

Eprog Java Überblick

HS 2023 Version 2.3 - Zoe Laroche Eprog23 TA

Dieser Java Überblick basiert auf der Vorlesung 252-0027-00 Einführung in die Programmierung des Professors Dr. Thomas Gross. Die Vorlesung ist (Stand HS23) Teil des Basisjahres des BSc Informatik an der ETHZ.

Dieser Überblick ist **kein** offizielles ETH Material, ist nicht vollständig und kann kleine Fehler enthalten.

Er dient dem groben Überblick über die behandelten Java Themen.

Ergänzungen, Fragen und Anmerkungen gerne an zlaroche@student.ethz.ch

Am wichtigsten ist es viele Aufgaben zu lösen und viel zu programmieren, um Java erfolgreich zu erlernen.

So keep coding :)

1 Typen und Variablen

Eine Variable wird verwendet, um einen Wert zu speichern und diesen (an verschiedenen Orten) wiederzuverwenden. Der Wert hat einen gewissen Typ, welcher angegeben werden muss, wenn die Variable definiert wird.

Die Zuweisung eines Wertes zu einer Variable heisst Initialisierung. Der Wert der Variable kann auch verändert werden.

1.1 Basistypen/primitive types

Die wichtigsten Basistypen (werden immer klein geschrieben):

<i>int</i>	ganze Zahlen	-5 3 0 42	
<i>double</i>	Dezimalzahlen	0.5 -0.234	Punkt (nicht Komma)
<i>char</i>	Zeichen	'a' 'd' 'B'	'x' (nicht "x")
<i>boolean</i>	logische Werte	<i>true false</i>	

```
int ganzeZahl = 3;
double dezimalZahl = 0.5;
char zeichen = 'a';
boolean logik = true;
```

```
=> typ name;      //Definition der Variable
    name = wert;  //Initialisierung der Variable
=> typ name2 = wert; //beides gleichzeitig
```

1.2 Strings

Mit Strings können ganze Wörter oder Sätze gespeichert werden.

```
String st = "Hello World!";
```

Nützliche Methoden für Strings:

```
st.charAt(index); //gibt Zeichen an Stelle index zurueck
st.equals(string2); //gibt true zurueck, wenn alle Elemente von st mit string2
    uebereinstimmen
st == string2 //NICHT verwenden! (vergleicht Adresse, nicht Inhalt)
st.contains(string2); //gibt true zurueck, wenn string2 in st enthalten ist
st.startsWith(string2); //gibt true zurueck, wenn st mit string2 beginnt
st.endsWith(string2); //gibt true zurueck, wenn st mit string2 endet
st.substring(start,end); //gibt den Teilstring von Index "start" (inklusive) bis "end"
    (exklusiv) zurueck
st.indexOf(substring); //gibt den Index des ersten Auftretens von substring zurueck
st.toUpperCase(); //alle Buchstaben werden gross
st.toLowerCase(); //alle Buchstaben werden klein
st + string2 //neuer String aus st und string2 zusammen
st.length(); //gibt Anzahl chars zurueck
```

1.3 Sichtbarkeit/Scope von Variablen

Variablen sind erst nach ihrer Definition sichtbar und nur innerhalb des Blockes (zB. Methode, *if*-Block, Schleife), in dem sie definiert wurden.

Eine Variable die zum Beispiel in einer Methode definiert wurde, ist ausserhalb dieser Methode nicht mehr sichtbar: sie “stirbt” am Ende der Methode.

```
{
    //hier ist die Variable noch nicht sichtbar
    int variable = 42;
    //hier ist die Variable sichtbar
}
//hier ist die Variable nicht mehr sichtbar
```

In verschachtelten Blöcken ist eine Variable in allen inneren Blöcken ebenfalls sichtbar:

```
{
    int i = 0; //AB hier ist i sichtbar
    if (i==0) {
        //hier ist i immernoch sichtbar
        int j = 1; //j ist nur hier sichtbar
    }
    //BIS hier ist i sichtbar (j ist nicht mehr sichtbar)
}
```

2 if else Verzweigungen

Um Unterscheidungen zu machen oder einen Codeabschnitt nur unter einer gewissen Bedingung auszuführen, verwenden wir Verzweigungen.

Eine Verzweigungen wird genommen, wenn deren Bedingung *true* ist.

