstract method
Eiffel	Current	deferred feature
C++	this	
Simula		virtual procedure

Smalltalk

ursprünglich: **Smalltalk-80**

Sprachdefinition: Goldberg/Robson 1984

Sprache | Klassenbibliothek | Programmierumgebung

Visualworks (ParcPlace)
IBM Smalltalk (VisualAge)
Squeak
ObjectStudio
Smalltalk/MT
Dolphin Smalltalk
Envy/Developer
...

Little Smalltalk

GNU Smalltalk
Smalltalk Express

Warum Smalltalk?

Eigenschaften von Smalltalk:

- durchgehendes OO Konzept ⇒ “automatisch” objektorientierte Betrachtungsweise
- Garbage Collection, Objektreferenzen statt Pointer
- klein und elegant
- (weitgehend) standardisierte Klassenbibliothek
- hohe Produktivität (Faktor 3-5 zu C++, ~2 zu Java)

Smalltalk besitzt nicht:

- Strong Typing, Multiple Inheritance, Parametrisierte Klassen, benutzerwählbare Stufen der Kapselung

Implementierungen

- traditionelles Implementierungsmodell: virtuelle Maschine exekutiert “Bytecode”
- 90er Jahre: Dynamische Übersetzung in direkt exekutierten Code (z.B. Visualworks) bzw. C als Zwischensprache (Squeak)
- binärkompatibel innerhalb eines Anbieters (VisualAge, Visualworks)
- inzwischen ANSI Standard für Sprache

Überblick

Grundkonstrukte

  Literale, Variable, Messages

Abfragen, Schleifen

Klassenvariable, Metaklassen

Klassenbibliothek

Beispiel

Literale

Literale sind Objekte, die direkt niedergeschrieben werden können.

- Zahlen: 125, 23.5
- Zeichen: $a, $M, $-, $$, $1
- Strings: ‘hi’, ‘the Smalltalk environment’
- Symbole: #John, #x235
- Arrays: #(1 2 3)

  #(‘Array’ ‘of’ 5 ‘strings ‘and’ 2 ‘integers’)
  #((‘eins’ 1) (‘nicht’ negativ’) 0 -1 $4)

  Achtung: Das sind keine Konstanten!

Variable

Variablennamen bestehen aus Folgen von Buchstaben und Zahlen, beginnend mit Buchstaben.

index, initialIndex, textEditor, bin14

Rectangle, HouseholdFinances, (private/shared)

- Kleingeschriebene Variable: lokal (Methodenparameter, lokale Variablen, Instance-Variablen)
- Großgeschrieben: Globale Variablen, Klassenvariablen
- Pseudovariable: nil, true, false

Variablen stellen grundsätzlich Referenzen auf Objekte dar.

Messages

Message = Empfänger + **Selektor** + *Parameter*

3 **+** *4*

index **+** *1* index **>** *limit*

theta **sin** flaeche **sqrt**

list **addFirst:** *newComponent*

list **removeLast**

rectangle **center**

rectangle **containsPoint:** *cursorLocation*

HouseholdFinances **spend:** *32.50* **on:** *'food'*

Arten von Messages

- Unäre Messages

  Nur ein Objekt ist betroffen (der Empfänger):
  theta sin
  HouseholdFinances cashOnHand

- Binäre Messages

  Ein Argument, Operator aus max. 2 Sonderzeichen:
  3 + 4, summe **-** 1, index <= n, 3/7, 4 @ 2

- Keyword-Messages

  Jeweils ein Keyword (mit abschließendem Doppelpunkt)
  vor einem Argument:
  array at: 3 put: 'three'

Ausdrücke/Statements

- Jede Message ist ein Ausdruck.
- Einzige andere Art von Ausdruck: Zuweisungen

  sum := 3 + 4 (früher: sum ← 3 + 4)
  sum := 3 + (tangens := 27 tan)
  index := index + 1

- Punkt als Statement-Separator:
  list addFirst**:** 'x'.
  list addFirst**:** 'y'
- *Kaskaden*:
  list addFirst**:** 'x';
  addFirst**:** 'y'

gleichwertig!

Resultate/Geschachtelte Messages

- *Jeder* Ausdruck liefert ein Resultat zurück
  ⇒ geschachtelte Ausdrücke/Message sends
- Unäre messages: Von links nach rechts
  1.5 tan rounded
- Binäre messages: Von links nach rechts
  index + offset * 2
  Achtung!! 2 + 3 * 4 = 20 (!)

