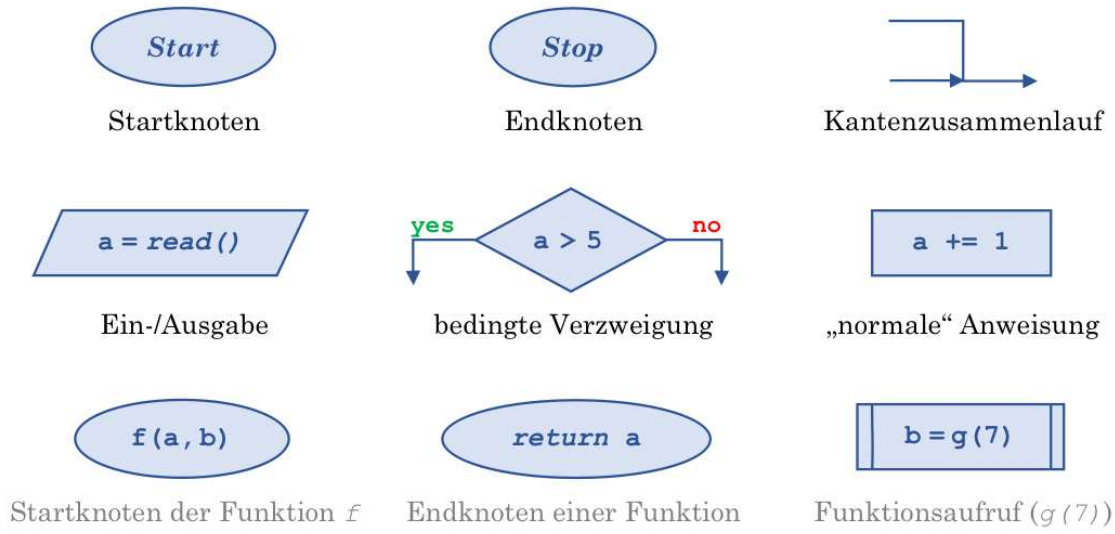
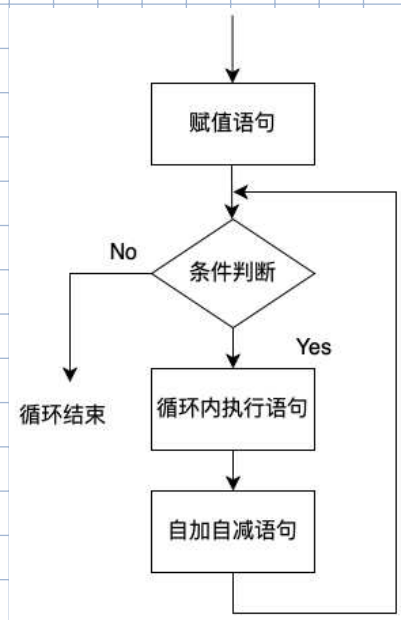
Kontrollflussdiagramm

Knotenarten

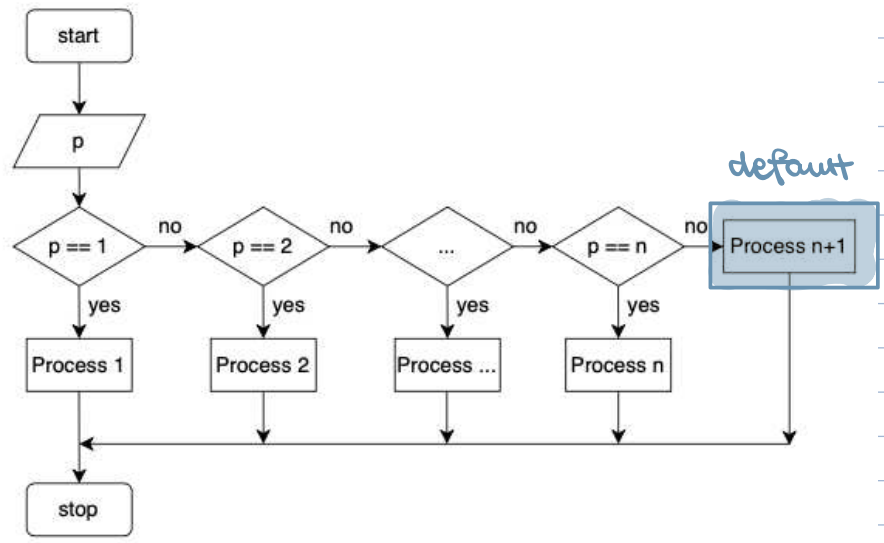


for-Schleife

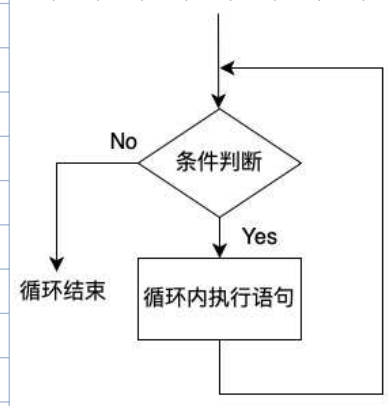


switch

```
// 情况1: 每个case后都有break
    switch(p) {
        case 1:
            process 1;
            break;
        case 2:
            process 2;
            break;
        ...
        case n:
            process n;
            break;
        deafult:
            process n+1;
            break;
    }
```

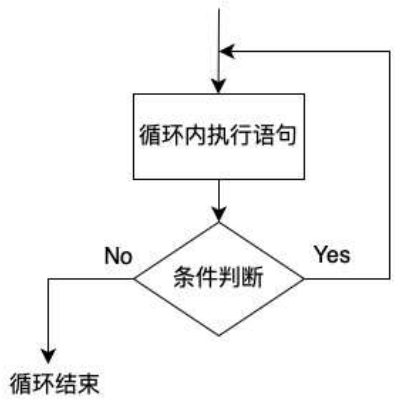


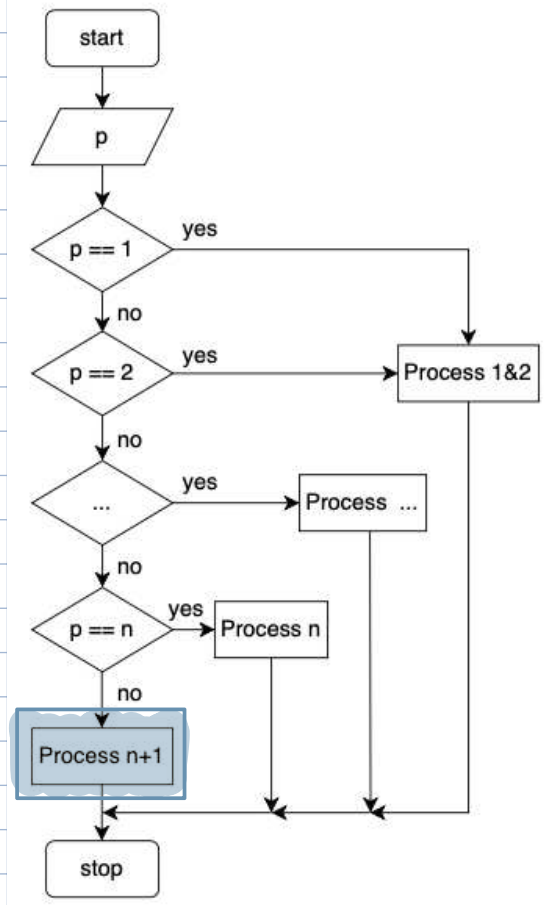
while-Schleife




```
//情况2: 每个case后不一定有break
    switch(p) {
        case 1:
        case 2:
            process 1&2;
            break;
        ...
        case n:
            process n;
            break;
        default:
            process n+1;
            break;
    }
```

do-while-Schleife





MiniJava-Syntax

```
<program>   ::=  <decl>* <stmt>*
<decl>      ::=  <type> <name> (, <name>)* ;
<stmt>      ::=  <expr>? ;
             |   { <stmt>* }
             |   <name> = <expr> ;
             |   <expr> [ <expr> ] = <expr> ;
             |   if ( <cond> ) <stmt> (else <stmt>)?
             |   while ( <cond> ) <stmt>
             |   do <stmt> while ( <cond> );
             |   return <expr> ;
<expr>      ::=  <number>
             |   <name>
             |   new int [ <expr> ]
             |   <expr> [ <expr> ]
             |   length( <expr> )
             |   read ( )
             |   write ( <expr> )
             |   <name> ( (ϵ | <expr> (, <expr>)*) )
             |   ( <expr> )
             |   <unop> <expr>
             |   <expr> <binop> <expr>
<cond>      ::=  true
             |   false
             |   ( <cond> )
             |   <expr> <comp> <expr>
             |   <bunop> ( <cond> )
             |   <cond> <bbinop> <cond>
<unop>      ::=  - (NEG)
<binop>     ::=  - | + | * | / | %
<bunop>     ::=  !
<bbinop>    ::=  && | ||
<comp>      ::=  == | != | <= | < | >= | >
<type>      ::=  int | int[]
<name>      ::=  [A-Za-z_] [A-Za-z_]*
<number>    ::=  0 | ([1-9] [0-9]*)
```

Ein Syntaxbaum kann für ein Programm gezeichnet werden, auch wenn es Kompilierungsfehler enthält, weil der Baum nur die strukturelle Syntax des Codes repräsentiert, nicht seine Semantik z.B. korrekte Deklaration von Variablen

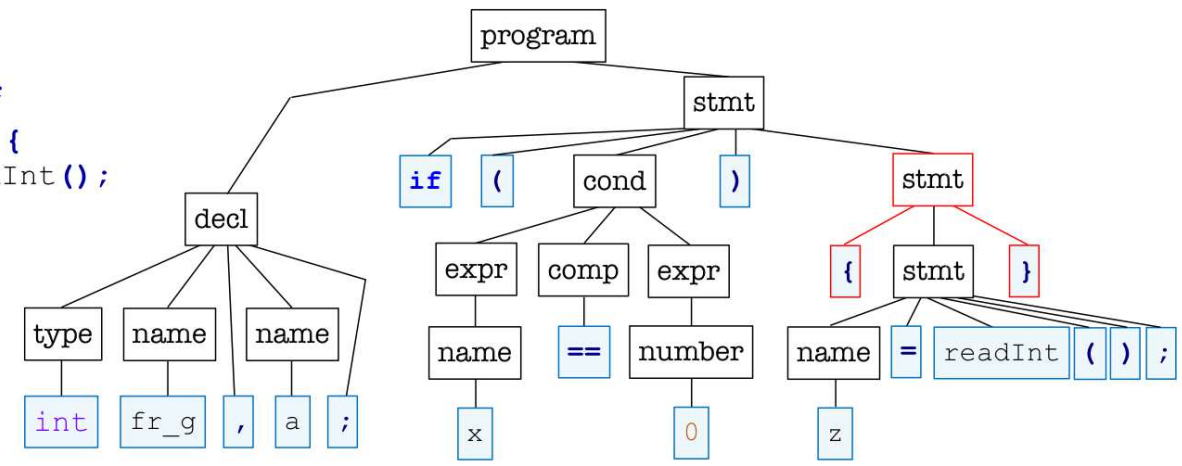
Relevante Operatoren:

		Bindungsstärke:
a	Literal/Terminalzeichen (a)	
x?	**Option** (keinmal oder einmal)	stark
x*	**Iteration** (keinmal, einmal oder beliebig oft)	
x+	Iteration (mindestens einmal)	
x y	**Konkatenation** (Verknüpfung)	mittel
x | y	**Alternative** (entweder x oder y)	schwach
a | ... | z	Alternativen (eines der Zeichen von a bis z)	