Wenn eine *if* oder *else if* Verzweigungen genommen wird, werden alle anderen, danach kommenden *else if* Verzweigungen ignoriert.

Else ist wie das Auffangbecken, wenn bis dorthin alle Bedingungen *false* waren.

```
if (Bedingung 1) { //wenn Bedingung 1 true ergibt, wird Statement 1 ausgefuehrt
    Statement 1;
}
else if (Bedingung 2) {
    Statement 2;
}
//hier koennten noch beliebig viele "else if" folgen
else {
    letztes Statement;
}
```

3 Schleifen/Loops

Wenn ein Codeabschnitt mehrmals ausgeführt werden muss, kann eine Schleife verwendet werden, anstatt denselben Code x mal zu schreiben.

3.1 for

Wenn vor Beginn der Schleife berechnet werden kann, wie oft ein Codeabschnitt wiederholt werden soll, dann ist die *for* Schleife nützlich.

Der Body wird ausgeführt, bis der Zähler (oft *i* genannt) die Bedingung nicht mehr erfüllt (oft $i < \text{eineZahl}$). Der Zähler wird nach jedem Durchlauf verändert (oft um eins erhöht: $i++$ oder verringert $i--$).

```
for (int i = 0; i < 100; i++) { //dieser loop wird 100 mal durchlaufen
    //body
}
```

3.2 while

Wenn ein Codeabschnitt so lange ausgeführt werden soll, bis eine Bedingung nicht mehr erfüllt ist, dann ist die *while* Schleife nützlich.

Solange die Bedingung gilt, wird der Body des Loops ausgeführt. Nach jeder Ausführung wird die Bedingung wieder geprüft.

Aufpassen dass das Programm nicht in einer Endlosschleife landet (bsp wenn die Bedingung nie *false* sein kann).

```
while (Bedingung) {
    //body
}
```

4 Arrays

Ein Array ist eine Möglichkeit mehrere Objekte zu speichern. Diese müssen alle den selben Typ haben (zB. *int*, *double*, *String*, *Object*, ...).

Der Index der Elemente beginnt bei **0**. Das letzte Element hat folglich Index Länge-1.

Beispiel eines *int* Arrays der Länge 5:

index	0	1	2	3	4
einträge	1	3	2	7	0

Ein Array erstellen:

```
int[] array = new int[5]; //neues int Array mit Laenge 5 erstellen
array[0] = 1; //erster Eintrag = 1 setzen
int[] array2 = {1, 4, 2}; //direkt mit Eintraegen fuellen
String[] array3 = new String[2]; //mit Strings

=> Typ [] name = new Typ[länge];
=> Typ [] name = {werte};
=> name[index] //Zugriff auf Element an Stelle index
```

Nützliche Methoden für Arrays:

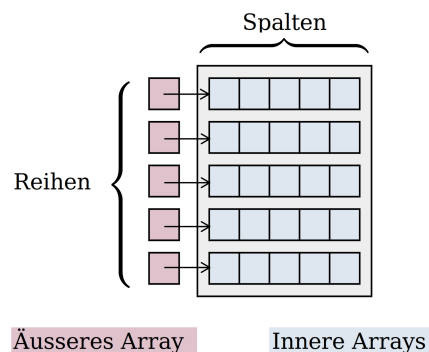
```
import java.util.Arrays; //diese Klasse muss importiert werden
Arrays.toString(array); //so wird das Array schoen ausgegeben
Arrays.sort(array);      //sortiert das Array
Arrays.binarySearch(array, key); //gibt den Index des gesuchten keys zurueck (geht nur,
    wenn array sortiert ist)
Arrays.equals(array, array2); //gibt zurueck ob alle Elemente von array mit array2
    uebereinstimmen
//NICHT!!:
array == array2
array.equals(array2);
//diese beiden vergleichen nur die Adresse und nicht den Inhalt

Arrays.copyOf(array, laenge); //gibt eine Kopie von "array" mit Laenge "laenge" zurueck
array.length //gibt die Anzahl Elemente zurueck
```

Wenn ein Array definiert, aber noch nicht initialisiert wird, ist es default mässig mit 0 (für *int* Arrays), 0.0 (für *double* Arrays), dem Null char ('`\000`') (für *char* Arrays) respektive *false* (für *boolean* Arrays) gefüllt.

