Funktionale Dekomposition

Vorgehensweise (Verfeinerungsentwurf):

1. Abstrakte Spezifikation der Funktion, die das Softwaresystem realisieren soll, d.h. möglichst genaue Beschreibung, welche Ausgaben auf welche Eingaben erfolgen sollen.

2. Hierarchisch absteigende Zerlegung der Funktion in immer elementarere Teilfunktionen, die jeweils von höheren Funktionen aufgerufen werden und selbst kleinere Teilfunktionen aufrufen.

3. Ausprogrammieren der Funktionen

Beispiel: Taschenrechner in UPN

UPN steht für Umgekehrte Polnische Notation.

Beispiel: (1+2)*(3+4)+5*6 => 1 2 + 3 4 + * 5 6 * +

Verwenden wir der Einfachheit halber, da Syntaxanalyse auf Bindungszahlen von Operatoren verzichten kann.

Abstrakte Spezifikation der Funktion (Stufe 1):

Berechnung des Wertes wohlgeformter arithmetischer Ausdrücke mit Operanden aus dem Bereich `float` in UPN.

Grobe Beschreibung einer Realisierung (Stufe 2a):

1. Eingabe in Token zerlegen
2. Token analysieren
   2.1. Ist Token eine Zahl? => speichern
   2.2. Ist Token ein Operator?
      2.2.1. Argumente holen
      2.2.2. Operation ausführen
      2.2.3. Ergebnis speichern
3. Ergebnis ausgeben.

Besonderheiten bei der weiteren Verfeinerung des Entwurfs (Stufe 2b, 2c, ...):

Realisierung des Speichers für die Zwischenergebnisse als Stapelspeicher mit Funktionen `push()` und `pop()`.

Zeichenweise Analyse der Token: Ist Token eine Zahl, so wird das Tokenende erst *nach* dem letzten zugehörigen Zeichen erkannt. Deshalb wird ein Zwischenpuffer für Zeichen benötigt.

Ausprogrammieren des Beispiels (Stufe 3), siehe `upn.c`

Nachteile der funktionalen Dekomposition:

Durch Konzentration auf Funktionalität wird Zustand der zu modellierenden Objekte der Umwelt nur ungenügend erfasst.

Strukturiert Programme vorrangig nach funktionalen Gesichtspunkten. Strukturelle Aspekte werden nicht ausreichend berücksichtigt (Nachteil, der sich besonders bei hoher Problem-Komplexität auswirkt)

Objektorientierte Modellierung

Prinzip: Der funktionalen Implementierung geht ein sorgsamer Entwurf der **Datenarchitektur** voraus. Oft programmieren sich dann Funktionen „von selbst“.

Phasen des objektorientierten Ansatzes zur Software-Entwicklung

1. Analyse: Untersuchung der realen Welt auf die Existenz von (relevanten) Objekten und Objektbeziehungen (externe Modellierung)

Fixierung der Art der späteren Nutzung des Softwaresystems

Frage: **Was** soll mit den Objekten **warum** geschehen ?

2. Entwurf: Das OO-Modell der realen Welt wird in die Welt der Software übertragen. Es entsteht ein Modell der Softwarearchitektur (interne Modellierung)

Frage: **Wie** soll etwas **im Prinzip** geschehen ?

3. Implementierung: Die Software-Architektur wird zum lauffähigen Programm konkretisiert.

Objektzustände	=>	Datenstrukturen
Objektfunktionalität	=>	algorithmische Realisierung

Frage: **Wie** soll alles **im Einzelnen** geschehen

Geschichte der objektorientierten Modellierung

Krisen der Informatik:

ca. 1970: Softwarekrise. Antwort: strukturiertes, ingenieurmäßiges Programmieren

ca. 1990: Wiederverwendungskrise. Antwort: Objektorientiertes Programmieren

ca. 2010 (geplant): Parallelitätskrise.

Antwort auf die zweite Krise: Probleme werden nicht mehr dadurch gelöst, dass man die Lösung von Grund auf neu entwickelt, sondern man versucht, aufbauend auf einem Grobkonzept, Klassen zu finden, deren Zusammenwirken möglicherweise die Fragestellung lösen kann.

Ausgangspunkt: Programmiersprache SMALLTALK-80 (1970 - 1980 entwickelt am Palo Alto Research Center der Firma Xerox)

ca. 1989-1994 Forschungsarbeiten zu objektorientierten Modellierungsansätzen (in textueller Repräsentation)

ab 1994/95 Entwicklung der UML durch Booch, Rumbaugh und Jacobson (Firma Rational Software)

seit 1995 Unterstützung durch die Object Management Group (OMG) - Entwicklung eines Standards

OMG umfasst heute über 800 Firmen weltweit.

Grundbegriffe der OOP

Objekt

gedankliche oder reale Einheit in der Umwelt (externes Objekt) oder in der Software (internes Objekt)

gegeben durch **Zustand** und **Funktionalität** (Verhalten)

Name
Zustand
Funktionalität

Zustand: charakterisiert durch Attributwerte

Verhalten: charakterisiert durch die vom Objekt ausführbaren Operationen (Methoden)

Jedes Objekt besitzt eine unveränderliche Identität, das es von allen anderen Objekten unterscheidet. Es kann jedoch mehrere Referenzen auf dasselbe Objekt geben.

Zugriff auf Attributwerte gewöhnlich nicht direkt, sondern über **Selektoren** `getAttribut()` (lesend) oder `setAttribut()` (schreibend)

Beispiel: Fernsehapparat

TV2000
EinAus z; Seriennummer s; Kanal k; Lautstärke l;

ein();
aus();
wähleKanal();
wähleLautstärke();
getSeriennummer();

Beispiel: Motor

Motor
EinAus z; Vorwärtslauf v; Rückwärtslauf r; Drehzahl d;

ein();
aus();
vorwärts();
rückwärts();
setDrehzahl();
getDrehzahl();

Klassen

Objekte „gleicher Bauart“ werden zu Klassen zusammengefasst. Eine Objektklasse T repräsentiert den **Typ** des Objekts.

Bündelt Eigenschaften (**Attribute**) und Operationen (**Methodenschnittstelle**) dieser Objekte zu einem **abstrakten Datentyp**. Verallgemeinert den Begriff des Datentyps aus dem ersten Semester.

Grundbereich $M(T)$ einer Klasse = Menge aller denkbaren Objekte.

Klasse = **Verbund** von Attributen und Methoden.

Klassisch: Methoden leben in einem objektunabhängigen Namensraum.

OO: Objekte kennen nur die Methoden der eigenen Klasse. Zugriff auf Methoden nur über Objektnamen (Objektmethoden) oder Klassennamen (Klassenmethoden).

Attribute und Operationen einer Klasse sind **genauer spezifiziert**.

Jedes Attribut hat einen **Typ**, der den Wertebereich und die Methoden des Attributs festlegt.

Jede Operation hat eine **Signatur**, die deren Aufrufsyntax festlegt.

Jede Klasse besitzt Methoden, um Objekte zu erzeugen (**Konstruktoren**). Klassen sind in diesem Sinne Objekt**schablonen**.

Attribute und Operationen einer Klasse sind in der zugehörigen **Dokumentation** beschrieben. Java erlaubt eine quelltextnahe Dokumentation, die mit `javadoc` extrahiert werden kann.

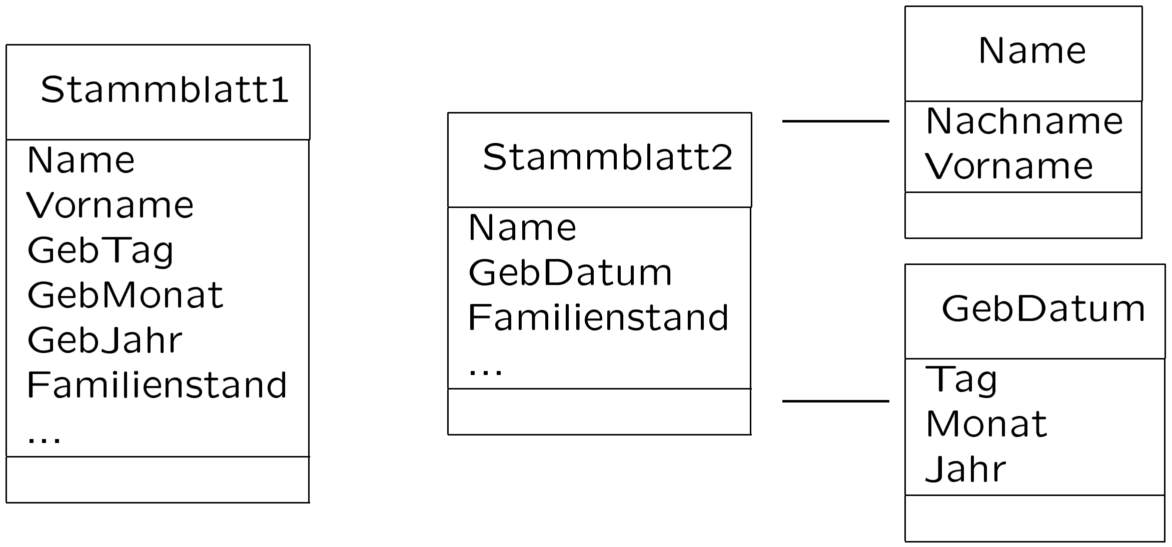
Objektbeziehungen

Die Kunst der OOP besteht in der richtigen Indentifizierung der Klassen und deren interner Struktur, zu denen (reale) Objekte zusammengefasst werden.

Klassisch werden solche Beziehungen über Verbunde realisiert. Verbunde dienen zur Gruppierung von Daten (Attributwerten), die in einem engeren semantischen Zusammenhang stehen.

Der OO-Zugang ist wesentlich mächtiger, da er neben den Daten auch die entsprechenden Funktionen bündelt.

Beispiel Stammblatt:

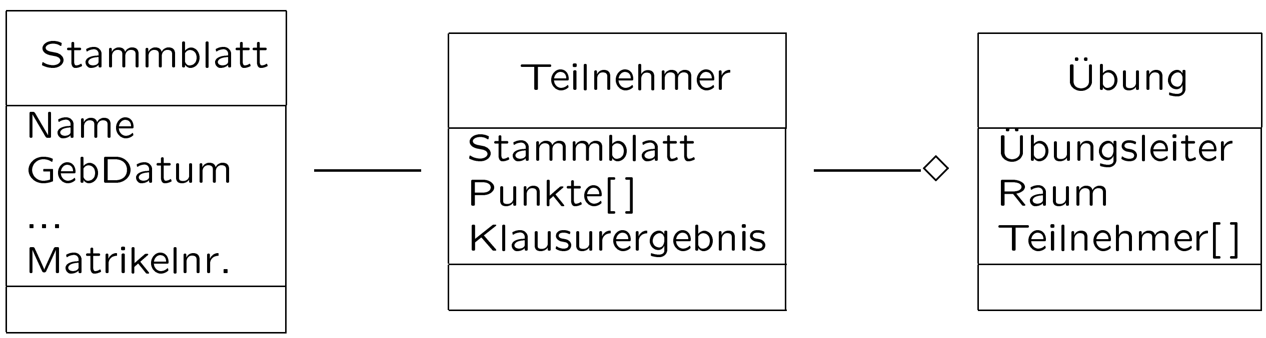


Zugriff

s1.Name="Meier"; s2.Name.Nachname="Meier";

Enthaltenseins durch Wert und durch Referenz

Beispiel: Aus der Modellierung des Übungsbetriebs am Institut



Generell sollte immer nach der Realität modelliert werden.

Java kennt nur den Zugriff auf Objekte als Attributwerte über Referenzen. Kopien müssen von referenzierten Objekten explizit mit `new()` angelegt werden.

Der Zugriff auf Objektmethoden erfolgt als Referenz über die zugehörige Klasse, da dort die (für alle Objekte der Klasse gleichen) Methodendefinitionen lokalisiert sind.

Die wichtigsten Objektbeziehungen

Assoziation (Benutzt-Beziehung), allgemeine Beziehung zwischen Objekten

Aggregation (Einschluss-Beziehung, has-a), Zusammensetzung aus Teilobjekten mit eigenständiger Existenz

Komposition, spezielle Aggregation, bei dem die Einzelteile das Ganze konstituieren

rekursive Assoziation, Beziehungen zwischen verschiedenen Objekten derselben Klasse

mehrgliedrige Assoziation, Beziehungen zwischen mehreren Klassen

Vererbung (Subtyp-Beziehung, is-a)

Objektbeziehungen werden eingeteilt in

- strukturelle Beziehungen
- verhaltensbezogene Beziehungen

Wichtige spezielle Beziehungsarten

1. Informationsfluss oder Nachrichten
2. Funktionsaufruf oder Kunde/Lieferant (client/server)
3. Einschluss
4. Subtyp oder Vererbung

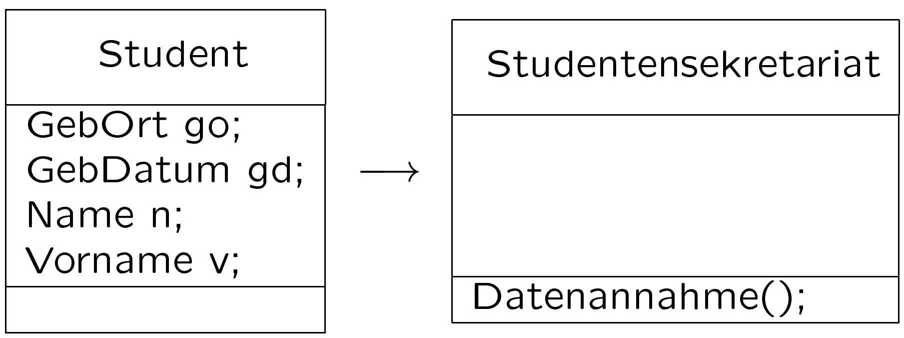
Informationsfluss und Client/Server-Beziehungen

Versenden einer **Nachricht** als einfachste Interaktionsform

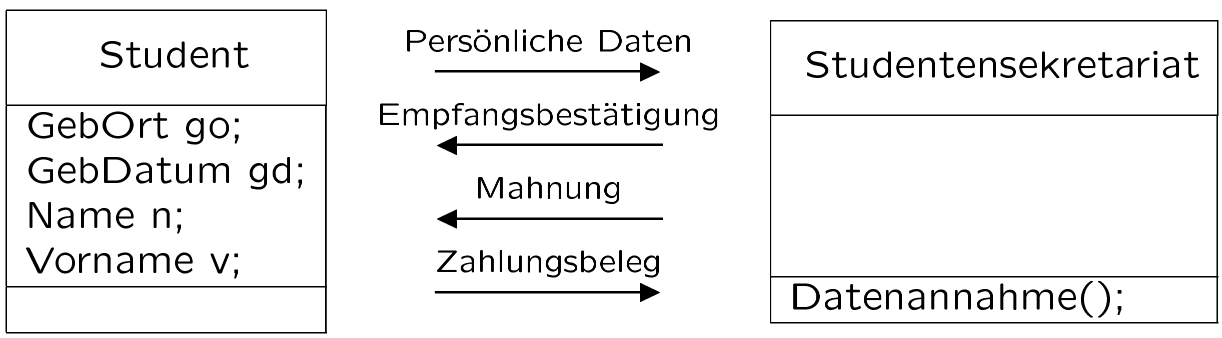
Notwendig ist eine Beziehung zwischen Sender und Empfänger mit einem Verbindungsdienst

asynchrone Kommunikationsform

Beispiel: Student schickt Studentensekretariat die Personaldaten



Beispiel: Interaktionsdiagramm einer komplexen Interaktion „Anmeldung mit Mahnung" zwischen (den Klassen) Student und Studentensekretariat



Client/Server-Beziehung

Kunde (Client) fordert Dienstleistung an.