Syntaxbaum

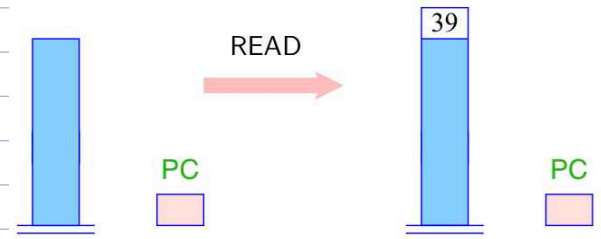
Beispiel:

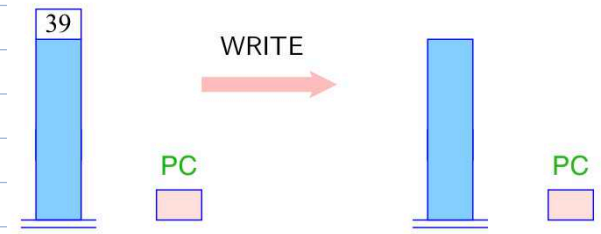
```
int fr_g, a;
if (x == 0) {
    z = readInt();
}
```

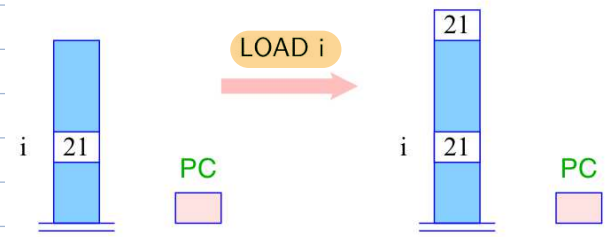


MiniJVM-Befehl

int-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
boolean-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendung des Programms:	HALT









- LOAD i erzeugt eine Kopie der Speicherzelle i oben auf dem Stack
- STORE i überschreibt Speicherzelle i mit dem obersten Wert auf dem Stack und löscht diesen dann oben

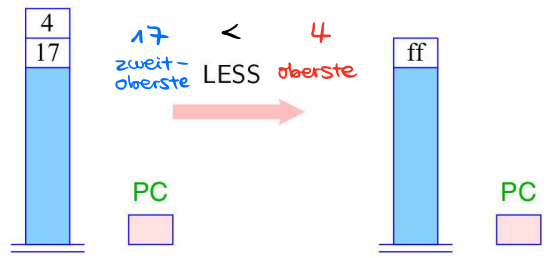
Beispiel für CALL

```
int a;
a = Math.pow(3, 2);
```

```
ALLOC 1
CONST 3
CONST 2
CALL Math.pow
STORE 0
```

das unterste Stack-Element -> das erste Argument der Methode
das zweite Stack-Element -> das vorletzte Argument der Methode
das oberste Stack-Element -> das letzte Argument der Methode

CONST 17
CONST 4
LESS



Übersetzung: MiniJava zu MiniJVM-Bytecode

Unäre Operation

modifizieren die oberste Zelle

-expr

```
<Übersetzung von expr>
NEG
```

!cond

```
<Übersetzung von cond>
NOT
```

Binäre Operation

verkürzen den Stack

$expr_1$ - $expr_2$

```
<Übersetzung von expr1>
<Übersetzung von expr2>
SUB
```

$expr_1$ <= $expr_2$

```
<Übersetzung von expr1>
<Übersetzung von expr2>
LEQ
```

$cond_1$ || $cond_2$

```
<Übersetzung von cond1>
<Übersetzung von cond2>
OR
```

if-Statement

if (cond) stmt

```
   <Übersetzung von cond>
   FJUMP A
   <Übersetzung von stmt>
A: ...
```

if-else-Statement

if (cond) $stmt_1$ else $stmt_2$

```
   <Übersetzung von cond>
   FJUMP A
   <Übersetzung von stmt1>
   JUMP B
A: <Übersetzung von stmt2>
B: ...
```

while-Statement

while (cond) stmt

```
A: <Übersetzung von cond>
   FJUMP B
   <Übersetzung von stmt>
   JUMP A
B: ...
```

Primitive Datentypen

Variablennamen dürfen nur Groß- oder Kleinbuchstabe (ü oder ß auch erlaubt), Ziffern (aber nicht am Beginn des Variables), Unterstriche und Dollarzeichen enthalten, alle anderen Sonderzeichen, Leerzeichen oder Reservierte Wörter (z.B. `int`, `public`, `final`) sind verboten

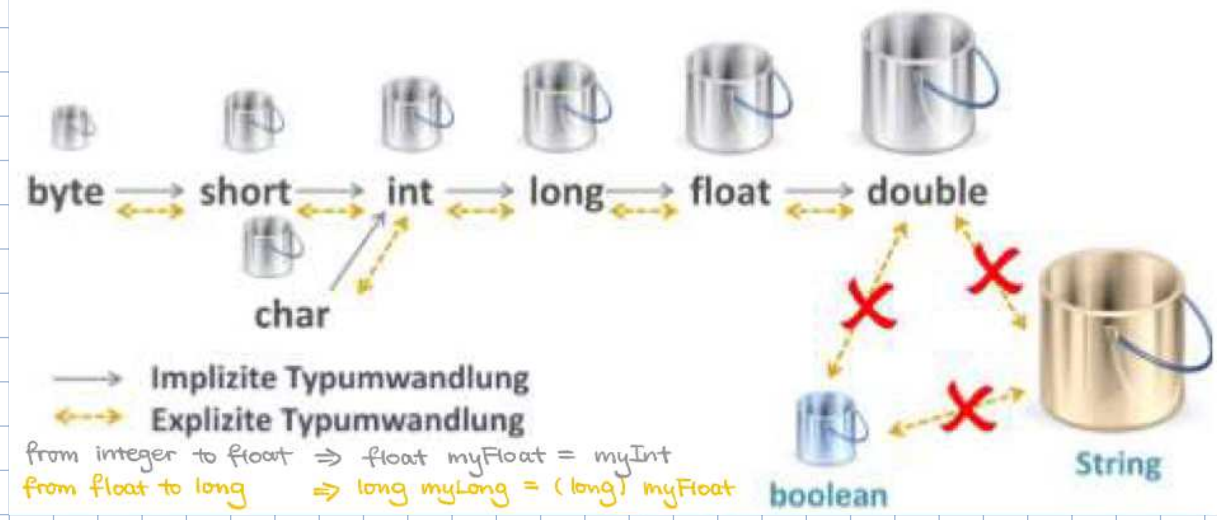
	Typ	Größe (Bit)	Wertebereich	Suffix	*Wrapperklasse* (in `java.lang`)
boolescher Wert	`boolean`	undefiniert	`false` / `true`	-	`Boolean`
Ganze Zahlen	`byte`	8	-2^7 ... 2^7-1 -128 ... 127	-	`Byte`
	`short`	16	-2^{15} ... $2^{15}-1$ -32768 ... 32767	-	`Short`
	`int`	32	-2^{31} ... $2^{31}-1$ -2147483648 ... 2147483647	-	`Integer`
	`long`	64	-2^{63} ... $2^{63}-1$ ca. $-9 \cdot 10^{18}$ ... $9 \cdot 10^{18}$	`l` / `L`	`Long`
Gleitkommazahlen (nach IEEE 754)	`float`	32 » Vorzeichen: 1 » Mantisse: 23 » Exponent: 8	ca. $\pm 1{,}4 \cdot 10^{-45}$ ... $\pm 3{,}4 \cdot 10^{38}$ und `NaN`, `±Infinity`, `±0` (Exponent: -126 ... 127)	`f` / `F`	`Float`
	`double`	64 » Vorzeichen: 1 » Mantisse: 52 » Exponent: 11	ca. $\pm 4{,}9 \cdot 10^{-324}$ ... $\pm 1{,}7 \cdot 10^{308}$ und `NaN`, `±Infinity`, `±0` (Exponent: -1022 ... 1023)	(`d` / `D`)	`Double`
Unicode-Zeichen (UTF-16)	`char`	16	0 ... $2^{16}-1$ 0 ... 65535 (z. B. 65 ≙ `'A'`)* `'\u0000'` ... `'\uFFFF'`	-	`Character`

* Für uns interessant sind i. d. R. nur die ersten 128 Zeichen (ASCII), vgl. *www.torsten-horn.de/techdocs/ascii.htm*

int(wert) casten: Zahl wird abgeschnitten, nicht gerundet. d.h. nur Ganzzahlanteil beibehalten, Dezimalanteil weggelassen - z.B. (int) 2.99 = 2

double a = 10/4 = 2.0 (wenn alle Operand Integer, dann erst 10/4 = 2, und dann als double 2.0)
double a = 10.0/4 = 2.5 und double a = 10/4.0 = 2.5 (mindestens ein Operand als double)

Typenumwandlung



falls hier ohne explizite Typenumwandlung `long myLong = myFloat` -> Compiler-Fehler

Enumeration

Zugriff mit Punkt-Operator `Weekday.MONDAY`

```
enum Weekday {
    MONDAY(8),   //1.提供当前枚举类的对象
    TUESDAY(10),
    WEDNESDAY(8),
    THURSDAY(8),
    FRIDAY(6),
    SATURDAY(0), //多个对象之间用","隔开
    SUNDAY(0);   //结尾以";"结束

    private int workingHours; //2.声明该类的属性

    Weekday(int workingHours){ //3.提供类构造器
        this.workingHours = workingHours;
    }
    public int getWorkingHours(){ //4.提供类的方法
        return this.workingHours;
    }
}
```

Operatoren Priorität - Bindungsstärke Präferenz

stark

Rang	Beschreibung	Operator / Symbol
0	Klammerung	`(...)`

1	Funktionsaufruf
	Array-Indizierung
	Objekterzeugung
	Memberzugriff (z. B. Attribute)
		`someFunction(...)`
		`someArray[someIndex]`
		`new ...`
		`someObject.someMember`

2	Postinkrement/-dekrement	`x++, x--`

3	Präinkrement/-dekrement
	unäre Operatoren
	Typecast
		`++x, --x`
		`+x, -x, !x, ~x`
		`(someType) x`

4	Multiplikation, Division Modulo (Teilerrest)	`a * b, a / b` `a % b`
5	Addition, Subtraktion Stringkonkatenation (Verknüpfung)	`a + b, a - b` `s + t`
6	Bitverschiebung (Shifts)	`d << k, d >> k, d >>> k`
7	Vergleich	`a < b, a > b, a <= b,` `a >= b, obj instanceof Cls`
8	Gleich / Ungleich	`a == b, a != b`
9	bitweises „Und“	`b & c`
10	bitweises „exklusives Oder“	`b ^ c`
11	bitweises „Oder“	`b | c`
12	logisches/boolesches „Und“	`B && C`
13	logisches/boolesches „Oder“	`B || C`
14	Bedingung (ternärer Operator)	`B ? y : n`

15	Zuweisung
	bzw.
	zusammengesetzte Zuweisungsoperatoren
	a Δ= b entspricht a = (*Typ von a*) (a Δ b)
		`a = b`
		`a += b, a -= b, a *= b,`
		`a /= b, a %= b, b &= c,`
		`b ^= c, b |= c, d <<= k,`
		`d >>= k, d >>>= k`

Kurzschluss-auswertung

x++ erhöht x um 1 und wertet zum *alten* Wert von x aus.

```
y = x++;
```
≡
```
y = x;
x = x + 1;
```

```
y = ++x;
```
≡
```
x = x + 1;
y = x;
```

++x erhöht x um 1 und wertet zum *neuen* Wert von x aus.