  Aber: offset * 2 + index funktioniert wie erwartet
- Klammern möglich:
  index + (offset * 2) liefert normales Resultat

Geschachtelte Messages 2

- Unäre Messages haben höchste Priorität
  frame width + border width * 2
- Änderung mit Klammern
  theta := 0.
  2 * theta cos.
  (2 * theta) cos
- Keyword messages haben niedrigste Priorität
  bigRect width: smallRect width * 2
  #(1 12 24 36) includes: 4 factorial

  rectangle scale: factor max: 5 -- 1 Message!
  rectangle scale: (factor max: 5)

Kontrollstrukturen - Abfragen

```
(number \\ 2) = 0
   ifTrue: [ parity := 0 ]
   ifFalse: [ parity := 1 ].
```

Block

gleichwertig!

parity := (number \\ 2) = 0 ifTrue: [0] ifFalse: [1].

Außer ifTrue:ifFalse gibt es noch ifTrue: und ifFalse: :

```
index <= limit
   ifTrue: [ summe := summe + (array at: index) ]
```


Implementierung von Abfragen

i < 1 ifTrue: [i := i + 1]

Wie paßt diese Schreibweise in das Message-Schema?

⇒ Blocks sind auch Objekte

- Ein Block steht für eine Folge von unausgeführten Anweisungen (andere Sicht: namenlose Funktion).
- Ausführung durch Senden der Message **value**

```
index := index + 1

[index := index + 1 ] value

b := [ index := index + 1 ].
b value
```

gleichwertig

Variable

- Variable sind Referenzen
- Speicherfreigabe durch Garbage Collection (autom.)

| rec1 help | “lokale Variablendefinition”
rec := 2 @ 5 corner 10 @ 8. “Definiert Rechteck”

rec, help → 2 @ 5, 10 @ 8

help := rec.

rec, help → 2 @ 5, 10 @ 8

rec := nil.

rec, help → 2 @ 5, 10 @ 8

help := nil.

rec, help, 2 @ 5, 10 @ 8

self und super

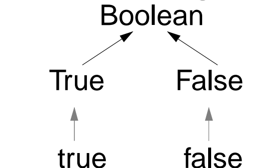
- "*Pseudovariable*": wie Variablen, aber keine Zuweisung möglich
- **self** bezieht sich immer auf das aktuelle Objekt (`this` in C++)
- **super** bezieht sich auf das aktuelle Objekt, aber die Suche beim Dynamic Binding beginnt bei der Oberklasse

Klasse Bahnticket, Methode **initialize**:

    timeOfIssue := Time now.

Klasse Seniorenticket, Methode **initialize**:

    tariff := 'Senior'.
    super initialize.



Methodendefinition

```
gcd: anInteger                                   ← Signatur
   "Euclid's algorithm"
 | m n t |                                        formaler Parameter
 m := self abs max: anInteger abs.                lokale Variablen
 n := self abs min: anInteger abs.
 [ n = 0 ]
   whileFalse:
       [ t:= n.
       n := m \\ n.
       m := t ].
 ^m
                                                  Methodenresultat
factorial
   "compute the factorial of the receiver"
 self > 1 ifTrue: [ ^self * (self - 1) factorial ]
          ifFalse: [ 1 ]
```


Implementierung von Abfragen (2)

Boolean

True False

true false

- true: einzige Instanz der Klasse True
  false: einzige Instanz der Klasse False
- Implementierung von ifTrue:ifFalse: für die Klasse True:

  ifTrue: block1 **ifFalse:** block2
  ^block1 value

  für die Klasse False:

  ifTrue: block1 **ifFalse:** block2
  ^block2 value

Kontrollstrukturen - Schleifen

```
index := 1.
[ index <= array size ]
   whileTrue: [ array at: index put: 0.
               index := index + 1 ]
```

- Ein Block kann mehrere Statements enthalten, der Wert des Blocks ist der Wert des letzten Statements

  Bsp: [a := 5. b := 4] liefert bei Ausführung 4
- Andere Schleifenkonstrukte
  ⇒ siehe Collections

Blocks mit Argumenten

```
sum := 0.
1 to: n do: [ :i | sum := sum + i ]
```

Auswertung von Blocks mit einem Argument:

```
block value: wert
```

Block mit mehreren Argumenten:

```
z := [ :x :y | (x * x + (y * y)) sqrt ].
z value: 2 value: 3 ⇒ 3.60
```


Klassendefinition

Object **subclass:** #Rectangle
instanceVariableNames: 'origin corner'
classVariableNames: ''
poolDictionaries: ''

Rectangle **superclass** ⇒ Object

- Klassen sind Objekte
- Klassennamen sind globale Variable (darum Klassennamen großgeschrieben)
- Die Klasse Object ist die Wurzel der Klassenhierarchie

Klassenvariable

Object subclass: #Ship
instanceVariableNames: 'course speed position'
classVariableNames: 'Icon'

!Ship methods!
display
ScreenManager displayIcon: Icon at: position speed: ...

