Übersicht Propädeutikum

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

Einführung

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`

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⌝;
```

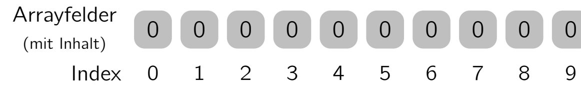
- → Änderung des Inhalts von bereits deklarierter Variable

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]`



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

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

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

`main`-Methode

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

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

Parameter `String[] args`

- → **Array** mit den **Argumenten** die Programm übergeben wurden

Beispiel: Aufruf von `HelloWorld` auf Kommandozeile

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

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

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
}
```

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++;
}
```

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

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

Konvertieren von Datentypen

String → Zahl

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

Objekt → String

```
String str = String.valueOf(obj);
String str = obj.toString();
```

⚠ nur wenn Objekt `toString`-Methode implementiert

Typecasting

→ Compiler sagen, dass Objekt A vom spezifischeren Typ B ist

```
Object o = "foo";
String s = o; // ⚠ Fehler!
String s = (String)o; // ✔
```

Scopes und Rekursion

Scopes

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

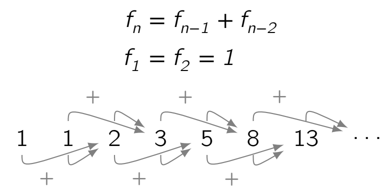
```
public static void main(String[] args) {
  int foo = 0; // gilt in diesem und allen
      ↪ inneren Bloecken
  {
    int bar = 0; // gilt nur in diesem Block
    int foo = 0; // ⚠ Fehler: Bereits im Block
        ↪ darueber deklariert!
    System.out.println(foo); // ✔
    System.out.println(bar);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

Rekursiv vs. Iterativ

Beispiel: Berechnung der n-ten Zahl in der Fibonacci-Folge

$$f_n = f_{n-1} + f_{n-2}$$
$$f_1 = f_2 = 1$$



Iterative Implementation

```
public static int iterativFibonacci(int x) {
  if (x == 1 || x == 2) {
    return 1;
  }
  int fn = 0;
  int fn_minus_1 = 1;
  int fn_minus_2 = 1;
  for (int i = 3; i <= x; i++) {
    fn = fn_minus_1 + fn_minus_2;
    fn_minus_2 = fn_minus_1;
    fn_minus_1 = fn;
  }
  return fn;
}
```

Rekursive Implementation

```
public static int rekursivFibonacci(int x) {
  if (x== 1 || x == 2) {
    return 1;
  }
  return rekursivFibonacci(x-1) +
      ↪ rekursivFibonacci(x-2);
}
```

Objektorientierung

Klasse

→ “Bauplan” für ein **Objekt**
→ Definiert **Attribute** und **Prozeduren** der Objekte
  → **Attribute:** Was *hat* das Objekt? (*Objektvariablen*)
  → **Prozeduren:** Was *kann* das Objekt? (*Objektmethoden*)

Beispielklasse: `Student`

```
public class Student {
    // Attribute/Objektvariablen
    String name;
    int matrikel;
    // Prozeduren/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 **aktuelle 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
  }
}
```

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`

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 "."

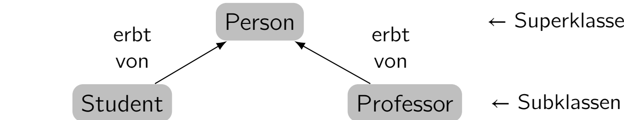
```
⌞Objekt⌝.⌞Variable/Methode⌝
stu.variablenname
stu.methodenname()
```

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

Sichtbarkeitsmodifizierer

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

Vererbung




```
public class Person {
  // Code
}
```

```
public class Student extends Person {
  // Code
}
```

```
public class Professor extends Person {
  // Code
}
```

→ Keyword: `extends`
  → **Sub**klasse erbt von **Super**klasse
→ Subklasse kann nur von jeweils *einer* Superklasse erben
  → *keine* Mehrfachvererbung in Java

Was ermöglicht Vererbung?