Zugriffsübersicht

Pro Datei kann nur eine `public class` haben und die Datei muss nach dieser Klasse benannt, sonst kompiliert nicht

	Überall	Subklasse	Package	Klasse
`public`	☑	☑	☑	☑
`protected`	☐	☑	☑	☑
default	☐	☐	☑	☑
`private`	☐	☐	☐	☑

Sichtbarkeit des Members darf bei Vererbung oder implements nicht verringert werden

Sonst -> Compiler - Fehler

In Java findet stets "Call-by-Value" statt. Im Falle von Objekten wie Arrays ist der übergebene "Wert" die Kopie von Referenz auf das Array

Call by Value (Wertparameter): Parametervariable erhält eine Kopie des Werts der übergebenen primitiven Typs (z.B. int). Innerhalb der aufgerufenen wird Funktion mit der Kopie gearbeitet und sich Änderungen nicht auf den ursprünglichen Wert auswirken

Call by Reference (Referenzparameter): Parametervariable erhält eine Referenz auf des übergebenen Referenzdatentyps (z.B. Objekt oder array). Bei einer Übergabe als Referenz wirken sich Änderungen an den Parametern auf den ursprünglichen Objekt aus, da die beiden variable referenzieren auf dasselbe Objekt

`==` Operator -> Referenzvergleich (Adressvergleich): überprüft, ob beide auf gleiche Objekt (Speicherort) zeigen

`equals()` Methode -> Inhaltsvergleich (ABER: Wenn eine Klasse von Objekt erbt und die equals()-Methode nicht überschreibt, tut sie gleich wie der `==` Operator, prüft auf Referenzgleichheit)

`instanceof`-Operator überprüft, ob das linke Objekt `a` mindestes so speziell wie die rechts genannte Klasse `b`

Falls `!(a instanceof b)` (bzw. false), können die Objekte nicht gleich sein

Heap-Speicherung für globale variable und Objekte (String und Arrays)

Stack-Speicherung für lokale variable (integer, boolean, byte, char) und Referenzen auf Objekte

Alle Threads teilen denselben Heap-Speicherbereich

Fallunterscheidungsarten

- **Einfach-Verzweigung „if – else“**
  - Zwei-Möglichkeiten-Entscheidung
  - Wenn-Dann/Sonst-Struktur

```
if (<Bedingung>){  //Anweisungen   }
else{              //Anweisungen   }
```

- **Hierarch. Verzweigung „if – else if“**
  - durch hierarchische Verschachtelung von Einfach-Verzweigungen

```
if(<BedingungA>)
   if(<BedingungB>)
   else if(<BedingungC>)
     // Anweisungen
```

- **Mehrfachverzweigung „if – else if“**
  - durch „flache“ Verschachtelung von Einfach-Verzweigungen

```
if(<BedingungA>) ...       //Anweisungen
else if(<BedingungB>) ... //Anweisungen
else if(<BedingungC>) ... //Anweisungen
```

- **Reinspring-Routine „switch“**
  - Bei Prozessen mit beliebig vielen diskreten Zuständen/Werten

```
switch(<variable>){
     case <wert1>: ... //Anweisungen
     case <wert2>: ... //Anweisungen
     …
     default:      ... //Anweisungen     }
```

- **Fallunterscheidung „switch“**
  - Bei Auswahlen zwischen beliebig vielen diskreten Zuständen/Werten

```
switch(<variable>){
     case <wert1>: ... break; //Anweisungen
     case <wert2>: ... break; //Anweisungen
     …
     default:      ... break; //Anweisungen}
```

`yield` innerhalb des switch-Ausdrucks dient dazu, einen Wert zurückzugeben und die Ausführung des switch-Ausdrucks zu beenden

Schleifen-Arten

Ohne geschweifte Klammern `{}` wird in Java nur die unmittelbar folgende Anweisung als Teil der Schleife betrachtet

muss ein boolescher Ausdruck sein -> sonst Compiler-Fehler

- **„for“-Schleife**
  - idR.: als „Zählschleife“

```
for(<Init.>; <Schleifenbed.>; <Zählvorschrift>){
      //Anweisungen
}
```

- **„while“-Schleife**
  - Kopfgesteuerte Schleife
  - Eingangskontrolle der Bedingung

```
while(<Bedingung>){
        //Anweisungen
}
```

- **„do-while“-Schleife**
  - Fußgesteuerte Schleife
  - Bedingungsprüfung erst nach dem ersten Schleifendurchlauf

```
do{
      //Anweisungen
} while(<Bedingung>);
```

- **Schleifenbefehle (überprüfen von Bedingungen in Schleifen und dann):**
  - **break**: Springt hinter die Schleife
  - **continue**: Springt an den Anfang der Schleife

`static` Statische Attribute und Methode (Klassenattribute und -methode)

- Statische Attribute haben in allen erzeugten Instanzen der Klasse den selben Wert (Ändert man in einer Methode einer Instanz den Wert eines statischen Attributes, so ändert sich automatische der Wert auch in allen anderen Instanzen, da alle Instanzen sich statische Attribute teilen)

- Statische Attribute und Methode gehören zur Klasse, nicht zu einem Objekt -> Aufruf erfolgt über `类名.静态变量` oder `类名.静态方法`
  ↳ lokale Variable kann nicht statisch sein
- Statische Attribute und Methode werden gelanden, wenn die Klassen geladen wird, und vernicht wenn die Klasse stirbt

- Einschränkungen für statische Methoden: 1) haben nur Zugriff auf statische Attribute/ Methode der Klasse; 2) `this` und `super` Referenz kann nicht verwendet werden; 3) Es darf keine statische und nicht-statische Methode mit der gleichen Signatur geben; 4) Statische Methoden können nicht `final` sein

- non-statische Methode kann Zugriff auf statische und non-statische Attribute/ Methode haben

- Globale Variablen (in Klasse definiert) werden automatisch mit Standardwerte initialisiert. int=0, boolean=false, Strings=null, Arrays mit 0 als Element
- Lokale Variablen (in Methode deklariert) werden nicht automatisch initialisiert -> müssen vor Verwendung explizit initialisiert werden

Rekursion

- ***Lineare Rekursion*** (häufigste Form): In jedem (rekursiven) Aufruf der Methode findet maximal ein rekursiver Aufruf statt, d. h. es wird entweder ein Rekursionsanker erreicht oder es findet exakt ein rekursiver Aufruf statt. Beispiele: Fakultät, GGT
- ***Endrekursion***: Nach Durchführung des rekursiven Aufrufs kann die Methode direkt verlassen werden, d. h. es sind keine weiteren Berechnungen oder Methodenaufrufe mehr nötig. Bei `void`-Methoden bedeutet das, dass der rekursive Aufruf das letzte sein muss, was vor dem Verlassen der Methode passiert. Bei allen anderen Methoden muss das Ergebnis des rekursiven Aufrufs unverändert zurückgegeben werden (z. B. `return f(n+1);`). Bei mehreren rekursiven Aufrufen innerhalb einer Methode muss diese Bedingung entsprechend für jeden rekursiven Aufruf erfüllt sein. Beispiele findest du im Unterkapitel „Endrekursion“.
- ***Verschachtelte Rekursion***: Rekursive Aufrufe werden als Parameter rekursiver Aufrufe verwendet (z. B. `f(f(n+1));`). Achtung: So etwas ist niemals endrekursiv.
- ***Kaskadenförmige Rekursion***: In mind. einem Fall finden mehrere rekursive Aufrufe statt (und werden ggf. miteinander verknüpft). Beispiele: Fibonacci, Binomialkoeffizient
- ***Verschränkte Rekursion***: Zwei verschiedene Methoden rufen sich wechselseitig auf. Beispiel: `f()` ruft `g()` auf und `g()` ruft `f()` auf.

Fibonacci-Zahlen

```java
// iterativ
public int fibIter(int n) {
    if (n < 0) return -1;
    if (n == 0) return 0;
    int a = 0;
    int b = 1;
    while (n > 1) {
        int tmpA = a;
        a = b;
        b += tmpA;
        n--;
    }
    return b;
}

// rekursiv
public int fibRec(int n) {
    if (n < 0) return -1;
    if (n == 0 || n == 1) return n;
    else {
        return fibRec(n-1) + fibRec(n-2);
    }
}
```

Fakultät

```java
// iterativ
public int facIter(int n) {
    int result = 1;
    while (n > 1){
        result *= n;
        n--;
    }
    return result;
}

// rekursiv
public int facRec(int n) {
    if (n <= 1) return 1;
    return n * facRec(n-1);
}
```

Print Sternendreieck

```java
public static void printTriangle(int size) {
    if (size == 0) return;
    printTriangle(size-1);
    System.out.println(stars(size));
}

private static String stars(int amount) {
    if (amount <= 0) return "";
    else {
        return "*" + stars(amount-1);
    }
}
```

Potenz “a hoch b”

```java
// iterativ
public int powerIter(int a, int b) {
    int result = 1;
    while (b > 0) {
        result *= a;
        b -= 1;
    }
    return result;
}
// rekursiv
public int powerRec(int a, int b) {
    if (b <= 0) return 1;
    return a * powerRec(a, b-1);
}
```

Palindrom

```java
// iterativ
public boolean isPalindromeIter(char[] c) {
    if (c == null) return false;
    for(int i = 0; i < c.length/2; i++) {
        if (Character.toLowerCase(c[i]) !=
            Character.toLowerCase(c[c.length-1-i]))
            return false;
    }
    return true;
}
// rekursiv
public boolean isPalindrome(char[] c) {
    if (c == null) return false;
    if (c.length == 0) return true;
    return isPalindrome(c, 0);
}
private boolean isPalindrome(char[] c, int pos) {
    if (pos >= c.length/2) return true;
    if (c[pos] != c[c.length-1-pos]) return false;
    return isPalindrome(c, pos+1);
}
```

größte gemeinsamer Teiler

```java
// rekursiv
public static int ggt(int a, int b) {
    if (a > b) return ggt(a-b, b);
    if (a < b) return ggt(a, b-a);
    return a;
}
```

Binomialkoeffizient (n über k)

```java
// iterativ
public int binomialDef(int n, int k) {
    return facRec(n) / (facRec(k) * facRec(n-k));
}
// rekursiv
public int binomialRec(int n, int k) {
    if (k < 0 || k > n) return -1;
    if (k == 0 || k == n) return 1;
    return binomialRec(n-1, k-1) + binomialRec(n-1, k);
}
```

alle durch 3 teilbare Zahlen

```java
// rekursiv
public static void multiplesOfThree(int n) {
    multiplesOfThree(n, 0 );
}
private static void multiplesOfThree(final int n, int i) {
    if (i > n) return;
    if (i != 0) System.out.print(", ");
    System.out.print(i);
    multiplesOfThree(n, i + 3);
}
```

alle digit durch newDigit ersetzen

```java
// rekursiv
public long replaceDigit(long number, byte digit, byte newDigit) {
    byte currentDigit = (byte) (number % 10);
    if (currentDigit == digit) currentDigit = newDigit;
    if (number < 10) return currentDigit;
    return replaceDigit(number / 10, digit, newDigit) * 10 + currentDigit;

}
```

Quersumme

```java
//iterativ
public int quersumme(int a) {
    int res = 0;
    while (a != 0) {
        res += a % 10;
        a /= 10;
    }
    return res;
}
// rekursiv
public int quersumme(int a) {
    if (a != 0){
        return a % 10 + quersumme(a / 10);
    }
    return 0;
}
```