Dienstleister (Server) erbringt die Dienstleistung.

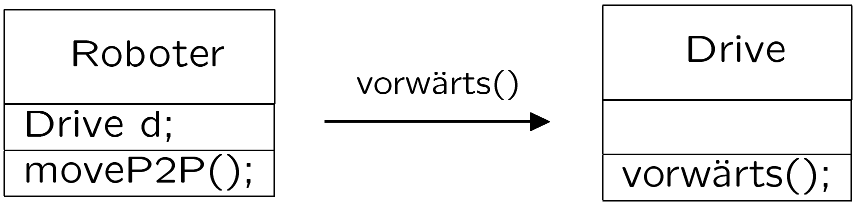
Dienstleistungen des Servers sind aufrufbare Methoden.

Kunde ruft die Methode des Servers (mit den erforderlichen Parametern) auf.

synchrone Kommunikationsform

Methodenaufruf

Erledigung von Aufgaben durch die Delegierung von Teilaufgaben an andere Objekte.



Client (Roboter) schickt eine **Botschaft** an den Server (Drive), die dort die **Operation** gleichen Namens auslöst und den Rückgabewert als **Antwort** zurückschickt.

Realisierung als Methodenaufruf in Java:

```
{ ... d.vorwärts(); ... }
```


Einschlussbeziehungen

Realisierung als Verbund in Java wie in C:

```
class Auto {
  Rad vr, vl, hr, hl;
  Motor m; ...

  m.ein();
  ...
}
```

Jede Einschlussbeziehung ermöglicht im Allgemeinen eine Informationsflussbeziehung.

Deshalb typischer Methodenaufruf bei Aggregationen:

Methoden zur Zusammenarbeit mit anderen („fremden“) Objekten vom *Gesamt*objekt zur Verfügung gestellt

Aufruf der Obermethode wird von dieser an geeignete Methoden der Teilobjekte weitergereicht.

Rückgabewerte werden von der Obermethode weiter verarbeitet.

Beispiel: Der Wert eines Warenkorbs berechnet sich als die Summe der Einzelpreise der darin befindlichen Waren.

Subtyp- bzw. Vererbungsbeziehung

Bezeichnung: Unterklasse / Oberklasse
abgeleitete Klasse / Basisklasse

Unterklasse ist **Spezialisierung** der Oberklasse

Objekte abgeleiteter Klassen sind zugleich Instanzen der Basisklasse

Unterklasse **erbt** die Attribute und Methoden der Oberklasse, kann aber weitere Attribute und Methoden besitzen.

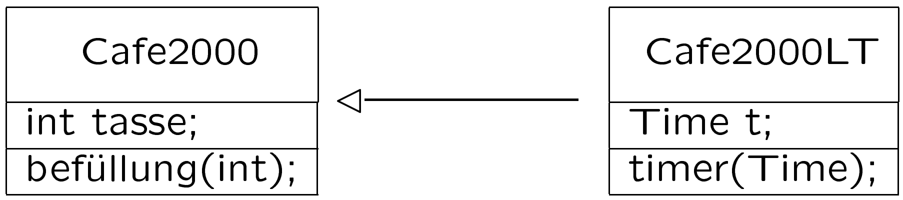
Oberklasse ist **Generalisierung** der Unterklassen

Typischer Ansatz, um gemeinsame Eigenschaften und Methoden nahe verwandter Klassen kohärent zu verwalten und Replikationen zu vermeiden.

Gutes Design von Oberklassen ist zentrales Mittel der OOP, um Wiederverwendbarkeit von Code zu ermöglichen.

Im Java Runtime Environment steht eine ganze Hierarchie von Klassen zur Verfügung, die Grundfunktionalitäten implementieren und von denen Nutzerklassen gewöhnlich erben.

Beispiel:



Realisierung in Java:

```
class Cafe2000LT extend Cafe2000 {
  Time t;
  void timer(Time u) { ... }
  ...
}
```

2. Java — Eine Einführung

2.1. Entwicklung von Java

Die Java-Vorgeschichte

Anfang 90er Jahre: neben großen monolithischen Programmen gewinnen kleine verteilte Programmkomponenten an Bedeutung

- 1991-92: Sun startet Green-Projekt (J. Gosling, Patrick Naughton)

  Ziel: Entwicklung von Hard-/Software für Konsumelektronik
  => Interpreter OAK
- 1992 Firmengründung zur Vermarktung des Prototyps Star Seven (*7) erfolglos
- 1994 Webbrowser HotJava (Basis: OAK), Integration in Netscape Navigator
- 1996 Umbenennung von OAK in Java, JDK 1.0, Gründung von JavaSoft

Webbrowser stellen das klassische Anwendungsfeld für verteilte Rechnungen dar, da jede Webverbindung wenigstens zwei Rechner betrifft.

Klassisch (HTML): Clientseite kann Inhalte nur nach vorgegebenen Verfahren rendern. Browser = Rendersoftware

neu: Intelligente Clients mit eigener dynamisch nachladbarer Funktionalität (Einbeziehung **aktiver Objekte**)

Erfordernis: Plattformunabhängigkeit (Java) oder homogene Welt der Betriebssysteme (Microsoft)

Technologischer Ansatz (Das JVM-Konzept)

- Spezifikation einer (höheren) **Programmiersprache** (Java)
- Spezifikation der Java-**Klassenbibliothek** mit Standardschnittstellen zu allen wichtigen Programmierfunktionalitäten.
- Spezifikation einer plattformunabhängigen **Zwischensprache** (Java-Bytecode), in die Java-Programme übersetzt werden (Java Virtual Machine – JVM)
- Interpretation des Bytecode in einer (plattformabhängigen) **Java-Laufzeitumgebung** (Java Runtime Environment – JRE)

Vorteile

- Portabilität
- Unterstützung von Code-Mobilität (Applets)
- Unterstützung von Nebenläufigkeit

Weiter

Suns neue Ausrichtung, Java nicht als Produkt, sondern als plattformübergreifende Infrastruktur zu positionieren.

seit 1998: Rasantes Wachstum des Interesses, JRE macht Klassenbibliotheken auf allen gängigen Plattformen verfügbar

1998: JDK 1.2, Umbenennung im Dez. 98 in Java 2 SE, um besser zwischen der (von SUN freizügig lizensierten) Technologie und der Entwicklungsumgebung SDK zu unterscheiden

2000: JDK 1.3 SE (Kestrel-Projekt) mit deutlichen Geschwindigkeitsgewinnen

2002: JDK 1.4 SE, erste Arbeiten an einer Enterprise Edition (mit zusätzlicher serverbasierter Funktionalität)

Unterscheide bei Java also zwischen

Java 2 Plattform SE (J2SE)
: die abstrakte Plattform, wie sie in der Sprachbeschreibung spezifiziert ist

Java 2 SDK SE (J2SDK)
: Suns Software Development Kit, mit dem Applikationen für die J2SE erstellt werden können

Java Virtual Machine (JVM)
: Spezifikation der Bytecode-Zwischensprache

Java 2 Laufzeitumgebung (JRE)
: Suns Laufzeitumgebung, in der Applikationen abgearbeitet werden können

Die wichtigsten Konzepte von Java

- einfacher Sprachaufbau, der an C angelehnt ist und die wichtigsten Konzepte der OOP verwirklicht: Klassen, Methoden, Vererbung
- Plattformunabhängigkeit durch JVM (Java Virtual Machine)
- keine Zeiger, automatische Speicherverwaltung
- Robustheit (Garbage Collection, Ausnahmebehandlung)
- Multithreading erlaubt nebenläufige Programmierung
- flexibles Sicherheitsmodell
- Syntax der Kontrollstrukturen ist an C angelehnt
- umfangreiche Bibliothek vordefinierter Klassen

2.2. Werkzeuge, Quellen, Webadressen, Dokumentation

Java-Quellen

Die aktuelle Version 1.4 der Java 2 Standard Edition (J2SE^{TM}) kann sowohl für Windows als auch für Linux von Suns Java-Seiten unmittelbar kostenfrei heruntergeladen werden.

Adresse: `http://java.sun.com/j2se/1.4/download.html`

Die Distribution enthält die wichtigsten Java-Werkzeuge in Kommandozeilenversion. Dazu ist das entsprechende `bin`-Verzeichnis in den aktuellen Pfad aufzunehmen.

Unter Windows können Sie **cygwin** installieren, womit Ihnen eine bequeme Linux-artige Kommandozeile sowie alle wichtigen GNU-Werkzeuge zur Verfügung stehen.

Adresse: `http://www.cygwin.com`

Java-Werkzeuge

`javac`	Java-Compiler
`java`	Java-Interpreter
`appletviewer`	Interpreter für Java-Applets (*.html)
`jdb`	Java-Debugger
`javadoc`	Dokumentationsgenerierung
`jar`	Verwaltung von Java-Archiven (jar-Dateien)
`javap`	Deassemblierer für Java-Klassendateien

Java-Dokumentation

Die Spezifikation der Java2-API-Schnittstelle kann ebenfalls von Suns Java-Seiten heruntergeladen werden.

Die Spezifikation liegt als Baum im HTML-Format vor und kann mit geeigneten Browsern (Netscape, MSIE) angezeigt werden. Die Quellen enthalten die Spezifikation aller Klassen, geordnet nach Paketen.

Weitere Quellen

- Aktuelle Buchliste auf `http://java.sun.com/docs/books`
- The Java Tutorial, `http://java.sun.com/docs/books/tutorial`
- JUG – die deutsche Java User Group `http://www.java.de`
- diverse Bücher auch in deutscher Sprache, auch online

Siehe auch die Webseiten zur Vorlesung.

2.3. Grundlegender Aufbau eines Java-Programms

Java-Programm = Summe von Klassendefinitionen

keine Koexistenz von prozeduralen und objektorientierten Konzepten (wie in C++)

Hello World

Beispiele: HelloWorld0, HelloWorld,

Übersetzen mit `javac HelloWorld.java`

Aufruf mit `java HelloWorld Fritz`

Verwendung der Java 2 API Spezifikation als Dokumentationsquelle

Vererbung

```
class Boot {
  int Kurs;

  void setzeKurs(int neuerKurs) {
    Kurs=neuerKurs;
  }
}

class Segelboot extends Boot {
  int SegelFläche;

  void setzeSegel(int neueSegelFläche) {
    SegelFläche=neueSegelFläche;
  }
}

class Motorboot extends Boot {
  int Drehzahl;

  void erhöheDrehzahl(int mehrGas) {
    Drehzahl+=mehrGas;
  }
}
```


Applikationen und Applets

Javaprogramme können (u.a.) als Applikationen (Text-Inferface) oder als Applets (grafisches Interface) aufgerufen werden.

Applikationen haben eine einfache Struktur, da nach dem Programmstart der Kontrollfluss vollständig an das Betriebssystem übergeht.

Unsere Konvention für Applikationen:

Beispiel: Beispiel1

Applets haben eine komplexere Struktur, da der Kontrollfluss ereignisgesteuert ist und durch Interaktionen mit dem Nutzer beeinflusst werden kann. Dazu ist eine umfangreichere Klassenhierarchie in Bewegung zu setzen.

Beispiel: Applet1-Demo   Aufruf `appletviewer Applet1.html`

Beispiel: AppletUmgebung, Applet1-Code

Übungsaufgaben und das PPS-Applet-Projekt

Beispiel: Beispiel2, Applet2

Dualität prozeduraler und objektorientierter Programmierung

prozedural

Funktionen aktiv, Daten passiv
Daten als Parameter von Funktionen
globaler Raum von Funktionen

=> **zentralistisch organisierte Datenlandschaft**

objektorientiert

Daten aktiv, Methoden passiv
Funktionen als Botschaften von Daten
Methoden sind Klassen zugeordnet

=> **marktförmig organisierte Datenlandschaft**

Vergleich der Interaktionsmodelle

	prozedural	objektorientiert
1)	globale Variablen	Objekte
2)	Funktionsparameter	Methodenparameter
3)	lokale Variablen	lokale Variablen

1) gültig über den gesamten Lebenszyklus (langlebig)
2) gültig nur während des Aufrufs (kurzlebig)
3) gültig nur im jeweiligen Block (kurzlebig)

Klassen- und Instanzvariablen und -methoden

OO-Ansatz: Privatisierung des Gemeineigentums an Funktionen.
Genossenschaftliches Eigentum = Klassenmethoden
Privateigentum = Instanzmethoden

Der Modifikator **static** kennzeichnet Klassenvariablen und Klassenmethoden.

