Objekte enthalten: lokale Variablen [→ Zustand], lokale Prozeduren [→ Methoden].

Zugriffsbefugnisse über „public“ [→ jeder], „private“ [→ nur objekteigene Methoden] oder „protected“ [→ nur eigene und unterklassen Methoden].

Vererbung von Basisklassen [„Oberklassen“ der vererbten] auf abgeleitete Klassen [„Unterklassen“ der Basisklassen]; dabei in abgeleiteter zusätzliche Variablen/Methoden zu denen der Basisklasse. U.u. Reimplementierung [Überschreiben] der Basisklassen-Methoden in Unterklassen nötig.

Polymorphie: Eigenschaft einer Variablen für Objekte verschiedener [→ abgeleiteter] Klassen stehen zu können.

Dynamisches Binden: Aufruf von reimplementierten Methoden von Unterklassen während der Laufzeit.

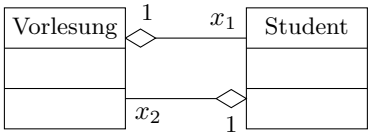
Abstrakte Klassen: Klassen mit mindestens einer definierten aber nicht implementierten Methode.

[Es gibt keine Objekte abstrakter Klassen!]

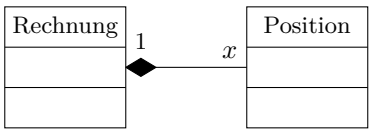
Phasen objektorientierter Programmierung: $\underbrace{\text{Analyse, Entwurf}}_{90\,\%}$, $\underbrace{\text{Implementation}}_{10\,\%}$ Zeitaufwand

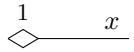
UML [Unified Modelling Language] ist eine standardisierte Sprache zur Beschreibung von Modellen mittels Diagrammen; z.B. *Klassendiagrammen* - beschreiben statische Beziehungen zwischen Klassen

Aggregation:

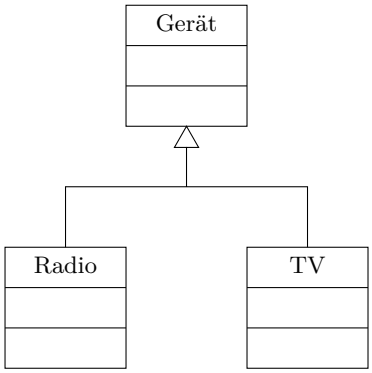


Komposition:



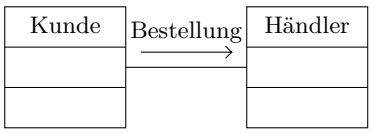
wobei  meint, dass 1 Objekt der Klasse links [hier...] x Elemente der Klasse rechts besitzt. Dabei ist auch $x = 1 \ldots *$ möglich [$*$ - beliebig groß]. Bei der Komposition sind die Einzelteile der Aggregation vom Aggregat existenzabhängig.

Vererbung:



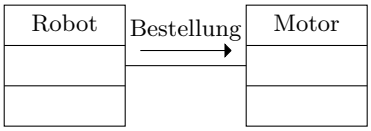
oder *Kollaborationsdiagramme* - beschreiben Kommunikation

Asynchron:



→ Senden und Empfangen zeitversetzt.

Synchron:

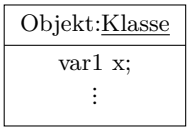


→ Senden und Empfangen nahezu gleichzeitig.

Zur Modellerstellung objektorientierte Analyse in Form grammatikalischer Inspektion.

[Substantiv → Objekt, Verb → Funktionalität, „hat“/„ist“ → Objektbeziehung]

Ein spezifisches Objekt einer Klasse:



.

Vererbung: Basis-/Oberklasse → abgeleiteter / Unterklasse

Überschreiben: Reimplementierung von Methoden der Oberklasse[n] in Unterklasse[n] [gleiche Signatur].

Überdecken: in Unterklasse[n] neue Variable[n] gleichen Namens definieren.

Überladen: gleichnamige Methode[n] in Unterklasse[n] mit anderer Signatur als in Oberklasse[n].

