Objektorientierte Programmierung

Prof. Dr. Uwe Kastens

WS 2013 / 2014

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 100

Ziele

Die Studierenden sollen lernen,

- Konzepte und Konstrukte objektorientierter Sprachen **planvoll in der Programmentwicklung einzusetzen**,
- **höhere Paradigmen** zur objektorientierten Programmierung anzuwenden und
- **Probleme und Grenzen** objektorientierter Programmierung zu erkennen.

Durchführung

- Die in der Vorlesung vermittelten Methoden und Techniken werden in **Übungen praktisch erprobt**.
- Als Programmiersprache wird **Java** verwendet.
- Es werden **Fallstudien** durchgeführt und vorgegebene Programme untersucht und weiterentwickelt.
- Es wird **unter Anleitung in kleinen Gruppen** an vorbereiteten Aufgaben gearbeitet.

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 101

Ziele:

Ziele bewusst machen

in der Vorlesung:

Erläuterungen dazu

Verständnisfragen:

- Stimmen die Ziele mit Ihren Vorstellungen überein?

Inhalt

Thema	Semesterwoche	Buch
1. Grundlagen		1, 2, 3, 12
Allgemeine OO-Konzepte	1, 2	[Bruce]
Statische Typisierung in OO-Sprachen, Generik	3 - 5	
2. Einsatz von Vererbung Spezialisierung, Klassifikation, Rollen, Eigenschaften, inkrementelle Weiterentwicklung	6 - 8	8 - 11
3. Entwurfsmuster zur Entkopplung von Modulen Factory Method, Bridge, Observer, Strategy	9	Gamma
4. Programmbausteine, Bibliotheken, Programmgerüste Kopplung von Bausteinen, Strukturkonzepte von Bibliotheken	10	Budd[2] P-H
5. Entwurfsfehler Missbrauch der Vererbung, Antipatterns, OO-Überraschungen	11	
6. Jenseits von Java Mehrfachvererbung, konsequent OO, kontrollierte Vererbung, prototypbasiert	12, 13	Budd[2]

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 102

Ziele:

Überblick über den Inhalt bekommen

in der Vorlesung:

Die Struktur wird erläutert.

Verständnisfragen:

- Was stellen Sie sich unter den Themen vor?

Literatur

Elektronisches Skript:

- **http://ag-kastens.upb.de/lehre/material/oop**

Buch zur Vorlesung:

- **Timothy Budd: Understanding Object-Oriented Programming with Java, Updated Edition, Addison-Wesley, 2000**
- Timothy Budd: An Introduction to Object-Oriented Programming, Third Edition, Addison-Wesley, 2002
- **Kim B. Bruce: Foundations of Object-Oriented Languages, MIT Press, 2002**

Weitere Bücher zu Thema:

- Timothy Budd: Object-Oriented Programming, Addison-Wesley, 1991
- Arnd Poetzsch-Heffter: Konzepte Objektorientierter Programmierung, Springer, 2000
- E.Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
- Arnold, Ken / Gosling, James: The Java programming language, Addison-Wesley, 1996
- Antero Taivalsaari: On the Notion of Inheritance, ACM Computing Surveys, Vol. 28, No. 3, September 1996
- Peter Coad, David North, Mark Mayfield: Object Models: Strategies, Patterns & Applications, 2nd ed., Yourdon Press, Prentice Hall, 1997

Weitere Hinweise im Vorlesungsmaterial unter *Internet*

© 2010 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 103

Ziele:

Passende Bücher kennenlernen

in der Vorlesung:

Erläuterungen zu den Referenzen

Elektronisches Skript

Vorlesung Objektorientierte Programmierung WS 2013/2014

ag-kastens.upb.de/lehre/material/oop/

UK Lehre Suchen Wörterbücher WissOrg Reisen Dienste News Das Institut Google Maps Wikipedia Apple

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Objektorientierte Programmierung WS 2013/2014

Folien
Aufgaben
Organisation
Hinweise

SUCHEN:

Vorlesung Objektorientierte Programmierung WS 2013/2014

Vorlesungsfolien

- Kapitelübersicht
- Folienverzeichnis
- Drucken

Übungsaufgaben

- Aufgabenblätter
- Drucken

Organisation

- Personen, Termine, Regeln
- Aktuelles

06.10.2013 Vorlesungsbeginn 15.10.2013 um 09:15 in F0.530

Wissenswertes

- Ziele
- Literatur
- Inhalt Budd: Understanding OOP with Java
- Internet-Links

Veranstaltungs-Nummer: L.079.05700

Generiert mit Camelot | Probleme mit Camelot? | Geändert am: 06.10.2013

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 104

Ziele:
Skript kennenlernen

in der Vorlesung:
Hinweise auf die Gliederung und Verwendung des Skriptes.

Verständnisfragen:
- Suchen Sie das Skript im Netz.
- Explorieren Sie es.

Organisation

Personen

Sprechstunde Uwe Kastens:

- `Mi  16:00 - 17:00 Uhr`
- `Die 11:00 - 12:00 Uhr`

Übungsbetreuer:

- Peter Pfahler

Termine

Vorlesung

- `Di, 9:15 - 10:45 Uhr F0.530`

Beginn: Di, 15. Oktober 2013 um 9:15 Uhr

Übungen

Die Übungen werden im 14-tägigen Abstand 2-stündig angeboten. Das Vorlesungsverzeichnis sieht 4 Übungsgruppen vor:

- **G1**: Dienstag 11:00 Uhr, *ungerade Wochen*, Beginn 22.10.2013, erst in F0.530, dann im Rechner-Pool F1 (hinterer Teil)
- **G2**: Dienstag 11:00 Uhr, *gerade Wochen*, Beginn 15.10.2013, erst in F0.530, dann im Rechner-Pool F1 (hinterer Teil)
- **G3**: Donnerstag 09:15 Uhr, *ungerade Wochen*, Beginn 24.10.2013, erst in F2.211, dann im Rechner-Pool F1 (hinterer Teil)
- **G4**: Freitag 09:15 Uhr, *gerade Wochen*, Beginn 18.10.2013, erst in F2.211, dann im Rechner-Pool F1 (hinterer Teil)

Prüfungstermine

Mündliche Prüfungen von ca 30 min Dauer im Rahmen von Modulprüfungen; für Studierende anderer Studiengänge als Informatik auch Einzelprüfungen.
Es werden zwei Prüfungszeiträume angeboten:

1. 12.-14. Februar 2014
2. 01.-03. April 2014

Zu Anmeldung in PAUL und Terminvergabe siehe http://www.cs.uni-paderborn.de/studierende/pruefungswesen/pruefungsanmeldung.html

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 105

Ziele:

Termine fixieren

in der Vorlesung:

Abstimmung der Termine

1. Grundlagen, 1.1 Allgemeine OO-Konzepte Objektorientierte Denkweise

- Modellierung und Entwurf von **Systemen interagierender Objekte**
  statt Programmabläufe mit passiven Daten

- handelnde Objekte mit **Eigenschaften**, veränderlichem **Zustand** und eigenen **Operationen** darauf
  statt Funktionen auf passiven Daten

- Objekt bestimmt, wie es einen Methodenaufruf ausführt:
  Aufrufer schickt Nachricht (message) an Empfänger-Objekt (receiver)
  Objekt führt Methode zu der Nachricht im Objektzustand aus
  Eigenverantwortlichkeit, **Selbständigkeit der Objekte**

- **Schnittstelle** (Protokoll):
  von außen beobachtbare Eigenschaften, benutzbare Methoden des Objektes
  Schnittstelle **beschreibt das Was und verbirgt das Wie.**

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 106

Ziele:

Konzepte aus objektorientierter Modellierung

in der Vorlesung:

Erläuterungen dazu

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 1

nachlesen:

Vorlesung Softwareentwurf (SE)

Klassen-basierter Entwurf

- Jedes **Objekt** gehört einer **Klasse** an.
- Klasse **definiert das Verhalten** ihrer Objekte
- Klasse als **abstrakter Datentyp** mit Implementierung
- **Klassen-Hierarchie**:
  Unterklasse **erbt** Eigenschaften und Methoden von ihrer Oberklasse, erweitert und verändert sie, z. B. zur Spezialisierung
- **Polymorphie:**
  allgemeine Variable enthält spezielles Objekt;
  Typ der Variable erlaubt Methodenaufruf mit bestimmter Signatur, Klassenzugehörigkeit des Objektes bestimmt wie der Aufruf ausgeführt wird: dynamische Methodenbindung (zur Laufzeit)
- **dynamische Methodenbindung:**
  Software-Module können **unabhängig entwickelt** werden,
  zur Laufzeit umkonfiguriert werden

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 107

Ziele:

Grundlegende Begriffe zu Klassen und Objekten

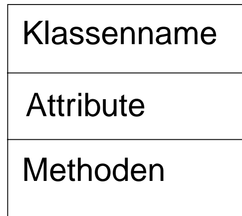
in der Vorlesung:

Erläuterungen dazu

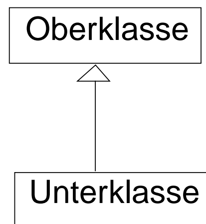
nachlesen:

Budd: Understanding OOP with Java, Abschnitt 1

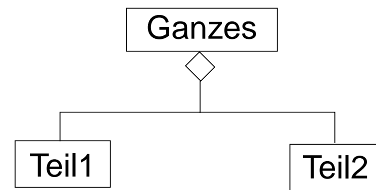
Entwurfsnotation UML Klassendiagramme



Klasse

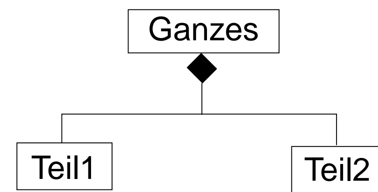


Vererbung



Aggregation
Teile existieren auch unabhängig

Assoziation



Komposition
Teile existieren nicht unabhängig

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 108

Ziele:
Graphische Notation zur Dokumentation

in der Vorlesung:
Erläuterungen dazu

nachlesen:
Vorlesung TSE

Verständnisfragen:
- Schlagen Sie die Bedeutung der Diagramme in den Unterlagen der Vorlesung Softwareentwurf nach.

Objektorientierte Programmiersprachen

1966	**Simula**	Klassenhierarchie mit Einfachvererbung, statisch typisiert, dynamische Methodenbindung, diskrete Simulation, Algol 60 ist Teilsprache
1980	**Smalltalk**	Klassenhierarchie mit Einfachvererbung, dynamisch typisiert, dynamische Methodenbindung, konsequent objektorientiert, interpretierte Zwischensprache
1986	**C++**	Klassenhierarchie mit Mehrfachvererbung, statisch typisiert, dynamische Methodenbindung, generische Klassen, ANSI-C ist Teilsprache
1988	**Eiffel**	Klassenhierarchie mit Mehrfachvererbung, statisch typisiert, dynamische Methodenbindung, generische Klassen, explizite Vererbung
1994	**Java**	Klassenhierarchie mit Einfachvererbung, Interfaces, statisch typisiert, dynamische Methodenbindung, Internet-Bezüge, Prozesse, interpretierte Zwischensprache, umfassende Bibliotheken
1995	**PHP**	Skriptsprache; Server-seitige Web-Progr.; Klassen-basierte Vererbung
1995	**JavaScript**	Skriptsprache; Client-seitige Web-Progr.; Objekt-basierte Vererbung
2001	**C#**	Microsoft; Eigenschaften wie Java

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 108a

Ziele:

Charakteristische Eigenschaften von OO-Sprachen

in der Vorlesung:

Erläuterungen dazu

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 2

nachlesen:

Vorlesung GdP

Grundlegende objektorientierte Sprachkonstrukte
Klassen und Objekte

Eine Klasse definiert **gleichartige Objekte**: Daten mit Operationen (**Methoden**) darauf.
Operationen abstrahieren von der Implementierung.

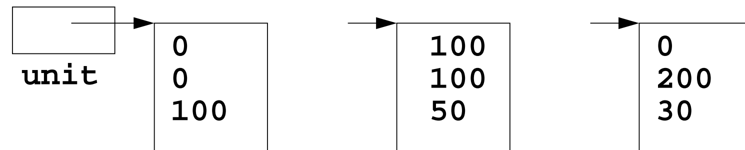
Klasse	`Ball`

Attribute	`Rectangle location`
	`double dx, dy`
	`Color color`

Methoden	`getX ()`
	`getY ()`
	`getRadius ()`
	`move ()`
	`paint (...)`

Konstruktormethode	`Ball (int x, int y, int r)`

3 Objekte der Klasse `Ball`



`Ball unit = new Ball (0, 0, 100);`   `unit.move();`

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 109

Ziele:

Grundbegriffe zu Klassendeklarationen

in der Vorlesung:

Erläuterungen dazu

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 5 (Fallstudie)

Verwendung von Attributen

Objektbezogene Attribute: Variable oder Konstante eines jeden Objektes

- Eigenschaft des Objektes, unveränderlich oder explizit veränderlich — `Color color;`
- Zustand des Objektes, veränderlich, auch durch andere Operationen — `boolean visible;`
- Teil des Objektes — `Flap leftFlap, rightFlap;`
- Bezug zu anderem Objekt — `Monitor registers;`

Klassenbezogene Attribute: Information für alle Objekte der Klasse gemeinsam

- feste Codierung von Werten — `final static int CLUBS = 1;`
- globale Information für oder über Objekte der Klasse — `static int ballCount = 0;`

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 110

Ziele:

Verwendungsart bewusst machen

in der Vorlesung:

- Verwendungsarten abgrenzen,
- Beispiele dazu

Verständnisfragen:

- Geben Sie Beispiele zu den Verwendungsarten an.

Zugriffsrechte für Attribute und Methoden

Regeln schränken ein, **wo** Zugriffe auf Attribute/Methoden **welcher Klasse** stehen dürfen:

`private`: `class C { private int x; ... }`
Zugriff nur **im Rumpf der Klasse** `C`,
unqualifiziert (`x`) auf `x` des „eigenen" `C`-Objektes und
qualifiziert (`q.x`) auf `x` eines beliebigen `C`-Objektes

Package-weit (default): `class C { int x; ... }`
Zugriff in jeder Klasse des **Package**, in dem `C` deklariert ist;
unqualifiziert in `C` und seinen Unterklassen oder
qualifiziert auf Attribute/Methoden beliebiger `C`-Objekte

`protected`: `class C { protected int x; ... }`
Zugriff **Package-weit und zusätzlich auch in Unterklassen** von `C`,
die nicht zum Package von `C` gehören

`public`: `class C { public int x; ... }`
unbeschränkter Zugriff

qualifizierter Name: `a.x` mit `a` als Ausdruck vom Typ `C` oder einer Unterklasse von `C`

unqualifizierter Name: `x` im Rumpf der Klasse `C` oder einer Unterklasse von `C`, entspricht `this.x`

© 2009 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 111

Ziele:

