Iterable

```
public interface Iterable<T> {
  Iterator<T> iterator();
}
```

Collection

```
public interface Collection<E>
    extends Iterable<E> {
  // True, falls sich die Collection durch
  // add / addAll verändert hat
  public boolean add(E e);
  public boolean addAll
      (Collection<? extends E> c);

  public void clear();
  public boolean contains(Object o);
  public boolean containsAll(Collection<?> c);
  public boolean isEmpty();
  public int size();

  // Reihenfolge ist nur bei bestimmten Collection-
  // Klassen definiert, z.B. LinkedList
  // Änderungen am zurückgegebenen Array ändern
  // die Collection selbst nicht (nicht jedoch
  // bei Änderungen an Array-Elementen).
  public Object[] toArray();

  // Entfernt erstes Element e, welches o.equals(e)
  // erfüllt. True, falls ein solches e existiert.
  public boolean remove(Object o);

  // Enterfernt alle Elemente, die auch in c
  // vorkommen. True, falls sich die Collection
  // dadurch ändert.
  public boolean removeAll(Collection<?> c);

  // Info: hashCode(), .equals() sind definiert
}
```

Iterator

```
public interface Iterator<E> {
  public boolean hasNext();
  public E next();

  // Löscht Element, welches zuletzt von next()
  // zurückgegeben wurde aus der dahinterliegenden
  // Collection. Nur 1x pro next()-Aufruf verwendbar.
  public void remove();
}
```

Comparable

Definiert Totalordnung auf implementierende Objekte.

```
public interface Comparable<T> {
  // Ist this größer als other, return > 0
  // Ist this gleich wie other, return 0
  // Ist this kleiner als other, return < 0
  // Muss transitiv sein
  public int compareTo(T other);
}
```

Comparator

```
public interface Comparator<T> {
  // Ist o1 größer als o2, return > 0
  // Ist o1 gleich wie o2, return 0
  // Ist o1 kleiner als o2, return < 0
  public int compare(T o1, To2);

  // Return value ist Comparator mit umgekehrter
  // Reihenfolge
  public default Comparator<T> reversed();
}

// Vorgefertigt in Klasse String
// (String.CASE_INSENSITIVE_ORDER)
public static final Comparator<String>
    CASE_INSENSITIVE_ORDER;
```

Set

```
public interface Set<E> extends Collection<E> {}
```

Interface, welches Collection erfüllt; beachte:

- Ein Set enthält keine zwei Elemente e1 und e2, sodass e1.equals(e2)
- Ein Set darf sich nicht selbst enthalten
- Objekte im Set sollten nicht modifiziert werden, sodass nachträglich e1.equals(e2) gilt, sonst undefined behaviour.

HashSet

```
public class HashSet<E> implements Set<E> {
  public HashSet();
  public HashSet(Collection<? extends E> c);
}
```

Verwendet HashMap im Hintergrund. Hat keine Ordnung.

SortedSet

```
public interface SortedSet<E> extends Set<E> {
  // Verwendeten Comparator erhalten, null für
  // die durch Comparable induzierte Ordnung
  Comparator<? super E> comparator();

  // first = niedrigstes, last = höchstes bzgl. Ordnung
  E first();
  E last();
}
```

Die Iteration erfolgt in *aufsteigender* Reihenfolge. Nur interface!

NavigableSet

```
public class NavigableSet<E>
    extends SortedSet<E> {
  // ceiling: Kleinstes Element >= e (non-existent: null)
  // floor: Größtes Element <= e (non-existent: null)
  public E ceiling(E e);
  public E floor(E e);

  // higher = ceiling, lower = floor; aber mit > / <
  public E higher(E e);
  public E lower(E e);

  // headSet(): Alle Werte < to (oder <= falls incl.)
  // tailSet(): Alle Werte > from (oder >= falls incl.)
  public SortedSet<E> headSet(E to,
      boolean inclusive);
  public SortedSet<E> tailSet(E from,
      boolean inclusive);

  // iterator() (da Iterable) ist aufsteigend
  // descendingIterator() ist absteigend
  public Iterator<E> descendingIterator();

  // Niedrigstes (first) / höchstes (last) Element
  // zurückgeben und entfernen (non-existent: null)
  public E pollFirst();
  public E pollLast();
}
```