Hauptmethode: **public static void** main(String args[])

Sichtbarkeitsoperatoren: **public**, **private**, **protected**

Namenskonventionen:
- sprechende Bezeichner [CamelCase]
- Namen einacher Bezeichner vollständig klein [**boolean**, **byte**, **char**, **double**, **float**, **int**, **long**, **short**, **void**]
- Konstanten vollständig groß
- Klassennamen haben großen Anfangsbuchstaben
- Variablen, Methoden, Datenelemente haben kleine Anfangsbuchstaben
- Pakete werden vollständig klein geschrieben

If-Bedingung: **if** (...) { ... } **else** { ... }

Switch: **switch**(**int**/bool) {**case** 0: ...; **break**; **case** 1: ...;}
case 1: einzig Sprungadresse → **break**; oder **continue**!

While-Schleife: **while** (...) { ... } , do-Schleife: **do**{ ... }**while**(...);

For-Schleife: **for**(init ; beding; modif) { ... }

Logische Operatoren: & Und, | Oder, ^ exklusives Oder [XOR], ! Negation, mit Kurzschlussauswertung: $\begin{cases} \&\& \text{ Und} \\ || \text{ Oder} \end{cases}$.

Typüberprüfung: **instanceof** [z.B. **if**(var **instanceof** Integer)]

Konstanten mittels **final** [z.B. **final int** var = 1;]; bei Konstanten Wertzuweisung bei Deklaration nötig!

Java nutzt **Call by Value**; für Referenzen heißt dies jedoch nur, dass die Referenz kopiert wird.

2-er Komplement: $\underbrace{0}_{n}\ \underbrace{0}_{n-1}\ \ldots\ \underbrace{0}_{1}$ über $\underbrace{0}_{n}\ \underbrace{1}_{n-1}\ \ldots\ \underbrace{1}_{1}$, $\underbrace{1}_{n}\ \underbrace{0}_{n-1}\ \ldots\ \underbrace{0}_{1}$ bis $\underbrace{1}_{n}\ \underbrace{1}_{n-1}\ \ldots\ \underbrace{1}_{1}$ entspricht 0 über $2^{n-1}-1$, -2^{n-1} bis –1.

Bit-weise Operatoren: << [·2], >> [/2 - ganzzahlig, positiv], >>> [/2 - auch für negative 2-er-komplement-Zahlen]

Bei **Division durch Null**:
- für Ganzzahlen → Laufzeitfehler [ArithmeticException]
- Gleitkommazahlen → ±∞ [entsprechend ±0] oder NaN

Automatische Konvertierung von Primitivtypen: **byte** → **short** → **int** → **long** → **float** → **double** [und: **char** → **int**]

Links-/Rechtsassoziativität von Operatoren [meisten links-, Zuweisungsoperatoren rechts-assoziativ]

Garbage Collector [System.gc();] statt free (); [löscht Objekte ohne Referenzen auf sie].

Destruktor heißt finalize ();.

Referenz auf das aufrufende Objekt selbst [während der Laufzeit]: **this** .

```
public class Name {

    public int var1;
    private int var2;

    // one constructor
    public Name(int var1, int var2) {
        this.var1 = var1;
        this.var2 = var2;
    }

    // methods
    public int Method1(int newValue) {
        return(var2);
    }
}
```

Abbildung 1: Typisches Beispiel einer Klasse.

Keine aktive Speicherfreigabe [auch im Destruktor nicht]; stattdessen gibt der **Garbage Collector** [System.gc();] alle unreferenzierten Bereiche frei.

Variablenarten: **Instanzvariablen** [objektspezifisch], **Klassenvariablen** [global innerhalb einer Klasse, **static**].
[Konstanten sind auch Variablen.]

Methoden: **Instanzmeth.** [→ wirken auf jeweiliges Objekt], **Klassenmeth.** [wirken nur auf Klassenobjekt, **static**].
[Aufruf: Objektreferenz.Objektmethode(arg1,...); , Klassenname.Klassenmethode(arg1,...) oder Objektreferenz.Klassenmethode(arg1,...); .]