4.1 2D Arrays/Matrizen

Matrizen werden als zwei dimensionale Arrays gespeichert. Das heisst, es sind verschachtelte Arrays: Jedes Element des äusseren Arrays ist wieder ein Array.



1	3	2	7	0
0	5	-2	16	1
7	3	2	7	9
1	3	2	0	4

Beispiel einer 4 mal 5 Matrix mit Typ *int*

Eine Matrix erstellen:

```
int[][] matrix = new int[5][2]; //neue int Matrix mit Groesse 5 x 2 erstellen
matrix[0][1] = 1; //Eintrag an Stelle 0,1 = 1 setzen
String[][] matrix2 = {"a", "b", "c"}, {"d", "e", "f"}; //direkt mit Eintraegen fuellen

=> Typ [][] name = new Typ[rows][cols]; //Matrix der Groesse rows x cols erstellen
=> name[i][j] //Zugriff auf Element an Stelle i,j
```

5 Reference type

Neben den primitive types gibt es die reference types. Eine primitive type Variable speichert die Information direkt als **Wert**. Im Gegensatz dazu speichert eine reference type Variable (Referenzvariable) eine **Referenz** auf die Information. Alle *Objects* sind reference types. *String* ist ebenfalls reference type, aber ein Spezialfall. Reference types werden immer gross geschrieben und ein neues Objekt wird mit dem *new* Keyword erstellt.

Wenn beispielsweise ein Array erstellt wird, bekommt man eine Referenz auf das Array zurück. Die Referenz ist wie ein **Verweis** auf das Array.

Wenn das Array an eine Methode übergeben wird, so werden nicht die Werte übergeben, sondern ein Verweis auf das Array. Wenn in der Methode Einträge des Arrays verändert werden, so ist diese Veränderung auch noch sichtbar, wenn die Methode wieder zurückgesprungen ist.

Es können auch mehrere Verweise auf ein Objekt zeigen. Dann haben verschiedene Referenzvariablen zugriff auf das Objekt.

6 Methoden/Funktionen

Methoden werden verwendet, um mehrmals gebrauchte Codeabschnitte auszulagern. Der Codeabschnitt wird in die Methode geschrieben und die Methode wird an der benötigten Stelle aufgerufen. Die **Signatur** der Methode ist der formale Rahmen und definiert den Namen der Methode und die Anzahl, Typ und Reihenfolge der Parameter der Methode. In derselben Klasse dürfen nicht mehrere Methoden dieselbe Signatur haben (sie müssen also unterschiedliche Namen und/oder unterschiedliche Parameteranzahl, Parametertypen oder Parameterreihenfolgen haben). Wenn mehrere definierte Methoden den selben Namen (aber unterschiedliche Parameter) haben, dann sprechen wir von Overloading.

Definition einer Methode:

```
public static Rückgabotyp name(Typ parameter){  
    //Inhalt der Methode  
    return rückgabewert; //muss den oben definierten Typ haben  
}
```

Aufruf der Methode (zB. in der *main* Methode):

```
Rückgabotyp r = name(parameter);  
name(parameter); //Aufruf einer Methode ohne Rueckgabewert
```

Wenn die Methode aufgerufen wird, springt das Programm zur Methodendefinition, führt diese aus und springt anschliessend wieder zurück dorthin, wo die Methode aufgerufen wurde:

```
public static void main(String[] args){  
    ① ↓ //... Programm  
    foo(); //Methodenaufruf  
    ⑤ ↓ //... Programm Fortsetzung  
}  
    ④ ↺  
    public static void foo(){  
    ③ ↓ //bla bla  
}
```

Das Diagramm zeigt den Kontrollfluss bei einem Methodenaufruf. Es besteht aus zwei Codeblöcken: `main` und `foo`. Im `main`-Block befindet sich ein Aufruf `foo();`. Rote Pfeile und Nummern in Kreisen markieren die Sprünge: Ein Pfeil führt von ① (Beginn von `main`) zum Aufruf, ein weiterer von ⑤ (Fortsetzung von `main`) zum Aufruf. Ein Pfeil führt von ② (Ende von `foo`) zurück zum Aufruf in `main`. Ein weiterer Pfeil führt von ④ (Ende von `main`) zurück zum Aufruf. Ein Pfeil führt von ③ (Beginn von `foo`) zum Aufruf. Ein Pfeil führt von ② (Ende von `foo`) zurück zum Aufruf.