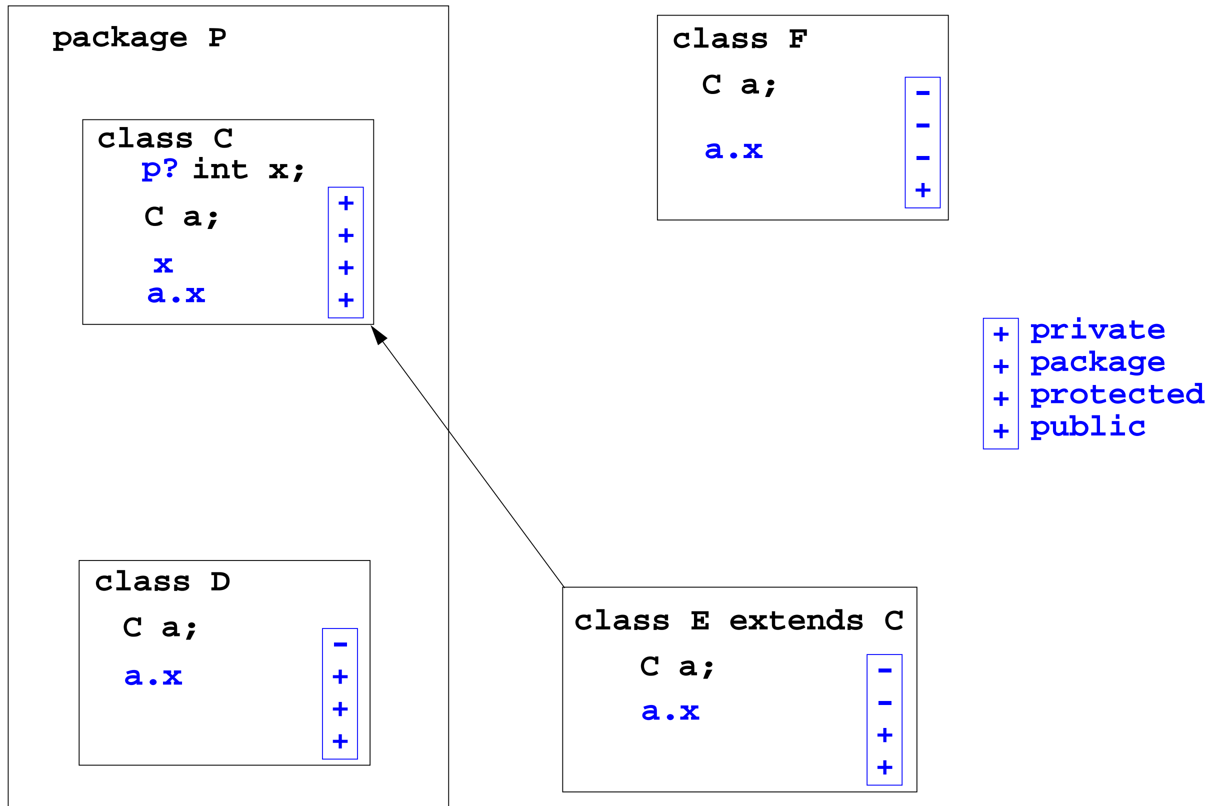
Zugriffsrechte verstehen

in der Vorlesung:

Erläuterungen dazu:

- Regeln unterscheiden nicht, ob auf das "eigene" oder auf ein anderes Objekt zugegriffen wird.
- Besonderheiten von protected

Beispiel zu Zugriffsrechten in Java



© 2010 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 111a

Ziele:

Zugriffsrechte am Beispiel

in der Vorlesung:

Erläuterungen dazu:

- Regeln unterscheiden nicht, ob auf das "eigene" oder auf ein anderes Objekt zugegriffen wird.
- Besonderheiten von protected

Zugriff auf Attribute

Attribute in der Regel `private`:

- **keine unkontrollierte Zustandsänderung**,
- keine Verpflichtung, das Attribut nicht zu ändern

Vorsicht mit `protected`: Package-weit sichtbar, Verpflichtung gegen Unterklassen

Attribute lesen und zuweisen nur mit **get- und set-Methoden**
diese mit passenden Rechten versehen

Zustandsattribute meist nicht von außen zugänglich,
werden von Methoden benutzt und zugewiesen

unveränderliche Objekte
kein Attribut wird nach der Initialisierung verändert (auch nicht in Unterklassen),
z.B. `java.lang.String`;
Kopien sind nicht unterscheidbar, referenzielle Transparenz

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 112

Ziele:

Konventionen für den Zugriff

in der Vorlesung:

Begründung der Regeln

Verständnisfragen:

- Diskutieren Sie die Anwendung der Konventionen an der ersten Fallstudie.

Konstruktor-Methoden

Konstruktor-Methode dient der **Initialisierung der Attribute** bei der Objekterzeugung.
Danach muss das Objekt in einem benutzbaren Zustand sein.

Der **Hauptkonstruktor** hat alle einstellbaren Werte als Parameter.

```
public Ball (int x, int y, int r, Color c)
{  location = new Rectangle (x-r, y-r, 2*r, 2*r);
   color = c;
   dx = 0; dy = 0; // initially no motion
}
```

Weitere Konstruktoren setzen einige Attribute mit default-Werten (überladene Konstruktoren):

```
public Ball (int x, int y, int r)
{  this (x, y, r, Color.blue);
}
```

Alle Konstruktoren rufen den Hauptkonstruktor auf:
zentrale Stelle, wo jedes Objekt initialisiert wird.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 113

Ziele:

Systematischer Entwurf von Konstruktoren

in der Vorlesung:

Erläuterungen

- von überladenen Konstruktoren,
- des Aufrufes this(...);

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 5

Verständnisfragen:

- Diskutieren Sie die Systematik an der ersten Fallstudie.

Klasse und Objekte - Typ und Werte

1. **Klasse** definiert die Eigenschaften ihrer **Objekte**
2. Objekte **existieren im Speicher**
3. Objekte haben **Identität,** ihre Objektreferenz
4. `new` erzeugt ein **neues Objekt** verschieden von allen anderen
5. **Objekte werden** nicht kopiert sondern **geklont**.
6. **Klasse als Typ von Variablen**: Variable kann eine Referenz auf ein Objekt derKlasse aufnehmen
7. **Klasse als Typ von Ausdrücken**: Auswertung des Ausdrucks liefert eine Referenz auf ein Objekt der Klasse
8. **Typanpassung** verändert weder das Objekt noch seine Referenz; erlaubt dem Übersetzer, die Referenz, als Referenz eines anderen Typs zu behandeln

1. **Typ** definiert die Eigenschaften seiner **Werte**
2. Werte können **Speicherinhalte** sein
3. Werte haben keinen Speicher, **keine Identität**
4. zwei **gleiche Werte** sind **nicht unterscheidbar**
5. **Werte werden kopiert**.
6. **Typ von Variablen**: Variable kann Wert des Typs aufnehmen
7. **Typ von Ausdrücken**: Auswertung des Ausdrucks liefert einen Wert eines Typs.
8. **Typanpassung** Verändert ggf. die Repräsentation des Wertes, z.B. von `int` nach `float`

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 114

Ziele:

Klasse und Typ unterscheiden können

in der Vorlesung:

Erläuterungen zu

- Behälter und Inhalt,
- Kopieren und Klonen,
- problematisch in C++: Objekte als Werte von Variablen,
- Typanpassung

Verständnisfragen:

- Erklären Sie den Unterschied zwischen Kopieren und Klonen.

Methoden

Operationen einer Klasse:

```
public void setMotion (double ndx, double ndy)
   {dx = ndx; dy = ndy;}
```

Typen der Parameter und des Ergebnis ist die **Signatur** der Methode
(in Java: nur Typen der Parameter - ohne Ergebnistyp)

```
setMotion: double x double -> void
```

Aufruf kann als Seiteneffekt den **Objektzustand ändern**;
hier: Geschwindigkeit zuweisen.