Beispiel: `System.out.println()`

```
class System {
   static PrintStream out;
   ...
}

class Printstream {
   public void println();
   ...
}
```

`static` deklarierte Attribute werden nur einmal pro Klasse angelegt (Klassenvariablen), andere Attribute für jedes Objekt gesondert (Instanzvariablen)

Beispiel: Beispiel2

Es gibt Klassen, die **nur** Klassenmethoden enthalten. Wirkung wie Namensräume in C++

Beispiel: `Math`, Wrapperklassen `Integer`, `Float` etc. für primitive Datentypen

Weitere Konzepte

- Geheimnisprinzip (Sichtbarkeit kann eingestellt werden)
- Polymorphismus (Methoden werden nach Namen **und** Signatur bestimmt)
- abstrakte Klassen und Schnittstellen
- Möglichkeit, Klassen zu Paketen zusammenzufassen
- Klassen können zur Laufzeit dynamisch geladen werden

Merkmale, die in Java nicht vorhanden sind

- keine Mehrfachvererbung
- striktes Klassenkonzept, keine prozeduralen Elemente
- keine Möglichkeit, Operatoren zu überladen
- keine Zeiger, nur Referenzdatentypen, keine Bitfelder (Robustheit, Sicherheit)
- keine Verbunde, Aufzählungstypen, Typdefinitionen, nur Klassen (überdeckt diese Funktionalitäten)
- kein `goto` (verwende `break` und `continue`)
- kein Präprozessor

2.4. Lexikalische Elemente

Zeichensatz

UNICODE (2 Byte) statt ASCII (1 Byte)

- Kompatibel zu ASCII-Zeichensatz (unteres Byte)
- Umlaute und Sonderzeichen können auch in Bezeichnern verwendet werden

Bezeichner

- sollten mit Buchstaben, dürfen auch mit '_' oder '$' beginnen
- keine Schlüsselwörter zulässig
- Konventionen:
  - Klassennamen groß: `EineKlasse`
  - Variablen-, Attribut- und Methodennamen klein: `eineMethode`
  - Konstantennamen vollständig groß: `LIMIT`

Kommentare

Wie in C bzw. C++

```
// Einzeiliger Kommentar bis Zeilenende

/* Mehrzeilige Kommentare, dürfen nicht
      geschachtelt werden */
```


Dokumentationskommentare für Bearbeitung mit `javadoc`

```
/**
* beschreiben Methoden verbal und kennzeichnen
* Parameter und Rückgabewerte
* <i>HTML</i> ist zulässig
*/
```

Beispiel: HelloWorldmitDoku.java

Aufruf:

```
javadoc -d Doku -author -private HelloWorldmitDoku.java
```

2.5. Operatoren und Kontrollstrukturen

weitgehend wie in C, ebenso

- Gültigkeitsbereiche lokaler Variablen (jedoch Überdeckung lokaler Variablen nicht erlaubt)
- Aufrufmechanismus und Parameterübergabe an Funktionen (Methoden) als call by value

aber

- keine Null-Konvention für boolesche Ausdrücke. Schleifentests müssen explizit zum Typ `boolean` auswerten.
- automatisches Dereferenzieren durch den Komponentenzugriffsoperator: `a.feld` ist die Komponente `feld` des von `a` referenzierten Objekts.

Beispiel: Quersumme

```
int Quersumme(int n) {
  int s=0;
  while(n>0) { s+=n%10; n/=10; }
  return s;
}
```

Beispiel: Quadratwurzel mit Newtonapproximation

```
double sqrt(double y, double epsilon) {
  double x_old,x_new;
  x_new=y;
  do {
    x_old=x_new;
    x_new=(x_old+y/x_old)/2;
  } while (Math.abs(x_new-x_old)>epsilon);
  return x_new;
}
```

Realisierung: `Beispiel3.java`

2.6. Datentypen

Java kennt **einfache (primitive) Datentypen**

```
boolean char byte short int long float double
```

mit eingebauter Funktionalität (statische Speicherallokation)

und **Referenzdatentypen** (Objekte und Arrays), die zur Laufzeit mit `new` generiert werden (dynamische Speicherallokation)

Primitive Datentypen

Aus Portabilitätsgründen sind die primitiven Datentypen nicht maschinenabhängig wie bei C. Sie sind mit einem Defaultwert vorinitialisiert. Lokale Variablen müssen **immer** initalisiert werden.

Datentyp	Beschreibung	Literale	Default
boolean	Boolescher Wert	`true, false`	`false`
char	16-bit Unicode-Zeichen	`'a'`	`'\u0000'`
byte	8-bit Integer	`122`	0
short	16-bit Integer	`122`	0
int	32-bit Integer	`122`	0
long	64-bit Integer	`122L`	0L
float	32-bit Real	`1.22F`	0.0F
double	64-bit Real	`1.22`	0.0

Integer = vorzeichenbehaftete ganze Zahl im Zweierkomplement

Real = vorzeichenbehaftete Floatzahl nach IEEE 754

Integerarithmetik wird entsprechend den modularen Regeln umgebrochen. Zahlbereichsüberlauf dabei löst keine Ausnahmen aus.

`int b1=2147483647`	int-Bereich ist $-2^{31} \dots 2^{31} - 1$
`b2=b1+1;`	ergibt $-2147483648 = -2^{31}$

Java kennt die im Standard IEEE 754 festgelegten 'Zahlen' `NaN`, `NEGATIVE_INFINITY` und `POSITIVE_INFINITY`. Float-Fehler (Division durch 0.0, Wurzel aus negativer Zahl) lösen deshalb (nach IEEE 754) keine Ausnahmen aus.

Status kann mit entsprechenden Methoden der Wrapperklasse abgefragt werden.

Beispiele:

```
double    inf  =  1.0/0.0;
double neginf  = -1.0/0.0;
double negzero = -1.0/inf;
double NaN     = 0.0/0.0;
```


Arrays

- Deklaration: `int[] a;`
- Einfache Erzeugung zur Laufzeit möglich: `a=new int[100];`
- Indextyp ist `int`. Zugriff auf Feldelemente mit `a[i]`.
- Länge wird beim Erzeugen festgelegt.
  Attribut `length` kann nur gelesen werden.
- Laufzeitindexprüfung $0 \leq i <$ `a.length`.
  Fehler löst Ausnahme `ArrayIndexOutOfBoundsException` aus.
- Array von Objekten ist ein Array von Objektreferenzen
- Felder werden automatisch initialisiert (mit dem Defaultwert primitiver DT oder durch `null` für Objekte).
- Bilden von Unterklassen ist nicht möglich.

Beispiele:

```
int[] a;
a=new int[100]; // Array mit Defaultwerten 0 belegt

boolean b[][];
b=new boolean[n][m]; // m,n zur Laufzeit bekannt
```

Literal-Initialisierung ist möglich:

```
int[] a={5,9,3,2};
int[][] b={{0},{1,2},{3,4,5}};
```

Beispiel: Berechnung und Ausgabe einer Multiplikationstabelle

Realisierung: `Beispiel4.java`

Besonderheit der Array-Darstellung:

Mehrstufige Arrays sind Arrays von Arrays (also von Referenzen) und keine zusammenhängenden Speicherbereiche.

```
int[][] a=new int[10][12];
```

ist gleichwertig zu

```
int[][] a=new int[10][];
for(int i=0;i<a.length;i++) a[i]=new int[12];
```

Zeichenketten

Zeichenketten sind Objekte der Klassen

`java.lang.String`	konstante Zeichenketten
`java.lang.StringBuffer`	veränderliche Zeichenketten

Konstruktionsmethoden von `String` sind über Methoden von `StringBuffer` realisiert, insbesondere '+' über die Methode `append()`.

'+' löst eine automatische Typkonvertierung primitiver Datentypen in eine String-Darstellung mit `String.valueOf(..)` aus.

```
String s1a="Mein String";
String s1b=new String("Mein String");

Stringbuffer s2=new StringBuffer("Eine Zeichenkette");

boolean b; int u;
System.out.println(b+": Länge ist "+u);
```


2.7. Typkonvertierung

Typkonvertierung erfolgt automatisch, wenn dies möglich ist, also

wenn der Grundbereich des Zieldatentyps größer ist:

```
byte > short > int > long > float > double
char > int
```

oder die Funktionalität des Zieldatentyps eingeschränkt ist (Konversion vom Subtyp zum Basistyp)

Beispiel:

```
int    i=13;
double d=i;   // okay
byte   b=i;   // nicht zulässig
```

Einschränkende Konvertierung erfordert explizites Casting:

```
int i = (int) 13.445;       // Konvertierung des double-Literals

double x = 3.0, y = 2.0;
int  i = (int) x/y;        // = 1
```

Einige Operationen führen automatische Typkonversionen aus.

```
byte b1 = 127, b2 = 1;
byte sum = (byte) b1 + b2;  // Überlauf des Byte-Bereichs
```


2.8. Namen und Gültigkeitbereiche

Ein **Name** ist ein **Bezeichner**, der dazu dient, in einem Programm deklarierte Einheiten anzusprechen.

einfache Namen: `i`, `einName`, `eineMethode`

qualifizierte Namen:
`einObjekt.eineMethode`, `EineKlasse.einAttribut`, `java.lang.String`

Qualifizierte Namen haben mit der Dateistruktur der Laufzeit-Klassenbibliothek (`rt.jar` im Bibliotheksverzeichnis des JDK) zu tun.

Beispiel: `javac -verbose Beispiel3.java`

vollqualifizierte Namen: enthalten den vollständigen Paketpfad

Importieren von Namensräumen

```
import java.util.Vector;
import java.util.*;
```

Namenskonflikte

Beispiel: `java.util.Date`, `java.sql.Date`

falsch:

```
import java.util.Date;
import java.sql.Date;
...
    Date einUtilDatum;
    Date einSQLDatum;
...
```

richtig:

```
...
    java.util.Date einUtilDatum;
    java.sql.Date einSQLDatum;
...
```


Global einheitliche Namenskonvention für Pakete

Wichtig insbesondere für Anwendungen, die im Web über mehrere Rechner verteilt sind und dezentral verwaltet werden.

Suns Vorschlag: Paketname setzt sich zusammen aus dem Internet-Domänennamen und dem Pfad auf dem Rechner, wo das Paket abgelegt ist.

Angabe des Internet-Domänennamens in umgekehrter Reihenfolge der Felder.

Beispiel: `COM.sun`, `DE.uni-leipzig.informatik.pps.beispiele`

2.9. Ein komplexes Beispiel: Einfach verkettete Listen ganzer Zahlen in Java

Lösung in C (aus Vorlesung DigInf)

```
/* intlist.h */
typedef struct uhu { int eintrag; struct uhu *next; } node ;
typedef node* intlist;

extern intlist new();
extern intlist l_random(int n);
...
```

Lösung in Java: Zwei Klassen `TList` und `TNode`

`TList`: Interface zur Außenwelt (Methoden), verbirgt Interna

`TNode`: Klasse als Verbund, nur Zugriffsmethoden


```
public class TList {

  /* innere Klasse (besser in extra Datei!) */
  private class TNode {
    // Felder
    int data; TNode next;
    // Konstruktor
    TNode(int a, TNode n) { data=a; next=n; }
  }

  /* Attribute von TList */
  private TNode head;

  /* Konstruktoren von TList */
  TList() { head=null; }
  TList(int n) { ... }

  /* Methoden von TList */
  public String toString() { ... }

}
```

Realisierung in `TList.java` und `Beispiel5.java`

2.10. Fehlerbehandlung in Java

Das Konzept der **Ausnahme**: Ausnahmen sind Programmzustände, in denen die Arbeit nicht entsprechend dem geplanten Kontrollfluss fortgesetzt werden kann.

klassische Reaktion: Programmabbruch
modern: vorhersehbare Ausnahmen werde „aufgefangen“

Java kennt dazu spezielle **Ausnahmeobjekte** (abgeleitet von der Klasse `Throwable`), die an einer Fehlerstelle generiert („geworfen“) werden und an einer anderen Stelle (durch eine im aktuellen Aufrufstack weiter oben liegende Funktion) ausgewertet („aufgefangen“) werden.

Ausnahmen wandern den Aufrufstack hinauf, bis sie von einer Methode ausgewertet („aufgefangen“) werden.

Ausnahmen werden spätestens von der JVM abgefangen, die Ausnahmeart und Aufrufstack ausgibt.

Ausnahme werfen: `throw` *Ausdruck*

Ausdruck wertet zu einem Ausnahmeobjekt aus (meist wird dieses anonym erzeugt)

Beispiel:

```
double factorial(int x) {
  if (x < 0)
    throw new IllegalArgumentException("x muss >= 0 sein!");
  double fact;
  for (fact=1.0; x>1; x--) fact*=x;
  return fact;
}
```

Java-Realisierung: `Beispiel6 12 3 -2`

weiteres Beispiel: `Beispiel3 F` (NumberFormatException)

2.10. Fehlerbehandlung in Java

Das Konzept der **Ausnahme**: Ausnahmen sind Programmzustände, in denen die Arbeit nicht entsprechend dem geplanten Kontrollfluss fortgesetzt werden kann.

klassische Reaktion: Programmabbruch
modern: vorhersehbare Ausnahmen werde „aufgefangen“

Java kennt dazu spezielle **Ausnahmeobjekte** (abgeleitet von der Klasse `Throwable`), die an einer Fehlerstelle generiert („geworfen“) werden und an einer anderen Stelle (durch eine im aktuellen Aufrufstack weiter oben liegende Funktion) ausgewertet („aufgefangen“) werden.

Ausnahmen wandern den Aufrufstack hinauf, bis sie von einer Methode ausgewertet („aufgefangen“) werden.

Ausnahmen werden spätestens von der JVM abgefangen, die Ausnahmeart und Aufrufstack ausgibt.

Ausnahme werfen: `throw` *Ausdruck*

Ausdruck wertet zu einem Ausnahmeobjekt aus (meist wird dieses anonym erzeugt)

Beispiel:

```
double factorial(int x) {
  if (x < 0)
    throw new IllegalArgumentException("x muss >= 0 sein!");
  double fact;
  for (fact=1.0; x>1; x--) fact*=x;
  return fact;
}
```

Java-Realisierung: `Beispiel6 12 3 -2`

weiteres Beispiel: `Beispiel3 F` (NumberFormatException)

Ausnahmetypen

Die Ausnahmebasisklasse `Throwable` hat zwei unmittelbare Unterklassen:

Die Klasse `Error`	nicht behebbare Probleme (werden selten gefangen)
Die Klasse `Exception`	weniger schwerwiegende Probleme

Methoden, die Ausnahmen werfen, können bzw. müssen mit `throws` im Prozedurkopf als solche deklariert werden.

Beispiel: `Integer.parseInt(String s)`

Geprüfte und ungeprüfte Ausnahmen

Der Compiler kann verfolgen, ob gewisse Arten von Ausnahmen, die zur Compilezeit explizit als solche deklariert sind (**geprüfte Ausnahmen**), aufgefangen werden. Dazu sind alle Methoden, die explizit oder implizit eine solche Ausnahme auslösen können, als *Ausnahme werfend* zu deklarieren.

Java-Methoden *können* Ausnahmen

abfangen (mit `try { ... } catch (Exception e) { ... }`)

oder weiterreichen (mit `throws` im Prozedurkopf)

Da jede Methode Ausnahmen auslösen kann (etwa `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`, ...), werden Ausnahmen, die von `Error` oder `RunTimeException` abgeleitet sind, aus Performancegründen **nicht geprüft**. Alle anderen Ausnahmen werden vom Compiler geprüft („überwacht").

Methoden, die geprüfte Ausnahmen auslösen oder weiterreichen, **müssen** als solche gekennzeichnet sein.