Vererbung:
In Java keine Mehrfachvererbung. Zugriff auf Objekte der Oberklasse mittels **super**.
[Z.B. Konstruktor **super**();, Methode **super**.methode();, auch mehrfach **super.super**.methode();.]

Konstruktoren und statische Variablen / Methoden werden nicht mitvererbt.

Jede Klasse standardmäßig [i.e. ohne explizite Angabe von **extends**] von java.lang.Object abgeleitet.

[**Signatur** in Java ohne Berücksichtigung des Rückgabewerts.]

Überdeckte Variablen existieren trotzdem in jedem Objekt der abgeleiteten Klasse [Zugriff über **super**].

Automatisches Casten zu Obermengen, nicht andersherum; Datentyphierarchie der Basisdatentypen:
byte ⊆ **short** ⊆ **int** ⊆ **long** ⊆ **float** ⊆ **double** und **char** ⊆ **int**

Pakete: Sammlung von Klassen; Zugehörigkeit eine Klasse mittels **package** packageName; in 1. Zeile der Quelldatei; Importieren in anderen Paketen **import** packageName.ClassName; [wobei mittels **import** packageName.∗; alle Klassen des Pakets importiert werden].

Sichtbarkeits-modifier:

	Klasse	Unterklasse	Paket	Welt
default	×	(∗)	×	
private	×			
protected	×	×	×	
public	×	×	×	×

[(∗) - nur im selben Paket]

Dynamisches Binden: Nach Zuweisung eines Unterklassenobjekts auf eine Objektreferenz der Oberklasse wird während der Laufzeit die in der Unterklasse überschriebene Funktion auch mittels der Oberklassenreferenz aufgerufen.

final-Methode: Methoden können in Unterklassen nicht neu implementiert werden.

final-Klasse: es können keine Unterklassen dieser Klasse erstellt werden.

Auflösen von Methodenaufrufen:
1. Bestimmung der Signatur zur Übersetzungszeit
2. Auswahl der letzten, speziellsten Implementierung zur Laufzeit

Seien $A(\text{typ}_{A1}, \ldots, \text{typ}_{An})$, $B(\text{typ}_{B1}, \ldots, \text{typ}_{Bn})$ in Frage kommende Methoden; Methode A ist spezieller als Methode B, falls:

1. für alle primitiven Parametertypen von A und B $\text{typ}_{Ai} \subseteq \text{typ}_{Bi}$.
2. alle Referenztypen von A aus den entsprechenden Paramtertypen von B abgeleitet werden können.

abstract - Schlüsselwort zur Bezeichnung einer Klasse oder Methode als **abstrakt**. Abstrakte Klassen können nicht zu Objekten instanziert werden; enthält eine Klasse mindestens eine abstrakte Methode, so ist auch die Klasse abstrakt zu deklarieren. Erst die Unterklasse, in der für alle abstrakten Methoden der Oberklassen auch eine Implementierung gegeben ist, kann instanziert werden.

interface - Ist eine vollständig abstrakte Klasse [Methoden **abstract**, Variablen **static final**]. In Java Mehrfachvererbung von Schnittstellen, nicht Klassen!

Dynamisches Binden mittels Methodentabellen in Objekten.

Fehlerbehandlung:

– Exceptions [leichte Fehler - Nulldivision, Zugriff auf Nullreferenz, ...] und Errors [schwere Fehler, „nicht behebbar“].

```
try {
        ...; // Auszuführendes
} catch( FirstException e ) {
        ...; // im Falle einer FirstException im Auszuführenden
} catch( SecondException | ThirdException e) {
        ...; // falls SecondException oder ThirdException im Auszuführenden
} finally {
        ...; // wird immer nach obigen Anweisungen ausgeführt
}
```

Abbildung 2: Exception-Handling-Block. Der **finally**-Block ist optional.

– Manuelles Werfen eines Fehlers via **if** (...) { **throw new** FirstException(); } . FirstException ist dabei eine Unterklasse von Throwable; geworfen wird ein entsprechendes Objekt.
Standardmäßig 2 Konstruktoren: mit Fehlermeldung [FirstException(String s)], ohne Fehlermeldung [FirstException()].