6.1 Parameter

Parameter sind Übergabewerte. Wenn die Methode aufgerufen wird, können Werte mitgegeben werden, sogenannte Parameter. Wenn die Methode definiert wird, muss auch definiert werden, wieviele Parameter erwartet werden, von welchem Typ und in welcher Reihenfolge. Es kann mehrere, nur einen oder keinen Parameter geben.

6.2 Rückgabewert/return value

Die Methode kann Werte zurückgeben. Wenn die Methode definiert wird, muss auch definiert werden, ob es einen Rückgabewert gibt und wenn ja welchen Typ er hat.

void: kein Rückgabewert

Wenn ein Rückgabewert definiert wird, so muss in der Methode *return* (verlässt die Methode sofort und gibt den gegebenen Wert zurück) gefolgt vom gewünschten Rückgabewert (vom definierten Typ) vorkommen.

7 Klassen und Objekte

Java ist eine objektorientierte Programmiersprache, sie ist also durch aufeinander einwirkende Objekte organisiert.

Mit einer Klasse können neue Typen definiert werden (zB. Auto → Klassenname). In der Klasse werden die Eigenschaften dieses Typs definiert (zB. Farbe und Grösse → Attribute) und was man mit Objekten dieses Typs tun kann (zB. fahren → Methoden). Zudem wird definiert, wie ein Objekt dieses Typs erstellt werden kann (→ Konstruktor).

Klassen beschreiben also einen Typ. Eine Klasse ist wie eine Vorlage/Schablone für ein Objekt dieses Typs. Mit dieser Vorlage können Objekte erstellt werden, die alle dieselbe Form und Funktionalität haben. Objekte sind also Exemplare/Instanzen einer Klasse und somit eines Typs. Alle Objekte sind reference types.

Struktur einer Klasse:

```
class KlassenName {  
  
    Typ attribut; //Attribut (kann mehrere haben)  
  
    KlassenName(Typ parameter){ //Konstruktor (hat den Namen der Klasse)  
        this.attribut = parameter;  
    }  
  
    public Rückgabetypp methodenName(Typ parameter){ //non-static Methode  
  
        return rückgabewert;  
    }  
  
    public static void main(String[] args){ //main Methode  
  
    }  
}
```

Erstellen eines neuen Objekts dieser Klasse:

```
KlassenName obj = new KlassenName(); //neues Objekt der Klasse "KlassenName" erstellen  
obj.methodenName(); //Aufruf einer non-static Methode  
obj.attribut //Zugriff auf ein Attribut des Objekts
```

7.1 Attribut/field

Attribute sind Variablen (mit beliebigem Typ) innerhalb des Objekts. Sie sind wie die Eigenschaften der Objekte und speichern für jedes Objekt die Werte der Eigenschaften.

7.2 non-static Methode

Die Methoden beschreiben, was man mit den Objekten der Klasse machen kann. Die Methoden für Objekte sind im Gegensatz zu den bisher gesehenen Methoden nicht *static*. Das heisst, man kann sie nur mit einem Objekt dieser Klasse aufrufen.

7.3 Konstruktor

Der Konstruktor gibt vor, wie ein neues Exemplar/Objekt/Instanz dieser Klasse erstellt werden kann und wie die Attribute bei der Erstellung initialisiert werden.

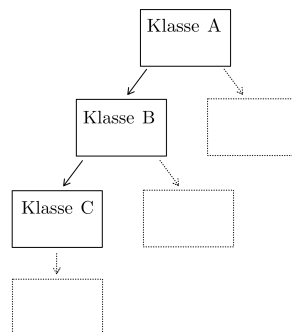
Wenn kein Konstruktor definiert wurde, so existiert der Defaultkonstruktor. Dieser nimmt keine Parameter entgegen und setzt alle Attribute auf deren Defaultwert (*null* für *Objects*, 0 für *int* usw.). Es können auch mehrere Konstruktoren erstellt werden, die sich in der Anzahl, Typ und Reihenfolge der Parameter unterscheiden. So können die Attribute unterschiedlich initialisiert werden.

