Übersicht Propädeutikum (Beta-Version)

Version vom 25.11.2019

⚠ Hat **keinen** Anspruch auf Vollständigkeit. Soll lediglich als Gedächtnisstütze und Übersicht der besprochenen Themen dienen.

Wird im Laufe der Zeit noch öfter geupdated und verändert, weswegen sowohl Layout als auch Inhalt noch nicht optimiert sind.

Vorlesung 1

Grundlegende Datentypen

Typ	Beschreibung	Default Value
`byte`	8-bit ganze Zahl	`0`
`short`	16-bit ganze Zahl	`0`
`int`	32-bit ganze Zahl	`0`
`long`	64-bit ganze Zahl	`0L`
`float`	32-bit Gleitkommazahl	`0.0f`
`double`	64-bit Gleitkommazahl	`0.0d`
`boolean`	Wahrheitswerte	`false`
`char`	einzelne Zeichen	`'\u0000'`
`String`	Zeichenketten	`null`

Primitive Type vs. Reference Type

→ **Primitive Types**
  → Sind *keine* Klassen: z.B. `int` ≠ Klasse
    → Besitzen aber *Wrapperklassen*: `int` → `Integer`
  → Liegen "direkt" in Variablen, *nicht* als Speicheradresse
→ **Reference Types**
  → Sind Klassen: z.B. `String` = Klasse ("echte" Objekte)
  → Liegen als Speicheradresse (*Referenz*) in Variable
  → Default Value eines Objekts ist immer `null`

`main`-Methode

```
public static void main(String[] args){
    // Anweisungen
}
```

→ **Einstiegspunkt** für das Ausführen des Programms
→ Klasse *kann*, aber *muss* keine `main` haben

Methoden zur Ausgabe auf Terminal

→ `System.out.println(String s)`
Gibt `s` gefolgt von Linebreak aus
→ `System.out.print(String s)`
Gibt `s` ohne Linebreak aus

Handhabung von Variablen

Deklaration

```
int number;
⌞Typ⌝ ⌞Variablenname⌝;
```

→ *Objektvariablen:* autom. Initialisierung mit default value
→ *lokale Variablen: keine* autom. Initialisierung

Deklaration mit Initialisierung

```
int number = 42;
⌞Typ⌝ ⌞Variablenname⌝ = ⌞Wert⌝;
```

→ gleichzeitige Deklaration & Zuweisung

Zuweisung (ohne Deklaration)

```
number = 777;
⌞Variablenname⌝ = ⌞Wert⌝;
```

→ Änderung des Inhalts von bereits deklarierter Variable

Einfache Operatoren

arithmetische Operatoren		**Vergleichsoperatoren**	
`+`	Addition	`==`	gleich
`-`	Subtraktion	`!=`	ungleich
`*`	Multiplikation	`>` / `>=`	größer (gleich)
`/`	Division	`<` / `<=`	kleiner (gleich)
`%`	Modulo		
unäre Operatoren		**logische Operatoren**	
`++`	Inkrementiert um 1	`&&`	logisches und ($\wedge$)
`--`	Dekrementiert um 1	`||`	logisches oder ($\vee$)
		`!`	Negation ($\neg$)
		`^`	entweder-oder (XOR)

⚠ Vergleich primitiver Typen mit ==
Vergleich Reference Types mit `equals`-Methode

Fallunterscheidungen

if

```
if (⌞boolescher Ausdruck⌝) {
    // Anweisungen wenn Ausdruck = true
}
```

if-else

```
if (⌞boolescher Ausdruck⌝) {
    // Anweisungen wenn Ausdruck = true
} else {
    // Anweisungen wenn Ausdruck = false
}
```

Verkettung von Fallunterscheidungen

```
if (⌞boolescher Ausdruck 1⌝) {
    // Anweisungen wenn Ausdruck1 = true
} else if (⌞boolescher Ausdruck 2⌝) {
    // Anweisungen wenn Ausdruck2 = true
} else {
    // Anweisungen wenn kein Ausdruck true
}
```

Vorlesung 2

Arrays

```
⌞Typ⌝[] ⌞Arrayname⌝ = new ⌞Typ⌝[⌞Länge⌝]
```

→ Zugriff auf Feld mit Index *n*: ⌞*Arrayname*⌝[⌞*n*⌝]
→ Zugriff auf Länge des Arrays: ⌞*Arrayname*⌝`.length`