Aufruf kann als Seiteneffekt den **Zustand von Parameterobjekten ändern,**
z. B. `Wand.reflektiere (ball);`

Zugriffsrechte für Methoden wie für Attribute

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 115

Ziele:

Methoden verstehen

in der Vorlesung:

Erläuterungen zu

- Signaturen,
- Seiteneffekten

nachlesen:

Vorlesung GdP, 6. Funktionen, Parameterübergabe

Methodenaufrufe

Notation von Methodenaufrufen

in der Klasse, in der die Methode deklariert ist, **unqualifiziert**

```
setMotion (1.0, 1.0);
```

qualifiziert für „fremdes“ Objekt

```
theBall.setMotion (1.0, 1.0);
```

Ausführung eines Aufrufs `a.m(p)`:

1. Ausdruck `a` auswerten, Ergebnis ist eine Referenz auf ein Objekt `obj`; mit dem (dynamischen) Typ des `obj` die aufzurufende Methode `m` bestimmen (dynamische Methodenbindung)
   (Überladung wird anhand der Signatur schon zur Übersetzungszeit aufgelöst)
2. In der Umgebung von `obj` eine Schachtel `s` für einen Aufruf der Methode `m` von `obj` bilden;
   statischer Vorgänger braucht nicht auf dem Laufzeitkeller zu liegen;
   Werte der aktuellen Parameter `p` berechnen und an formale Parameter in `s` zuweisen;
3. Rumpf der Methode mit der Schachtel `s` ausführen;
   ggf Ergebnis an die Aufrufstelle liefern; hinter die Aufrufstelle zurückkehren

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 115a

Ziele:

Methodenaufrufe verstehen

in der Vorlesung:

Erläuterungen zu

- Parameterübergabe,
- Umgebungen
- siehe GPS-4-6 und GPS-6-2a

nachlesen:

Vorlesung GdP, 6. Funktionen, Parameterübergabe

Verständnisfragen:

- Welche Variablen sind bei der Ausführung eines Methodenaufrufes zugänglich (Umgebung)? Wo liegen sie im Speicher?

Schnittstelle und Implementierung

Abstraktion: Außensicht **Was**; Innensicht **Wie**

Schnittstelle einer Methode:

Signatur: `setMotion: double x double -> void`

Sprachkonstrukt:

abstrakte Methode, die in Unterklassen implementiert wird:

```
abstract void setMotion (double, double);
```

Schnittstelle einer Klasse:

Menge der Signaturen der von außen aufrufbaren Methoden (`public`)

Implementierung einer Klasse:

Implementierung der Methoden der Schnittstelle
mit Methodenrümpfen und
dazu notwendige Attribute und Hilfsmethoden

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 116

Ziele:

Konzept Schnittstelle verstehen

in der Vorlesung:

Erläuterungen zu

- Schnittstellen von Klassen,
- Schnittstellen als Typen

Schnittstelle als selbständiges Sprachkonstrukt

in Java: `interface`,
in C++: rein abstrakte Klasse,
in SML und Haskell: `signature`

benannte Menge von Methoden-Signaturen

```
public interface Enumeration
{  boolean hasMoreElements();
   Object nextElement()
      throws NoSuchElementException;
}
```

Klasse `C` **implementiert ein Interface** `Intf`,
d. h. `C` implementiert alle Methoden aus `Intf`

Schnittstelle `Intf` als Abstraktion für Objekte von Klassen, die `Intf` implementieren:

Schnittstelle als Typ, z. B.

```
Enumeration e = new A();
while (e.hasMoreElements())
{  Object x = e.nextElement();
   ...
}
```

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 116a

Ziele:

Schnittstelle als Abstraktion verstehen

in der Vorlesung:

Erläuterungen zu

- Schnittstellen als Sprachkonstrukte,

Verständnisfragen:

- Sei V eine Variable vom Typ eines Interface I. Geben Sie genau an, welche Werte V enthalten kann und wozu man V verwenden kann.

Oberklassen und Vererbung

Klasse als **Unterklasse einer Oberklasse** definieren.
(verschiedene Zwecke siehe Kapitel 2)

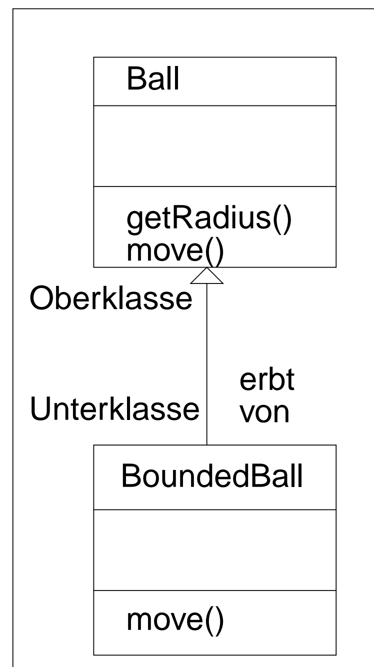
Unterklasse **erbt** von der Oberklasse
Attribute, Methoden,
Implementierungsverpflichtungen (d. h. abstrakte Methode)

Unterklasse **definiert zusätzliche**
Attribute, Methoden, Implementierungsverpflichtungen

Unterklasse **überschreibt** Methoden der Oberklasse
bei gleichem Methodennamen und -signatur.
Unterklassenobjekt führt Methodenaufruf mit seiner
statt der überschriebenen Methode aus
(dynamische Methodenbindung)

Klasse als **Typabstraktion**:
Variable vom Oberklassentyp können Referenzen von
Unterklassenobjekten aufnehmen

```
Ball b = new Bounded Ball (...); ...b.move();
```



© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 117

Ziele:

Vererbung technisch verstehen

in der Vorlesung:

Erläuterungen dazu

- Vererbungsrelation,
- überschreiben,
- Oberklassentypen

Verständnisfragen:

- Erklären Sie die dynamische Methodenbindung am Beispiel der Variablen b.

Vererbung und verwandte Konzepte

Sei **`A`** eine Oberklasse von **`B`** und in **`A`** sei eine Methode m implementiert, z. B.

```
class A { C m (String s) {...} ...}   class B extends A {...}
```

1. Wenn **`m`** in **`B`** nicht definiert ist, **erbt** **`B`** die Methode **`m`**.
2. Wenn in **`B`** eine Methode **`m`** mit gleichen Parametertypen definiert ist, **überschreibt** sie das **`m`** aus **`A`**. Seit Java 5 muss der Ergebnistyp gleich **`C`** oder eine Unterklasse von **`C`** sein (Java: substitutable); kovariante Ergebnistypen.
3. Wenn in **`B`** eine Methode **`m`** definiert ist, deren Parametertypen sich von denen von **`m`** in **`A`** unterscheiden, dann sind die beiden Methoden in **`B`** **überladen**.
   Ebenso, wenn in **`A`** eine weitere Methode **`m`** mit anderen Parametertypen definiert wird.
4. Wenn in **`B`** eine Klassenmethode (**`static`**) **`m`** definiert ist, deren Parametertypen mit denen von **`m`** in **`A`** übereinstimmen und die Ergebnistypen kovariant sind, dann **verdeckt** **`m`** in **`B`** die Methode **`m`** in **`A`**, d. h. **`m`** aus **`A`** ist in **`B`** nicht sichtbar. **`m`** in **`A`** muss dann auch **`static`** sein.

Sei **`A`** eine Oberklasse von **`B`** und in **`A`** sei eine Methode **`m`** spezifiziert, z. B.

```
abstract class A { abstract C m (String s); ...}
class B extends A {...}
```

5. Wenn **`B`** nicht abstract ist, muss **`m`** in **`B`** mit denselben Parametertypen und einem kovarianten Ergebnistyp **implementiert** werden.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 117a

Ziele:

Spracheigenschaften festigen

in der Vorlesung:

Erinnerung an Spracheigenschaften (SWE-82b)

Objektorientierte Polymorphie

Polymorph: vielgestaltig; hier im Sinne der Typen von Sprachkonstrukten

Referenzen auf **Objekte unterschiedlicher Klassen** können
in einer Variable enthalten sein oder
Ergebnis der Auswertung eines Ausdruckes sein.