7.4 Vererbung

In Java können Klassen erweitert werden, so kann auf schon existierenden Klassen aufgebaut werden. Die erweiterte Klasse (Subklasse) erbt dabei alle Attribute und Methoden von der Superklasse. Konstruktoren werden nicht vererbt. Die Erweiterung wird mit dem *extends* Keyword in der Klassendefinition festgelegt. Diese Erweiterungen können über mehrere Stufen gehen (Bsp. C *extends* B und B *extends* A) dadurch entsteht eine Hierarchie von Klassen. Eine Klasse kann aber nur genau eine andere Klasse erweitern. *Object* ist eine Superklasse aller Typen, die durch Klassen definiert wurden. Mithilfe des *super* Keywords können Attribute, Methoden oder Konstruktoren der Superklasse aufgerufen werden (*super.methodName()*).

```
class Superklasse {  
    public int attribut; //Attribut  
    public void foo(){ } //Methode  
}  
  
class Subklasse extends Superklasse { //erbt das Attribut und die Methode  
    //in diesem Fall wird die Methode foo ueberschrieben:  
    @Override //nicht noetig, aber zeigt an, dass diese Methode ueberschrieben wird  
    public void foo(){ }  
}
```

Ein Objekt der Subklasse ist immer auch ein Objekt der Superklasse
ZB. ein Velo ist ein Fahrzeug (wenn Velo *extends* Fahrzeug).



Vererbungshierarchie (hier: *class B extends A* und *class C extends B*)

7.4.1 Polymorphismus und dynamische Bindung

Die vererbten Methoden können überschrieben werden. Dabei kann eine Methode beliebig verändert werden, solange die Signatur der Methode mit der zu überschreibenden Methode der Superklasse übereinstimmt. Zudem darf die Sichtbarkeit der Methode in der Subklasse nicht restriktiver sein, als die der Superklasse.

Dadurch existieren also mehrere Implementierungen einer Methode in verschiedenen Klassen (→ Polymorphismus). Welche dieser Versionen ausgeführt wird, wenn die Methode aufgerufen wird, wird bei Laufzeit ermittelt (→ Dynamische Bindung).

```
L obj = new R();    //Referenzvariable "obj" hat Typ L und zeigt auf ein Objekt vom Typ R
R obj2 = (R) obj;   //Von L zu R casten (explizit)
Object obj3 = obj;  //Von L zu Object casten (implizit)
obj instanceof L    //gibt zurueck ob "obj" auf ein Objekt von Typ L verweist
```

- R definiert den Typ des Objekts und L den Typ der Referenzvariable.
- Eine Referenzvariable von Typ L kann eine Referenz auf ein Objekt vom Typ L oder einem Subtyp von L haben. Mit anderen Worten: die Klasse L muss immer über oder gleich der Klasse R sein in der Vererbungshierarchie.
- L können wir verändern, indem wir die Referenz von Typ L in eine Referenz eines anderen Typs des Vererbungszeitastes umwandeln (casten). Dies geht aber nur, solange die Hierarchie noch stimmt.
 - nach “oben” casten (zb. von L zu $Object$): implizites Casting (wird automatisch von Java erledigt)
 - nach “unten” casten (zb von L zu R): expizites Casting (muss manuell gemacht werden. D. h. der gewünschte Typ muss in Klammern vor die Variable geschrieben werden.) Achtung Casts werden erst bei Laufzeit geprüft (also nicht schon vom Compiler) und erst dann allfällig eine Exception geworfen.
- L bestimmt, welche Methoden für Objekte aufgerufen werden können. Also welche Methoden zur Verfügung stehen: alle Methoden die in L und allen Superklassen von L definiert sind.
- R bestimmt, welche Version der Methode ausgeführt wird, wenn die Methode aufgerufen wird. Es wird immer die Version der Methode in der Klasse R ausgeführt und wenn diese Methode dort nicht überschrieben wurde, so wird die als nächstes implementierte Version aufwärts in der Klassenhierarchie (Vererbungszeitast) ausgeführt.
- L bestimmt welche Version des Attributs verwendet wird. Attribute werden nicht überschrieben, sondern “versteckt”. D. h. Attribute mit gleichem Namen in verschiedenen Klassen sind separate Attribute und es wird die Version aus L (oder die nächstobere) verwendet.