Primzahlen

```java
// rekursiv
public boolean isPrime(int n) {
    if (n <= 1) return false;
    return isPrime(n, n-1);
}
private boolean isPrime(int n, int div) {
    if (div <= 1) return true;
    if (n % div == 0) return false;
    return isPrime(n, div-1);
}
```

Vererbung

- Gemeinsamkeiten in Oberklassen zusammenfassen und auf Unterklassen vererben
- Oberklasse bietet eine Generalisierung ihrer Unterklassen; Unterklassen sind Spezialisierungen Ihrer Oberklassen
- Eine Klasse kann nur eine Oberklasse erben -> alle `public` Attribute und Methode von Oberklasse erben
- Innerhalb der Unterklasse kann man den Konstruktur der Oberklasse mit der Referenz `super` (in erster Konstruktur-Zelle) aufrufen
- (innerhalb der Konstruktur darf `this` und `super` nicht gelichzeitig angewendet - NUR entweder `this` oder `super`)

Polymorphie

- die gleichnamige Methode auf unterschiedliche Art Innerhalb der Unterklasse implementieren
- Überschreibungsverbot mit Schlüsselwort **final** ~~oder static~~ (kann vererbt oder überladen, aber nicht überschrieben)
- **Überschreiben (Override):** Implementierung einer gleichnamigen Methode mit gleicher Anzahl/ Art der Parameter und Rückgabetyp -> darf geändert werden - NUR innerhalb der geschweifte Klammern
- **Überladung (Overload):** Implementierung einer gleichnamigen Methode mit veränderter Anzahl/ Art der Parameter (und Rückgabetyp) -> darf geändert werden - in den runden UND geschweiften Klammern

```java
class Animal{
    public void sound(){
        System.out.println("动物叫唤 ");
    }

    //override
    public void sound(){
        System.out.println("小狗汪汪 ");
    }

    //override
    public void sound(){
        System.out.println("小猫喵喵 ");
    }
}
```

Signatur unverändert

Implementierung verändert

```java
class Sum {
    int add(int a, int b){
        return a + b;
    }
}

class MultiSum {
    //overload
    int add(int a, int b, int c){
        return a + b + c;
    }
    //overload
    int add(int a, int b, int c, int d){
        return a + b + c + d;
    }
}
```

Signatur verändert

Implementierung verändert

Aufgerufene Methode bestimmen

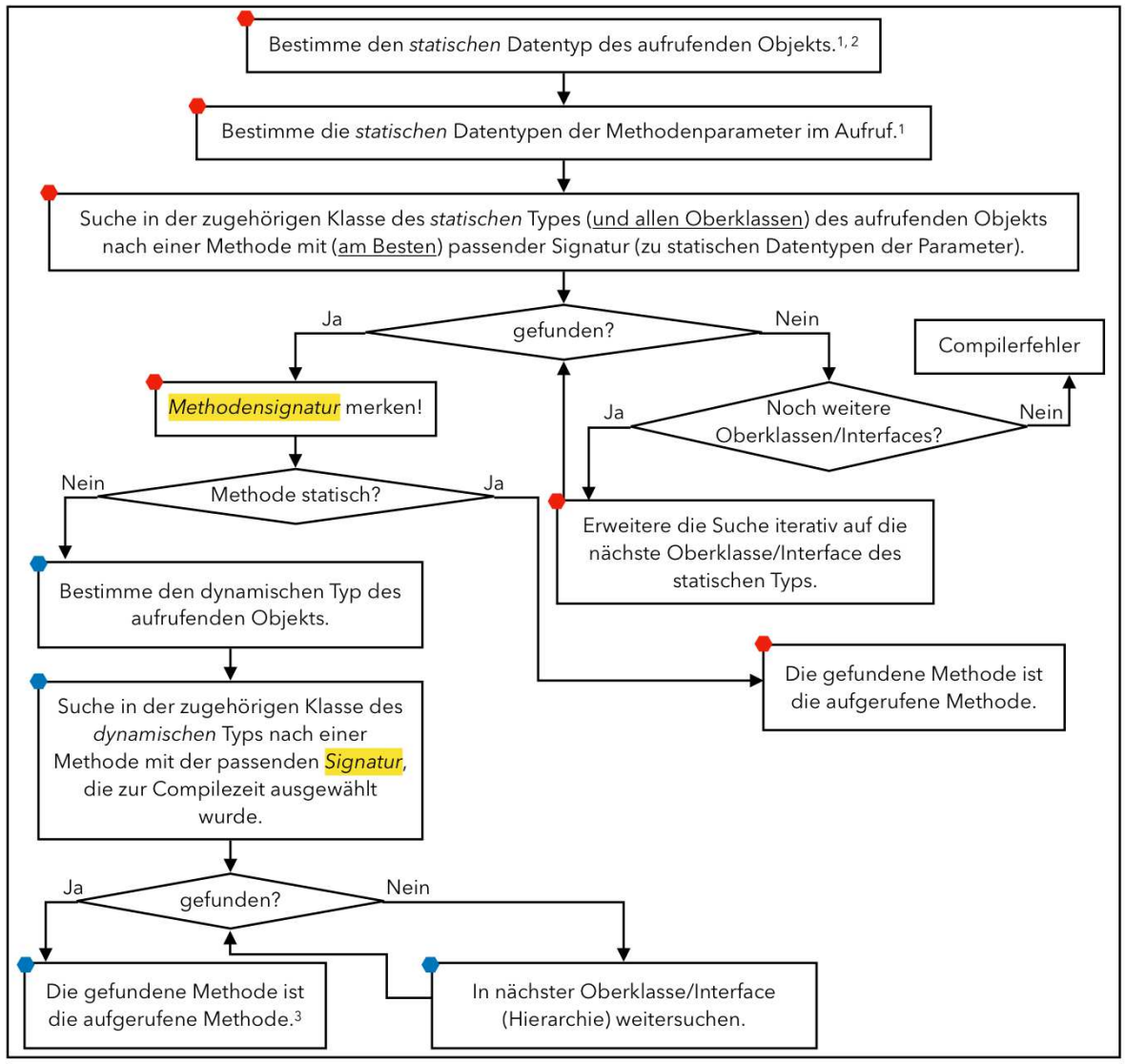
Statischer Typ: wird durch den Deklaration-Typ bestimmt und kann durch **Casts** verändert werden

Dynamischer Typ: wird mit der Instanzierung eines Objektes (also mit **new**) festgelegt -> kann durch neue Zuweisung geändert

links rechts

```java
Object o = new Fahrzeug();
Fahrzeug f = (Fahrzeug) o;
```

- **Compilezeit:** Während das Programm kompiliert wird, sind ausschließlich die statischen Typen bekannt. Entsprechend wird in dieser Phase nur mit statischen Typen gearbeitet
- **Laufzeit:** Während das Programm läuft, sind auch die dynamischen Typen bekannt. In der Phase wird die finale Methodenauswahl mit der Kenntnis über den dynamischen Typ durchgeführt



动态绑定机制

调用对象方法: 方法与该对象的运行类型(dynamischTyp)绑定

调用对象属性: 没有动态绑定机制，哪里声明，哪里使用

für Attribute => nur statischTyp relevant

```java
Top c = new Top();
c.m1(new Medium());
```

Beide statischen und dynamischen Typen von `new Medium()` sind Medium, unabhängig von anderes, da ein neue Instanz in Klasse Medium

Compiler-Fehler: aus statischer Sicht wird keine passende Methode gefunden

```
static class B extends A {
    public void f(A a) {
        System.out.println("B.f(A)");
        a.f(this);
    }
```

statischTyp B
dynamischTyp C

statischTyp A
dynamischTyp B

```
B b = new B();
C<A> c = new C<A>();
c.f(b);
```

<? extends A> -> Oberschranke A

< ? super A> -> Unterschranke A

TypeCasting

Compiler-Sicht: Steht die Klasse des stat. Typs der Variable, die gecastet wird, in einer Relation (d. h. Vererbungshierarchie) mit der „Cast"-Klasse? Wenn die beiden Klassen etwas miteinander zu tun haben, also eine von der anderen erbt (ggf. indirekt) oder die Klassen identisch sind, dann könnte der Cast theoretisch möglich sein. Anderenfalls gibt es einen **Compiler-Fehler**

- Bei der Polymorphie wählen wir immer die am besten passende Methode! Problem hier ist aber, dass beide gleich gut/schlecht passen. Man spricht von einem **ambiguous method call (mehrdeutiger Methodenaufruf) -> Compiler-Fehler**

Dynamische Sicht: Ist das Objekt der Variable, die wir gerade versuchen zu casten, auch tatsächlich vom Typ des „Casts" (oder spezieller)? Wir sehen uns jetzt also den dyn. Typ der Variable an (siehe unten) und prüfen, ob diese Klasse (des dyn. Typs) von der „Cast"-Klasse erbt (ggf. indirekt, bzw. identisch sind). Ist der Typ des Objekts nicht mindestens so speziell wie der „Cast-Typ", so tritt ein **Laufzeitfehler (ClassCastException)** auf

In Java sind generische Typ-Parameter `T` ohne explizite obere Grenze implizit als `T extends Object` definiert, d.h. Compiler übersetzt `T` mit deren allgemeinste Form `Object`. Deshalb wird jede Methode wie `f(T)` in einer generischen Klasse `A<T>` nach Type Erasure als `f(Object)` behandelt, sodass nur `f(Object)` als überschreibenden Methode für `f(T)` möglich

Wenn `T` ein Integer ist, wird `f(Number)` als überschreiende Methode für `f(T)`. Wenn `f(Number)` nicht existiert, wird `f(Object)` gewählt. Nur wenn keine dieser Methoden existiert, wird Integer zu `int` entpackt und `f(int)` aufgerufen

`abstract class 类名` Abstrakte Klasse

- Eine abstrakte Klasse kann Memeber wie normal haben, einschließlich (statische) Attribute, Konstruktur, (abstrakte) Menthode)
- Eine abstrakte Methode verfügt lediglich über Methodenkopf `public abstract methode()` , fehlt Implementierung (bzw. Rumpf {})
- Eine solche Methode ist also definiert aber nicht implementiert und kann deshalb auch nicht ausgeführt und instanziniert werden
- Eine von einer abstrakte Klasse erbende Unterklasse muss alle abstrakte Methode der Oberklasse implementieren
- Klasse mit abstrakte Methode muss abstrakte Klassen sein, abstrakte Klasse kann aber auch nicht abstrakte Methode beinhalten

`public/ default` Interface

- Interface besitzen keinen Konstruktur und kann daher keine Instanzen erzeugen
- innerhalb Interfaces standardmäßig implizit gesetzt: Methode `public abstract` ; Attribute `public static final` (Zugriff: `接口名.属性名`)
- Eine normale Klasse implements interface, muss alle Methode des Interfaces implementieren
- ABER eine abstrakte Klasse muss Methode des Interfaces nicht implementieren ↳ muss default-Methode nicht
- Eine Klasse kann mehrere Interfaces implements `class A extends B implements C, D, E`
- Ein Interface kann mehrere Interfaces erben `interface IA extends IB, IC`

```
interface 接口名 {
  // 属性 public static final
  // 抽象方法 public abstract
  // 静态方法 public static
  // 默认方法 public default
}

class 类名 implements 接口 {
  // 自己属性
  // 自己方法
  // 接口的必须实现的抽象方法
}
```

Class A implements Interface X
A objA = new A()
X objX = (X) objA
✓ kein ClassCastException

Class A implements Interface X
C<A> c = new C<A>;
C<X> c → Kompilerfehler
(Generics invariant)