Wird darauf eine Methode aufgerufen, so bestimmt
die **Klassenzugehörigkeit des Objektes (zur Laufzeit)**
welche Methode ausgeführt wird: **dynamische Methodenbindung**.

Der **statische Typ der Variable** (des Ausdruckes) garantiert zur
Übersetzungszeit, dass eine Methode mit passender Signatur existiert.

Einsatz zur

- Abstraktion
- Entkopplung von Programmmodulen
- Umkonfigurierung während der Ausführung
(z. B. Erscheinungsbild der GUI-Komponenten)

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 118

Ziele:

OO-Polymorphie verstehen

in der Vorlesung:

Erläuterungen dazu an den Beispielen

- dynamischer Typ

Taxonomy of type systems

[Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4):471–523, 1985.]

monomorphism: Every entity has a unique type. Consequence: different operators for similar operations (e.g. for `int` and `float` addition)

polymorphism: An operand may belong to several types.

ad hoc polymorphism:

overloading: a construct may different meanings depending on the context in which it appears (e.g. + with 4 different signatures in Algol 60)

coercion: implicit conversion of a value into a corresponding value of a different type, which the compiler can insert wherever it is appropriate (only 2 add operators)

universal polymorphism: operations work uniformly on a range of types that have a common structure

inclusion polymorphism: sub-typing as in object-oriented languages

parametric polymorphism: a type denotation may have formal type parameters, e.g. `('a x 'a)`; they are substituted by type **inference** (e.g. in SML) or by **generic instantiation** (C++, Java)

© 2010 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 118a

Ziele:

Understand characteristics of type systems

in der Vorlesung:

Reminder of prerequisites from PLaC:

- different polymorphisms are explained using examples of different languages;
- consequences for type analysis are pointed out.

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 6.1

Verständnisfragen:

- Which characteristics are exhibited in Java and in C?

Monomorphism and ad hoc polymorphism

```
monomorphism                          (1)
polymorphism
├── ad hoc polymorphism
│       overloading                   (2)
│       coercion                      (3)
└── universal polymorphism
    ├── inclusion polymorphism        (4)
    └── parametric polymorphism       (5)
```

monomorphism (1):
4 different names for addition:

```
addII: int   x int   -> int
addIF: int   x float -> float
addFI: float x int   -> float
addFF: float x float -> float
```

overloading (2):
1 name for addition `+`;
4 signatures are distinguished by actual operand and result types:

```
+: int   x int   -> int
+: int   x float -> float
+: float x int   -> float
+: float x float -> float
```

coercion (3):
`int` is acceptableAs `float`,
2 names for two signatures:

```
addII: int   x int   -> int
addFF: float x float -> float
```

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 118b

Ziele:

Examples illustrate monomorphism and ad hoc polymorphism

in der Vorlesung:

- The examples are explained

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 6.1

Dynamische Methodenbindung in Java

3 verschiedene Situationen für dynamische Methodenbindung in Java:

1. Schnittstellentyp:
```
interface Enumeration { boolean hasMoreElements(); ... }
Enumeration enum; .... enum.hasMoreElements() ...
```
2. Überschriebene Methode:
```
class Ball { public void move (){...} }
class BoundedBall extends Ball { public void move (){...} }
Ball b; .... b = new BoundedBall (...); ... b. move(); ...
```
3. Implementierung abstrakter Methode:
```
Class Animal { abstract void makeNoise(); ...}
Class Frog extends Animal
  {void makeNoise(){ System.out.print("QuakQuak");} }
Animal which; ... which.makeNoise();
```

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 119

Ziele:

Programmiertechnik zur dynamischen Methodenbindung verstehen

in der Vorlesung:

Erläuterung der Beispiele mit OOP-118; siehe auch GPS-1-18a.

nachlesen:

Budd: Understanding OOP with Java, Abschnitt 12

Verständnisfragen:

- Welche Alternativen gibt es zur Laufzeit?
- Unter welchen Bedingungen kann doch schon vor der Ausführung entschieden werden, welche Methode aufgerufen wird?

Überladene Methoden

Überladene (overloaded) Methoden:

Mehrere Methoden mit **gleichem Namen**, aber **unterschiedlicher Anzahl** oder **Typen der formalen Parameter** sind an einer Aufrufstelle gültig.

Die **Typen der aktuellen Parameter** im Aufruf bestimmen, welche Methode aufgerufen wird.

Methoden einer Klasse können untereinander überladen sein.

Methoden einer Unterklasse können mit geerbten Methoden überladen sein (ohne sie zu überschreiben!).

Konstruktoren können überladen sein.

Operatoren (+, /, usw.) sind meist überladen.

Überladen wird häufig auch als **Variante der Polymorphie** bezeichnet.

© 2005 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 120

Ziele:

Überladen und Überschreiben unterscheiden

in der Vorlesung:

Erläuterungen und Beispiele dazu

Verständnisfragen:

- Geben Sie Beispiele für Überladen in Java.
- Geben Sie Beispiele für Überladen in C.

1.2 Statische Typisierung in OO-Sprachen

Dieser Abschnitt folgt dem Buch
F.O.O.L von Kim B. Bruce, Kap 2, 3, 5, 6.
Darin werden Eigenschaften von OO-Sprachen, insbesondere die Typisierung, allgemeiner und konsequenter dargestellt als sie aus gängigen Sprachen wie C++, Java, C# bekannt sind. Diese Darstellung soll das Verständnis und die kritische Beurteilung von OO-Konzepten vertiefen.

Folgende Begriffe werden hier gegenüber dem vorigen Abschnitt verfeinert oder verschärft:

- Typen getrennt von Klassen
- Untertypen (subtyping) getrennt von Unterklassen (subclassing)
- Typisierungsdefekte in gängigen OO-Sprachen rigoros hergeleitet

© 2007 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 130

Ziele:

OO Typisierung als Thema erkennen

in der Vorlesung:

- Auf Typisierungsdefekte in realen Sprachen hinweisen
- Subtyping begründen
- Buch empfehlen

Begriffe zu Klassen am Beispiel

Es sollte verschiedene Namen geben für

- Klasse `CellClass`
- Typ der Objekte `CellType`
- Konstruktor

Zunächst werden alle

- Instanzvariablen als `private`
- Methoden als `public`

angesehen.

Bruce verwendet eine Notation, die daran erinnern soll, dass es sich nicht um Java, C++, etc handelt

```
class CellClass {
   x: Integer := 0;

   function get (): Integer is
   { return self.x }

   function set (nuVal: Integer): Void is
   { self.x := nuVal }

   function bump (): Void is
   { self <= set(self <= get()+1) }
}
```

Um die besondere Operation ***Senden einer Botschaft an ein Objekt*** (d.h. Aufruf einer Methode) hervorzuheben notiert Bruce `o <= m(x)` statt `o.m(x)`

`self` benennt das Objekt, für das ein Aufruf der Methode ausgeführt wird - `this` in Java.

In den meisten Sprachen kann `x` statt `self.x` und `get()` statt `self<=get()` geschrieben werden

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 131

Ziele:

OO Begriffe wiederholen und präzisieren

in der Vorlesung:

Begriffe werden erläutert

Objekttypen

Ein **Typ** ist eine Menge von Werten und den darauf anwendbaren Operationen.

Alle **Objekte**, auf die die **gleichen Operationen anwendbar** sind, sollten **denselben Typ** haben.