!Ship class methods!
initialize: anIcon
Icon := anIcon

Ship

Freighter Tanker

Ship initialize:

Klassenvariable

s := Ship new. ...
s display.

sp 22.4kn
c 238.4°
p 241°W 17°N

f := Freighter new. ...
f display.

sp 17.4kn
c 18.4°
p 221°W 14°N

Anderes Bild für Frachter:
Freighter initialize:

s := Ship new. ...
s display.

sp 22.4kn
c 238.4°
p 241°W 17°N

f := Freighter new. ...
f display.

sp 17.4kn
c 18.4°
p 221°W 14°N

Klassenvariable sind für alle Unterklassen gleich!

Class Instance Variables

- Instanzvariablen sind unterschiedlich für jede Instanz
- Klassen sind Instanzen von Metaklassen und können daher Klasseninstanzvariable haben (nicht identisch mit Klassenvariablen!)

Ship class instanceVariableNames: ‘Icon’.

Ship initialize: Freighter initialize:

s := Ship new. ...
s display.

sp 22.4kn
c 238.4˚
p 241˚W 17˚N

f := Freighter new. ...
f display.

sp 17.4kn
c 18.4˚
p 221˚W 14˚N

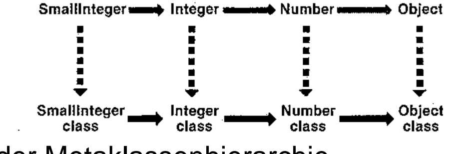
Metaklassen (1)

- Zu jeder Klasse gibt es exakt eine Metaklasse (daher kein Name für die Metaklasse nötig)
- Die Vererbungshierarchie der Klassen ist identisch strukturiert zu der der Metaklassen

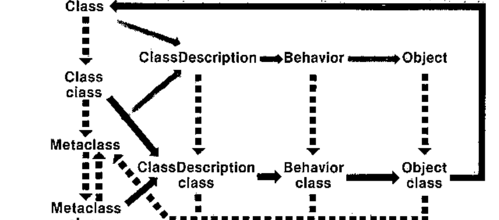
Ship class
Ship
Freighter class
Tanker class
Freighter
Tanker

Metaklassen (2)

Ausschnitt aus der normalen Klassenhierarchie + Metaklassen

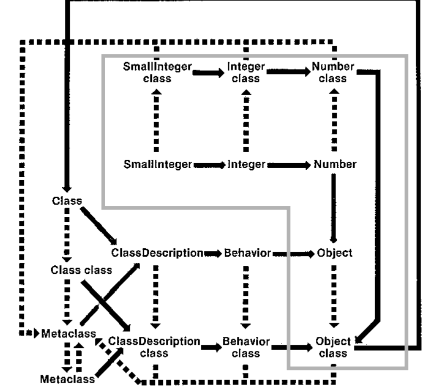


Spitze der Metaklassenhierarchie



Metaklassen (3)

Beides zusammen:



Programmvergleich

```
program frequency;
const size = 80;
var s: string[size];
    i, k: integer;
    c: char;
    f: array[1..26] of integer;
begin
writeln('enter line');
readln(s);

for i := 1 to 26 do f[i] := 0;
for i := 1 to size do begin
    c := lowerCase(s[i]);
    if isLetter(c) then begin
          k := ord(c) - ord('a') + 1;

          f[k] := f[k] + 1
          end; end; end;
```

```
| s c f k |


f := Array new: 26.

s := Prompter
    prompt: 'enter line'
    default: ''.
1 to: 26 do: [ :i | f at: i put: 0 ].
1 to: s size do: [ :i |
    c := (s at: i) asLowerCase.
    c isLetter ifTrue: [
          k := c asciiValue -
          $a asciiValue + 1.
          f at: k put: (f at: k) + 1 ] ]
^f
```