Beispiel:

```
double factorial(int x) throws Exception {
  if (x < 0)
    throw new Exception("x muss >= 0 sein!");
  double fact;
  for (fact=1.0; x>1; x--) fact*=x;
  return fact;
}
```

Java-Realisierung: `Beispiel7.java`

Ausnahmen auffangen

Dies geschieht in einem `try-catch-finally` Konstrukt.

```
try {
    /* Code, der normalerweise problemlos läuft, aber (direkt oder indirekt)
       Ausnahmen auslösen kann. */
} catch(eineAusnahme e1) {
    /* Reaktion auf Ausnahmen eines ersten Typs. Zugriff auf die Daten
       im Ausnahmeobjekt über dessen Namen e1. */
} catch(andereAusnahme e2) {
    /* Reaktion auf Ausnahmen eines weiteren Typs. Es wird höchstens
       einer der catch-Blöcke abgearbeitet. */
} finally { /* Aufräumcode */ }
```

Die `finally`-Klausel kann fehlen. Sie wird *immer* ausgeführt, entweder nach dem normalen Ende des `try`-Blocks oder nach der Abarbeitung der abgefangenen Ausnahme.

Nicht abgefangene Ausnahmen werden auch von einem `try`-Block weitergereicht.

`try-finally` Konstrukte sind auch im Zusammenhang mit `break`- oder `return`-Anweisungen interessant.

Beispiel (auf der Basis von Beispiel3): Auch ungeprüfte Ausnahmen können abgefangen werden.

```
try {
    x=Integer.parseInt(argv[i]);
    qs=Quersumme(x);
    System.out.println("Quersumme von "+x+" ist "+qs);
}
catch (NumberFormatException e) {
    System.out.println("NumberFormatException wurde ausgelöst");
}
```

Realisierung: `Beispiel8.java`

2.11. Beispiel: Taschenrechner mit UPN-Notation

Ideen:

1) Token liefert der Kommandozeileninterpreter
2) Stack für Ablage der Zwischenergebnisse in eigener Klasse
3) Taschenrechner in eigener Klasse mit einer Methode `calculate`
4) Testprogramm wie immer in weiterer Klasse

Realisierung: Stack $\Rightarrow$ Hausaufgabe
Taschenrechner-Klasse: `TR1.java`
Testapplikation: `Beispiel9.java`

Beispiele:

```
java Beispiel9 1 2 +              Ergebnis:    3.0
java Beispiel9 2 10 ^             Ergebnis: 1024.0
java Beispiel9 1 2 3 4 x - /      Ergebnis:   -0.1
```

3. Objektorientierung in Java

Klassen und das Client/Server-Prinzip

Klassen bestehen aus

Attributen (oder Datenfeldern)
- gemeinsame interne **Struktur** der Objekte
- Aggregation, Klasse als Container

Methodendeklarationen oder **-definitionen**
- gemeinsame **Funktionalität** der Objekte
- Schnittstelle

Klassen fassen Daten zu einer logischen und syntaktischen Einheit zusammen.

- Client/Server-Prinzip
- Kontrakt

3.1. Klassen und Objekte

Eine **Klasse** enthält die Definition der Datentypen der **Attribute** sowie die Deklarationen der Signaturen und ggf. die Implementierungen der **Methoden**, die alle Objekte der Klasse gemeinsam haben.

Objekte sind Instanzen der Klasse, denen während der Erzeugung (`new`) durch **Konstruktoren** konkrete Speicherbereiche auf der Halde zugeordnet werden.

Aufteilung der Funktionalität zwischen Objekt und Klasse.

Die Methode `getClass()` von `Object`.

Instanzmethode als Klassenmethode mit zusätzlichem Parameter

dezentral: Instanzmethode `a.machwas(Order)`

Nachricht (von außen) an `a`, `machwas(Order)` auszuführen

zentral: Klassenmethode `Klasse_von_a.machwas(a,Order)`

Nachricht (von außen) an `Klasse_von_a`, den Untertan (die Instanz) a `machwas(Order)` ausführen zu lassen.

Selbstreferenz (Methodenaufruf *innerhalb* des Klassenrumpfs):

dezentral: Instanzmethode `machwas(Order)`

Erinnerung an sich (`this`), `machwas(Order)` auszuführen

zentral: Klassenmethode `machwas(a,Order)`

Befehl an Untertan a, `machwas(Order)` auszuführen

Die Basisklasse `Object` der Java-Klassenhierarchie

Wichtige Methoden:

`toString()`	Stringdarstellung eines Objekts
`equals(Object o)`	Vergleich von zwei Objekten (`this` und `o`)
`clone()`	Kopie eines Objekts
`getClass()`	siehe oben

Beispiel: `Beispiel5.java` und `TList.java` ohne Methode `toString()`

Im Normalfall werden die (eher syntaktisch definierten) Methoden der Basisklasse `Object` in abgeleiteten Klassen durch semantisch sinnvolle Methoden **überschrieben**.

Die Klasse `Class`

3.2. Ein Beispiel

Basisklasse `Circle` mit Klassenattributen, Klassenmethoden, Instanzattributen und Instanzmethoden

Das Schlüsselwort `this`

`Circle.java` und `Beispiel10`

Subklasse `PlaneCircle`. Geerbte, verdeckte und überschriebene Attribute und Methoden.

`PlaneCircle.java` und `Beispiel11`

Klassen, Objekte und Vererbung

`class B extends A { ... }`

- Unterklasse B erbt alle Felder und Methoden der Oberklasse A und ergänzt diese um eigene Felder und Methoden. (`getFields()` und `getDeclaredFields()` etc.)
- Klassenzeiger auf die Oberklasse: `B.getSuperClass()`
- Instanzen der Unterklasse B sind damit auch Instanzen der (eingeschränkteren) Oberklasse A.
- Variablen vom Typ A können auch mit Objekten vom Typ B belegt werden. Objektvariablen sind in diesem Sinne **polymorph**.
- Prüfung mit dem Operator `instanceof`.

```
A a=new A(); B b=new B();
a instanceof A : true      b instanceof A : true
a instanceof B : false     b instanceof B : true
```

3.3. Überladen, Überdecken, Überschreiben

Unter allen (geerbten und neudefinierten) Methoden und Feldern können solche mit gleichem Namen sein (Prinzip des **Polymorphismus** der Objektorientierung).

Überladen

Es können mehrere Methoden mit gleichem Namen, aber unterschiedlicher Signatur definiert werden. Solche Methoden sind **überladen**.

```
class Point { ...
  void set(int x0,int y0) { x=x0; y=y0; }
  void set(int x0,int y0,int c0) { x=x0; y=y0; color=c0; }
... }
```

Von den Methoden wird die mit der passenden Signatur aufgerufen. Das kann (und wird) bereits zur Übersetzungszeit entschieden. (**statische Bindung**)

Passen mehrere Methoden, so wird die speziellste aufgerufen.

```
class Number { ...
  void add(int i) { ... }
  void add(float i) { ... }
... }

Number a; a.add(5);
```

Es können sowohl Klassen- als auch Instanzmethoden überladen sein. Mehrfache Konstruktoren einer Klasse sind grundsätzlich überladen.

Das Verhalten überladener Methoden sollte „gleichartig“ sein, um Programmierfehler zu vermeiden. (Semantisch) komplizierte Überladungen sollte man vermeiden.

Überdecken

Wird in einer Subklasse ein Feld mit gleichem Namen wie in der Oberklasse deklariert, so **überdeckt** dieses das Feld der Oberklasse.

```
public class Circle {
    // Ein Klassenfeld
    public static final double PI=3.14159;
    // Ein Instanzfeld
    double r; // Der Radius
...
}

public class PlaneCircle extends Circle {
    // Ein Klassenfeld
    public static final double PI=3.14159265358979323846264;
    // Instanzfelder
    double cx, cy; // Zentrumskoordinaten
    double r; // Der Abstand zum Ursprung
...
    this.r=Math.sqrt(cx*cx+cy*cy);
}
```


Überdecken ist eine Frage der **Sichtbarkeit**. Beide Felder sind **erreichbar**:

Zugriff mit qualifiziertem Bezeichner (für Klassenfelder)

```
System.out.println("Circle.PI ist "+Circle.PI);
System.out.println("PlaneCircle.PI ist "+PlaneCircle.PI);
```

oder über `this` und `super` bzw. Casting (für Instanzfelder)

```
r                   // Das Feld r in PlaneCircle
this.r              // Das Feld r in PlaneCircle
super.r             // Das Feld r in Circle
((Circle) this).r   // Das Feld r in Circle
```

Dasselbe gilt für **Klassenmethoden** mit gleichem Namen und gleicher Signatur. Auf jede von ihnen kann über ihren qualifizierten Namen zugegriffen werden.

Überschreiben

Instanzmethoden mit gleichem Namen und gleicher Signatur einer Basisklasse, die in einer Subklasse redefiniert werden, werden **überschrieben**.

```
class Ellipse extends Circle {
    double a,b; // Halbachsen
    ...
    public double inhalt() { return PI*a*b; }
    public double umfang() { return ... /* ein ellipt. Integral */ }
    ...
}
```

`toString()` in `Circle` und `PlaneCircle`

Überschreiben von Methoden ist eine der wichtigsten und nützlichsten Techniken der OO-Programmierung: (Syntaktisch) gleichnamige geerbte Methoden werden durch (semantisch) angemessene neue Methoden **ersetzt**.

Damit können an Unterklassen einer Klasse dieselben Botschaften geschickt werden, die jedoch objektspezifisch auf unterschiedliche Weise umgesetzt werden.

```
Circle[] cc=new Circle[2];
cc[0]=new Circle(4);
cc[1]=new PlaneCircle(1,2,3);

for(int i=0;i<cc.length;i++) System.out.println(cc[i]);
```

Beispiel12.java

Die korrekte Methode kann erst zur Laufzeit aus der (genauen) Klasse des Objekts o (mit `o.getClass()`) abgeleitet werden (**dynamisches Binden**).

Typischer Anwendungsfall: Zeichenroutinen für grafische Objekte, die als Sammlung (Collection) in einem Container (Array, Liste oder andere Containerklasse) enthalten sind:

```
Container einContainer;
einContainer.draw() ==> for c in einContainer do c.draw()
```

Benötigt gemeinsame Oberklasse (`Figure`) für die Grafikelemente (Rechteck, Kreis, Dreieck, . . .), deren Methode `draw()` in den Unterklassen überschrieben wird.

Mehr dazu im Kapitel „Abstraktionskonzepte“.

3.4. Aufbau von Klassendefinitionen

```
[Modifikatoren] class Bezeichner [extends Basisklasse]
                             [implements Schnittstelle]
{  Klassenrumpf }
```

Modifikatoren:

public: Klasse öffentlich zugänglich (sonst nur innerhalb der Datei bzw. des Pakets).

Jede Datei kann nur eine `public class` enthalten, deren Name mit dem Dateinamen übereinstimmen muss.

abstract: Klasse ist abstrakt, d.h. nicht alle Methoden sind implementiert

final: Klasse ist final, d.h. es können keine Subklassen abgeleitet werden

Klassen, in denen nicht alle Methoden implementiert sind heißen **abstrakt**. Von diesen können keine Instanzen gebildet werden.

Abstrakte Methoden müssen durch den Modifikator `abstract` gekennzeichnet sein. Klassen, die als `abstract` gekennzeichnete Methoden enthalten, müssen selbst als `abstract` gekennzeichnet sein.

Die Verwendung abstrakter Methoden ist ein grundlegendes Prinzip der generischen OO-Programmierung: Sie können in „generischen" Methodendefinitionen verwendet werden, die (ohne Änderung) korrekt laufen, wenn die abstrakten Methoden in Subklassen korrekt überschrieben sind.

Schnittstellen sind spezielle abstrakte Klassen, die nur aus Methodendeklarationen (und evtl. Klassenvariablen) bestehen. Kennzeichnung durch Schlüsselwort `interface` statt `class`.

Schnittstellen werden verwendet, um Eigenschaften zu deklarieren (**Marker-Interface**) oder zu implementierende Methoden vorzuschreiben (**Methoden-Interface**).

Beispiele:
`Cloneable`, `Serializable` (Marker-Interfaces)
`Runnable`, `Comparable` (Methoden-Interfaces)

Eine Klasse kann mehrere Schnittstellen implementieren (einzige Möglichkeit zur Mehrfachvererbung in Java).

Attribute

`[Modifikatoren] Typ Bezeichner [ = Initialisierung]`

Eine Attributdeklaration besteht aus

- optionalen Modifikatoren
- dem Attributtyp (primitiver oder Referenztyp)
- einem oder durch Kommata getrennten Bezeichnern
- optionaler Initialisierung

Wichtige Modifikatoren:

Sichtbarkeit	`public`, `protected`, `private`
Konstanten	`final` (nach Initialisierung nicht änderbar)
Klassenattribute	`static`

Beispiele:

```
int x,y;
static float f = 3.4f;
final java.lang.String test=new String("Ein Test");
double x=0.0, y=1.0, z;
double[] d=new double[5];
```

Ein klassischer Verbund (in C) entspricht einer Klassendefinition, die nur Attribute enthält (also keine Methoden)

Beispiel: `class Messwert { int x,y; double wert; }`

Siehe auch Klasse `TNode`.

Methoden

```
[Modifikatoren] Resultattyp Bezeichner(Parameterliste)
                         [Ausnahmespezifikationen] {
   /* Rumpf */
}
```

Eine Methodendefinition besteht aus

- optionalen Modifikatoren
- dem Resultattyp (primitiver oder Referenztyp bzw. `void`)
- dem Methodennamen
- der Liste formaler Parameter (kommaspearierte Einträge `Typ Name`)
- optionalen Ausnahmespezifikationen (`throws Exc1, Exc2, ...`)
- dem Methodenrumpf

wichtige Modifikatoren:

Sichtbarkeit	`public protected private`
Konstanten	`final` (Methode kann in Subklassen nicht redefiniert werden)
Klassenmethoden	`static`
Abstrakte Methoden	`abstract` (nur Methodendeklaration, kein Rumpf)

Beispiel:

```
class Point {
  private int x,y,color;

  public void print() {
    System.out.println("["+x+","+y+"]");
  }
}
```

Ein komplexes Beispiel: Taschenrechner für ganze Zahlen großer Länge

Verwende im Beispiel `TR1` statt des primitiven Datentyps `double` als Zahlbereich `BigInteger`.

Demo für das Zusammenwirken von drei Klassen (`TR_BI`, `Stack_BI` und `BigInteger`)

Erforderliche Modifikationen des `TR1`:

- Klasse `Stack` muss auf `BigInteger` umgeschrieben werden.
- Alle Operatoraufrufe in `TR1` sind durch entsprechende Methodenaufrufe aus `BigInteger` zu ersetzen.
- Ersetze in Beispiel9 `TR1` durch `TR_BI` und den Rückgabewert `double` durch `java.math.BigInteger` (oder importiere das Package vorher).

Siehe `Beispiel13.java`

Aufbau einer eigenen Klasse Long1 zum Rechnen mit long-Zahlen analog zu BigInteger

Siehe `Long1.java`

Aufbau eines Taschenrechners auf dieser Basis: Definiere analog die Klassen `TR_L`, `Stack_L` und `Beispiel14.java`

3.5. Klassenhierarchien, Kapselung und Zugriffskontrolle

Neben einer vertikalen Zuordnung von Klassen (Vererbung) gibt es in Java auch die Möglichkeit, inhaltlich zusammengehörende Klassen (horizontal) zusammenzufassen (Packages).