– speziellere Exceptions müssen früher abgefangen werden [da Überprüfung über **instanceof**].

– Exceptions wandern Aufrufhierarchie hoch, bis sie abgefangen werden.

– Zusicherungen [Assertions]

~ dienen zur Fehlersuche während Entwicklungs-/Test-Phase

~ können paket-/klassenweit ein-/ausgeschaltet werden [−ea - enable assertions, −da - disable assertions]

~ mit [assert booleanExpression : ErrorString;] oder ohne Fehlermeldung [assert booleanExpression;]

Aufzählungstypen:

– intern als Unterklasse von Enum realisiert; Elemente sind Attribute [**public static final**].

– jedem Element ist ein Ordinal zugewiesen [Integer von 0 bis ...]

– Beispiel enum Day { Mon, Tue, Wed, Thu, Fri, Sat, Sun;
boolean isWeekend() { **return**(**this** == Sat || **this** == Sun); } }

– obwohl Attribute Konstanten sind, wird nur der erste Buchstabe gezwungenermaßen groß geschrieben

Generische Programmierung in Java

– größtmögliche Abstraktion eines Algorithmus', um die Verwendbarkeit mit möglichst vielen verschiedenartigen Datentypen zu realisieren

– mittels der Wurzelklasse Object möglich [dann z.B. in einfach verketteten Listen, Binärbäumen, ...]

– mittels „generischer Klassen“

· Definition einer Klasse: **class** Name<T1,T2> {...} , Instanzierung: Name<String,Integer> var;

· nur auf Compiler „aufgesetzt“!

· entspricht Klassen-/Schnittstellen-/Methoden-Schablonen unter der Verwendung von Typvariablen [in Java keine Typvariablen für primitive Datentypen]

· ein „generischer Typ“ heißt die Instanzierung einer generischen Klasse mit einem konkreten Typ-Argument

· generische Typen verschiedener Datentypen sind inkompatibel [Name<String> ns = **new** Name<Integer>(3); liefert Fehler]; kein automatisches Casten!

· generische Typen sind vollständig [Name2<Integer, Name2<Integer,Double>> var; möglich]

· Einschränkung möglicher Typvariablen möglich [z.B.: **class** Name<T **extends** type1 & type2> {...} , type1 kann eine Klasse oder eine Schnittstelle, type2 und Folgende nur Schnittstellen sein]

· abgeleitete Klassen von Oberklassen, die ein Interface implementieren, welches für die Typvariable gefordert ist, werden nicht akzeptiert

· Generische Wildcard-Klasse name<?> var; [„name of unknown“];

Rückgabewerte immer in Object speicherbar, Attributreferenzen immer **null** setzbar.

– in Java nur eine Implementierung einer generischen Klasse
[Compiler: Type-Erasure von T mit Object in generischer Klasse, aber Casten im Code bei Verwendung von Objekten]
– „Rawtype“ existiert, Verwendung nicht empfohlen [Implementierung **class**<T> {}, Verwendung **class** obj]
– Einschränkungen:
· keine Typvariablen für statische Variablen [**static** T obj; ↯]
· Verwendung von **instanceof** verboten [(x **instanceof** T) ↯]
· Typkonvertierung nutzlos [(T) x; - wegen Type-Erasure durch Compiler ↯]
· keine Reihungen und Konstruktoraufrufe für Typvariablen [T[] obj = **new** T[10]; und T obj = **new** T(); ↯]
· Typvariablen können keine Oberklassen bilden

– **polymorphe Methoden**
· Deklaration: [modifiers] <type1, ..., typeN> returnType fctName([formalParameters]) {...}
· Aufruf: returnTarget = reference.<type1, ..., typeN> fctName([actualParameters]);
· Typinferenz erlaubt das Weglassen von Typargumenten beim Aufruf polymorpher Funktionen
[Vorsicht, fehleranfällig!]
· polymorphe Funktion nur compiliert/verwendet, wenn nicht bereits eine „normale“ Funktion besteht

flaches Kopieren: bit-weise Kopie des Objekts; **tiefes** Kopieren: bit-weise Kopie des Objekts und der Objekte aller enthaltenen Referenzen [und der Objekte von Referenzen in den Objekten der Referenzen in dem Objekt ...]