→ Subklasse erbt Variablen und Methoden der Superklasse
  → können wie gehabt genutzt werden
→ Subklasse kann *neue* Variablen/Methoden definieren
  → existieren dann *nur* in Subklasse
→ Subklasse kann Methoden der Superklasse *überschreiben*
  → Neudefinition einer Methode mit selber Signatur

Vererbung & Konstruktoren

→ Konstruktoren werden *nicht* vererbt
→ Zugriff auf Konstruktor der Superklasse in Subklasse durch Aufruf von `super(⌞Parameter⌝)`

```
public Student(⌞Parameter⌝) {
  super(⌞Parameter für Konstruktor Person⌝);
  // Restliche Zuweisungen fuer Student
}
```

Erben von `Object`

→ *Jede* Klasse erbt implizit von `Object`
→ Manche `Object`-Methoden sind sinnvoll zu überschreiben
  → `boolean equals(Object o)`
    Zur Gleichheitsüberprüfung
  → `String toString()`
    Für sinnvolle String-Repräsentation des Objekts

Abstrakte Klassen

```
public abstract class AbstractDemo {
  // muss in Subklasse ueberschrieben werden
  public abstract int methodeEins(String s);
  // muss nicht ueberschrieben werden
  public int methodeZwei(String s);
}
```

→ Keyword `abstract`
→ Können verschiedene Methoden enthalten:
  → `abstract` deklarierte Methoden (= nur Kopf d. Methode)
  → fertig implementierte Methoden
  → aber **keinen** Konstruktor
→ Subklassen können von nur einer abstrakten Klassen erben (gilt wie eine nichtabstrakte Klasse in der Vererbung)
  → dann **müssen** alle abstr. Methoden überschrieben werden
  → nicht-abstrakte Methoden jedoch nicht
→ Wozu: Sicherstellen, dass erbende Klassen *definitiv* alle eine *eigene Implementation* der abstrakten Methoden haben

Interfaces

```
public interface InterfDemo {
  // NB: kein abstract weil implizit abstrakt
  public int methodeEins(String s);
  public int methodeZwei(String s);
}
```

```
public class ClassDemo implements InterfDemo {...}
```

→ Keyword `interface`
→ Kann *nur* abstrakte Methoden enthalten
  → Alle Methoden implizit `abstract`
  → ebenfalls **kein** Konstruktor
→ Klasse kann *mehrere* Interfaces *implementieren*
  → Keyword `implements`
  → **Alle** "geerbten" Methoden **müssen** überschrieben werden

Exceptions und I/O

Arten von Exceptions

→ Checked Exceptions
  → Exceptions die ein gutes Programm abfangen sollte
  → *Beispiel:* File öffnen der nicht existiert
  ⚠ Catch or specify requirement
→ Runtime Exceptions
  → logische Fehler im Code
  → *Beispiel:* Zugriff auf nichtexistentes Arrayfeld
→ Errors
  → Nicht/schlecht antizipierbare Fehler
  → *Beispiel:* File wegen Hardwarefehler unlesbar

try-catch Block

```
try {
  // Code der Exception schmeissen kann
} catch (⌞Exceptiontyp⌝ e) {
  // "Handling" der Exception
} finally {
  // "Cleanup" das immer ausgefuehrt wird
  // Egal ob Exception auftritt oder nicht
}
```

→ Verkettung mehrerer `catch`-Blöcke möglich
→ `catch`-Block optional (z.B. wenn man ein Cleanup für Code schreiben möchte, in dem garkeine Exception geworfen wird)
→ `finally`-Block ist optional (z.B. wenn kein Cleanup nötig)

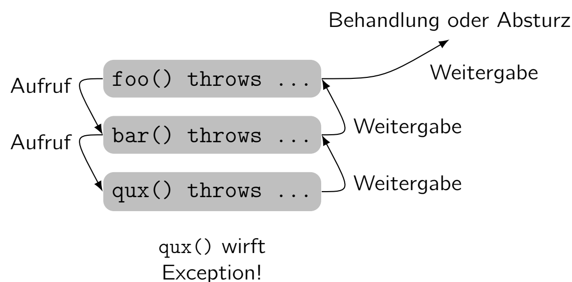
try-with-resources

```
try (
  // Initialisierung der Ressourcen nach try
  BufferedReader br = new BufferedReader(new
      ↪ FileReader("input.txt"));
  BufferedWriter bw = new BufferedWriter(new
      ↪ FileWriter("output.txt"));
) {
  String c;
  while ((c = br.readLine()) != null) {
    bw.write(c);
    bw.newLine();
  }
}
```

→ Alternative zu try-catch für **Ressourcen**-Objekte
  → Ressourcen = “Objekte die man schließen muss”
→ Kein schließen in `finally`-Block mehr nötig, übernimmt JVM

throws-Spezifikation

```
public void foo() throws ⌞Exceptiontyp⌝ {
  // Code der Exception vom Typen
  // ⌞Exceptiontyp⌝ werfen kann
}
```