Packages

Ein Package ist eine benannte Sammlung von Klassen (und evtl. Subpackages), die dazu dient, verwandte Klassen zusammenzufassen und für diese einen gemeinsamen Namensraum zur Verfügung zu stellen.

Kennzeichnung durch package-Befehl als erste Zeile

```
package <name>;
```

Dateien ohne Package-Befehl gehören zu einem gemeinsamen (unbenannten) Default-package.

Beispiele aus der Java-Plattform

`java.lang`	Java-Basisklassen (wird automatisch importiert)
`java.io`	Klassen zur Manipulation von Datenströmen und Dateizugriffen
`java.net`	Klassen, die Netzwerkprogrammierung unterstützen
`java.awt`	Der *abstract windowing toolkit*, eine Sammlung von Klassen zur Entwicklung plattformunabhängiger GUI
`java.applet`	Klassen zur Entwicklung von Applets
`java.util`	Sammelpaket für Hilfsklassen, die komplexe Datentypen (Listen, Hashes, Stacks, Tabellen, ...) zur Verfügung stellen.

Beispiele von Erweiterungen

`javax.*`	von Sun definierte Standarderweiterungen der Java-Plattform (z.B. die swing-Plattform)
`org.omg`	Java-Implementierungen von Standards der OMG (`www.omg.org`)

Kapselung und Zugriffskontrolle

public	Klasse/Attribut/Methode kann von allen anderen Klassen und Objekten verwendet werden
protected	Klasse/Attribut/Methode kann nur innerhalb der Klassenhierarchie oder des gleichen Pakets verwendet werden, wird also vererbt
(default)	(innere) Klasse/Attribut/Methode ist nur innerhalb der Klasse und des Pakets, dem die Klasse angehört, sichtbar
private	(innere) Klasse/Attribut/Methode ist nur für Objekte dieser Klasse sichtbar und wird auch nicht vererbt

Ist die Klasse nicht sichtbar, so sind auch deren Attribute und Methoden nicht sichtbar, unabhängig von deren Modifikator.

Sichtbarkeit und Vererbung

Beim Überschreiben ist es meist sinnvoll, die Sichtbarkeitsklasse beizubehalten.

Allgemein darf die Unterklasse die Sichtbarkeit nicht einschränken (da die Methoden der Basisklasse geerbt werden):

Oberklasse		**Unterklasse**
public	=>	public
protected	=>	public oder protected
default	=>	default, public oder protected

Für überladene, überdeckte oder neu definierte Methoden gibt es keine solchen Einschränkungen.

Faustregeln zur Anwendung von Sichtbarkeitsmodifikatoren

public: Nur für Methoden, die Bestandteil des öffentlichen API der Klasse (Klassenkontrakt) sind. Felder sollten nicht public, sondern Zugriffsmethoden gekapselt sein.

protected: Felder und Methoden, die Programmierer von Anwendungen nicht brauchen, die aber für abgeleitete Klassen in einem anderem Package relevant sind. Sollten ebenso wie public-Methoden gut dokumentiert sein.

default: Innere Methoden und Felder, die aber von kooperierenden Klassen aus demselben Package benötigt werden. Dazu ist das Anlegen eines Package erforderlich.

private: Felder und Methoden, die nur in dieser Klasse verwendet werden und überall sonst versteckt sein sollen.

Beispiel: `Circle1.java`

3.6. Der Objekt-Lebenszyklus

Konstruktoren

Jede Klasse enthält einen oder mehrere Konstruktoren, um Instanzen der Klasse zu erzeugen und zu initialisieren.

- Konstruktoren haben den gleichen Namen wie die Klasse.
- Konstruktoren haben keinen Ergebnistyp (denn sie reagieren nicht auf Botschaften).
- Eine Klasse kann mehrere (überladene) Konstruktoren besitzen.
- Ein Konstruktor kann andere Konstruktoren aufrufen. Jedoch muss dieser Aufruf als erste Anweisung erfolgen.
- Ein Konstruktor einer abgeleiteten Klasse muss zunächst einen Konstruktor der Basisklasse aufrufen (mit `super(...)`).
  Sonst wird implizit `super()` aufgerufen (das dann in der Basisklasse auch definiert sein muss).

- Ist für die Klasse `K` kein Konstruktor definiert, so wird ein Konstruktor `K()` implizit hinzugefügt, der (nur) Speicherplatz für die zu erzeugende Instanz reserviert und deren Felder mit den Standardwerten initialisiert.

Beispiel:

```
class Point { ...
  Point() { x=0; y=0; }
  Point(int x0,int y0) { x=x0; y=y0; }
... }
```


Objekte (Klasseninstanzen) erzeugen

Beispiel:

```
Point A;            // Objektdeklaration
A=new Point(2,3);   // Objekterzeugung
```

Objekterzeugung bewirkt:

- Allokation von Speicherplatz
- Aufruf eines Konstruktors zur Initialisierung des Objekts
- Rückgabe einer Referenz auf das Objekt

Der Konstruktor einer Klasse muss alle Attributwerte, die selbst Objektreferenzen sind, initialisieren.

falsch:

```
class Strecke {
  Point A,B;
  // impliziter Konstruktor Strecke()
  void print() { A.print(); B.print(); }
}
```

richtig:

```
class Strecke {
  Point A,B;
  Strecke() { A=new Point(); B=new Point(); }
  void print() { A.print(); B.print(); }
}
```


Objekte aufgeben (finalisieren)

Geschieht automatisch, wenn auf das Objekt keine expliziten (aus Codeteil) oder impliziten (aus anderen Objekten) Referenzen mehr existieren (automatische Speicherbereinigung – **garbage collection** als Thread der JVM im Hintergrund).

Manche Objekte müssen vorher „dekonstruiert“ werden (Dateien und Kanäle schließen, Kindprozesse abbrechen etc.).

Methode `finalize()` der Klasse `Object`.

Manchmal ist explizite Speicherfreigabe sinnvoll.

```
int computeAndProcessAverage() {

  int[][][] cube =
  initializeCube(cube);
    /* enthält new int[100][100][100],
            was 2 MB Speicher allokiert */

  Average avg = computeAverage(cube);
    /* Analyse von cube, kleinere Datenmenge */

  cube = null;
    /* Sonst wäre Speicher bis zum Ende blockiert */

  doSomethingTimeConsuming(avg);
}
```


4. Abstraktionskonzepte

Ziel: Wiederverwendbarkeit von Code und Daten sicherstellen. Reduktion von Code-Wiederholungen auf verantwortbares Minimum.

Beispiele:

Stack, Liste: Natur der gespeicherten Objekte ist zweitrangig. Gefordert ist ein Konzept, die wesentlichen Merkmale dieser Datentypen einmal zu implementieren. Dazu muss dieses „Wesentliche“ herausgearbeitet werden (Abstraktion).

Java-Dokumentation von `Stack` und die darum existierende Klassenhierarchie.

Abstrahiert verschiedene Arten, nicht näher spezifizierte Objekte zu sammeln.

Sortierverfahren: Natur der zu sortierenden Objekte ist weitgehend zweitrangig, sie müssen jedoch vergleichbar sein.

Siehe in der Java-Dokumentation `Comparable`, `Array.sort()`, `Collections.sort()`.

Typisch eingesetztes Instrumentarium:

- Sprachmittel zur Beschreibung solcher Abstraktionen:

  gut zugeschnittene Klassenhierarchien, Templates und parametrisierte Datentypen bzw. abstrakte Klassen und Schnittstellen
- Sprachmittel zur Rückbindung von Abstraktionen an reale Daten:

  Überladen, Überschreiben und (Re)implementieren von Methoden

4.1. Überladen und Überschreiben von Instanzmethoden

Überladen

Zu einem Objekt o können mehrere Methoden mit gleichem Namen, aber unterschiedlicher Signatur gehören. Solche Methodennamen nennt man **überladen**. (Prinzip: ein Name – mehrere Methoden)

Die entsprechenden Implementierungen koexistieren **nebeneinander**, siehe `o.getClass().getMethods()`, und sind alle (von o aus direkt) erreichbar.

Der Compiler entscheidet zur Übersetzungszeit (**statisches Binden**) an Hand der Typen der Aufrufparameter, welche der Methoden verwendet wird.

Für polymorphe Objektvariablen (Instanzen mehrerer Klassen einer Hierarchie) als Aufrufparameter kann es mehrere passende Methoden geben. Von diesen wird immer die **speziellste** aufgerufen.

```
class A { ... void call(A a) { ... } ... }

class B extends A { ...  void call(B b) { ... } }
```

Objekt o der Klasse B kennt damit die Methoden `o.call(A)` und `o.call(B)`. Unterscheide deshalb:

```
B o=new B(); A a=new A(); B b=new B();
o.call(a); o.call(b); o.call((A) b);
```

Spezieller sind Methoden einer spezielleren Klasse mit spezielleren Parametertypen (Halbordnung! Objekttyp und Parametertypen gleichberechtigt)

Compilerfehler, wenn es eine solche speziellste Methode nicht gibt.

Entscheidung wird **statisch**, also zur Übersetzungszeit getroffen.

Überschreiben

Wenn eine Subklasse S eine Instanzmethode mit gleichem Namen, Rückgabetyp und Parametern wie eine Methode der Basisklasse definiert, dann **überschreibt** sie die Methode der Basisklasse.

In `S.getMethods()` wird die alte durch die neue Methode **ersetzt**. Die alte Methode ist nicht nur **nicht sichtbar**, sondern (von der Methodentabelle von S aus) **nicht erreichbar**.

Zum Aufrufverhalten überschriebener Methoden:

```
...
  print(Circle[] cc) {
    for(int i=0;i<cc.length;i++) System.out.println(cc[i]);
  }
...
Circle[] cc=new Circle[2];
cc[0]=new Circle(4);
cc[1]=new PlaneCircle(1,2,3);
print(cc);
```

Woher weiß der Compiler, welche Methode er aufzurufen hat? Er weiß es in der Tat nicht, denn die jeweils zu verwendende Methode wird **dynamisch** zur Laufzeit aus der genauen Klasse des Objekts o (mit `o.getClass().getMethod(name, paramTypes)`) deduziert.

Ein (impliziter oder expliziter) Typecast hat darauf keinen Einfluss, denn er ändert das Objekt nicht. (Ganz anders bei überladenen Methoden!)

Trotz dynamischer Bindung verwendet Java eine **statische Typprüfung**. Deshalb müssen überschriebene Methoden dieselbe Signatur implementieren: `name` und `paramTypes` wird in obigem Aufruf zur Compilezeit eingesetzt. Die Typprüfung wird zur Compilezeit abgeschlossen.

Dynamisches Binden verlangt einen zusätzlichen Aufwand im Laufzeitsystem. In C++ müssen solche in Unterklassen überschreibbaren Methoden deshalb speziell als „virtuell“ gekennzeichnet werden.

In Java sind alle Methoden dynamisch gebunden, da jede in Subklassen überschrieben werden kann.

Überschreiben ist nicht überdecken: In `o.getClass().getMethods()` wird die geerbte Methode durch die reimplementierte ersetzt. Von o aus besteht also kein direkter Zugriff auf die geerbte Methode mehr.

Nur Methoden, die als `final` gekennzeichnet sind, müssen nicht dynamisch gebunden werden (und können durch native 'just in time' Compiler optimiert werden)

```
class A {
  int i=1;                        // Instanzfeld
  int f() { return i; }           // Instanzmethode
  static char g() { return 'A'; } // Klassenmethode
}

class B extends A {
  int i=2;                        // überdeckt Feld i
  int f() { return (-i); }        // überschreibt Methode f
  static char g() { return 'B'; } // überdeckt Klassenmethode g
}
...
B b=new B();

System.out.println(b.i);     // B.i      ->  2
System.out.println(b.f());   // B.f()    -> -2
System.out.println(B.g());   // B.g()    ->  B
System.out.println(b.g());   // so geht's auch

A a=b;                       // Typumwandlung
System.out.println(a.i);     // A.i      ->  1 (in B überdeckt)
System.out.println(a.f());   // B.f()    -> -2 (überschrieben !)
System.out.println(a.g());   // A.g()    ->  A (überdeckt !)
System.out.println(A.g());   // so ist's besser
```


Überschriebene Methoden sind nur über eine explizite Referenz auf die Oberklasse erreichbar:

```
class A {
  int i=1;
  int f() { return i; }
}

class B extends A {
  int i;                   // verdeckt A.i
  int f() {                // überschreibt A.f()
    i=super.i + 1;         // Zugriff auf A.i, auch ((A) this).i
    return super.f()+i;    // Aufruf von A.f(), nicht ((A) this).f()
  }
```

Achtung: Klassenmethoden (gleicher Signatur) verhalten sich ganz anders: Sie können nicht überschrieben, sondern nur überdeckt werden, da sich ihre (qualifizierten) Namen **immer unterscheiden**.

Typprüfung und Bindung

Programmtext enthält nur *Bezeichner* zur Benennung von Objekten und Methoden.

Die Zuordnung Bezeichner – Objekt/Methode heißt *Bindung*. Unterscheide Variablenbindung und Methodenbindung.

Polymorphie: Bezeichner können in unterschiedlichem Kontext unterschiedliche Objekte/Methoden benennen. Mehrdeutigkeit muss zur Bindung durch Auswertung von Zusatzinformationen aufgelöst werden.

statisch	Informationen werden vom Compiler zur Übersetzungszeit ausgewertet.
dynamisch	Informationen werden (erst) vom Laufzeitsystem ausgewertet.

OO-Ansatz kennt *Polymorphie von Bezeichnern* für Objektreferenzen: Eine Variable vom Typ A kann eine Instanz jeder Unterklasse `B extends A` aufnehmen. Bindung erfolgt in zwei Etappen: statische Deklaration und dynamische Objekterzeugung.

```
A a; a=new B();
```

Polymorphie von Methodenbezeichnern: Ein Methodenbezeichner kann mit unterschiedlichen Signaturen (überladen) oder mit gleicher Signatur in unterschiedlichen Klassen (überschrieben) vorkommen.

Methodenbindung in Java erfolgt in zwei Schritten:

(1) Auflösung der Polymorphie an Hand der Typsignatur: Typprüfung und Anlegen eines Verweises `o.getClass().getMethod(`T_1`, ..., `T_n`)` in die Methodentabelle des Objekts o

(2) Binden des Verweises an eine konkrete Funktion

(2) geht (für potenziell überschriebene Methoden) nur dynamisch, da die genaue Objektklasse von o erst zur Laufzeit feststeht.