Boxing: automatisches Casten von Referenztypen in Wrapperklassen [Character, Short, Integer, Long, Boolean, Float, Double]; **Unboxing** ist das Gegenteil. [Vorsicht: jedoch nur wenn nötig!]

Thread-Konzepte

– Threads sind Programmfragmente, die „parallel“ ablaufen können [arbeiten auf dem selben Datenbereich]
– Rechenzeitvergabestrategien
· nicht-verdrängend [z.B. first in first serve [FIFS]]
· verdrängend [z.B. Round-Robin Verfahren [u.U. prioritätengesteuert; dabei wechseln sich die Prozesse/Threads ab, typ. Zeitscheibe ~ 100 ms] ⇒ „parallel“]
– in Java über Objekte der Klasse java.lang.Thread
· über Unterklasse-Objekte der Klasse Thread
· über an einen Thread übergebene Objekte, welche das Interface Runnable implementieren
– Implementation in der Methode **public void** run() {...}
– Aufruf über Thread- oder Thread-Unterklassen-Objekt mittels obj.start(); , nicht obj.run() !
– Prioritäten
· Java: Threads mit höherer Priorität sollen mehr Rechenzeit bekommen
· MAX_PRIORITY = 10; , NORM_PRIORITY = 5; , MIN_PRIORITY = 1; ,
public final void setPriority(**int** newPriority); , **public final int** getPriority();

– Synchronisation
  · Programmtiel, mit dem auf gemeinsame Daten zugegriffen wird, heißt „kritischer Bereich“; dieser darf währen der Ausführung nicht unterbrochen werden
  · Semaphor:
    ~ Integer-Variable S in der die Anzahl der momentan laufenden, die gleichen Daten betreffenden Threads gespeichert wird
    ~ initialisierung von S mittels Init (S) [n - Anzahl gleichzeitig ausführbarer kritischer Bereiche]
    public void Init(S) {S = n;}
    ~ vor kritischem Bereich P(S) [passieren]
    public void P(S) {**if** (S > 0) { S = S – 1; } **else** { stoppe Thread und in Warteliste von S eintragen }
    ~ nach kritischem Bereich V(S) [verlassen]
    public void V(S) { S = S + 1;
    if (Warteliste von S nicht leer) { Wähle Q aus Warteliste von S und springe zu dessen P(S), welches Q zuvor gestoppt hat } }
    ~ Fehler bei Verwendung durch Programmierer jedoch möglich!

  · Monitor:
    ~ besteht aus gemeinsamen Variablen, Oprationen [„Eintrittspunkte“] und Initialisierungscode
    ~ nur eine Operation darf ohne Unterbrechung auf den Variablen arbeiten
    ~ in Java werden Methoden mit **synchronized** gekennzeichnet
    ~ Java erstellt für jedes Objekt mit mindestens einer **synchronized**-Methode einen Monitor; da drin eine Sperre für dieses Objekt
    ~ auch Blöcke können synchronisiert werden; dazu Objekt mit zu verwendender Sperre anzugeben
    [**synchronized** (myObject) { ... *//Anweisungsblock }*]
    ~ Ausführung eines Threads außerhalb eines kritischen Bereichs kann bewusst unterbrochen und „blockiert“ [i.e. wird nicht in Warteschlange eingetragen] werden mit wait()
    ~ notify() hebt die Blockierung auf [i.e. Thread wird in Warteschlange wieder eingetragen]

  · Speichermodell
    ~ ab Java 1.5 keine vollständige sequentielle Konsistenz mehr; jeder Thread hat lokalen Cache
    ~ lesen von [„refresh“] / schreiben auf [„flush“] Hauptspeicher nach Speichermanager-Regeln
    ~ bei Verwendung **volatile** definierter Variablen refresh / flush des gesamten lokalen Cache
    ~ **final**-Variablen lösen partiellen refresh / flush aus