Generics invariant: List<String> kein Untertyp von List<Object>, obwohl String ein Untertyp von Object, deshalb deren cast nicht erlaubt

Wrapper-Klasse → final

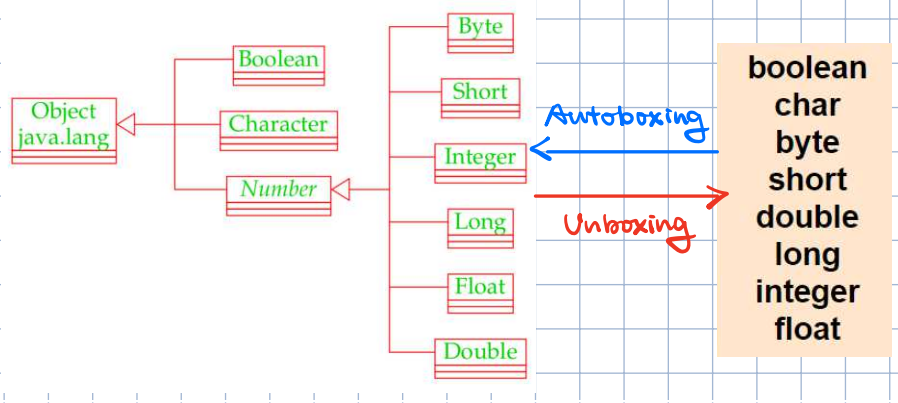
- Generische Klassen (z.B. ArrayList) arbeiten nur mit Objekten und unterstützen keine primitiven Datentypen
- Wenn wir also mit ihnen arbeiten wollen, müssen wir primitive Werte in Wrapper-Objekte umwandeln
- (Auto)Boxing - automatisches Konvertieren vom Primitiv-Wert ins Wrapper-Objekt

```
//converting int into Integer
int a = 20;
Integer i = Integer.valueOf(a); //explicitly
Integer j = a; //autoboxing (automatisch)
```

- Unboxing - automatisches Konvertieren vom Wrapper-Objekt in Primitiv-Wert

```
//converting Integer to int
Integer a = new Integer(3);
int i = a.intValue(); // explicitly
int j = a; //unboxing (automatisch)
```

Konstrukturaufrufe für generische Typen verboten
- **d.h. falls `new T()` -> führt zum Compiler-Fehler**



String

`System.out.println(e);` ruft automatisch die `toString()`-Methode des Objekts `e` auf, um deren String-Repräsentation auszugeben

- `char charAt(int index)`: Gibt das Zeichen an der Stelle `index` zurück.
- `int length()`: Gibt die Länge des Strings zurück.
- `boolean equals(Object anObject)`: Gibt `true` zurück, falls `anObject` ein String ist, der *zeichenweise* gleich zu diesem String ist.
- `String substring(int beginIndex)`: *bzw.* `String substring(int beginIndex, int endIndex)` Gibt den Teilstring beginnend bei `beginIndex` (bis Ende) zurück. Wird zusätzlich ein zweiter Parameter übergeben, so ist der Teilstring bis zum Index `endIndex-1` beschränkt.
- `boolean startsWith(String prefix)`: Gibt zurück, ob der String mit `prefix` beginnt.
- `boolean endsWith(String suffix)`: Gibt zurück, ob der String mit `suffix` endet.
- `String replace(char oldChar, char newChar)`: Ersetzt jedes Zeichen `oldChar` durch `newChar` und gibt das Ergebnis zurück. *geht auch mit zwei `String`s als Parameter, z. B.:* `"aba".replace("a", "xx")` gibt `"xxbxx"` zurück.
- `String[] split(String regex)`: Trennt den String an den Stellen `regex` in ein Array, z. B. `"axbcxd".split("x")` = `{"a","bc","d"}`
- `int indexOf(int ch)`: *bzw.* `int indexOf(String str)` *analog dazu gibt es* `lastIndexOf(...)` Gibt den Index des ersten Vorkommens des Zeichens `ch` (bzw. Strings `str`) im String zurück (`-1`, falls nicht enthalten). Es kann ein zweiter Parameter `int fromIndex` übergeben werden, um eine Start-Suchposition zu spezifizieren.
- `char[] toCharArray()`: Gibt ein Array zurück, das alle Zeichen des Strings unverändert enthält.
- `String toUpperCase()`: *bzw.* `toLowerCase()`: Gibt einen String zurück, der dem String in Groß- bzw. Kleinschreibung entspricht.

- ☑ `String C = (String) new Object();` ⇒ **Laufzeitfehler** (`ClassCastException`)
- ☑ `String[] q = new String[2];` ⇒ erzeugt String Array `{null, null}`
- ☑ `String f = null + " World";` ⇒ Konkatenation zu `"null World"`
- ☑ `String j = '5' + 5.0 + "5" + 5;` ⇒ → `"58.055"`, denn `(int)'5'` ist 53
- ☒ `String k = 'J';` ⇒ kann `char` nicht implizit zu `String` casten
- ☒ `String l = (String) 'K';` ⇒ kann `char` auch nicht explizit zu `String` casten

firstUniqueChar

```
public int firstUniqeChar(String s) {
    Map<Character, Integer> frequency = new HashMap<Character, Integer>();
    for (int i = 0; i < s.length(); i++) {
        char ch = s.charAt(i);
        frequency.put(ch, frequency.getOrDefault(ch, 0)+1);
    }
    for (int i = 0; i< s.length(); i++){
        if(frequency.get(s.charAt(i)) == 1){
            return i;
        }
    }
    return -1;
}
```

filterAlphaToUpper

```
public static String filterAlphaToUpper(String s) {
    String result = " ";
    for (int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if (c >= 'a' && c <= 'z') {
            c = c - 'a' + 'A';
        }
        else if ( c < 'A' || c > 'Z'){
            c = '_';
        }
        result = result + c;
    }
    return result;
}
```

invertString

```
public static String invert(String s) {
    String result = "";
    for (int i = s.length()-1; i >= 0; i--){
        result += s.charAt(i);
    }
    return result;
}
```

contain SubString

```
public static boolean contains(String s, String sub) {
    int maxIndex = s.length() - sub.length();
    for (int start = 0; start <= maxIndex; start++) {
        boolean found = true;
        for (int j = 0; j < sub.length(); j++) {
            if (sub.charAt(j) != s.charAt(start + j)){
                found = false;
            }
        }
        if (found) return true;
    }
    return false;
}
```

longestCommonPrefix

```
public String longestCommonPrefix(String[] strs) {
    if(strs == null || strs.length == 0) return " ";

    String prefix = strs[0];
    int count = strs.length;
    for(int i = 1; i < count; i++){
        prefix = longestCommonPrefix(prefix, strs[i]);
        if(prefix.length() == 0) break;
    }
    return prefix;
}

public String longestCommonPrefix(String str1, String str2){
    int length = Math.min(str1.length(), str2.length());
    int index = 0;
    while(index < length && str1.charAt(index) == str2.charAt(index)){
        index++;
    }
    return str1.substring(0, index);
}
```

reverseInteger

```
public int reverseInteger(int x) {
    boolean sign = x < 0;
    if (sign) x = -x;
    int result = 0;
    int t;
    while(x > 0 && result >= 0){
        t = result * 10 + (x % 10);
        if(t/10 != result) return 0;
        result = t;
        x /= 10;
    }
    if(result < 0) return 0;
    return sign ? -result : result;
}
```

isPalindrome (验证回文串)

```
public boolean isPalindrome(String s) {
    for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
        while (i < j && !Character.isLetterOrDigit(s.charAt(i))) i++;
        while (i < j && !Character.isLetterOrDigit(s.charAt(j))) j--;
        if (Character.toLowerCase(s.charAt(i)) !=
            Character.toLowerCase(s.charAt(j))){
            return false;
        }
    }
    return true;
}
```

hexToDez

```
public static long hexToDez(String hex) {
    hex = hex.toLowerCase();
    if(!hex.startsWith("0x")) return -1;
    hex = hex.substring(2);

    // Führende Nullen zählen
    int zerosIndex = 0;
    while (zerosIndex < hex.length() && hex.charAt(zerosIndex) == '0'){
        zerosIndex++;
    }
    if (zerosIndex == hex.length()) return 0;

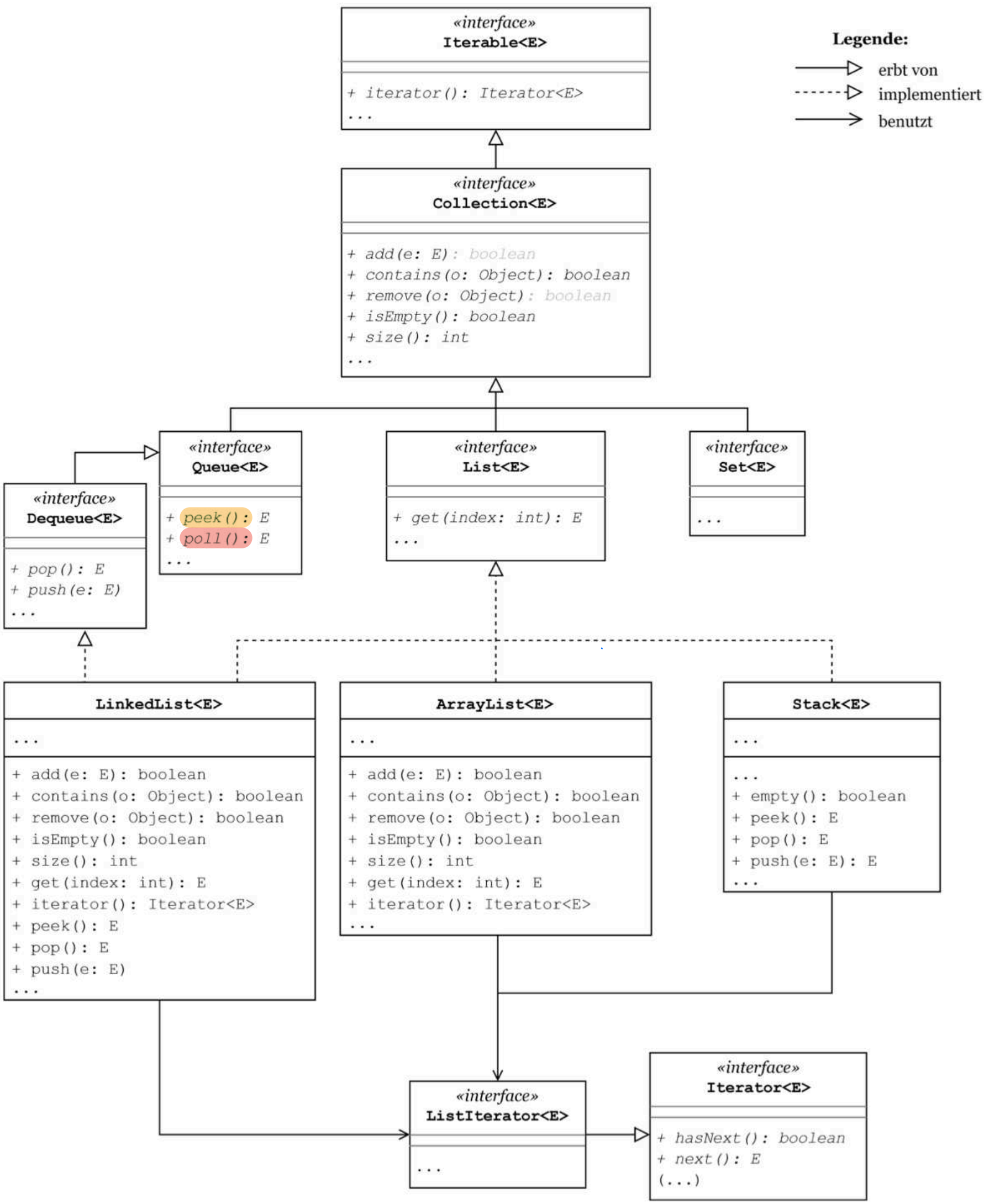
    // Führende Nullen entfernen
    hex = hex.substring(zerosIndex, hex.length());
    // Fehlerprüfung: Mehr als 16 Hex-Ziffern sind nicht als long darstellbar
    // Bei exakt 16 Ziffern muss die erste kleiner 8 sein, sonst nicht darstellbar
    if (hex.length() > 16 || hex.length() == 16 && hex.charAt(0) < '8'){
        return -1;
    }
    long multiplier = 1, result = 0;
    for (int i = hex.length() - 1; i >= 0; i--){
        int digit = hex.charAt(i);
        // Konvertiere die Hex-Ziffer in ihren äquivalenten Dezimalwert
        if (digit >= '0' && digit <= '9') digit -= '0';
        else if (digit >= 'a' && digit <= 'f') digit = digit - 'a' + 10 ;
        else return -1;
        result += digit * multiplier;
        multiplier *= 16;
    }
    return result;
}
```