(1) kann statisch oder dynamisch erfolgen

Statisch getypte Sprachen

- Compiler kann bereits alle Typsyntaxfehler aufdecken, bevor der Code läuft
- keine Typprüfung zur Laufzeit erforderlich (Code damit kompakter und schneller)
- Zur Laufzeit können (bekannte) übersetzte Klassen geladen, aber keine neuen erzeugt werden.
- zur Laufzeit können alte Methoden nur durch solche mit gleicher Syntax (Java) überschrieben werden.

Beispiel: C, C++, Java

Dynamisch getypte Sprachen

Beispiel: Smalltalk, Lisp, Aldor, MuPAD

Eigenschaften:

- Flexiblere generische Konstruktionen sind möglich
- Neue Klassen können zur Laufzeit erzeugt werden
- komplexeres und im Allgemeinen mächtigeres Sprachkonzept
- durch Typprüfungs-Overhead langsamer
- Sicherheitskonzepte sind schwieriger zu realisieren
- Fehlersuche ist schwieriger

Java: statische Typprüfung und dynamische Methodenbindung.

C++: statische Typprüfung und statische Methodenbindung außer für `virtual` deklarierte Methoden. Nur diese werden in die VMT der Klasse eingetragen.

Ein zusammenfassendes Beispiel

```
class A {
    void m1() { System.out.println("A->m1()"); }
    void m2(A a) { System.out.println("A->m2(A)"); }
    A m3() { System.out.println("A->m3()"); return new A(); }
    A m4(A a) { System.out.println("A->m4(A)"); return new A(); }
}

class B extends A {
    void m1() { System.out.println("B->m1()"); }
    void m2(B a) { System.out.println("B->m2(B)"); }
    void m2(A a) { System.out.println("B->m2(A)"); }
    // B m3() { System.out.println("B->m3()"); return new B(); }
    /* Nicht erlaubt, weil gleiche Parameter, aber anderer
       Returntyp */
    B m4(B a) { System.out.println("B->m4(B)"); return new B(); }
}
```

PolymorphieTest.java

4.2. Abstrakte Klassen und Schnittstellen

Abstrakte Methoden und abstrakte Klassen

Beispiel: `package shapes` implementiert `Circle`, `Rechteck`, `Quadrat`, `Dreieck`, `Ellipse`, jede mit Methoden

```
public double inhalt() { ... }
public double umfang() { ... }
public void draw() { ... }
```

Gemeinsames Verwenden solcher Figuren erfordert Basisklasse `Figur` als gemeinsame Abstraktion.

Lösung als abstrakte Klasse

```
abstract class Figur {
  public abstract double inhalt();
  public abstract double umfang();
  public abstract void draw();
}

class Circle extends Figur { ... /* wie bisher */ }
class Rechteck extends Figur { ... /* wie bisher */ }

Figur[] f=new Figur[3];

f[0]=new Circle(2.0);
f[1]=new Rechteck(2.0,3.0);
f[2]=new Rechteck(4.0,2.0);

double total_area = 0;
for(int i=0; i < f.length; i++)
  total_area += f[i].inhalt();
```

Die jeweils richtige Methode `inhalt()` wird dynamisch ermittelt.

Regeln zum Einsatz abstrakter Methoden

- Eine Klasse mit abstrakten Methoden ist selbst automatisch abstrakt und muss als solche deklariert werden.
- Eine abstrakte Klasse kann nicht instantiiert werden.
- Eine abgeleitete Klasse kann nur instantiiert werden, wenn sie alle abstrakten Methoden mit eigenen Implementationen überschreibt. Solche Subklassen heißen **konkret**.
- Eine abgeleitete Klasse, die nicht alle abstrakten Methoden implementiert, ist selbst wieder abstrakt.
- `static`, `private` oder `final` deklarierte Methoden können nicht abstrakt sein, da sie nicht überschrieben werden können.
- Eine Klasse kann auch als abstrakt deklariert werden, wenn sie keine abstrakten Methoden enthält, aber auf irgend eine andere (semantische) Weise unvollständig ist. Solche Klassen können ebenfalls nicht instantiiert werden.

Zusammenfassung der Gemeinsamkeiten verwandter Klassen in einer gemeinsamen Superklasse wird als *Faktorisierung* bezeichnet.

Für jede Figur kann man Inhalt und Umfang berechnen sowie diese zeichnen, aber die Methoden nur für reale Figuren (Objekte der Subklassen) ausgeführen.

Instanzen der Superklasse können nicht gebildet werden, da einige Methoden nicht implementiert sind.

Variablen und Methodenparameter des Typs `Figur` können aber deklariert werden.

Instantiierung solcher Variablen und Aufruf solcher Methoden ist nur mit Objekten von konkreten Unterklassen möglich.

Mehrfachvererbung

Beispiel: `package shapes` implementiert auch Klassen `BCircle`, `BRechteck`, `BQuadrat`, `BDreieck`, `BEllipse`, die noch zusätzlich über einen Basispunkt verfügen, etwa um die Figur zu verschieben.

Idee:

```
abstract class BasePoint {
  double xBasis, yBasis;
  public abstract double getXBase();
  public abstract double getYBase();
  public abstract void setBasePoint(double x, double y);
}
```

Problem: Klassen können in Java nicht von mehreren Basisklassen erben, d.h. `class BCircle extends Figur, BasePoint` ist nicht möglich.

Grund:

```
class B { ... f(); .... }
class K1 extends B { int d; ... f() { ...} .... }
class K2 extends B { int d; ... f() { ...} .... }

class K extends K1, K2 { ... }

K v=new K();
v.d;           // Welches d ?
v.f();         // Welches f() ?
```

Lösung 1:

```
abstract class FigureWithBasePoint extends Figure {
  double xBasis, yBasis;
  public abstract double getXBase();
  public abstract double getYBase();
  public abstract void setBasePoint(double x, double y);
}

class BCircle extends FigurWithBasePoint { ...  }
class BRechteck extends FigurWithBasePoint { ... }
```


Schnittstellen

Lösung 2, wenn sich die Vererbungshierarchie nicht linearisieren lässt:

```
public interface BasePoint {
  public abstract double getXBase();
  public abstract double getYBase();
  public abstract void setBasePoint(double x, double y);
}

public interface Figur {
  public abstract double inhalt();
  public abstract double umfang();
  public abstract void draw();
}

class BCircle implements Figur, BasePoint { ...  }
class BRechteck implements Figur, BasePoint { ... }
```

Regeln zum Einsatz von Schnittstellen

- Schnittstellen sind spezielle abstrakte Klassen.
- Schnittstellen enthalten nur abstrakte Methodendeklarationen (also keine Klassenmethoden, da diese nicht abstrakt sein können).
- Alle Methoden sind implizit `abstract` und `public`.
- Schnittstellen dürfen keine Felddefinitionen enthalten. (Ausnahme: Konstantenfelder, die `static final` sind)
- Eine Schnittstelle kann nicht instantiiert werden und hat deshalb auch keinen Konstruktor.
- Eine Klasse kann mehrere Schnittstellen implementieren.

Schnittstellen ohne Methodendefinitionen heißen **Marker-Schnittstellen** (Bsp.: `Cloneable`, `Serializable`)

bzw. **Konstanten-Schnittstellen**, wenn sie Konstanten definieren (Bsp.: `java.io.ObjectStreamConstants`).

Schnittstellen und abstrakte Klassen im Vergleich

- Schnittstellen können von jeder Klasse implementiert werden, auch außerhalb von Vererbungs-Hierarchien.
- Klassen können mehrere Schittstellen implementieren, aber nur eine abstrakte Klasse erweitern.
- Konkrete Klassen müssen immer alle Methoden der Schnittstelle (neu) implementieren, auch wenn sich der Code wiederholt. Abstrakte Klassen können Default-Implementierungen enthalten.

  Jedoch können einige Default-Implementierungen in einer abstrakten Begleitklasse zusammengefasst werden (geht aber nur für je ein Interface, da konkrete Klassen von dieser abstrakten Klasse erben)

Beispiel:

```
public interface RectangularShape {
    /* Definition der Schnittstelle */
}

public abstract class AbstractRectangularShape
    implements RectangularShape {
    /* einige Default-Implementierungen */
}
```

- Verändern oder Erweitern der Schnittstelle macht alle implementierenden Klassen unbrauchbar (neue abstrakte Methoden!). Abstrakte Klassen können ohne Auswirkung auf abgeleitete Klassen um nicht-abstrakte Methoden erweitert werden.

4.3. Generisches Programmieren

Beispiel: Der Datentyp Stack

Hausaufgabe 3-4 (`Stack_double.java`)

```
public class Stack_double {
    private double[] val;
    private int sp;

    Stack_double(int n) { val=new double[n]; sp=0; }

    void push(double f) throws Exception {
        if (sp < val.length) val[sp++] = f;
        else throw new Exception("Fehler: Stack voll\n");
    }

    double pop() throws Exception {
        if (sp > 0) return val[--sp];
        else throw new Exception("Fehler: Stack leer\n");
    }

    boolean isEmpty() { return sp==0; }
}
```

Ähnlich Stack für BigInteger (`Stack_BI.java`)
und Long1 (`Stack_L.java`)

```
public class Stack_L {
    private Long1[] val;
    private int sp;

    Stack_L(int n) { val=new Long1[n]; sp=0; }

    void push(Long1 f) throws Exception {
        if (sp < val.length) val[sp++] = f;
        else throw new Exception("Fehler: Stack voll\n");
    }

    Long1 pop() throws Exception {
        if (sp > 0) return val[--sp];
        else throw new Exception("Fehler: Stack leer\n");
    }

    boolean isEmpty() { return sp==0; }
}
```


Ziele des generischen Programmierens

- Ein und derselbe Code soll für Daten verschiedener Typen verwendet werden.
- Die wesentliche Strukturverwandtschaft von Code über verschiedenen Datentypen soll offengelegt werden.
- Code-Redundanz, die insbesondere die Programmkonsistenz gefährdet und die Programmwartung erschwert, soll vermieden werden.
- Das Navigieren in höheren Datenstrukturen (Listen, Kollektionen, Bäumen, Hashes, Stacks) soll von den Elementtypen der gespeicherten Daten getrennt werden.

Generische Datenstrukturen

Ansatz: Höhere Datenstrukturen über verschiedenen Elementtypen werden durch *Vererbung* aus einer Datenstruktur über einem *generischen Elementtyp* abgeleitet, der alle denkbaren *konkreten Elementtypen* als Oberklasse überdeckt.

Als generischer Elementtyp wird dabei `Object` bzw. eine genau entworfene Schnittstelle oder (abstrakte) Klasse verwendet.

Die dort deklarierten bzw. implementierten *generischen (Instanz-) Methoden* werden in Anwendungen von den konkreten Methoden konkreter Instanzen überschrieben.

Beispiel: `GStack` als `Stack(Object)`

```
public class GStack {
    private Object[] val;
    private int sp;

    /* Konstruktor */
    GStack(int n) { val=new Object[n]; sp=0; }

    void push(Object f) ...
    Object pop() ...
    boolean isEmpty() ...

    // Für Ausgabezwecke noch zusätzlich

    public String toString() ...
}
```

Realisierung siehe `GStack.java`.

Beispiel: Generische Listen

```
public class GList0 {
    // Hilfsklasse als innere Klasse
    static protected class GNode {
        Object data; GNode next;
        GNode(Object a, GNode n) { data=a; next=n; }
    }

    // Instanzfelder
    protected GNode head;
    protected int length;

    // Hilfsfunktion zur Laengenbestimmung.
    protected int length() { ... }

    // Konstruktoren
    GList0() { head=null; length=0; }
    GList0(Object[] v) { ... }

    // Ueberschriebene Methode aus Object
    public String toString() { ... }
}
```

Realisierung siehe `GList0.java`

Beispiel: `GStack` als `Stack(Object)`

```
public class GStack {
    private Object[] val;
    private int sp;

    /* Konstruktor */
    GStack(int n) { val=new Object[n]; sp=0; }

    void push(Object f) ...
    Object pop() ...
    boolean isEmpty() ...

    // Für Ausgabezwecke noch zusätzlich

    public String toString() ...
}
```

Realisierung siehe `GStack.java`.

Beispiel: Generische Listen

```
public class GList0 {
    // Hilfsklasse als innere Klasse
    static protected class GNode {
        Object data; GNode next;
        GNode(Object a, GNode n) { data=a; next=n; }
    }

    // Instanzfelder
    protected GNode head;
    protected int length;

    // Hilfsfunktion zur Laengenbestimmung.
    protected int length() { ... }

    // Konstruktoren
    GList0() { head=null; length=0; }
    GList0(Object[] v) { ... }

    // Ueberschriebene Methode aus Object
    public String toString() { ... }
}
```

Realisierung siehe `GList0.java`

Eigenschaften:

- Eine solche generische Datenstruktur kann Objekte aller Art aufnehmen, die Instanzen von `Object` sind (also `BigInteger`, `Long1` usw., aber **nicht** `double`).

  Nützlich für *Containerklassen*, etwa bei Graphikobjekten.
- Kann auch Instanzen einer Klasse aufnehmen, die zur Compilezeit der generischen Datenstruktur noch gar nicht existiert, wenn sie nur eine Unterklasse von (in diesem Fall) `Object` ist.
- Instanzen der generischen Datenstruktur sind nicht notwendig homogen, d.h. können `BigInteger`, `Long1`, `Float` **nebeneinander** enthalten.
- Aufweichung der strengen Typprüfung zur Compilezeit (jede Instanz von `Object` geht durch). Typprüfung muss notfalls zur **Laufzeit** durch den Nutzer selbst hinzugefügt werden.

  Beispiel: `GStackTypeTest.java` in der Version 1

Details:

```
void push(Object f) throws ArrayStoreException {
    if (sp < val.length) val[sp++] = f;
    else throw new ArrayStoreException("Fehler: Stack voll\n");
}
```

Parametertyp ist `Object`, d.h. beim Aufruf der Methode erfolgt eine Typkonvertierung (Typaufweitung, ungefährlich).

```
Object pop() throws ArrayStoreException {
    if (sp > 0) return val[--sp];
    else throw new ArrayStoreException("Fehler: Stack leer\n");
}
```

Rückgabetyp ist `Object`, d.h. bei der weiteren Verwendung des Rückgabewerts ist eine Typkonvertierung (explizit anzugebende Typeinschränkung) erforderlich.

Beispiel: GTR – Taschenrechner mit `GStack` und `BigInteger`

```
public class GTR {

   GStack s=new GStack(100);

   BigInteger calculate(String[] v) throws Exception {
      BigInteger op1,op2;
      int exp;
      for(int i=0;i<v.length;i++) {
         if (v[i].equals("+")) {
            s.push(((BigInteger) s.pop()).add((BigInteger) s.pop()));
         }
...
```

`s.pop()` gibt eine Instanz von `Object` zurück, `((BigInteger) s.pop())` macht daraus (wenn möglich) eine Instanz von `BigInteger`, deren Methode `add(..)` mit dem zweiten Argument aufgerufen wird, das dazu auch in den Typ `BigInteger` konvertiert werden muss.