Programmvergleich (2)

```
| s f |
s := Prompter prompt: 'enter line' default: ''.
f := Bag new.
s do: [ :c | c isLetter ifTrue: [ f add: c asLowerCase ] ].
^f
```

⇒ Kenntnis der Klassenhierarchie bedeutet effektivere Programmierung

Klassenbibliothek (1)

- Klassenbibliothek von Smalltalk/V: 200 Klassen, 2000 Methoden. Visualworks: ~500 Klassen, 6000 Methoden
- Die meisten Methoden haben 2 bis 10 Implementierungen in unterschiedlichen Klassen
- Methoden sind üblicherweise ziemlich klein (10 Zeilen)
- Große Klassenbibliothek bedeutet großen Lernaufwand bis zur vollen Ausnutzung ⇒ integrierte Programmierumgebung hilft bei der Orientierung

Klassenbibliothek von Smalltalk/V DOS

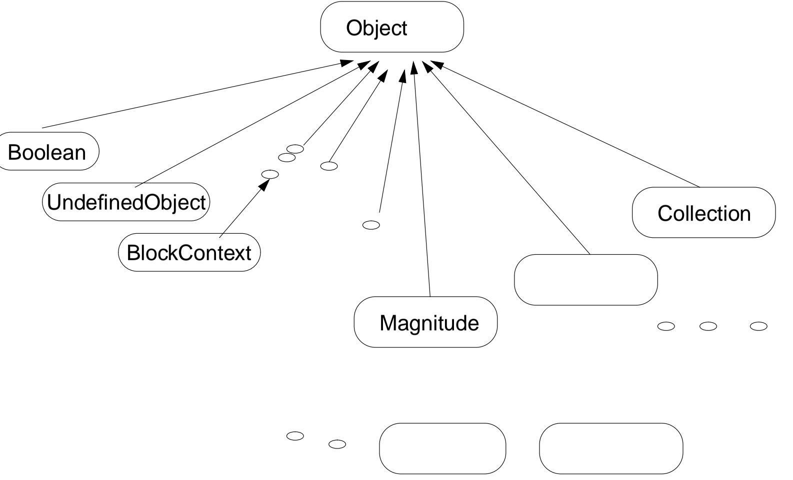
Entsprechungen in Visualworks (Smalltalk-80 Familie)

```
Object
  Behavior
    Class
    MetaClass
  BitBlt
    CharacterScanner
    Pen
      Animation
      Commander
  Boolean
    False
    True
  ClassBrowser
  ClassHierarchyBrowser
  ClassReader
  Collection
    Bag
    IndexedCollection
      FixedSizeCollection
        Array
        Bitmap
        ByteArray
          CompiledMethod
          FileControlBlock
        Interval
        String
          Symbol
      OrderedCollection
        Process
        SortedCollection
    Set
      Dictionary
        IdentityDictionary
          MethodDictionary
        SystemDictionary
      SymbolSet
  Compiler
  Context
  CursorManager
    NoMouseCursor
  DemoClass
  Directory
  DiskBrowser
  Dispatcher          <- Controller
    GraphDispatcher
    PointDispatcher
    ScreenDispatcher
    ScrollDispatcher
      FormEditor
      ListSelector
      TextEditor
        PromptEditor
      TopDispatcher
  DispatchManager
  DisplayObject
    DisplayMedium
      Form
        DisplayScreen
        SelectorForm
    DisplayString
    InfiniteForm
  File
  Inspector
    Debugger
    DictionaryInspector
  Magnitude
    Association
    Character
    Date
    Number
      Float
      Fraction
      Integer
        LargeNegativeInteger
        LargePositiveInteger
        SmallInteger
    Time
  Menu
  Message
  Pane                <- View
    SubPane
      GraphPane
      ListPane
      TextPane
    TopPane
  Pattern
    WildPattern
  Point
  Prompter
  Rectangle
  Stream
    ReadStream
    WriteStream
      ReadWriteStream
        FileStream
        TerminalStream
  StringModel
  TextSelector
  UndefinedObject
SwappedOutObject
```

Ursprung der Klassenbibliothek

- Die grundlegende Funktionalität, die notwendig ist, um Messages zu schicken, ist an der Spitze der Klassenhierarchie definiert (Object, Behavior, Class...)
- Wenn Klassen durch Messages erzeugt werden, wer erzeugt dann diese Klassen? (Henne-Ei-Problem)
- gewisse Methoden (etwa 50-100) sind in Assembler/C programmiert ("primitive methods")
- Programmierumgebung mit der gesamten Klassenbibliothek ausgeliefert
- Zustand einer Programmiersitzung wird abgespeichert - alle Objekte sind bis zwischen Sessions persistent