isAnagram (字母异位词)

```
public boolean isAnagram(String s, String t) {
    if(s.length() != t.length()){
        return false;
    }
    int[] counter = new int[26];
    for(int i = 0; i < s.length(); i++) {
        counter[s.charAt(i) - 'a']++;
        counter[t.charAt(i) - 'a']--;
    }
    for (int count : counter) {
        if(count != 0){
            return false;
        }
    }
    return true;
}
```

countVowels

```
public static int countVowels(String text) {
    int count = 0;
    for (int i = 0; i < text.length(); i++) {
        char c = text.charAt(i) :
        if (c >= 'A' d& c <= 'Z') {
            // Groß- zu Kleinbuchstabe
            c = (char) (c - 'A' + 'a');
        }
        if (c == 'a' || c == 'e' || c = 'i' ||
            c == 'o' || c == 'u' || c == 'y'){
                count++;
        }
    }
    return count;
}
```



peek(): returns the object at the top of the current queue, without removing it. If empty, it returns null

poll(): returns the object at the top of the current queue and removes it. If empty, it returns null

ArrayList

removeElement

```java
public int removeElement(int[] nums, int val) {
    int i = 0;
    for(int j=0; j < nums.length; j++){
        if(nums[j] != val){
            nums[i] = nums[j];
            i++;
        }
    }
    return i;
}
```

minAndMax

```java
public static void minAndMax(int[] a) {
    if (a.length == 0) return;
    int min = a[0];
    int max = a[0];
    for (int i = 0; i < a.length; i++) {
        if (a[i] < min) min = a[i];
        if (a[i] > max) max = a[i];
    }
    System.out.println("Minimum = " + min +
                       ", Maximum = " + max);
}
```

invertieren (in umgekehrter Richtung)

```java
public static void invert(int[] a) {
    for (int i = 0; i < a.length / 2; i++) {
        int temp = a[i];
        a[i] = a[a.length - 1 - i];
        a[a.length - 1 - i] = temp;
    }
}
```

hinzufügen

```java
private static int[] append(int[] array, int value) {
    int[] newArray = new int[array.length + 1];
    for(int i = 0; i < array.length; i++) {
        newArr[i] = arr[i];
    }
    newArray[newArray.length - 1] = value;
    return newArray;
}
```

rotieren

```java
public static void rotate(int[] array, int amount) {
    int length = array.length;
    if(length == 0) return;
    amount = amount % length;
    if(amount < 0) {
        amount += length;
    }

    int[] rotated = new int[length];
    for(int i = 0; i < length; i++) {
        rotated[(i + amount) % length] = array[i];
    }
    for(int i = 0; i < length; i++) {
        array[i] = rotated[i];
    }
}
```

lastIndexOf

```java
public int lastIndexOf(Object[] arr, Object elem){
    int i = arr.length;
    while (i-- > 0){
        if (arr[i] == elem) return i;
    }
    return -1 ;
}
```

contain

```java
private static boolean contains(int[] array, int value) {
    for (int i = 0; i < array.length; i++){
        if (array[i] == value) {
            return true;
        }
    }
    return false;
}
```

filter

```java
public static int[] filter(int[] array,int min,int max){
    int firstUnusedPosition = 0;
    int[] out = new int[array.length];
    for (int j = 0; j < array.length; j++){
        if(array[j] >= min && array[j] <= max) {
            out[firstUnusedPosition] = array[j];
            firstUnusedPosition++;
        }
    }
    return Arrays.copyOf(out,firstUnusedPosition);
}
```

removeDuplicates

```java
public static int[] removeDuplicates(int[] arr) {
    // zählen, wie groß das Ergebnisarray sein muss
    int size = 0;
    for (int i = 0; i < arr.length; i++) {
        boolean alreadyCounted = false;
        for (int j = 0; j < i; j++) {
            if (arr[j] == arr[i]){
                alreadyCounted = true;
            }
        }
        if (!alreadyCounted) size++;
    }
    // Array anlegen und befüllen
    int[] result = new int[size];
    int index = 0;
    for (int i = 0; i < arr.length; i++) {
        boolean alreadyInserted = false;
        for (int j = 0; j < i; j++) {
            if (arr[j] == arr[i]) {
                alreadyInserted = true;
            }
        }
        if (!alreadyInserted) {
            result[index] = arr[i];
            index++;
        }
    }
    return result;
}
```

zählen

```java
public int count(int[] numbers, int element) {
    int counter = 0 ;
    for (int i = 0; i < numbers.length ; ++i){
        if ( numbers[i] == element )
        counter++;
    }
    return counter;
}
```

ausschneiden (für length Element)

```java
public static int[] intersect(int[] a, int length) {
    if (length <= 0) {
        return new int[0];
    }
    int[] intersected = new int[length];
    for (int i = 0; i < length && i < a.length; i++) {
        intersected[i] = a[i];
    }
    return intersected;
}
```

für zweidimensionale Array

minAndMax

```java
public static int[][] minsAndMaxs(int[][] a) {
    int[][] out = new int[a.length][2];
    for(int i = 0; i < a.length; i++) {
        if(a[i].length <= 0) {
            out[i] = null;
            continue;
        }
        int min = a[i][0];
        int max = a[i][0];
        for(int j = 1; j < a[i].length; j++) {
            if(a[i][j] < min) {
                min = a[i][j];
            }
            if(a[i][j] > max) {
                max = a[i][j];
            }
        }
        out[i] = new int[]{min, max};
    }
    return out;
}
```

transponieren

```java
public static int[][] transpose(int[][] a) {
    if(a.length == 0 || a[0].length == 0) {
        return null;
    }
    int[][] transposed = new int[a[0].length][a.length];
    for(int i = 0; i < a.length; i++) {
        for(int j = 0; j < a[i].length; j++) {
            transposed[j][i] = a[i][j];
        }
    }
    return transposed;
}
```

```
{
    {1, 2, 3, 4},
    {5, 6, 7, 8}
}
```

→

```
{
    {1, 5},
    {2, 6},
    {3, 7},
    {4, 8}
}
```

linearisieren

```java
public static int[] linearize(int[][] a) {
    int length = 0;
    for (int i = 0; i < a.length; i++) {
        length = length + a[i].length;
    }
    int[] linearized = new int[length];
    int linIndex = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a[i].length; j++) {
            linearized[linIndex++] = a[i][j];
        }
    }
    return linearized;
}
```

`linearize(new int[][] {{1, 3}, {25}, {7, 4, 6, 9}})`

liefert ein Array `{1, 3, 25, 7, 4, 6, 9}`.

单链表

LinkedListNode

```java
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val=val;
    }
}
```

LinkedList

```java
class MyLinkedList {
    int size;
    ListNode head;
    public MyLinkedList() {
        size = 0;
        head = new ListNode(0);
    }
```

addAtIndex

```java
public void addAtIndex(int index, int val) {
    if (index > size) return;
    if (index < 0) index = 0;

    size++;
    ListNode prev = head;
    for (int i = 0; i < index; i++) {
        prev = prev.next;
    }
    ListNode toAdd = new ListNode(val);
    toAdd.next = prev.next;
    prev.next = toAdd;
}

public void addAtHead(int val) {
    addAtIndex(0, val);
}
public void addAtTail(int val) {
    addAtIndex(size, val);
}
```

reverse

```java
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode cur = head;
    ListNode temp = null;
    while (cur != null) {
        temp = cur.next;
        cur.next = prev;
        prev = cur;
        cur = temp;
    }
    return prev;
}
```

get

```java
public int get(int index) {
    if (index < 0 || index >= size) return -1;
    ListNode currentNode = head;
    for (int i = 0; i <= index; i++) {
        currentNode = currentNode.next;
    }
    return currentNode.val;
}
```

deleteAtIndex

```java
public void deleteAtIndex(int index) {
    if (index < 0 || index >= size) return;
    size--;
    if (index == 0) {
        head = head.next;
        return;
    }
    ListNode prev = head;
    for (int i = 0; i < index ; i++) {
        prev = prev.next;
    }
    prev.next = prev.next.next;
}
```

链表相交

```java
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    Set<ListNode> nodesInB = new HashSet<ListNode>();
    while(headB != null){
        nodesInB.add(headB);
        headB = headB.next;
    }
    while(headA != null){
        if(nodesInB.contains(headA)){
            return headA;
        }
        headA = headA.next;
    }
    return null;
}
```

合并两个有序链表

```java
public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    ListNode prevhead = new ListNode(-1);
    ListNode prev = prevhead;
    while(list1 != null && list2 != null){
        if(list1.val <= list2.val){
            prev.next = list1;
            list1 = list1.next;
        }
        else{
            prev.next = list2;
            list2 = list2.next;
        }
        prev = prev.next;
    }
    prev.next = list1 == null ? list2 : list1;
    return prevhead.next;
}
```

环形链表

```java
public ListNode detectCycle(ListNode head) {
    Set<ListNode> visited = new HashSet<ListNode>();
    ListNode node = head;
    while(node != null) {
        if(visited.contains(node)){
            return node;
        }
        visited.add(node);
        node = node.next;
    }
    return null;
}
```

双链表

DoublyLinkedListElement

```java
public class DoublyLinkedListElement {
    private int info;
    public DoublyLinkedListElement next;
    public DoublyLinkedListElement prev;

    public DoublyLinkedListElement(int startInfo) {
        info = startInfo;
        next = null;
        prev = null;
    }
    public int getInfo() return info;
}
```