Die (sauber geschriebene) Applikation stellt selbst sicher, dass alle im Stack gespeicherten Objekte vom Typ `BigInteger` sind.

Generische Methoden

Generische Methoden sind Methoden, die auf Methoden der generischen Elementklasse zurückgreifen. Diese Methoden sind für konkrete Elementklassen überschrieben. In Java können das also nur Instanzmethoden sein.

Beispiel: `toString()` in `GStack`

```
  private Object[] val; ...
  public String toString() { ...
     for(int i=0; i<sp; i++) sb.append(val[i].toString()+"|");
...  }
```

Methoden der generischen Elementklasse können `abstract` sein, ohne dass die definierte Methode der generischen Klasse `abstract` ist.

Beispiel: Vergleich zweier Listen. Implementierung einer generischen Methode `equals()` in der Klasse `GListCompare` (die ihrerseits `equals()` der Klasse `Object` überschreibt).

```
public class GListCompare extends GList0 {
...
    public boolean equals(Object o) {
        GListCompare o1=(GListCompare) o;
        if(length!=o1.length) return false;
        GNode x=head,y=o1.head;
        while( x!=null ) {
            if(!x.data.equals(y.data)) return false;
            x=x.next; y=y.next;
        }
        return true;
    }
}
```

Zur Laufzeit ist das generische `x.data.equals(..)` der Klasse `Object` durch das konkrete `equals(..)` der (genauen) Klasse des konkreten Objekts `x.data` überschrieben.

Generische Klassen mit neuem Elementtyp verwenden

Generische Klassen können Elementtypen verwenden, die erst nach der Compilezeit der generischen Klasse angelegt wurden.

Beispiel: Stack und Listen von Objekten der (neuen) Klasse `Point`

Spezifikation: Point hat Koordinaten (x, y) vom Typ `int`. Es soll die Möglichkeit vorgesehen werden, Punkte mit zufälligen Koordinaten zu erzeugen.

Siehe `Point.java`, `PointTest.java`

Generische Klassen und Typprüfung zur Laufzeit

Qualität der Eingabe + Qualität der Implementierung
= Qualität der Ausgabe

Deshalb muss nur die Eingabequalität (also `push(...)`) überwacht werden.

```
public class GStack1 extends GStack {
    private Class derGStackType=null;

    // Neuer Konstruktor
    GStack1(int n, String type) { .. }

    // Typtest-Methode
    protected final void typeTest(Object f) {
        if(f.getClass()!=derGStackType) ..
    }

    // push wird überschrieben
    void push(Object f) throws ArrayStoreException { .. }
}
```

Siehe `GStack1.java` und `GStackTypeTest.java` Version 2

Generische Klassen und generische Algorithmen

Generische Klassen werden auch zur Zusammenfassung generischer Algorithmen in „Pakete“ verwendet. Die Algorithmen werden dabei als Klassenmethoden vereinbart, generische Klasse hat also keine Objekte. Klasse wird oft auch als `final` deklariert, womit das Paket nicht modifiziert werden kann.

Beispiel: Paket von Array-Funktionen

```
public class ArrayFunctions {

    static Comparable min(Comparable[] F) {
        if(F.length==0)
            throw new RuntimeException("Feld ohne Elemente");
        Comparable o=F[0];
        for(int i=1;i<F.length;i++)
            if(o.compareTo(F[i])==1) o=F[i];
        return o;
    }
}
```


Weitere Funktionen in dem Paket etwa

- `void sort(Comparable[] F)` zum Sortieren eines Felds (Hausaufgabe 5-2)
- Verschiedene Sortierverfahren zum Vergleich
- Service-Funktionen

Dazu muss die Elementklasse die Schnittstelle `Comparable` implementieren. Siehe `Point1` als Erweiterung von `Point`.

Anwendung in `ArrayFunctionsTest`.

Siehe auch die Java-Klasse `Arrays`.

Einsatz des Konzepts generischer Klassen in Java

Java stellt in seiner Klassenbibliothek eine ganze Reihe generischer Klassen zur Verfügung.

Etwa die verschiedenen Containerklassen in `java.util`

> Set, List, Map, SortedSet, SortedMap, HashSet, TreeSet, ArrayList, LinkedList, Vector, Collections, Arrays, AbstractCollection, ...

Die verschiedenen Containerklassen spannen eine verzweigte Vererbungshierarchie mit generischen Klassen und Schnittstellen, abstrakten Methoden, Defaultimplementierungen usw. auf.

Beispiel:

Alle Klassen implementieren das **`interface Collection`** (Wurzel der Container-Klassenhierarchie).

Defaultimplementierungen einiger Methoden sind in der Begleitklasse **`abstract class AbstractCollection`** enthalten: Alle Methodendefinitionen werden im Wesentlichen auf eine (abstrakte) Methode **`iterate()`** zurückgeführt, die für konkrete Subklassen zu implementieren (überschreiben) ist.

Siehe Java-Dokumentation.

Das geht im Prinzip so:

```
for(Iterator i=s.iterator(); i.hasNext(); ) { ... i.next(); ... }
```

Grenzen des Konzepts generischer Klassen in Java

Exemplarisch am Beispiel Taschenrechner.

Frage: Kann man einen generischen Grundbereich installieren, der `Long1`, `BigInteger`, `Rational`, ... verallgemeinert?

Gemeinsamkeit: Begriff des *Rings*. Operationen $+, -, *$, spezielle Elemente 0 und 1, Inversenbildung eingeschränkt ausführbar.

Auf dem Weg zu einer generischen Klasse `Ring`:

1. Frage: `a+b` als `add(a,b)` oder als `a.add(b)`

2. Frage: Beziehung zwischen generischer und konkreter Methode

```
Ring add(Ring)
BigInteger add(BigInteger)
```


3. Frage: Konstanten `ONE`, `ZERO`

    Als Attribut `Ring ONE` – Instantiierung erforderlich

Konstanten als nullstellige Funktionen

    `Ring one() ...` – gut, aber wäre eine Klassenmethode

Was kann in eine Klasse `Ring` als Schnittstelle eingetragen werden?

```
interface Ring {
    public Ring add(Ring b);
    public Ring negate();
    public Ring subtract(Ring a);
    public Ring multiply(Ring b);
    public Ring inverse();
    public Ring divide(Ring a);
    public Ring pow(int exp);
}
```

Welche Default-Implementierungen kann eine Klasse `AbstractRing` als Begleitklasse der Schnittstelle `Ring` enthalten?

```
abstract public Ring add(Ring b);
abstract public Ring negate();

public Ring subtract(Ring a) { return add(a.negate()); }
```

Die folgende Lösung mit nullstelliger Klassenfunktion geht nicht, da Klassenfunktionen *sofort* implementiert werden müssen und nicht überschrieben werden können.

```
    abstract static Ring one();
    abstract static Ring zero();
...
    public Ring pow(int exp) {
        if (exp<0) return inverse().pow(-exp);
        Ring a=one();
        while(exp-->0) a=a.multiply(this);
        return a;
    }
```


Deshalb Verwendung von nicht initialisierten Klassenvariablen (leider nicht als `abstract` deklarierbar. Diese müssen für *konkrete* Ringklassen in einem Klassen-Initialisierungsblock angelegt werden, der während des Ladens der *Klasse* ausgeführt wird. Damit stehen die erforderlichen Instanzen der Klassenvariablen zur Laufzeit zur Verfügung.

```
abstract public class AbstractRing implements Ring {
    protected static Ring ONE, ZERO;
...
    public Ring pow(int exp) {
        if (exp<0) return inverse().pow(-exp);
        Ring a=ONE;
        while(exp-->0) a=a.multiply(this);
        return a;
    }
```

Einbettung in einen generischen Taschenrechner GTR1

Problem: GTR-Eingabe ist “String[]”. Woher weiß der GTR, in welche Zahlbereichsklasse umzuwandeln ist?

“konkrete Lösung”: Konstruktor `Long1(String)`

Anforderungen an eine generische Lösung:

(1) Zahlbereichsklasse muss beim Generieren einer GTR-Instanz mit angegeben und zugehörige Klasse gefunden werden (zur Laufzeit!).

(2) Konstruktoraufruf dieser (zur Übersetzungszeit nicht bekannten) Klasse muss zur Laufzeit abgesetzt werden.

Lösung: Geeignete Methoden aus `Class` verwenden.

(1) `c=Class.forName(String)` sucht vorcompilierte Klasse an Hand des Namens und lädt diese. Ausnahmebehandlung!

Einbau in den Konstruktor von GTR1.

```
private Class c;
GTR1(String s) {
    try { c=Class.forName(s);
        System.out.println("Instanz von GTR1 ueber "+c+" erzeugt");
    }
    catch (Exception e) {
        System.out.println("Klasse "+s+" nicht gefunden");
        throw new RuntimeException(e.getMessage());
    }
}
```

(2) `o=c.newInstance()` ruft den nullstelligen anonymen Konstruktor der Klasse c auf. Der stellt im Normalfall nur Speicherplatz zur Verfügung.

Initialisierung des Zahlbereichsobjekts o aus dem String s durch `o.fromString(s)`.

`Object` hat leider keine solche Methode. Syntax der Schnittstellenklasse `Ring` muss erweitert werden.

```
try {
    Object o=c.newInstance();
    s.push(((Ring) o).fromString(v[i]));
} catch(Exception e) { throw new RuntimeException(e.getMessage()); }
```

Gesamte Lösung: `GTR1.java`

Implementierung einer konkreten Klasse `Long2`

```
public class Long2 extends AbstractRing {
    private long value;

    /* Konstruktoren */
    Long2() { } // wird für newInstance() benötigt !
    Long2(long n) { value=n; } // wie in Long1
    Long2(String s) { value=new Long(s).longValue(); }
...
```

Methoden: nicht `Long2 add(Long2)`, sondern `Ring add(Ring)` !

```
...
public Ring add(Ring a0) {
    Long2 a=(Long2) a0;
    return new Long2(value+a.value);
}
...
```

Vielleicht noch Test (`a0 instanceof Long2`), aber sonst wird sowieso eine `ClassCastException` ausgelöst.

```
public Ring divide(Ring a0) {
    Long2 a=(Long2) a0;
    return new Long2(value/a.value);
}
```

Default aus `AbstractRing` wird überschrieben, da im Bereich $\mathbf{Z}$ nicht einfach $a/b = a \cdot 1/b$ gilt (Beispiel: $3/2 = 1$ aber $1/2 = 0$).

Initialisierung der beiden Klassenvariablen `ONE` und `ZERO` mit `Long2`-Instanzen (Subklasse von `Ring`):

```
static { ONE=new Long2(1); ZERO=new Long2(0); }
```

Siehe `Long2.java` und `GTR1Test.java`

4.4. Templates

Ansatz: Erzeuge einen Prototyp (*Klassen-Template*) `Stack`, aus dem sich die speziellen Klassen `Stack<BigInteger>`, `Stack<Long1>` usw. als *Klassen-Instanzen* erzeugen lassen.

Dieser Ansatz verallgemeinert das Konzept der Funktion bzw. Prozedur (Aufruf ein und desselben Codeteils mit unterschiedlichen Parametern) zu einem Konzept von *Datentypkonstruktoren*.

Klassenbezeichnungen sind damit nicht allein durch Typ*bezeichner*, sondern auch durch Typ*ausdrücke* möglich. Die **Sprache der Typausdrücke** ist eine Ergänzung oder Erweiterung der Programmiersprache.

Java: Klassen sind über die `class`-Datei eng an ihren *Namen* gebunden. Klassen-Prototypen gibt es in Java deshalb (bisher) nicht.

Besonderheiten des Konzepts

Unterscheide wie für Funktionen zwischen *Definition* und *Aufruf* eines solchen Datentypkonstruktors.

Definition: Es werden der Name des *Klassenprototyps* (etwa `Stack`), formale Aufrufparameter (Deklaration) sowie Erzeugungsverfahren (Implementierung) vereinbart. Die formalen Aufrufparameter sind Platzhalter für konkrete Klasseninstanzen, also Objekte vom (fiktiven) Typ *Type*.

Aufruf: Konkrete Klasseninstanzen werden aus dem Prototyp hergestellt, indem die Aufrufparameter mit speziellen Klassennamen (allgemein: Typausdrücken) belegt werden und das Erzeugungsverfahren abgearbeitet wird.

Das von Klasse – Objekt bekannte Bild von Stempel und Prägung wiederholt sich auf der Ebene Prototyp – Klasse.

Jeder (korrekte) Typausdruck wertet zu einer Klasseninstanz aus. In statisch getypten Sprachen muss diese Auswertung vom Compiler vorgenommen werden.

C++ - Templates

Definition:

```
template<class T>
class GStack {
    T* val; int len; int sp;

public:
    /* Konstruktor */
    GStack(int n) { val=new T[len=n]; sp=0; }

    /* Methoden */
    void push(T f) { if (sp < len) val[sp++] = f; }
    T pop() { if (sp > 0) return val[--sp]; }
};
```

Aufruf: Spezielle Stack-Objekte erzeugen

```
GStack<char> s1(10);
GStack<double> s2(100);
```

Erläuterungen:

- `template<class T>` vereinbart für die folgende Definition T als Argument vom Typ *Type*.
- In der Definition des *Klassenprototyps* `GStack` wird T als Platzhalter für einen konkreten Typ(ausdruck) verwendet.

  Analogie zu Funktionsdefinition und Wertparameter.
- Aufruf `GStack<char>` erzeugt (zur Compilezeit) eine *Klasseninstanz*, in der $[T = char]$ gesetzt ist und vereinbart (und initialisiert) $s1$ als Instanz dieser Klasse.

  Analogie zu Funktionsaufruf.
- Klasseninstanzen `GStack<int>` und `GStack<char>` sind bereits compilerseitig *homogene* Datenstrukturen.

Ergänze `GStack` um Methoden `isMember` und `print`:

```
int isMember(T o) {
  for(int i=0;i<sp;i++) if(val[i].equals(o)) return 1;
  return 0;
}

void print() {
  cout << "|$|";
  for(int i=0;i<sp;i++) cout << val[i].getValue() << "|";
  cout << '\b' << ">\n";
}
```

Ist der Code syntaktisch korrekt? Unklar, weil nicht bekannt ist, ob T die Methoden `int equals(T)` und `?? getValue()` bereitstellt. Das kann erst für einen konkreten Parameterwert T_0, also eine Klasseninstanz `GStack<`T_0`>` entschieden werden.

Gute Regel: Schnittstellendefinition von T als Kommentar in der Template-Implementierung angeben.