→ *Beispiel einfaches Array:* `int[] num = new int[10]`

Arrayfelder (mit Inhalt)	0	0	0	0	0	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

→ *Beispiel 2D-Array:* `int[][] num = new num[10][10]`

Schleifen

for-Schleife

```
// Allgemein
for (⌞Init. Zähler⌝; ⌞boolescher Ausdruck⌝; ⌞Update Zähler⌝) {
    // Anweisungen
}
// Beispiel: Inhalt v. Array a ausgeben
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

while-Schleife

```
// Allgemein:
while (⌞boolescher Ausdruck⌝) {
    // Anweisungen
}
// Beispiel: Inhalt v. Array a ausgeben
int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```

`break` und `continue`

→ `break` beendet komplette Schleife
→ `continue` beendet aktuellen Durchlauf der Schleife

Vorlesung 3

Block

→ Durch geschweifte Klammern (`{}`) gruppierte Anweisungen

Methoden

```
// Kopf der Methode
⌊Keywords⌉ ⌊Return Type⌉ ⌊Name⌉(⌊Parameter⌉) {
  // Koerper der Methode
}
```

→ **Keywords:** z.B. Sichtbarkeit, `static`, `abstract` etc.
→ **Return Type:** Typ der im Körper von `return` zurückgegeben werden *muss* (`void` wenn keine Rückgabe)
→ **Name:** Name zum Aufruf der Methode
→ **Parameter:** Kommaseparierte Liste:
`Typ1 parName1, Typ2 parName2, ...`

lokale Variablen

→ Variable in einem Block im Körper einer Methode
→ Nichtlokale Variable: Attribute einer Klasse

Klasse `String`

→ Strings entsprechen intern einem `char`-Array
⚠ `String` ist *kein* primitiver Typ, verhält sich aber oft so (z.B. kein `new` nötig, wie bei anderen Reference Types)

→ `+` (*überladener* Operator)
Bei Strings keine Addition, sondern Konkatenation
→ `int length()`
Gibt Länge des Strings aus
→ `char charAt(int n)`
Gibt Character an Position *n* aus
→ `String substring(int n, int m)`
Gibt Teilstring von Position *n* bis *m* aus

Konvertieren: String → Zahl

```
int i = Integer.parseInt(str);
double d = Double.parseDouble(str);
```

Scopes

→ **Gültigkeits-/Sichtbarkeitsbereich** einer Variable
→ Variable nur gültig in dem Block in dem sie deklariert wurde

```
public static void main(String[] args) {
  int foo = 0; // gilt in diesem und allen
      ↪ inneren Bloecken
  {
    int bar = 0; // gilt nur in diesem Block
    int foo = 0; // ⚠ Fehler: Bereits im Block
        ↪ darueber deklariert!
    System.out.println(foo); // ✔
    System.out.println(bar); // ✔
  }
  System.out.println(foo); // ✔
  System.out.println(bar); // ⚠ Fehler: existiert
      ↪ in diesem Block nicht!
}
public void someOtherMethod() {
  int foo; // ✔ Kein Konflikt: Neuer Block
}
```

⚠ Objektvariablen dürfen **überschattet** werden durch gleichnamige lokale Variablen, da auf Objektvariablen immer noch eindeutiger Zugriff möglich (z.B. durch `this`)

Vorlesung 4

Kommandozeilenparameter mit `String[] args`

→ **Array** mit den **Argumenten** die der `main`-Methode in das Array `String[] args` übergeben wurden

Beispiel: Aufruf von `HelloWorld` auf Kommandozeile

```
$ java HelloWorld foo bar quz
```

Erzeugt Array `args` = {"foo", "bar", "quz"}

Collections

→ "Container" für mehrere Objekte des selben Typs
  → spezialisierte Datenstrukturen je nach Verwendungszweck
  → bieten dynamische Größenanpassung
  → einfache vordefinierte Operationen