TreeSet

```
public class TreeSet<E>
    implements NavigableSet<E> {
  // Neues, leeres Set. Entweder in Comparable-Ordnung
  // oder durch comparator geordnet
  public TreeSet();
  public TreeSet
      (Comparator<? super E> comparator);

  // Automatisch nach Comparable-Ordnung sortieren
  public TreeSet(Collection<? extends E> c);

  // Behält Ordnung bei
  public TreeSet(SortedSet<E> s);
}
```

List

```
public interface List<E> extends Collection<E> {
  // Das index-te Elemente der Liste und alle folgenden
  // werden nach hinten verschoben und element bzw. c
  // davor eingefügt. Die Zählung beginnt bei 0!
  public void add(int index, E element);
  public void addAll(int index,
      Collection<? extends E> c);

  // Element an Stelle index abfragen / setzen
  // set gibt Element zurück, das zuvor an index stand
  public E get(int index);
  public E set(int index, E element);

  // indexOf (lastIndexOf) gibt Index des ersten
  // (letzten) Auftretens von o zurück (bzgl. equals)
  // oder -1, falls nicht gefunden
  public int indexOf(Object o);
  public int lastIndexOf(Object o);

  // Ist c == null, so wird nach Comparable-Ordnung
  // sortiert. Sortiert aufsteigend.
  public void sort(Comparator<? super E> c);

  // inklusive fromIndex; exklusive toIndex
  public List<E> subList(int fromIndex,
      int toIndex);
}
```

ArrayList

```
public class ArrayList<E> implements List<E> {
  public ArrayList();
  public ArrayList(Collection<? extends E> c);
}
```

Im Gegensatz zur LinkedList garantiert ArrayList schnellen Zugriff auf beliebige Indizes.

Deque

```
// Double-Ended Queue, "deck"
public interface Deque<E>
    extends Collection<E> {
  // Einfügen am Anfang / Ende, addLast = add
  public void addFirst(E e);
  public void addLast(E e);

  // Gibt erstes / letztes Objekt zurück oder null
  // falls Deque leer
  public E peekFirst();
  public E peekLast();

  // Löscht erstes / letztes Objekt und gibt es zurück,
  // oder null falls Deque leer
  public E pollFirst();
  public E pollLast();

  // Statt peek / poll gibt es auch get / remove
  // Diese werfen Exception falls Deque leer
}
```

LinkedList

```
public class LinkedList<E>
    implements List<E>, Deque<E> {
  public LinkedList();
  public LinkedList(Collection<? extends E> c);
}
```

Map

```
public interface Map<K,V> {
  public V get(Object key);

  // Gibt vorigen Wert an Stelle key zurück oder null
  public V put(K key, V value);
  public void putAll(Map<K,V> m);

  public boolean containsKey(Object key);
  public boolean containsValue(Object value);

  // Gibt entfernten Wert (oder null) zurück
  public V remove(Object key);
  public void clear();
  public boolean isEmpty();

  public int size();

  // Zum Iterieren geeignet:
  public Set<Map.Entry<K,V>> entrySet();
  public Set<K> keySet();
  public Collection<V> values();
}
```

Map.Entry

```
public interface Map.Entry<K,V> {
  public K getKey();
  public V getValue();

  // Verändert auch Eintrag in der Map selbst
  public V setValue();
}
```

HashMap

```
public class HashMap<K,V> implements Map<K,V> {
  public HashMap();
  public HashMap(Map<K,V> m);
}
```

SortedMap

```
public class SortedMap<K,V> extends Map<K,V> {
  public Comparator<? super K> comparator();

  // first = niedrigster Key, last = höchster Key
  public K firstKey();
  public K lastKey();
}
```

NavigableMap

```
public class NavigableMap<K,V>
    extends SortedMap<K,V> {
  // Siehe NavigableSet
  public Map.Entry<K,V> ceilingEntry(K key);
  public Map.Entry<K,V> floorEntry(K key);

  public NavigableMap<K, V> descendingMap();

  // Zurückgeben und entfernen
  public Map.Entry<K,V> pollFirstEntry();
  public Map.Entry<K,V> pollLastEntry();
}
```

TreeMap

```
public class TreeMap<K,V>
    implements NavigableMap<K,V> {
  public TreeMap();
  public TreeMap(Map<K,V> m);
}
```