Wenn eine Methode als Parameter eine Referez auf ein Objekt von Typ T erwartet, kann auch eine Referenz auf ein Objekt von einem Subtyp von T übergeben werden. Wenn der Paramter beispielsweise den Typ $Object$ hat, dann kann eine Referenz auf ein beliebiges Objekt übergeben werden. Dasselbe gilt für Arraytypen. Ein Array von Typ T kann Objekte von T oder jedem Subtyp speichern. Das ist insbesondere nützlich, wenn mehrere Typen zusammengefasst werden sollen oder wenn mehrere Typen zulässig sind.

7.5 Sichtbarkeit

Attribute, Methoden und Konstruktoren können verschiedene Zugriffsmodifikatoren haben. Diese definieren den Sichtbarkeitsbereich (scope). Ein *private* Attribut ist beispielsweise nur in der Klasse, in der es definiert wurde sichtbar, wo hingegen ein *public* Attribut überall sichtbar ist. Ein *protected* Attribut ist in der eigenen Klasse, im ganzen Paket (Struktur innerhalb eines Java projects, die zusammengehörige Klassen zusammenfasst) und in allen Subklassen (auch solchen in anderen Paketen) sichtbar. Wenn ein Attribut kein Modifikator hat, so ist es nur in der Klasse und im Paket sichtbar. Wenn ausserhalb des Sichtbarkeitsbereich auf das Attribut zugegriffen werden soll, dann können get- und set- Methoden definiert und aufgerufen werden.

	Klasse	Paket	Subklasse (ausserhalb Paket)	Welt
<i>public</i>	x	x	x	x
<i>protected</i>	x	x	x	-
kein Modifikator	x	x	-	-
<i>private</i>	x	-	-	-

```
modifikator typ name  
public int attribut = 5;
```

8 Input und Output

8.1 Input mit Scanner

Input von der Konsole:

```
import java.util.Scanner;  
Scanner sc = new Scanner(System.in);  
int input = sc.nextInt(); //geht auch mit anderen Typen
```

Input von einem File:

```
//in die Methodendefinition muss "throws FileNotFoundException"  
import java.io.File;  
Scanner sc = new Scanner(new File("fileName.txt"));  
String input = sc.next();
```

8.2 Output

Output in die Konsole:

```
System.out.println("Hello World!");
```

Output in ein File:

```
import java.io.File;  
import java.io.PrintStream;  
PrintStream fileOut = new PrintStream(new File("fileName.txt"));  
fileOut.println("text");
```

9 Bedingte Auswertung und Kurzformen

Kurzformen:

```
i = i + 1;    =>    i++; oder ++i;
i = i - 1;    =>    i--; oder --i;
//bei i++ wird zuerst der Wert von i zurueckgegeben und dann erst incremented
//bei ++i wird zuerst incremented und dann der neue Wert zurueckgegeben

i = i + x;    =>    i+=x;
i = i - x;    =>    i-=x;
i = i * x;    =>    i*=x;
i = i / x;    =>    i/=x;
i = i % x;    =>    i%=x; //modulo
```

Bedingte Auswertung:

Wenn ein logischer Term ausgewertet wird, kann die Auswertung teilweise schon früher abgebrochen werden.

```
x && y        //bricht ab, wenn x false ist (y wird gar nicht ausgewertet)
x || y        //bricht ab, wenn x true ist (y wird gar nicht ausgewertet)

//bsp:
int a = 0;
int b = 1;
(a != 0) && (b/a == 3) //da der erste Term schon false gibt, wird abgebrochen, bevor
                       1/0 gerechnet wird. Somit gibt es keine "divide by zero" Exception
```

10 Sonstiges

10.1 Zufallszahlen/Random

```
import java.util.Random;
Random rand = new Random();
rand.nextInt(bound); //gibt zufaelligen int zwischen 0 (inklusive) und bound (exklusiv)
                     zurueck
```

10.2 Import

Um vorgefertigte Klassen und Methoden nutzen zu können, müssen diese importiert werden. Die *import* Statements kommen immer am Anfang des Dokuments, vor der Klasse.

Viele Klassen die man benötigt sind im Package *java.util* enthalten. Um nicht jede Klasse separat importieren zu müssen, kann folgender Import benutzt werden:

```
import java.util.*; //importiert ganzes java util Package
```