Lesen und Schreiben mit BufferedReader/BufferedWriter

```
public class BufferedReaderWriterTest {
  public static void main(String[] args) throws
      ↪ IOException {
    BufferedReader br = null;
    BufferedWriter bw = null;
    try {
      // Initialisierung mit Quell-/Zieldatei
      br = new BufferedReader(new
          ↪ FileReader("input.txt"));
      bw = new BufferedWriter(new
          ↪ FileWriter("output.txt"));
      // c ist Zwischenspeicher fuer Gelesenes
      String c;
      // Lesen & Speichern in c und
      // Pruefen ob am Ende der Quelle
      while ((c = br.readLine()) != null) {
        // Schreiben des Gelesenen
        bw.write(c);
        bw.newLine();
      }
    } finally {
      // Ressourcen schliessen!
      if (br != null) br.close();
      if (bw != null) bw.close();
    }
  }
}
```

→ `br.readLine()` liest Zeile ein **und springt zur Nächsten Zeile**
→ `Buffered`-Klassen lesen/schreiben Zeile für Zeile
→ `FileReader` und `FileWriter` lesen/schreiben Zeichenweise

⚠ Sind Ressourcen: Müssen am Ende geschlossen werden!

Regex

→ Für die Syntax der Regex-**Patterns** gibt es ausführliche Cheatsheets, wird deswegen hier nicht behandelt
→ Nicht vergessen: Es gibt Tools um seine Patterns zu testen

Escaping in Java

→ Escaping = Bedeutungsveränderung eines Zeichen durch vorangestelltes \ (z.B. \a)
→ Java nutzt Escaping *unabhängig von Regex*
  → Daher in Regex-Pattern-String \ nochmals Escapen: \a → \\a

Klasse Pattern

→ `Pattern`-Objekte beschreiben Regex-Muster und werden in einem `Matcher` zum Auffinden der Muster genutzt
→ Erzeugung nicht mit Konstruktor, sondern:

```
Pattern p = Pattern.compile(⌞Pattern als String⌝);
```

Klasse Matcher

→ `Matcher`-Objekte werden mit String “gefüttert” in dem gematcht werden soll
→ Erzeugung ebenfalls nicht mit Konstruktor, sondern aus `Pattern`-Objekt:

```
Matcher m = p.matcher(⌞String zum Matchen⌝)
```

→ `boolean matches()`
  Überprüfen ob *gesamter* String mit Pattern übereinstimmt
→ `boolean find()`
  Matcher sucht nächstes Vorkommen vom Pattern in ⌞*String zum Matchen*⌝, returned `true` wenn ja, `false` wenn nein.
→ `String group(int n)`
  Returned die *n*-te Capturing Group im zuletzt durch `find()` gefundenen Match. *Ohne Argument* wird per default die *0*-te Gruppe ausgegeben.

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

Interface `Comparable<T>`

→ Klasse die es implementiert hat *natürliche Ordnung*
→ Ermöglicht Sortierung, z.B. durch
  → Verwendung von `TreeSet` oder `TreeMap`
  → Methode `Collections.sort(List l)`
→ Muss hierfür `compareTo(T obj)` überschreiben:

```
public class Foo implements Comparable<Foo> {
  public int id;
  // Rest der Klasse ...
  @Override
  // Beispiel: Sortierung nach Feld id
  public int compareTo(Foo f) {
    return Integer.compare(this.id, f.id);
  }
}
```

→ `compareTo` muss ein `int` value returnen:
  → value $< 0 \Rightarrow$ `this` $<$ `f`
  → value $= 0 \Rightarrow$ `this` $=$ `f`
  → value $> 0 \Rightarrow$ `this` $>$ `f`

Interface `Comparator<T>`

→ Idee: `Comparator`-Objekt das `sort`-Methode übergeben wird und bestimmt *wie* sortiert wird
→ Umsetzung: Neue `Comparator`-Klasse muss das Interface `Comparator` implementieren
  → Überschreiben von `compare(T obj1, T obj2)` aus Interface (analog zu `compareTo` oben)
→ Verwendung in `Collections.sort(List l, Comparator c)`

Empfehlungen

Think Java

Für alle die `null` Erfahrung mit dem Programmieren haben
→ http://thinkjava.org/

Oracle Java Tutorials

Größtenteils gute Tutorials für den Einstieg ins Programmieren mit Java
→ https://docs.oracle.com/javase/tutorial/index.html

Java ist auch eine Insel

Erklärt die Grundlagen (und mehr) gut, kompakt und verständlich
→ http://openbook.rheinwerk-verlag.de/javainsel/index.html

Rosalind

Bioinformatik-Programmierübungen von einfach bis komplex
→ http://rosalind.info/

Project Euler

Ähnlich zu Rosalind, Aufgaben sind aber eher mathematische Puzzles
→ https://projecteuler.net/