Generische Klassen

→ Collections sind *generische* Klassen
→ *generisch* = Arbeiten flexibel mit verschiedenen Typen
  ⚠ jedoch **nicht** mit primitiven Typen
→ Erkennbar an *Typparameter* in spitzen Klammern (<>)

```
ArrayList<Integer> myList =
        new ArrayList<Integer>();
ArrayList<⌊Typparameter⌉> myList =
        new ArrayList<⌊Typparameter⌉>();
```

→ Alternativ mit *Diamantoperator* rechte Seite leer lassen:

```
ArrayList<⌊Typparameter⌉> myList = new ArrayList<>();
```

Bulk Operations

→ komfortable Operationen zwischen ganzen Collections

→ `boolean containsAll(Collection c)`
Enthält `this` alle Elemente in `c`?
→ `boolean addAll(Collection c)`
Fügt zu `this` alles in `c` hinzu
→ `boolean removeAll(Collection c)`
Entfernt aus `this` alles was in `c` ist
→ `boolean retainAll(Collection c)`
Behalte in `this` alles aus `c`, entferne Rest
→ `void clear()`
Entfernt jeglichen Inhalt aus `this`

Klasse `ArrayList<T>`

→ **Indexbasiert:** Indexierte "Fächer" (analog zu Arrays)
→ **Ordered:** via *insertion order*
  → Reihenfolge des Hinzufügens = Reihenfolge in Liste
→ **Duplikate:** erlaubt

→ `boolean add(T e)`
Fügt Element ans Ende der Liste an
→ `boolean remove(Object e)`
Entfernt *erstes* Vorkommen des Objekts
→ `T remove(int i)`
Entfernt *i*-tes Element aus Liste (und returned dieses)
→ `T get(int i)`
Gibt *i*-tes Element aus Liste aus
→ `boolean contains(Object e)`
Prüft ob Objekt in Liste vorhanden ist
→ `int size()`
Gibt Anzahl der Elemente in der Liste aus
→ `boolean isEmpty()`
Prüft ob Liste leer ist

Klasse `HashMap<K,V>`

→ **Key-Value Paare:** Bildet (eindeutige) Keys vom Typ `K` auf beliebige Values vom Typ `V` ab
  → Wer mit Python gearbeitet hat: analog zu `dict`
→ **Unordered:** Iterationsreihenfolge kann sich ändern
  → `TreeMap` = geordnet nach *natural order* des Inhalts
  → `LinkedHashMap` = geordnet nach *insertion order*
→ Keine Collection per se, aber Teil des Collection Frameworks

→ `V put(K key, V val)`
Assoziiert `key` mit `val` in der Map (returned Value der vorher mit `key` assoziiert war, oder `null` wenn neues Key-Value-Paar)
→ `V get(Object key)`
Gibt Value der mit `key` assoziiert ist aus
→ `boolean containsKey(K key)` / `containsValue(V val)`
Prüft ob `key` bzw. `val` in Map ist
→ `int size()`
Gibt Anzahl an Key-Value Paaren in Map aus
→ `boolean isEmpty()`
Prüft ob Map leer ist
→ `Set<K> keySet()`

Gibt `Set`-Objekt mit allen Keys aus

→ `Collection<V> values()`

Gibt `Collection`-Objekt mit allen Values aus

→ `Set<Map.Entry<K,V>> entrySet()`

Gibt `Set`-Objekt mit allen Key-Value Paaren aus

Klasse `HashSet<T>`

→ **Modelliert Menge:** Duplikate *nicht* erlaubt

→ **Unordered:** Iterationsreihenfolge kann sich ändern

→ `TreeSet` = geordnet nach *natural order* des Inhalts

→ `LinkedHashSet` = geordnet nach *insertion order*

→ `boolean add(T e)`

Fügt Element zur Menge hinzu

→ `boolean remove(Object e)`

Entfernt Objekt aus Menge

→ `boolean contains(Object e)`

Prüft ob Objekt in Menge vorhanden ist

→ `int size()`

Gibt Anzahl der Elemente in der Menge aus

→ `boolean isEmpty()`

Prüft ob Menge leer ist

Iteration über Collections

Mit einem `Iterator`-Objekt

```
public class IteratorDemo {
  public static void main(String[] args) {
    ArrayList<String> names = new ArrayList<>();
    // Namen zu "names" hinzufuegen ausgelassen
    Iterator<String> itr = names.iterator();
    // Pruefen ob naechstes Element existiert
    while (itr.hasNext()) {
      // Einlesen des naechsten Elements
      String name = itr.next();
      // Beispiel: Alle Namen ausgeben
      System.out.println(name);
    }
  }
}
```