Templates und Methodenbindung

- In `GStack<`T_0`>` sind `equals` und `getValue` durch Methoden der als Parameter übergebenen Klasse T_0 zu binden.

  Für verschiedene T_0 sind das Methoden aus verschiedenen Klassen, die in keinerlei Vererbungsrelation stehen. Also liegt weder Überladen noch Überschreiben vor!

- Typprüfung, ob Methoden passender Signatur vorhanden sind, ist erst auf der Klasseninstanz möglich, da vorher die erforderliche Information fehlt.

- C++ erzeugt für jede Klasseninstanz einen eigenen Codeblock, der dann wie klassischer Quellcode behandelt wird. C++-Templates sind in diesem Sinne Makros.

  Vorteil: die Methoden von T_0 können auch statisch gebunden werden.

  Nachteil: der Code wird aufgebläht (code bloat).

Beispiel: Generischer Taschenrechner `GTR<T>`

Siehe `GTR.cc`

- T=Typparameter für den Zahlbereichstyp des Taschenrechners.

- Typecasts **nicht** erforderlich, da T die erforderliche Signatur „mitbringt“.

  Überprüfung erfolgt (statisch) erst auf der jeweils erzeugten Klasseninstanz `GTR<Int>`. Parameterklasse `Int` muss die erforderliche Signatur aufweisen.

  Siehe `Int.cc` und `GTRTest.cc`.

- Da nicht überladen oder überschrieben wird, kann auch auf Klassenmethoden (und sogar Felder) des Parametertyps Bezug genommen werden.

  Siehe `GTR1.cc`, `Int1.cc` und `GTR1Test.cc`.

Templates in C++

Templates spielen in C++ eine ähnlich zentrale Rolle als Abstraktionskonzept wie generische Klassen in Java.

Wichtigste Anwendung: die Standard Template Library (STL), die (wie `java.util`) Containerklassen, Algorithmen und Iteratoren enthält. Allerdings sind die Containerklassen homogen.

Wie in Java wird das Konzept des Iterators verwendet, um Containerklassen und Algorithmenklassen zu trennen. Iteratoren verallgemeinern die aus C bekannte Pointerarithmetik. Wegen der Homogenität ist die Bitgröße der einzelnen Elemente bekannt.

Siehe `http://www.sgi.com/tech/stl`

Zusammenfassung

- Templates werden zur Definition von Klassenprototypen eingesetzt.
- Klassenprototypen sind (in C++) Makros, aus denen zur Compilezeit Klasseninstanzen erzeugt werden.
- Dieser Mechanismus führt bereits compilerseitig zu homogenen Behälterklassen (Stacks, Listen, Bäume, . . . enthalten nur Elemente vorgegebenen Typs).
- Typprüfung der Methoden aus den Parameterklassen erfolgt erst auf den Klasseninstanzen. Die Bindung kann statisch erfolgen.
- Parametermethoden sind weder überschrieben noch überladen.

4.5. Parametrisierte Datentypen

Im Klassenbegriff überlagern sich zwei Abstraktionskonzepte der Informatik: Modularisierung und Typkonzept

Modularisierung

Abstraktionskonzept der „Programmierung im Großen“, mit dem Teile eines Programms zu Einheiten zusammengefasst werden, auf die nur über Schnittstellen zugegriffen werden kann.

Einzelheiten der Implementierung unterliegen (aus verschiedene Gründen) dem Geheimnisprinzip.

Modul-Implementierungen sind untereinander austauschbar, wenn sie nur dieselbe Schnittstelle bereitstellen (Schnittstellenkontrakt).

Aufteilung in **Schnittstelle** (interface) als öffentlichem Teil und **Implementierung** (implementation) als privatem Teil.

Pascal-Unit	`interface ... implements ... end.`
Modula-2	`module specification M; ... end.` `module implementation M; ... end.`
Ada	`package M is ... end M;` `package body M is ... end M;`
C und C++	Header- und Implementationsdateien (`*.h` und `*.cc`)
Java	Konzept der Trennung von Schnittstelle und Implementierung wird nicht unterstützt

Klassen: Schnittstelle und Implementierung

Klasse = Klassenname + Menge von Attributen und Methoden $m_1, \ldots, m_n$ (Oberbegriff: **Merkmale**)

Schnittstelle der Klasse = *Deklaration* der Merkmale

Attribute: Angabe von Name und Typ (Klassenname)
Methoden: Angabe von Name und Signatur

Schnittstelleninformation = Versprechen, dass an der beschriebenen Stelle zur Bindezeit etwas Vernünftiges steht.

Implementierung einer Klasse = *Definition* der Merkmale

Hier werden die Versprechen eingelöst, wobei auf die Einlösung der Versprechen anderer Klassen Bezug genommen wird.

Beispiel: `GStack.cc`

Klassisch: Quelltext einer Klasse übersetzen

Klassen werden gewöhnlich getrennt übersetzt (Effizienz, Möglichkeit zu Klassenbibliotheken).

Im Programm wirken dann mehrere solche übersetzten Klassen zusammen, deren Kontrollflüsse koordiniert werden müssen.

(1) Parsen des Quelltextes (lexikalische Analyse): Prüfen auf Einhaltung des Sprachstandards und Zerlegung in einzelne **Token**.

(2) Übertragung in die Zielsprache (Objektcode).

- Syntaxanalyse, Semantikverarbeitung
- einzelnen Token werden **Bedeutungen** (Konstrukte der Zielsprache) zugeordnet
- Organisation des Kontrollflusses (Datenflussanalyse)
- Bezug auf Merkmale anderer Klassen (über das Versenden von Botschaften)

Bezug auf andere Klassen legt Austritts- und Eintrittspunkte des Kontrollflusses fest, deren Beschreibung in einem geeigneten Format in den Objektcode einzufügen ist.

Konsistenz der eigenen Versprechen und Plausibilität des Verhaltens fremder Klassen wird vom **Typsystem** überprüft. Dazu werden die zugesicherten Eigenschaften der fremden Klassen (also deren Klassenschnittstellen) herangezogen.

Kommunikation mit fremder Klasse erfolgt also über deren **Schnittstelle**. Beziehung zwischen Schnittstelle (= Versprechen) und Implementierung (= Einlösung des Versprechens, **Binden**) wird nach einem im Objektcode-Sprachstandard festgelegten (also fest vorgegebenen) Verfahren realisiert.

Überladen: Existenz mehrerer unterschiedlich deklarierter Merkmale gleichen Namens (ist eine Eigenschaft der Schnittstelle)

Überschreiben: Binden desselben Merkmals an unterschiedliche Implementierungen (ist eine Eigenschaft des Binde-Verfahrens)

Zusammenfassung: Welche Information muss der Objektcode der übersetzten Klasse enthalten?

- Liste offener Referenzen auf andere Klassen mit Spezifikation der von dort einzubringenden fremden Methoden bzw. Attribute.
- Folglich auch Information über die eigene Klassenschnittstelle, insbesondere die Spezifikationen und Eintrittspunkte der einzelnen Methoden dieser Klasse.

Java

- `class`-Datei enthält Schnittstelle und Implementierung in einem. Beim Übersetzen der Klasse wird also das Schnittstellen-Versprechen verifiziert.
- Um eine Klasse zu übersetzen werden `class`-Dateien aller fremden Klassen benötigt (Schnittstelleninformation).
- Java-Projekte übersetzen dazu ggf. automatisch alle im Quelltext genannten fremden Klassen.
- Dynamisch geladene Klassen (etwa `Long2` in `GTR1Test`) sind davon nicht erfasst.
- Endgültige Bindung erfolgt zur Laufzeit: JVM lädt alle `class`-Dateien und interpretiert den Bytecode. `s.call(...)` wird realisiert als (fest eingebundener) Aufruf

  ```
  s.getClass().Method[*].invoke(s,...)
  ```

  Das ist genau der Mechanismus des dynamischen Bindens.

C++

- Interface und Implementierung sind normalerweise getrennt (Header- und Objektdatei). Die enge Verbindung zwischen Versprechen und Einlösung ist damit nicht gegeben.

  Beispiel: `GStack.cc`
- Zur Erstellung einer Objektdatei werden nur die Schnittstellen (Header) der fremden Klassen verwendet.
- Abhängige Klassen werden nicht automatisch übersetzt, sondern müssen in Projekten verwaltet werden (GNU-Werkzeug 'make').
- Endgültige Bindung zu Maschinencode erfolgt in Phase 2 des Compilers. Dabei kann es geschehen, dass uneingelöste Schnittstellenversprechen entdeckt werden.
- Maschinencode enthält keine Klasseninformation mehr. Methoden werden normalerweise statisch gebunden.

  Für dynamisch gebundene (virtuelle) Methoden muss eine VMT der Klasse in den Code eingebunden werden.

Schnittstellen und Vererbung

`class A extends B` $\Longleftrightarrow$ B-Schnittstelle ist (als Menge) Teilmenge von A-Schnittstelle

Kommunikation mit Instanzen der Klasse B als A-Instanzen ist damit trivial: der Zuordnungsmechanismus

$$\text{Schnittstelle} \longrightarrow \text{Implementierung}$$

ist auf eine Teilmenge einzuschränken.

Problem: Java erkennt Klassen an ihrem Namen, nicht an der Schnittstelle selbst. Schlüsselwort `extends` erforderlich.

Template + Typprüfung = parametrisierter Datentyp

Problem: Prototypdefinition `GTR<T>` kann vom Typsystem nicht auf die beschriebene Weise bearbeitet werden, weil T ohne „Versprechen“ kommt.

Lösung: T ist selbst ein „Typ“ (genauer: **Typklasse**) zuzuordnen, womit das erforderliche „Versprechen“ artikuliert werden kann.

Ansatz bewegt sich auf höherem Abstraktionsniveau als das bisherige Typsystem:

- **Klassen** garantieren Eigenschaften von **Objekten**.
- **Typklassen** garantieren Eigenschaften von **Klassen**.

Beispiel: `GTR<T::Ring>::GTRInterface`

`GTR`	Name des **Datentypkonstruktors**
`T`	formaler Typparameter, der in der Templatedefinition von `GTR` verwendet wird
`::`	Erweiterung des Typsystems um Typklassenversprechen

`Ring` und `GTRInterface` sind **Typklassenbezeichner**.

Eine **Typklasse** charakterisiert die Eigenschaft einer Klasse, eine bestimmte Klassenschnittstelle zu implementieren.

Java: Genau das sagt `...class A implements I...` aus. Allerdings ist `interface I` selbst eine Java-Klasse (mit `class`-Datei) und geht als Oberklasse von A durch.

Statt dessen: Klasse A ist (Typklassen)-Instanz von I.

Dazu ist eine **Erweiterung des Konzepts des Typsystems** erforderlich:

bisher: Jedem Objekt(bezeichner) und jedem Ausdruck sind eindeutige Typen (Klassennamen) zugeordnet, deren Konsistenz beim Übersetzen von Quelltext geprüft werden kann.

neu: Auch jedem Klassenbezeichner und Typausdruck sind eindeutige Typklassen (Klassenschnittstellen-Versprechen) zugeordnet, deren Konsistenz beim Übersetzen von Templates geprüft werden kann.

Templates sind damit (typ)-„sicher". Beim Erzeugen von Klasseninstanzen aus Templates muss nur noch die Konsistenz der Aufrufparameter (Typklasseninstanz-Eigenschaft) geprüft werden.

Vorteile:

- Templates können bereits zur Definitionszeit automatisch auf Typsicherheit geprüft werden.
- Template-Implementierungen müssen nicht vollständig kopiert werden, sondern können intelligenter zu Klasseninstanzen gebunden werden.

Nachteil:

- Das Typsystem wird beim Auswerten von Typausdrücken um neue Typen erweitert. Ist nun einen ganze Typ**sprache**.

  In statisch getypten Sprachen werden Typen **im** Compiler ausgewertet. Der Compiler muss diesen Teil der Sprache also nicht nur interpretieren, sondern auch verarbeiten.

  Der Compiler ist also um eine ganz wesentliche neue Komponente zu erweitern.

bisher: Typsystem ist syntaktisch einfaches Hilfsmittel, um den vom Compiler verarbeiteten Programm-Quelltext zu „verstehen“.

neu: Typsystem wird zu einer eigenen Sprache, die entweder zusammen mit dem Programm-Quelltext (dynamisch getypt) oder bei der Analyse des Programm-Quelltexts (statisch getypt) zu „verstehen“ ist.

Kategorielle Programmiersprachen

Konsequente Trennung der Ebenen Typsystem und Programmquelltext mit folgenden Bezeichnungen

Typklasse	Category
Typ / Klasse	Domain
Objekte	Domain-Elemente

Beispiel: MuPAD (`www.mupad.de`), Aldor (`www.aldor.org`)

Kategorielle Programmiersprachen unterscheiden zwischen Konstruktoren für Objekte und Datentypkonstruktoren (Domainkonstruktoren).

Problem: Nicht `GTRInterface` sondern

$$\texttt{GTRInterface<T> == } \{ \texttt{ calculate:Array<String>} \mapsto \texttt{T } \}$$

Kategorielle Programmiersprachen benötigen also auch Typklassen-Konstruktoren.

Beispiel (MuPAD):

`Dom::Matrix(R)`: Domain-Konstruktor für Matrizen über einem Grundbereich R der Category `Cat::Rng`.

Konstruierter Domain gehört zur Category `Cat::Matrix(R)`, die mit dem Typklassenkonstruktor `Cat::Matrix` erzeugt wurde.

Beispiel MuPAD:

```
r:=random(10);
Mat:=Dom::Matrix(Dom::Integer);
A:=Mat(3,3,(i,j)->r());
linalg::det(A);
A^3;
7*A;
A^(-1);
A::dom;

RMat:=Dom::Matrix(Dom::Rational);
R:=RMat(A);
R^(-1);

MMat:=Dom::Matrix(Dom::IntegerMod(2));
M:=MMat(A);
MInvers:=M^(-1);
M*MInvers;
expr(%);
```

Typklassen – die Lösung ?

Beispiel: Datentypkonstruktoren `Polynomial` und `Fraction` entsprechen folgenden mathematischen Konzepten:

R Integritätsbereich. Dann ist $Q(R)$ der Quotientenkörper und $R[x]$ der Ring der Polynome in x mit Koeffizienten aus R.

$f = 2 \cdot x + 3$ Typ: `Polynomial(Integer)`
$g = 1/2$ Typ: `Fraction(Integer)`

Welchen Typ hat $f + g$? Kann der Compiler das ggf. selbst herausfinden (Typinferenzproblem)?

Alternativen: `Polynomial(Fraction(Integer))` oder `Fraction(Polynomial(Integer))`

Derartige Probleme führen schnell an Fragen der prinzipiellen Entscheidbarkeit heran und markieren die theoretische Front, an der heute in der **Theorie der Programmiersprachen** gearbeitet wird.