String

```
// CharSequence ist Interface, das String implementiert!

public class String implements CharSequence,
    Comparable<String> {
  public String(); // leerer String
  public String(byte[] bytes);
  public String(char[] value);
  public String(String original);

  public char charAt(int index);
  public char[] toCharArray();

  public boolean contains(CharSequence s);
  public boolean startsWith(String prefix);
  public boolean endsWith(String suffix);
  public boolean isEmpty();

  public String replace(CharSequence target,
      CharSequence replacement);
  public String toLowerCase();
  public String toUpperCase();

  // "qbc".substring(2, 3) -> "c"
  // "Harbison".substring(3) -> "bison"
  public String substring(int begin, int end);
  public String substring (int beginIndex);

  // Whitespace am Anfang / Ende entfernen
  public String trim();

  // String an regex (bzw. einfach String) trennen
  public String[] split(String regex);

  // Index des ersten / letzten Auftretens von str
  public int indexOf(String str);
  public int lastIndexOf(String str);
  public int hashCode();
  public int length();

  // Implementiert Comparable mit lexikal Reihenfolge
  // (Unicode / Länge), return value > 0, falls other
  // früher im Alphabet
  public int compareTo(String other);

  // String-Darstellung der Werte. b kann sein:
  // boolean, char, double, float, int, long, Object
  public static String valueOf(b);
}
```

Object

```
public class Object {
  // Muss reflexiv, symmetrisch, transitiv
  // und gleichbleibend sein
  public boolean equals(Object obj);

  protected Object clone()
      throws CloneNotSupportedException;

  // Für Zwei Objekte obj1, obj2 muss gelten:
  // obj1.equals(obj2)
  // => obj1.hashCode() = obj2.hashCode()
  public int hashCode();

  public String toString();
}
```

Datentypen

Attribute und statische Attribute werden von Java auf vorgegebenen Standardwerte initialisiert. Bei lokalen Variablen geschieht dies nicht!

Type	Min	Max	Init	Wrapper
byte	-128	127	0	Byte
short	-2^{15}	$2^{15}-1$	0	Short
int	-2^{31}	$2^{31}-1$	0	Integer
long	-2^{63}	$2^{63}-1$	0L	Long
float	$-\infty$	∞	0.0f	Float
double	$-\infty$	∞	0.0d	Double
boolean	-	-	false	Boolean
String	-	-	null	-
Object	-	-	null	-

Die Wrapper-Klassen `Byte`, `Short`, `Integer`, `Long` enthalten die statischen Konstanten `MAX_VALUE` und `MIN_VALUE`. Die Konstruktoren aller Wrapperklassen akzeptieren auch einen String!
`Float` / `Double` enthalten beide die statischen Konstanten `NEGATIVE_INFINITY` und `POSITIVE_INFINITY`.

Arrays

Arrays haben ein Konstantes `.length`-Attribut, sind aber keine Klasse. Die Klasse `Arrays` enthält einige statische Array-Hilfsfunktionen:

```
public class Arrays {
  // T kann sein: byte, char, double, float int, long,
  // Object, short

  // Sucht key in aufsteigend sortiertem Array a; gibt
  // index zurück (negativ, falls nicht gefunden)
  public static int binarySearch(T[] a, T key);

  // Wahr, wenn gleiche Elemente in gleicher Reihenfolge
  public static boolean equals(T[] a, T[] b)

  public static void fill(T[] a, T val);

  // Sortiert aufsteigend (Quicksort, O(n log(n)))
  public static void sort(T[] a);
  public static void sort
      (T[] a, Comparator<T> c);

  // deepToString führ mehrdimensionale Arrays
  // Ausgabe z.B. "[1, 2, 3]", "[[1, 2], [2, 3]]"
  public static void toString(T[] a);
  public static void deepToString(T[] a);
}
```

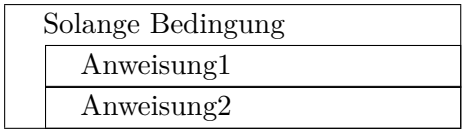
Reservierte Schlüsselwörter

`abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, false, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while`

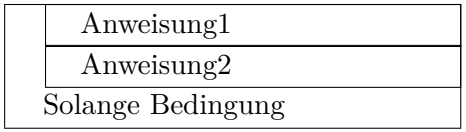
Math

```
public class Math {
  public static double E;
  public static double PI;

  // T kann sein: double, float, int, long
  public static T abs(T a);
  public static T max(T a, T b);
  public static T min(T a, T b);

  public static double sin(double a);
  public static double cos(double a);
  public static double tan(double a);

  public static double sinh(double x);
  public static double cosh(double x);
  public static double tanh(double x);

  public static double acos(double a);
  public static double asin(double a);
  public static double atan(double a);
  public static double atan2
      (double y, double x);

  // log = natürlicher logarithmus
  public static double exp(double a);
  public static double log(double a);
  public static double log10(double a);

  // Quadrat- / Kubikwurzel
  public static double sqrt(double a);
  public static double cbrt(double a);

  public static double pow(double a, double b);

  // Runden auf größere / kleinere / nächste Ganzzahl
  // Beachte Typen der Rückgabewerte!!
  public static double ceil(double a);
  public static double floor(double a);
  public static long round(double a);
  public static int round(float a);

  // Zufälliger Wert zwischen 0.0 und 1.0
  public static double random();
}
```