Die wichtigsten Klassen



Object

- Die bei Object definierten Operationen stehen allen Objekten zur Verfügung (Beispiele):

class
respondsTo: *aMessageSelector*
== *anObject*
~~ *anObject*
=, ~= *anObject*
isKindOf: *aClass*, **printString, storeString**
implementedBySubclass, become: *anObject*
doesNotUnderstand: *aMessage*
shallowCopy, deepCopy
yourself,

UndefinedObject / Boolean

UndefinedObject

- anObject **isNil**
  anObject **notNil**
  (dieselben Op. bei Object, mit umgekehrtem Resultat)

Boolean

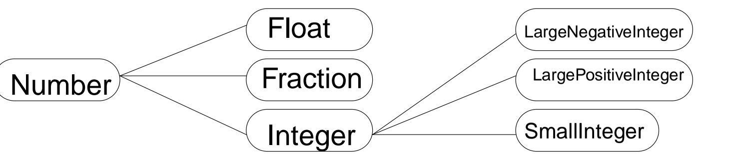
- "normale" boolesche Operationen: **&**, **|**, **not**
  aBoolean **&** *anotherBoolean*
- "Partielle Auswertung: **and:**, **or:**
  aBoolean **and:** *aBlock*

  (color **=** #red) **|** (color **=** #green)
  (i **<=** anArray **size**) **and:** [(anArray **at:** i) **>** 0]

Magnitude

- Abstrakte Oberklasse für skalare (geordnete) Typen

  Date, Time, Character, Number
- einheitlich: Vergleichsoperatoren
- Konversion, Tests (Bsp. Character):
  asciiValue, isDigit, isLetter, isLowerCase, isVowel, asUpperCase
- Zahlen



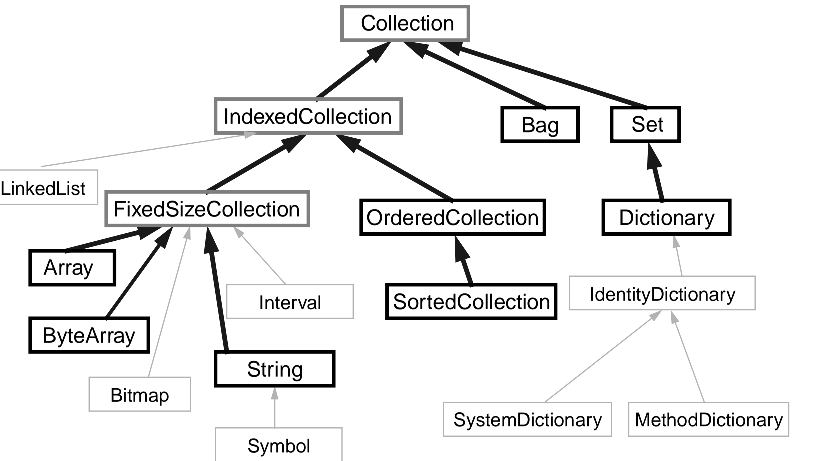
Zahlen

- Automatische Konversion (Integer ⇒ Fraction ⇒ Float)
- A **/** B (konvertiert nach Fraction): 2/4 ⇒ 1/2
  aber: 3 / 2.0 ⇒ 1.5
- Kein Integer-Overflow
- Übliche Operatoren: **rounded, negated, exp, ln, sqrt, even, odd, abs, sin, cos, tan**, ...
- *Schleifen*
  n **to:** *m* **do:** *aBlock*
  n **timesRepeat:** *aBlock*
- Auch legal: (n **to:** *m*) **do:** *aBlock*
  x := (n **to:** *m*). x **do:** *aBlock* ⇒ Klasse Interval

Collections

- “Arbeitspferd” des Smalltalk-Systems
- Behälter für beliebige Objekte
- Implementierungen für Standard-Datentypen wie Mengen, Listen, Arrays, Strings
- Unterstützung für Sortieren
- Enge Verwandschaft: Streams

Collections - Überblick (ST/V ST-80)



Die wichtigsten Collection-Klassen

Array	Feste Größe, Integer-Index beginnend mit 1
Set	Menge (ungeordnet, keine Duplikate)
Bag	Menge (mit Duplikaten)
ByteArray	Array,Einträge 0..255 (hptsl. systemintern benutzt
Dictionary	Wie Array, aber beliebige Objekte als Index
Interval	Zahlenintervall
OrderedCollection	Wie Array, kann aber neue Elemente hinzufügen (bevorzugt vorn und hinten)
SortedCollection	automatisch sortiert

Allgemeines Collection-Protokoll

Customer := Bag with: #John. Supplier := #(John Peter).