Anwendbarkeit einer Methode wird durch die Signatur (= Typ) der Methode bestimmt

```
get: Void -> Integer
```

Der Typ der Objekte, die zur Klasse `CellClass` erzeugt werden, ist die Menge der Methoden-Signaturen. (Instanzvariablen tragen nicht zum Objekttyp bei, da sie außen nicht sichtbar sind):

```
CellType = ObjectType {get: Void -> Integer,
                       set: Integer -> Void,
                       bump: Void -> Void }
```

Konsequenz:
Verschiedene Klassen können **Objekte desselben (bzw. äquivalenten) Typs** erzeugen, sofern die **Methoden paarweise dieselbe Signatur** haben!
(Strukturäquivalenz der Typen)

In **verbreiteten OO-Sprachen werden Klassen als Typen** aufgefasst: Alle Objekte einer Klasse C haben denselben Typ - verschieden vom Typ der Objekte jeder anderen Klasse D (Namensäquivalenz) - auch wenn die Mengen der Methodensignaturen übereinstimmen.

© 2012 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 132

Ziele:

Neues Konzept für Objekttypen verstehen

in der Vorlesung:

Strukturelle Definition von Objekttypen wird erläutert

Unterklassen und Vererbung

Eine Unterklasse erbt Instanzvariablen und Methoden der Oberklasse und kann Methoden zufügen und überschreiben, z. B.

modifies set verhindert versehentliches Überschreiben.

super macht die überschriebene Methode der Oberklasse zugänglich

```
class ClrCellClass inherits CellClass modifies set {
   color: ColorType := blue;

   function getColor (): ColorType is
   { return self.color }

   function set (nuVal: Integer): Void is
   {  super <= set (nuVal);
      self.color := red }
}
```

Typ der Objekte von `ClrCellClass:`

```
ClrCellType = ObjectType {get: Void -> Integer,
                          set: Integer -> Void,
                          bump: Void -> Void,
                          getColor: Void -> ColorType }
```

Wird **bump** für ein Objekt der Klasse `ClrCellClass` aufgerufen, dann führt `self <= set(self <= get()+1)` im Rumpf von **bump** zum Aufruf von `set` aus `ClrCellClass`.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 133

Ziele:

Vererbung wiederholen und präzisieren

in der Vorlesung:

Begriffe werden erläutert

- versehentliches Überschreiben (vergl. Java und C#),
- self und super,
- Teilmengenrelation für Objekttypen

Untertypen (Subtyping)

S ist ein **Untertyp (subtype)** von T, notiert als **S `<:` T**, wenn **ein Wert vom Typ S in jedem Kontext verwendet werden kann, wo ein Wert vom Typ T erwartet wird.** T heißt dann auch Obertyp von S.

Z.B. `ClrCellType <: CellType`

`<:` braucht **nicht für die Objekte jedes Paares von Unter- und Oberklasse** zu gelten, z.B. wenn die Unterklasse Methoden der Oberklasse löschen oder umbenennen kann (wie in Eiffel).

Hier sind **Untertypen anhand der Typeigenschaften definiert** (welche Methoden-Signaturen enthalten sie). Das ist ***structural subtyping***.
In den meisten OO-Sprachen wird die **Untertyprelation durch die Unterklassenrelation** festgelegt; diese wird durch die Klassen**namen** bestimmt.

In Sprachen mit **Subtyping** kann **ein Wert zu mehreren Typen gehören**, die in Untertyprelation stehen. Deshalb spricht man von ***subtype polymorphism***.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 134

Ziele:

Untertyprelation verstehen

in der Vorlesung:

Begriffe werden erläutert

- Subtyping ist verschieden von Subclassing
- Für jede Art von Typen wird die

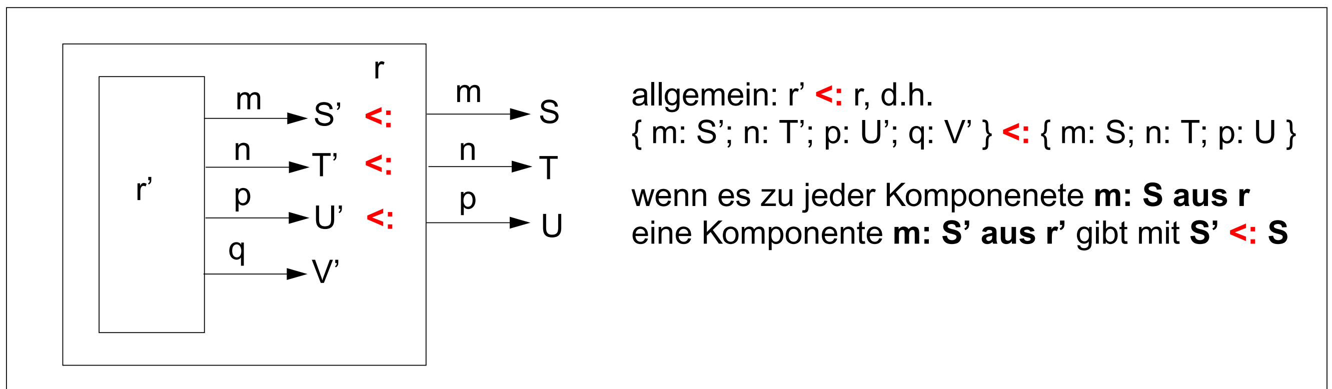
Untertypen von Record-Typen

Betrachten wir die **Typen von unveränderlichen Record-Werten**: Komponenten werden **nur gelesen**, d.h. Typen der Werte müssen **paarweise die <: Relation** erfüllen

Sei `CheeseType <: FoodType`, dann ist `CheeseSandwichType <: SandwichType`. Werte vom Typ `CheeseSandwichType` können die Rolle von Werten des Typs `SandwichType` spielen:

```
SandwichType = {         bread: BreadType;
                         filling: FoodType }

CheeseSandwichType = {   bread: BreadType;
                         filling: CheeseType;     ← tiefes
                         sauce: SauceType }       ← breites
                                                    Subtyping
```



allgemein: r' **<:** r, d.h.
{ m: S'; n: T'; p: U'; q: V' } **<:** { m: S; n: T; p: U }

wenn es zu jeder Komponenete **m: S aus r** eine Komponente **m: S' aus r'** gibt mit **S' <: S**

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 135

Ziele:

Untertyprelation für Record-Typen verstehen

in der Vorlesung:

Begriffe werden erläutert:

- breites und tiefes Subtyping;
- bei veränderlichen Komponenten: nur breites Subtyping (siehe Folie 138).

Untertypen von Funktionstypen

Betrachten wir die **Typen mit Signaturen P -> R (Parametertyp P und Resultattyp R)**.
Hat f die Signatur P -> R, dann kann g: P’ -> R’ die Rolle von f spielen, falls **P’ -> R’ <: P -> R**.
Die Parameterübergabe sei call-by-value.

Sei `CheeseType <: FoodType`, und `Integer <: Long`:

```
Integer -> CheeseType <: Integer -> FoodType
```

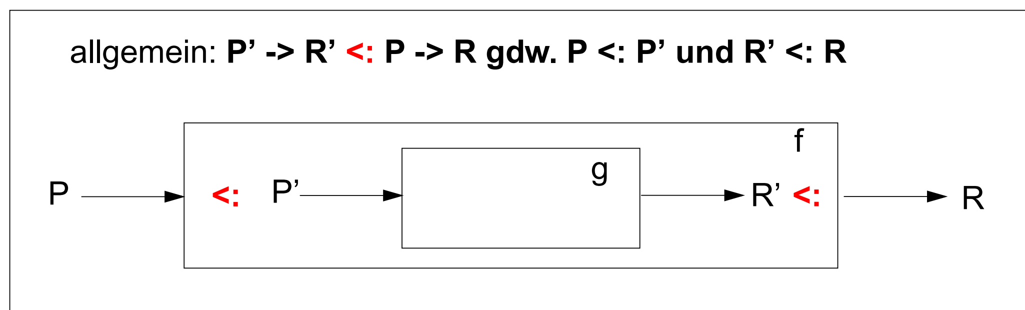
kovariante Ergebnistypen

speziellere Resultate der ersetzenden Funktion sind als allgemeinere Resultate akzeptabel

```
FoodType -> Integer <: CheeseType -> Integer
CheeseType -> Integer </: FoodType -> Integer
```

kontravariante Parametertypen

Die ersetzende Funktion muss **mindestens so allgemeine Parameter** akzeptieren wie die ersetzte.



© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 136

Ziele:

Untertyprelation für Funktionstypen verstehen

in der Vorlesung:

Herleitung für

- Parametertypen: contravariant,
- Resultattypen: covariant

Untertypen von Typen von Variablen

Unter welchen Umständen kann eine **Variable vom Typ T' die Rolle einer Variablen vom Typ T spielen**?

Mit einer Variable können **Lese- und Schreiboperationen** ausgeführt werden.

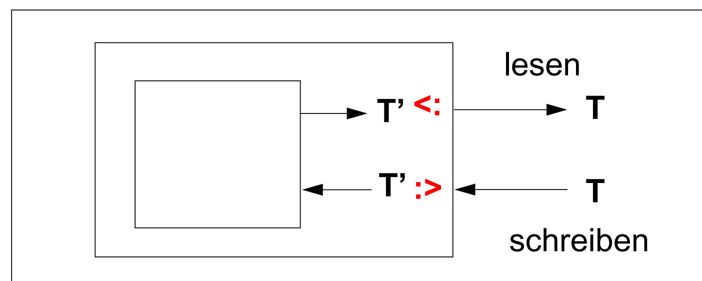
Sei **v'** vom Typ T' und **v** vom Typ T und T'<: T.

Für die **Leseoperation** kann v' die Rolle von v spielen: Jeder Wert aus v' ist auch ein Wert des Typs von v. Für das **Lesen wird Kovarianz** benötigt, T'<: T.

Für die **Schreiboperation** kann v' nicht die Rolle von v spielen: Nicht alle Werte des Typs von v können an v' zugewiesen werden. Es wird **Kontravarianz für das Schreiben** benötigt!

Insgesamt: T' **<:** T und T **<:** T', d.h. T' **~** T.

Es gibt **keine nicht-trivialen Untertypen von Variablen**. Sie können einander nur dann ersetzen, wenn ihre **Typen äquivalent** sind.



© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 137

Ziele:

Untertyprelation für Variablentypen verstehen

in der Vorlesung:

Herleitung für

- Lese- und Schreiboperationen,
- Konsequenz: äquivalente Typen statt Untertypen

Keine nicht-trivialen Untertypen von Variablen: Konsequenzen

Es gibt **keine nicht-trivialen Untertypen von Variablen**.
Sie können einander nur ersetzen, wenn ihre **Typen äquivalent** sind.

Untertyprelation für **Records mit veränderlichen Komponenten**:

Für jede Komponente **n: T des Obertyps** muss es eine Komponente **n: T' mit T ~ T' im Untertyp** geben, d.h. nur **breite nicht tiefe Untertypen**.

Untertyprelation für **Arrays mit unveränderlichen Elementen [Ind] of El:**

Ind ist der Indextyp, El der Elementtyp.
[Ind'] of El' <: [Ind] of El wenn **Ind <: Ind'** (kontravariant) und **El' <: El** (kovariant)

Untertyprelation für **Arrays mit veränderlichen Elementen [Ind] of El:**

[Ind'] of El' <: [Ind] of El wenn **Ind <: Ind'** (kontravariant) und **El' ~ El** (äquivalent)
Missachtung hat in Java eine Lücke in der statischen Typisierung verursacht, die durch dynamische Prüfung geschlossen wurde:

```
class Node { ... }
class Leaf extends Node { ... }
...
Node[] arr = new Leaf[42];

arr[0] = new Node();
```

Erlaubt in Java, obwohl nicht gilt
Leaf-Array <: Node-Array

statisch korrekte Zuweisung verursacht Laufzeitfehler: java.lang.ArrayStoreException: Node

© 2009 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 138

Ziele:

Konsequenzen aus Variablentypen erkennen

in der Vorlesung:

Herleitung für

- Record- und Array-Typen

Untertyprelation für Objekttypen: Präzisierung

- **Keine Änderung der Typen von Instanzvariablen** durch Überschreiben.
- **Zufügen von Methoden im Untertyp**: breite Untertyprelation wie bei Records.
- **Überschreiben von Methoden**:
  Signaturen brauchen nicht äquivalent zu sein (wie in Java):
  kontravariante Parametertypen und kovariante Resultattypen - wie bei Funktionen.
  (Eiffel erlaubt kovariante Parametertypen: Lücke in der Typsicherheit.)

```
    A a;   X x;   P p;                   C c; B b;

       a = x.m (p);
```

Übersetzer prüft Typen gegen Signatur von m in X und Methoden-Untertyp zwischen m in Y und m in X

```
class X { C m (Q q) { use of q;... return c; }

     v     v     ^

 class Y { B m (R r) { use of r;... return b; } }
```

© 2007 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 139

Ziele:

Untertyprelation für Objekte verstehen

in der Vorlesung:

- Herleitung aus Record-, Variablen- und Funktionstypen,
- kontravariante Parameter- und kovariante Resultattypen

Beispiel für Änderung des Resultattyps

Überschreiben der einer Methode **deepClone**, die einen Klon des Unterklassenobjektes bildet, benötigt **Änderung des Resultattyps**:

```
class CClass {
   var Cvar: XType;
   function deepClone (): CType is
   {  var newClone: CType := self<=clone (); -- shallow clone
      newClone<=setCVars ();
      return newClone;
   }
   function setCVars (): void is {self.Cvar<=deepClone();}
}

class SCClass inherits CClass modifies deepClone {
   var SCvar: YType;
   function deepClone (): SCType is
   {  var newClone: SCType := self<=clone (); -- shallow clone
      newClone<=setSCVars ();
      return newClone;
   }
   function setSCVars (): void is
      {  self<=setCVars (); self.SCvar<=deepClone(); }
}
```

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 140

Ziele:

Notwendigkeit der Änderung des Resultattyps erkennen

in der Vorlesung:

Das Beispiel wird erläutert:

- Der Resultattyp wird kovariant geändert.
- Daher hat ein Aufruf a.deepClone() den statischen Typ SCType, wenn a diesen Type hat.
- Die überschreibende Methode deepClone ruft Methoden auf, die für die Referenz-Variablen der Oberklasse und der eigenen Klasse deepClone aufrufen.

Typisierungsproblem: binäre Methode

Methode soll Operation mit Operanden gleichen Typs realisieren, z.B. equals.
Überschreiben in Unterklasse würde kovariante Änderung des Parametertyps erfordern:

```
class CClass {
   function equals (other: CType): Bool is
   { ... }
}

class SCClass inherits CClass modifies equals {
   function equals (other: CType): Bool is
   {  ... (SCType) other <= ... }
}
```

Wird teilweise durch parametrisierte Typen gelöst.

© 2010 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 141

Ziele:

Typisierungsproblem verstehen

in der Vorlesung:

- Beispiel wird erläutert.