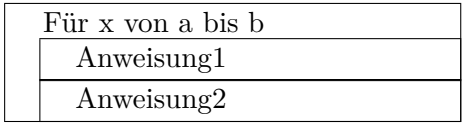
Struktogramm: while



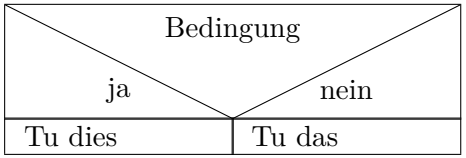
Struktogramm: do-while



Struktogramm: for

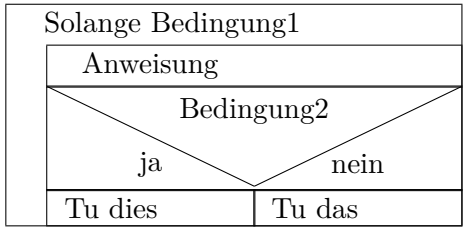


Struktogramm: if



Struktogramm: Äußere Merkmale

`SomeClass.someMethod()`



`Parameter: String param1, Integer param2`

Im Struktogramm sollen keine Java-Befehle stehen, sondern Allgemeine Beschreibungen. Für Parameter und Attribute können auch Buchstaben als Abkürzungen angegeben werden.

Wichtige Exceptions

```
// Alles was beim Ausführen so schief geht
public class RuntimeException
    extends Exception;

// Beispiel: Durch 0 teilen
public class ArithmeticException
    extends RuntimeException;

// Lesen / Schreiben von Elementen, deren
// Index nicht im Array / String ist
public class IndexOutOfBoundsException
    extends RuntimeException;
public class ArrayIndexOutOfBoundsException
    extends IndexOutOfBoundsException;
public class StringIndexOutOfBoundsException
    extends IndexOutOfBoundsException;

// Unerwartete Verwendung eines null-Werts
public class NullPointerException
    extends RuntimeException;
```

Eigene Exception

```
public class CustomException extends Exception {
  int value;

  public CustomException(int value) {
    this.value = value;
  }
}
```

`throws` sollte dann angegeben werden, wenn eine Methode eine Exception nicht selbst behandelt, sondern sie "nach oben"weiterleitet. Wenn throws spezifiziert wird, muss der Methodenaufruf mit try-catch abgesichert werden. `throws` muss aber nicht angegeben werden.

```
public class ExceptionMain {
  public void doSomething(int value)
      throws CustomException {
    int limit = 0;

    if (value > limit) {
      throw new CustomException(value);
    }
  }
}
```

Algorithmus: Bubblesort

```
public class BubbleSort {
  // Sortierung durch Vergleichen zweier nebeneinander liegender Elemente
  // und anschließendes Vertauschen, falls kleineres Element weiter oben im Array
  // Mit jedem Durchlauf durch die Liste ist ein weiteres Element sortiert

  // Iterativer Ansatz:
  public static void bubblesort(int[] array) {
    for (int i = 1; i < array.length; i++) {
      for (int j = 0; j < (array.length - i); j++) {
        if (array[j] > array[j + 1]) {
          int h = array[j];
          array[j] = array[j + 1];
          array[j + 1] = h;
        }
      }
    }
  }
}
```

Algorithmus: Binärsuche

```
public class BinarySearch {
  // Suche auf einem geordneten Array mit Rückgabe des Index
  public static int binarySearch(int[] array, int element) {
    int leftIndex = 0;
    int rightIndex = array.length - 1 ;

    while(leftIndex <= rightIndex) {
      int middle = leftIndex + ((rightIndex - leftIndex) / 2);

      if (array[middle] == element) {
        return middle;
      } else if (array[middle] > element) {
        rightIndex = middle - 1;
      } else {
        leftIndex = middle + 1;
      }
    }

    return -1;
  }
}
```