Ausdruck	*Ergebnis*/Wert v. Customer
Customer add: #Bob	Bag(John Bob)
Customer addAll: Supplier	Bag(John John Peter Bob)
Customer occurrencesOf: #John	*2*
Customer removeAll: Supplier	Bag(John Bob)
Customer remove: #Bob	Bag(John)
Customer isEmpty	*false*
Customer size	*1*
Customer includes: #John	*true*
Customer addAll: Supplier	Bag(John John Peter)

Iterieren über Collections

- Einfache Iteration: do

  | count |
  count := 0.
  customer **do**: [:aName | count := count + aName size].
  ⇒ *13*

- Auswahl

  Customer **select:** [:aName | aName == #John]
  ⇒ *Bag(John John)*

  Customer **reject:** [:aName | aName == #John]
  ⇒ *Bag(Peter)*

  Customer **detect:** [:aName | aName includes: #P]
  ⇒ *Peter*

Iterieren über Collections (2)

Customer **detect:** [:aName | aName = #Mary]
ifNone: ['Not found']
⇒ *'Not found'*

- Auswerten

  Customer **collect:** [:aName | aName first]
  ⇒ Bag ($J $J $P)

- Deklarative Schleifenkonstrukte
- Einheitlich für alle Collections
- **Selbst definierbar!** (für neue Subklassen, in neuen Varianten)

Exkurs - Queries in OODB's

Normale SQL-Abfrage:

angestellte(<u>svnr</u>,name,abtnr,adr,gehalt),
abteilung(<u>abtnr</u>,name,vorstand,gebäude)

select ang.svnr ← Relationen
  from angestellte ang, abteilung a
  where ang.abtnr = a.abtnr **and** a.name = "Vertrieb".

Smalltalk:

Person **subclass** Angestellt **instanceVariables**: 'svnr name abt ...' ...
Object **subclass** Abteilung **instanceVariables**: 'name ...' ...

angestellte ← Collection (z.B. Set) von Instanzen der Klasse Angestellt
  select: [:ang | ang abt name = "Vertrieb"]

Konversion

- x **asArray**
  x **asBag**
  x **asSet**
  x **asOrderedCollection**
  x **asSortedCollection** <= als Vergleichsop.
  x **asSortedCollection:** sortBlock
- Beispiel:
  #(4 5 3) asSortedCollection asArray ⇒ #(3 4 5)

  (#(4 5 3) asSortedCollection: [:x :y | x > =y]) asArray
  ⇒ #(5 4 3)

Arrays / Strings / Dictionaries

- Erzeugung, Änderung:

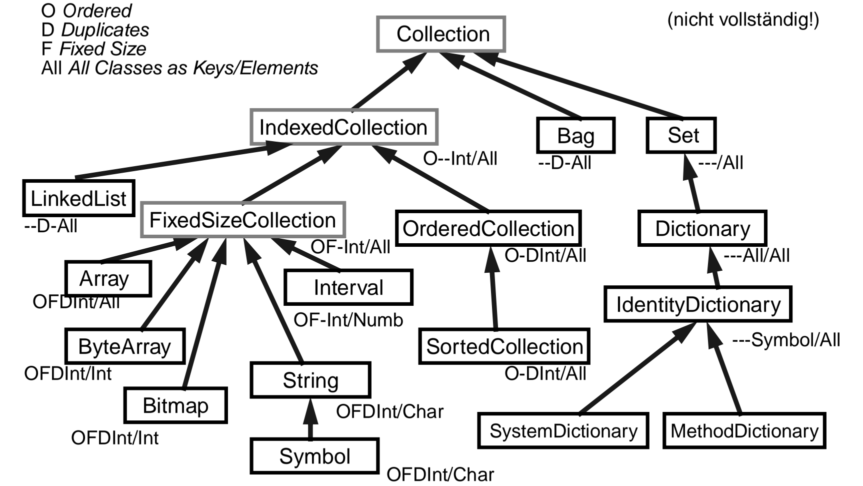
  x := (Array **new:** 100)
    at: 1 **put:** 'hello';
    at: 2 **put:** #(1 2 3 (5 6 7));
    at: 3 **put:** #smile;
    yourself.