1.3 Generik

Generik: zusätzliche Ebene der Parametrisierung zur Übersetzungszeit

Abgrenzung:

- **Funktionsaufruf**: Funktionen mit formalen Parametern werden mit aktuellen Parameterwerten zur **Laufzeit** aufgerufen.
- **Generik**: Aus **Schemata** für Definitionen von Programmkonstrukten (z. B. Klassen, Module, Funktionen) werden zur **Übersetzungszeit** Definitionen erzeugt.
  Ein Schema hat **formale generische Parameter** für z. B. Typen, Klassen, Funktionen.
  Dafür werden bei der **Instanziierung aktuelle generische Parameter** eingesetzt.
- **Makro-Substitution**: Makros sind **Symbolfolgen** mit formalen Parametern. An den Anwendungsstellen werden sie nach **Substitution** der aktuellen Parameter (auch Symbolfolgen) eingesetzt.
  Substitution durch den **Präprozessor** ohne Beachtung der syntaktischen Struktur und der statischen Semantik.

```
int ggt (int a, int b)
   { ...}

ggt (m, n)
```

Java 1.5:

```
   class Stack <E>
   { void push (E e)
        {...}
      ...
   }

Stack<Kreis> kreisStk;
```

```
#define PSEL(var,fld) \
   ((var)->fld)

PSEL (a[i], next)
```

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 151

Ziele:

Einordnung des Begriffes Generik

in der Vorlesung:

- Erläuterung der Abgrenzung an Beispielen.
- Grundbegriffe zur Generik.

Zweck der Generik

Abstraktion durch Programmschemata, z. B.

- Kellerimplementierung mit Elementtyp als generischem Parameter
- Sortierfunktion mit Elementtyp und Vergleichsfunktion als generische Parameter

Ermöglicht **Parametrisierung** mit Programmobjekten, die **nicht als Laufzeitdaten** existieren, nicht „first class“ sind; je nach Sprache: Typen, Klassen, Module, Funktionen.

Ergebnis der Instanziierung wird auf **Einhaltung der statischen Sprachregeln** überprüft, insbes. Typregeln;
z. B. Übersetzer garantiert homogene Behälter

Konsistenzbedingungen für aktuelle generische Parameter sind formulierbar und prüfbar, z. B. generische Sortierfunktion: Elementtyp mit passender Vergleichsfunktion

Insgesamt:
zusätzliche **Abstraktionsebene**
zusätzliche **Programmsicherheit** durch schärfere Regeln, schärfere statische Prüfungen

In Sprachen, die kaum statische Regeln haben, ist Generik nicht sinnvoll, z. B. Smalltalk.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 152

Ziele:

Abstraktion und Sicherheit

in der Vorlesung:

Erläuterungen dazu

Generische Definitionen (seit Java 1.5)

Eine **Generische Definition** hat **formale generische Parameter.** Sie ist eine **abstrakte Definition einer Klasse** oder eines Interfaces. Für jeden generischen Parameter kann ein **Typ (Klasse oder Interfce) eingesetzt** werden.
(Er kann auf Untertypen eines angegebenen Typs eingeschränkt werden.)

Beispiel in Java:
Generische Definition einer Klasse `Stack` mit generischem Parameter für den Elementtyp

```
class Stack<Elem>
{  private Elem [] store ;
   void push (Elem el) {... store[top]= el;...}
   ...
};
```

Eine **generische Definition wird instanziiert** durch Einsetzen von **aktuellen generischen Parametern**. Dadurch entsteht zur Übersetzungszeit eine Klassendefinition. Z. B.

```
Stack<Float> taschenRechner = new Stack<Float>();
Stack<Frame> windowMgr = new Stack<Frame>();
```

Generische Instanziierung kann im Prinzip durch **Textersetzung** erklärt werden: Kopieren der generischen Definition mit Einsetzen der generischen Parameter im Programmtext.

Der Java-Übersetzer erzeugt für jede generische Definition eine Klasse im ByteCode, in der `Object` für die generischen Typparameter verwendet wird. Er setzt Laufzeitprüfungen ein, um zu prüfen, dass die ursprünglich generischen Typen korrekt verwendet wurden.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 153

Ziele:

Generische Typen aus GPS wiederholen

in der Vorlesung:

- Beispiel wird erläutert.

Parametrisiertes Interface für binäre Methoden

Ein formaler **Typparameter im Interface** bestimmt den Typ des zweiten Operanden.
Die implementierende Klasse bindet den **eigenen Typ an den Parameter**.

```
interface Comparable <T> {
   function equals (other: T): Bool;
}


class CClass implements Comparable<CType> {
   function equals (other: CType): Bool is
   { ... }
}

class SCClass implements Comparable<SCType> {
   function equals (other: SCType): Bool is
   {  ... other <= ... }
}
```

In diesem Modell kann leider nicht `SCClass` Unterklasse von `CClass` sein und `equals` überschreiben.

© 2010 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 154

Ziele:

Nutzen generisch parametrisierter Interfaces erkennen

in der Vorlesung:

- Beispiel wird erläutert.

Generik in C++

Generische Programmschemata heißen in C++ **Templates**.

Templates für Klassen, deren separat definierte **Methoden**, für eigenständige **Funktionen**.

Generische Parameter können sein:
Klassen, Typen, Templates, Funktionen, Variable, Zahlkonstante
auch Default-Werte für formale generische Parameter

Generische Parameter **nur ohne Restriktionen**:

```
template<class Elem>
class Stack
{  ...
   void push (Elem e);
   ...
}
```

```
template<class Elem>
Stack::push (Elem e)
   {...}
```

Funktion als generischer Parameter:
Klasse:

```
template<class T, void(*err_fct)()>
class List {...}
```

Funktion:

```
template<class T>
void sort(Array<T>& a) {...}
```

Instanziierungen:

```
Stack<int> s1;

List<Book, error_handl> lb;

Array<Book> ab;
sort(ab);
```

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 155

Ziele:

C++ Templates kennenlernen

in der Vorlesung:

Erläuterungen zu den Beispielen

Generik in Eiffel

Eiffel hat ein **einheitliches Typkonzept**: Alle Typen sind durch Klassen definiert.

Generik: Klassendeklarationen können **Typen als generische Parameter** haben.
Instanziierung einer generischen Klasse liefert einen Typ.

Generischer Parameter **ohne Restriktionen**, z. B. für Behälterklassen:

```
class Stack [ELEM] feature
   ...
   push (e: ELEM) is ...
   ...
end -- class Stack
```

Instanziierung als Typangabe:

```
si:  Stack[INTEGER];
d:   Dictionary[Book, STRING];
```

Generische Parameter **mit Restriktionen**:

```
class Dictionary [G, KEY->Hashable]
  feature ...
end;
```

Der generische Parameter KEY ist auf Unterklassen von Hashable eingeschränkt. In der Klasse Dictionary können Hashable-Methoden für KEY-Objekte benutzt werden.

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 156

Ziele:

Generik in Eiffel kennenlernen

in der Vorlesung:

Erläuterungen

- zum einheitlichen Typkonzept; keine Sonderrolle für Grundtypen,
- zu Restriktionen generischer Parameter

Generik in Ada 95

Ada 95 hat ein **umfangreiches Typkonzept**;
Ableitung von Typen ist mit Vererbung zwischen Klassen vergleichbar;
weitere OO-Konzepte begründen den Anspruch OO-Sprache zu sein.

Generisch deklarierbar: **Funktionen, Prozeduren und Package**s (Module).

Generische Parameter können sein
Typen, auch mit vielfältigen Restriktionen, z. B. Obertyp oder Array-Typ,
Funktionen, Variable, Konstante

Beispiel:

```
generic Type T is private
procedure Swap (X, Y: in out T);

procedure Swap (X, Y: in out T) is
   tmp: constant T := X;
begin ... end;
```

Instanziierungen:

```
procedure Swap_Int is new Swap (Integer);

procedure Swap_Vect is new Swap (Vect);
```

© 2013 bei Prof. Dr. Uwe Kastens

Vorlesung Objektorientierte Programmierung WS 2013/2014 / Folie 157

Ziele:

Generik in Ada kennenlernen

in der Vorlesung:

Erläuterungen zu dem Beispiel