DoublyLinkedList

```java
public class DoublyLinkedList {
    private int size = 0;
    private DoublyLinkedListElement head;
    private DoublyLinkedListElement tail;

    public DoublyLinkedList() {
        head = null;
        tail = null;
    }
}
```

get

```java
public int get(int index) {
    if (index < 0 || size <= index) {
        System.out.println("Position ist größer als Liste!");
        return 0;
    }
    return getElement(index).getInfo();
}

private DoublyLinkedListElement getElement(int index) {
    if (index < size / 2) {
        DoublyLinkedListElement temp = head;
        for (int i = 0; i < index; i++, temp = temp.next)
        return temp;
    } else {
        DoublyLinkedListElement temp = tail;
        for (int i = size - 1; i > index; i--, temp = temp.prev)
        return temp;
    }
}
```

append

```java
public void append(int info) {
    DoublyLinkedListElement newElement
        = new DoublyLinkedListElement(info);
    if (size == 0) {
        head = newElement;
        tail = newElement;
    } else {
        tail.next = newElement;
        newElement.prev = tail;
        tail = newElement;
    }
    size++;
    }

public int size() {
    return size;
}
```

isEqual

```java
public boolean isEqual(DoublyLinkedList other) {
    if (other == null) return false;
    DoublyLinkedListElement mytemp = head;
    DoublyLinkedListElement othertemp = other.head;
    while (mytemp != null) {
        if (!mytemp.isEqual(othertemp)) {
            return false;
        }
        mytemp = mytemp.next;
        othertemp = othertemp.next;
    }
    return othertemp == null;
}
public boolean isEqual(DoublyLinkedListElement other) {
        return other != null && info == other.info;
}
```

appendList

```java
public void appendList(DoublyLinkedList other) {
    // if this list has elements, append
    if (size != 0) {
        tail.next = other.head;
    // otherwise set new head
    } else {
        head = other.head;
    }
    // if the other list is empty nothing needs to be done
    if (other.size != 0) {
        other.head.prev = tail;
        tail = other.tail;
        size += other.size;
    }
}
```

remove

```java
public void remove(int index) {
    if (size <= index || index < 0) {
        System.out.println("Position ist größer als Liste!");
    } else {
        DoublyLinkedListElement toRemove = getElement(index);
        // it might delete the head, check if prev exists
        if (toRemove.prev != null) {
            toRemove.prev.next = toRemove.next;
        } else {
            head = toRemove.next;
        }
        // it might be the tail as well, check if next exists
        if (toRemove.next != null) {
            toRemove.next.prev = toRemove.prev;
        } else {
            tail = toRemove.prev;
        }
        size--;
    }
}
```

toString

```java
@Override
public String toString() {
    StringBuilder result = new StringBuilder();
    for (DoublyLinkedListElement temp = head; temp != null; temp = temp.next) {
        result.append(temp.getInfo());
        if (temp.next != null) {
            result.append(',');
        }
    }
    return result.toString();
}
```

sum

```java
public int sum() {
    int result = 0;
    for (DoublyLinkedListElement temp = head; temp != null; temp = temp.next) {
        result += temp.getInfo();
    }
    return result;
}
```

copy

```java
public DoublyLinkedList copy() {
    DoublyLinkedList result = new DoublyLinkedList();
    for (DoublyLinkedListElement temp = head; temp != null; temp = temp.next) {
        result.append(temp.getInfo());
    }
    return result;
}
```

Stack (LIFO -> Last In First Out)

push

```
public void push(int number) {
    StackElement element = new StackElement(number);
    if(first == null) {
        first = last = element;
    } else {
        element.setNext(first);
        first = element;
    }
    size++;
}
```

pop

```
public int pop() {
    if(first == null) {
        return Integer.MIN_VALUE;
    }
    int poppedValue = first.getValue();
    first = first.getNext();
    size--;
    return poppedValue;
}
```

toArray

```
public int[] toArray() {
    int[] stackAsArray = new int[size];
    StackElement current = first;
    for(int i = 0; current != null;
       current = current.getNext(), i++) {
        stackAsArray[i] = current.getValue();
    }
    return stackAsArray;
}
```

size

```
public int size() {
    return size;
}
```

isEmpty

```
public boolean isEmpty() {
    return size == 0;
}
```

Queue (FIFO -> First In First Out)

push

```
public void push(int number) {
    QueueElement element = new QueueElement(number);
    if(first == null) {
        first = last = element;
    } else {
        last.setNext(element);
        last = element;
    }
    size++;
}
```

pop

```
public int pop() {
    if(first == null) {
        return Integer.MIN_VALUE;
    }
    int poppedValue = first.getValue();
    first = first.getNext();
    size--;
    return poppedValue;
}
```

toArray

```
public int[] toArray() {
    int[] queueAsArray = new int[size];
    QueueElement current = first;
    for(int i = 0; current != null;
       current = current.getNext(), i++) {
        queueAsArray[i] = current.getValue();
    }
    return queueAsArray;
}
```

size

```
public int size() {
    return size;
}
```

isEmpty

```
public boolean isEmpty() {
    return size == 0;
}
```

Deque (darf Element am Anfang oder am Ende einsehen/ hinzufügen/ entfernen)

einsehen

```
public int getFirst() {
    if (isEmpty()) {
        return Integer.MIN_VALUE;
    }
    return head.val;
}
```

```
public int getLast() {
    if (isEmpty()) {
        return Integer.MIN_VALUE;
    }
    return tail.val;
}
```

hinzufügen

```
public void addFirst(int val) {
    if (isEmpty()) {
        head = new Element(val);
        tail = head;
    } else {
        head = new Element(val, head);
        head.next.prev = head;
    }
    size++;
}
```

```
public void addLast(int val) {
    if (isEmpty()) {
        head = new Element(val);
        tail = head;
    } else {
        tail = new Element(tail, val);
        tail.prev.next = tail;
    }
    size++;
}
```

entfernen

```
public int removeFirst() {
    if (isEmpty()) {
        return Integer.MIN_VALUE;
    }
    int val = getFirst();
    if (size() == 1) {
        head = null;
        tail = null;
    } else {
        head = head.next;
        head.prev = null;
    }
    size--;
    return val;
}
```

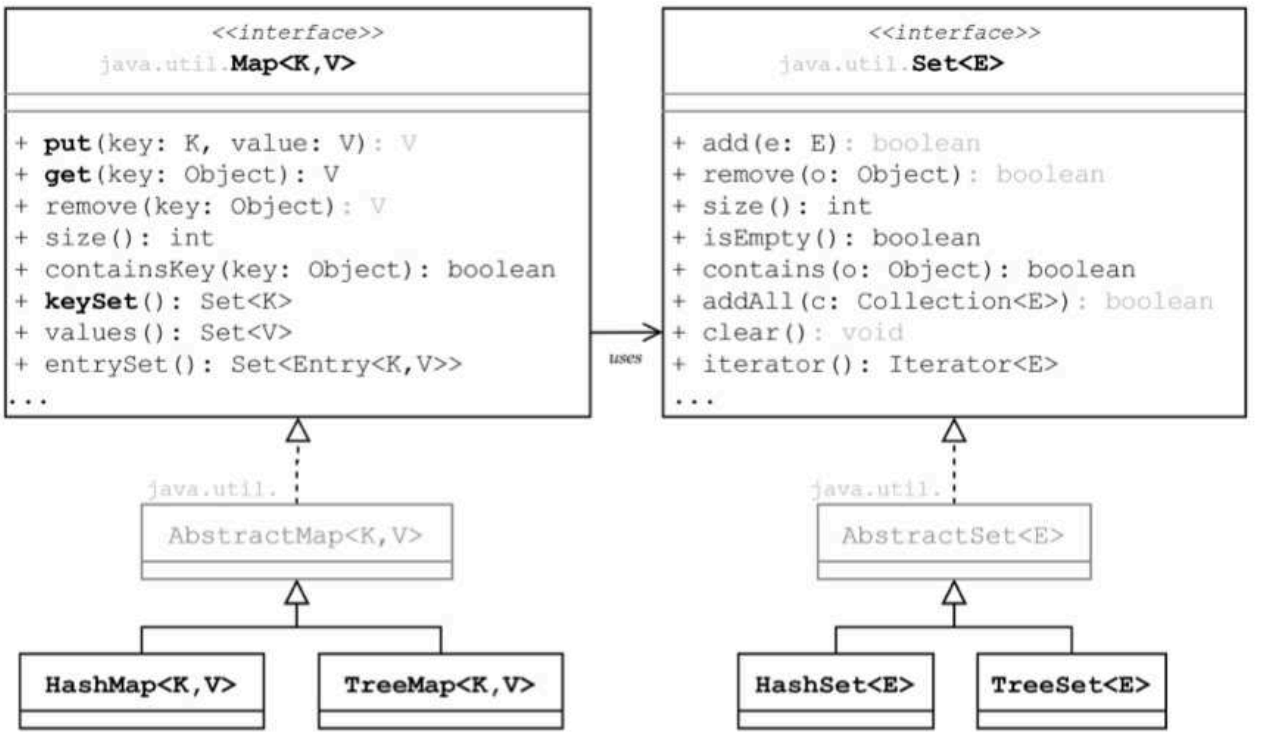
```
public int removeLast() {
    if (isEmpty()) {
        return Integer.MIN_VALUE;
    }
    int val = getLast();
    if (size() == 1) {
        head = null;
        tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }
    size--;
    return val;
}
```

size

```
public int size() {
    return size;
}
```

isEmpty

```
public boolean isEmpty() {
    return size() == 0;
}
```



两个数组的相同元素

```java
public int[] intersection(int[] nums1, int[] nums2) {
    HashSet<Integer> set1 = new HashSet<Integer>();
    for(Integer n : nums1) set1.add(n);
    HashSet<Integer> set2 = new HashSet<Integer>();
    for(Integer n : nums2) set2.add(n);

    if(set1.size() < set2.size()) return set_intersection(set1, set2);
    else return set_intersection(set2, set1);
}
public int[] set_intersection (HashSet<Integer> set1, HashSet<Integer> set2){
    int[] output = new int[set1.size()];
    int index = 0;
    for(Integer s : set1){
        if(set2.contains(s)){
            output[index++] = s;
        }
    }
    return Arrays.copyOf(output, index);
}
```

查找字符串的共用字符

```java
public List<String> commonChars(String[] words) {
    List<String> result = new ArrayList<>();
    if (words.length == 0) return result;
    int[] hash= new int[26];
    for (int i = 0; i < words[0].length(); i++){
        hash[words[0].charAt(i)- 'a']++;
    }
    for (int i = 1; i < words.length; i++) {
        int[] hashOtherStr= new int[26];
        for (int j = 0; j < words[i].length(); j++) {
            hashOtherStr[words[i].charAt(j)- 'a']++;
        }
        for (int k = 0; k < 26; k++) {
            hash[k] = Math.min(hash[k], hashOtherStr[k]);
        }
    }
    for (int i = 0; i < 26; i++) {
        while (hash[i] != 0) {
            char c= (char) (i +'a');
            result.add(String.valueOf(c));
            hash[i]--;
        }
    }
    return result;
}
```

twoSum = target

```java
public int[] twoSum(int[] nums, int target) {
    int[] res = new int [2];
    if(nums == null || nums.length == 0) {
        return res;
    }
    Map<Integer,Integer> map = new HashMap<>();
    for(int i = 0; i < nums.length; i++){
        int num1 = nums[i];
        int num2 = target - num1;
        if (map.containsKey(num2)) {
            res[0] = i;
            res[1] = map.get(num2);
            break;
        }
        map.put(num1,i);
    }
    return res;
}
```

快乐数 (数字各位平方之和=1)

```java
public boolean isHappy(int n) {
    Set<Integer> record = new HashSet<>();
    while(n != 1 && !record.contains(n)){
        record.add(n);
        n = getNextNumber(n);
    }
    return n == 1;
}

public int getNextNumber(int n){
    int totalSum = 0;
    while(n > 0){
        int d = n % 10;
        n /= 10;
        totalSum += d * d;
    }
    return totalSum;
}
```