  Kürzere Schreibweisen existieren
  x := Array **with:** 'hello' **with:** #(1 2 3 (5 6 7)) **with:** #smile.
- Zugriff: x **at:** 1
- Dictionaries: Indexwerte nicht auf Integers beschränkt. Index wird als Schlüssel (*key*) bezeichnet.
- Strings: Matching, textbezogene Op., Ein/Ausgabe

OrderedCollections

- Im wesentlichen wie dynamische Arrays
- Einfügen/Löschen am Anfang bzw. Ende:
  x **addFirst:** #John
  x **addLast:** #Peter
  x **removeLast**
- Einfügen in der Mitte (Löschen analog):
  x **add:** #John **after:** #Mary
  x **add:** #Peter **before:** #John
- Einfügen an bestimmten Positionen:
  x **at:** 7 **put:** #John

Collections (ST/V ST-80)



Woher diese Hierarchie?

4 wesentliche Gruppen von Operationen

- Erzeugen, Iterieren, Tests: alle, mit speziellen Erweiterungen (z.B. Literale für Arrays)
- Hinzufügen, Entfernen (add:, remove:): Bags, Sets, OrderedCollections, SortedCollections
- Random Access (at:, at:put:): Dictionaries, IndexedCollections
- Entfernen nach Reihenfolge (addFirst, removeFirst): IndexedCollections, aber nicht FixedSizeCollections
- Alle Collections außer FixedSizeCollections verwalten ihre Größe selbst (Eingreifen nur f. Effizienz sinnvoll)

Semantische Beziehungen

- Vererbung bedeutet Übereinstimmung von Schnittstellen
- Abstrakte Klassen: Collection, IndexedCollection, FixedSizeCollection
- Dictionary und OrderedCollection sind funktional gesehen, Erweiterungen von Array.
  Strings, Symbole, ByteArrays und Bitmaps sind spezielle Arrays.
- ⇒ Die Collection-Hierarchie entspricht *ausgewählten* Kriterien, nicht *allen* Kriterien.

Implementierungsbeziehungen

- Vererbung bedeutet Wiederverwendung von Implementierung (Code)
- SortedCollection: implementiert als OrderedCollection
- Set: implementiert als Hash-Tabelle über den enthaltenen Objekten
- Dictionary: implementiert als Hashtabelle über den Schlüsseln der Objekte. Einträge: sog. *Associations* (speichern die Kombination Schlüssel/Wert)
- intern praktisch, semantisch nicht einsichtig
- ⇒ Kapitel über Vererbung

Alternative Hierarchie

- Collection
  - LinkedList
    - Set
  - IndexedCollection
    - Array
      - ByteArray
        - String
          - Symbol
    - OrderedCollection
      - SortedCollection
    - Dictionary
      - ...
  - Interval

Collections in Little Smalltalk (Version 3)

Klassen/Methoden

	Col	Index	Array	Str	Dict	Intv	List	Set
add:	✎						✎	✎
at:		✎						
at:ifAbsent:		✎			✎		✎	
at:put:			✎		✎			
binaryDo:			✎		✎			
do:	✎		✎			✎	✎	
includes:	✎							
includesKey:		✎	✎					
isEmpty	✎							
occurrencesOf:	✎							
printString	✎			✎				
reject:	✎							
select:	✎	✎						
size	✎		✎	✎				
sort	✎							

✎ ... Implementierung der Methode (Little Smalltalk Ver. 3)

Streams

Stream
- ReadStream
- WriteStream
  - ReadWriteStream
    - FileStream ...

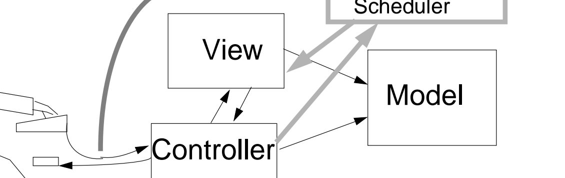
Protokoll:

next	position	position: n
nextPut: anObject	reset	skip: n
atEnd	isEmpty	skipTo: x
contents	upTo: x	next: n

Verwendung: Filehandling, Terminal-Ein/Ausgabe, Generierung von Zufallszahlen

Window-System (1)