Mit for-each-Konstrukt

```
public class ForEachDemo {
  public static void main(String[] args) {
    ArrayList<String> names = new ArrayList<>();
    // Namen zu "names" hinzufuegen ausgelassen
    // Lesbar als: "fuer jeden 'name' in 'names'"
    // NB: Name der linken Var frei waehlbar
    for (String name : names) {
      // Beispiel: Alle Namen ausgeben
      System.out.println(name);
    }
  }
}
```

→ **Protip:** Funktioniert auch bei normalen Arrays

Vorlesung 5

Klasse

→ "Bauplan" für ein **Objekt**

→ Definiert **Attribute** und **Methoden** der Objekte

→ **Attribute:** Was *hat* das Objekt? (*Objektvariablen*)

→ **Methoden:** Was *kann* das Objekt? (*Objektmethoden*)

Beispielklasse: `Student`

```
public class Student {
    // Attribute/Objektvariablen
    String name;
    int matrikel;
    // Methoden/Objektmethoden
    public Student(String name, int matrikel) {
        this.name = name;
        this.matrikel = matrikel;
    }
    // (...)
}
```

Konstruktor

```
public Student(String name, int matrikel) {
    this name = name;
    this.matrikel = matrikel;
}
```

→ Besondere Methode zur **Erzeugung** eines Objekts

→ Unterschiede zu normalen Methoden:
kein Returntype; Methodenname == Klassenname

Keyword `this`

→ Um in Methoden Zugriff auf das **aufrufende Objekt** zu haben

→ Beispiel: Aufruf von `stu.setName("Maria")`
⇒ `this` wird durch das aufrufende Objekt `stu` ersetzt

```
public class Student {
  String name;
  public void setName(String name) {
    this.name = name; // NB: Ueberschattung
  }
}
```

Objekt

→ **Instanz** einer **Klasse** (aus "Bauplan" gebaut)

Erzeugung eines Objekts mit `new`

```
Student stu = new Student("Max",1234567);
└Typ┐ └Varname┐ = new └Konstruktor┐(└Parameter┐);
```

→ `new`-Operator *instanziiert eine Klasse*

→ Syntax: Benötigt als Argument nur einen *Konstruktor*

→ geht daher auch ohne vorheriges ablegen in Variable, z.B.:

```
demoArrayList.add(new Student("Max",1234567));
```

Punkt-Operator "."

```
└Objekt┐.└Variable/Methode┐
stu.variablenname
stu.methodenname()
```

→ Erlaubt Zugriff auf Variablen und Methoden die für das Objekt in seiner jeweiligen Klasse definiert wurden

Vorlesung 6

Getter-/Settermethoden

```
private └Typ┐ foo;
public └Typ┐ getFoo() { return this.foo; }
public void setFoo(└Typ┐ s) { this.foo = s; }
```

→ Uneingeschränkter Zugriff auf Variablen eventuell nicht erwünscht, sind daher oft auf `private` gesetzt

→ Kontrollierten Zugriff trotzdem ermöglichen durch:

→ **Getter**-Methoden die gespeicherten Wert returnen

→ **Setter**-Methoden die Wert der Variable ändern

⚠ **kein** Muss, sind auch keine "besonderen" Methoden

Keyword `static`

```
public static int statischeVar
public static void statischeMethode(){...}
```

→ Deklariert, dass sich **Variable** oder **Methode** von *jedem* Objekt einer Klasse **geteilt** werden (*Klassenvariablen, -methoden*)

→ Kein Objekt nötig für Zugriff auf `static` Variablen/Methoden

→ Daher: Zugriff mit Klassenname verdeutlicht, dass `static` z.B. `Student.statischeVar` anstatt `stu.statischeVar`

Sichtbarkeitsmodifizierer

	sichtbar in/zugreifbar aus			
Eigenschaft ist	**Klasse**	**Package**	**Unterklasse**	**Welt**
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	✖
ohne	✔	✔	✖	✖
`private`	✔	✖	✖	✖