- 3 Subhierarchien (MVC-Paradigma)
  - Model: entspricht der eigentlichen Applikation
  - View: Das Window, zuständig für Darstellung am Schirm
  - Controller: zuständig für die Eingabe (in manchen Implementierungen in View integriert)



Window-System (2)

- Vorteil des MVC-Systems: Saubere Trennung Eingabe/Ausgabe/Anwendung ⇒ hohe Flexibilität, geringer Änderungsaufwand bei Varianten der Standardklassen
- Nachteil: Lernaufwand durch feine Unterteilung, vorausgesetzt, es wird nicht Standardfunktionen verwendet.
- Alternativen: z.B. Zusammenlegung von View und Controller zu einer einzigen Klasse (z.B. IBM Smalltalk)
- Es existiert keine klar überlegene Lösung. Das Problem liegt nicht bei der Graphik, sondern in der Multi-Window-Oberfläche.

Mini-Applikation - Lotsenbeispiel

- Schiffe haben unterschiedlich riskante Ladung
- Lotsen verfügen über eine gewisse Erfahrung, die bestimmt, ob sie ein bestimmtes Schiff führen dürfen
- Die Zuordnung von Lotsen zu Schiffen wird über einen jeweils ausgestellten Lotsenschein festgehalten
- Die Menge aller Lotsen muß irgendwie verwaltet werden, damit man einen Lotsen finden kann, wenn ein Schiff ankommt

Lotsenbeispiel - Klassen

```
Object subclass: #Lotse
    instanceVariableNames: ‘name dienstzeit belegt’
    classVariableNames: ‘AktiveLotsen’.
Object subclass: #Schiff
    instanceVariableNames: ‘reederei name bRT
        geschwindigkeit ladung versicherungskl’.
Schiff subclass: #Tanker
    instanceVariableNames: ‘’
Object subclass: #Document
    instanceVariableNames: ‘ausstellungszeit’
Document subclass: #Lotsenschein
instanceVariableNames: ‘fahrzg beauftragter anfangszeit’!
```


Klasse Schiff/Tanker

```
!Tanker methods!
ladung: aSymbol
        “Neuer Wert für ladung. Bereichsüberprüfung notwendig”
    (#(Rohoel Erdgas Heizoel ...) includes: aSymbol) not
        ifTrue: [ self error: ‘Falsche Ladungsangabe’ ]
        ifFalse: [ ladung := aSymbol ]
versicherungsklasse: anInt
        “Neuer Wert für versicherungskl. Bereichsüberprüfung notwendig”
    (anInt between: 4 and: 6)
        ifTrue: [ versicherungskl := anInt ]
        ifFalse: [self error ...]
versicherungsklasse
    ^versicherungskl
```

Klasse Lotse

```
!Lotse class methods !
initialize
        "beim Systemstart aufrufen"
    AktiveLotsen := Set new.

new
        "Verwaltung der Liste aktiver Lotsen nicht vergessen"
    ^AktiveLotsen add: super new initialize! !
            "super: Bezug auf Methode inOberklasse"

aktiveLotsen
        "liefert den Wert der Klassenvariablen"
    ^AktiveLotsen! !
```


Klasse Lotse (2)

```
!Lotse methods!
initialize
    dienstzeit := 0. ...!

geleistet: arbeitszeit          "Erfahrungswert erhöhen"
    dienstzeit := dienstzeit addTime: arbeitszeit !

zulässigFür: schiff             "abhängig von der Erfahrung"
    ^dienstzeit hours/ 500 > schiff versicherungsKlasse!

übernimmt: einLS
    belegt := einLS!

istFertig
    belegt := nil!

belegt
    ^belegt notNil! !
```


Klasse Lotsenschein

```
!Lotsenschein class methods!
ausfüllenFür: einSchiff
    ^self new ausfüllenFür: einSchiff!

!Lotsenschein methods!
ausfüllenFür: einSchiff
        "Lotsen finden und Belegung festhalten"
    beauftragter := Lotse aktiveLotsen
        detect: [ :lo |
            lo belegt not and:
            [ lo zulässigFür: einSchiff ] ]
        ifAbsent: [ ^... "Fehlerbehandlung" ].
    beauftragter übernimmt: self
    anfangszeit := Time now.
    fahrzg := einSchiff.
```


Klasse Lotsenschein (2)

```
anlegen
        "Lotsen freigeben und Erfahrung erhöhen"
    beauftragter
        geleistet: (Time now subtractTime: anfangszeit);
        istFertig!
!
```

Ablauf