最频繁元素

```java
public static Set<Integer> mostFrequentElements(int[] arr) {
    Map<Integer, Integer> counts = new HashMap<>();
    //Elemente zählen in der HashMap aufsummieren
    for (int elem : arr) {
        Integer oldCount = counts.get(elem);
        counts.put(elem, oldCount == null ? 1 : oldCount+1);
    }
    // Finde die maximale Häufigkeit
    int max = 0;
    for (int count : counts.values()){
        if (count > max){
            max = count;
        }
    }
    // Finde die Elemente (Keys) mit der größten Häufigkeit (Value)
    HashSet<Integer> mostFrequent = new HashSet<>();
    for (int elem : counts.keySet()) {
        if (counts.get(elem) == max) {
            mostFrequent.add(elem);
        }
    }
    return mostFrequent;
}
```

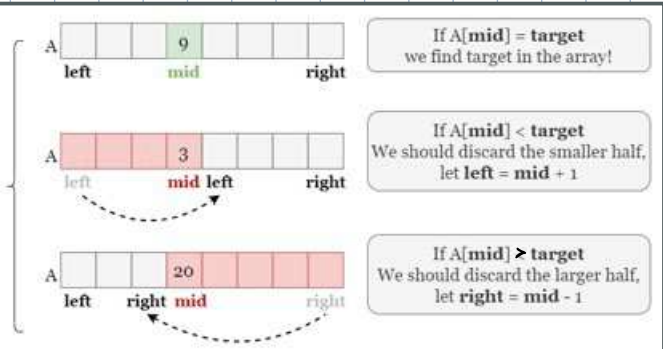
Sortier- und Such-Algorithmen

```
/* standardmäßig Zahlen aufsteigend
und Zeichenkette alphabetisch sortiert */
Arrays.sort(int[] {5,2,1,6,7,3});
```

```
private static void swap(int[] array, int a, int b){
    int temp = array[a];
    array[a] = array[b];
    array[b] = temp;
}
```

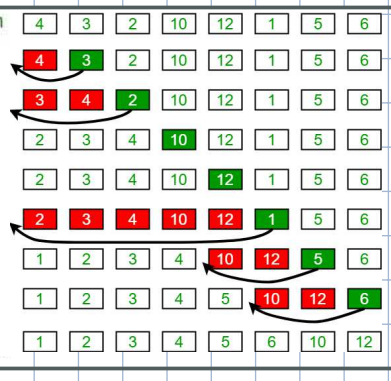
Binary Search

```
public int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length-1;
    while(left <= right) {
        int mid = left + (right - left)/2;
        if(nums[mid] == target) {
            return mid;
        }
        else if (nums[mid] < target){
            left = mid + 1;
        }
        else if (nums[mid] > target){
            right = mid - 1;
        }
    }
    return -1;
}
```



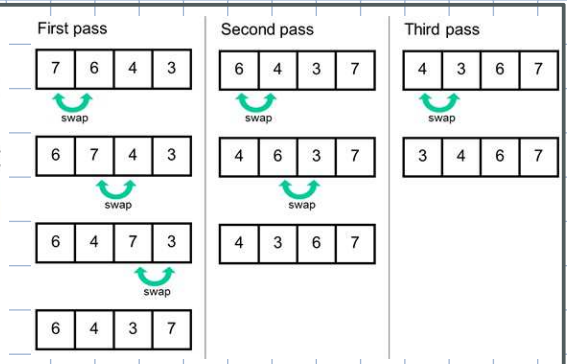
Insertion Sort

```
/* InsertSort: Zum Einfügen der neuen Karte müssen
alle auf der Hand nachfolgenden eine position
weiter nach rechts wandern */
public static void insertSort(int[] array){
    for(int i = 1; i < array.length; i++){
        for(int j = i; j >= 1; j--){
            if(array[j] < array[j-1]){
                swap(array, j, j-1);
            }
            else break;
        }
    }
}
```



Bubble Sort

```
/* BubbleSort (aufsteigende Blasen):
falls aktuelle Element > Nachfolgeelement
-> Position miteinander getauscht */
public static void bubbleSort(int[] array){
    int n = arr.length;
    for (int i = 0; i < n - 1; i++){
        for (int j = 0; j < n - i - 1; j++){
            if (arr[j] > arr[j+1]){
                // swap arr[j+1] and arr[j]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

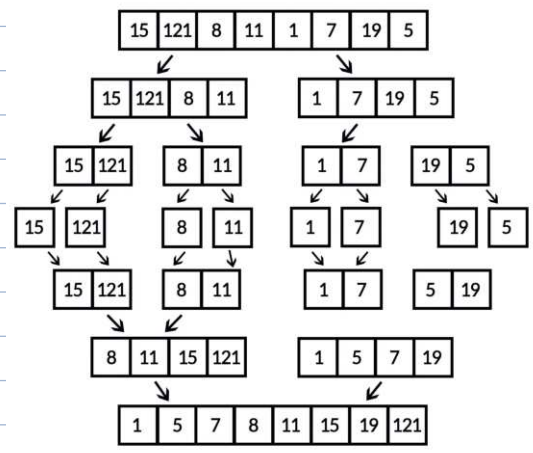


Merge Sort

```
/* MergeSort: zu sortierende Array wird rekursive
in zwei Teile aufgeteilt, TeilArray werden sortiert
 und im Ausgangsarray zusammengeführt */
public static void mergeSort(int[] array){
    // 1) Abbruchbedingung: nichts zu sortieren
    if(array.length <= 1) return;

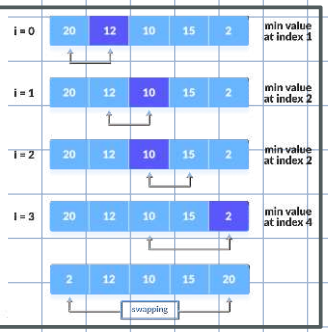
    // 2) Array in linke und rechte Hälfte teilen
    int middel = (int)(array.length / 2);
    int[] leftHalf = new int[middel];
    for(int i = 0; i < leftHalf.length; i++){
        leftHalf[i] = array[i];
    }
    int[] rightHalf = new int[array.length - middel];
    for(int i = 0; i < rightHalf.length; i++){
        rightHalf[i] = array[middel + i];
    }

    // 3) beide Hälfte sortieren
    mergeSort(leftHalf);
    mergeSort(rightHalf);
    merge(array, leftHalf, rightHalf);
}
// 4) sortiete Hälfte ins Ursprungarray mergen
private static void merge(int[] array, int[] leftArray, rightArray){
    int left = 0;
    int right = 0;
    int res = 0;
    while(left < leftArray.length && right < rightArray.length){
        if(leftArray[left] < rightArray[right]){
            array[res++] = leftArray[left++];
        }
        else {
            array[res++] = rightArray[right++];
        }
    }
    while(left < leftArray.length) {
        array[res++] = leftArray[left++];
    }
    while(right < rightArray.length){
        array[res++] = rightArray[right++];
    }
}
```

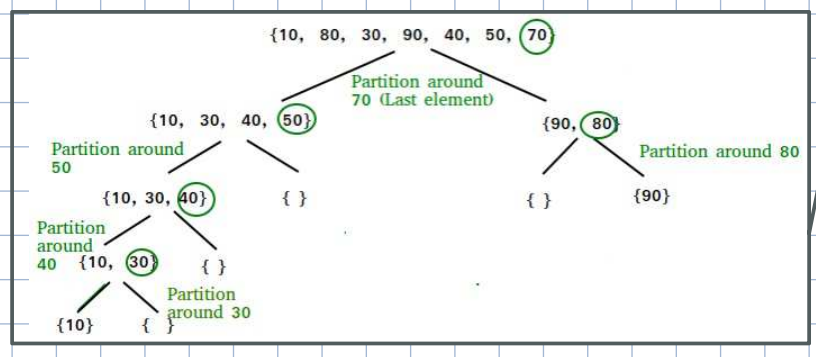


Selection Sort

```
/* selection Sort: muss jedes Mal den unsortierten
  Teil durchlaufen, um Minimum zu finden */
public static void selectionSort(int[] array){
    for(int i = 0; i < array.length-1; i++){
        int minIndex = i;
        for(int j = i; j < array.length; j++){
            if(array[j] < array[minIndex]){
                minIndex = j;
            }
            swap(array, i, minIndex);
    }
}
```



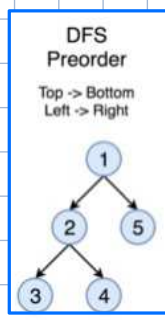
Quick Sort



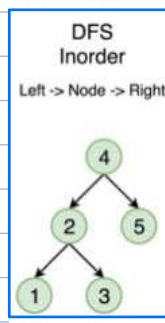

```
/* QuickSort: sortierte Array in zwei Hälfte
 teilen und dort wieder quickSort anwenden
kleiner als Pivotelement -> linke Hälfte
größer als Pivotelement -> rechte Hälfte */
public static void quickSort(int[] array){
    quickSort(array, 0, array.length-1);
}
private static void quickSort(int[] array, int from, int to){
    int length = to - from + 1;
    if(length >= 2){
        int left = from;
        int right = to;
        while(left != right){
            while(left != right && array[left] < array[to]){
                left++;
            }
            while(left != right && array[right] >= array[to]){
                right--;
            }
            swap(array, left, right);
        }
        swap(array, right, to);
        int pivotIndex = right;
        quickSort(array, from, pivotIndex-1);
        quickSort(array, pivotIndex+1, to);
    }
}
```

Binärbaum (Binary Tree 二叉树)

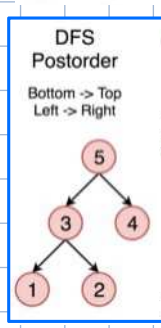
DFS (Depth First Search) 深度优先搜索




```java
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<Integer>();
    preorder(root, result);
    return result;
}
public void preorder(TreeNode root, List<Integer> result){
    if(root == null){
        return;
    }
    result.add(root.val);
    preorder(root.left, result);
    preorder(root.right, result);
}
```



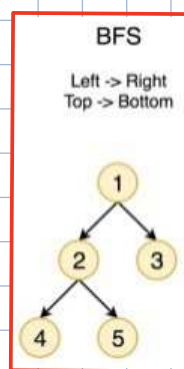

```java
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<Integer>();
    inorder(root, result);
    return result;
}
public void inorder(TreeNode root, List<Integer> result){
    if(root == null) {
        return;
    }
    inorder(root.left, result);
    result.add(root.val);
    inorder(root.right, result);
}
```




```java
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<Integer>();
    postorder(root, result);
    return result;
}
public void postorder(TreeNode root, List<Integer> result) {
    if(root == null){
        return;
    }
    postorder(root.left, result);
    postorder(root.right, result);
    result.add(root.val);
}
```

On the following figure the nodes are numerated in the order you visit them, please follow `1-2-3-4-5` to compare different strategies.

BFS (Breadth First Search) 广度优先搜索




```java
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> levels = new LinkedList<>();
    if(root == null) return levels;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int level = 0;
    while(!queue.isEmpty()){
        levels.add(new ArrayList<Integer>());
        int level_length = queue.size();
        for(int i = 0; i < level_length; i++){
            TreeNode current = queue.poll();
            levels.get(level).add(current.val);
            if(current.left != null) queue.add(current.left);
            if(current.right != null) queue.add(current.right);
        }
        level++;
    }
    return levels;
}
```

Average of Levels

```java
public List<Double> averageOfLevels(TreeNode root) {
    List<Double> list = new ArrayList<>();
    Deque<TreeNode> deque = new LinkedList<>();

    if(root == null) return list;
    deque.offerLast(root);
    while(!deque.isEmpty()){
        int levelSize = deque.size();
        double levelSum = 0.0;
        for(int i = 0; i < levelSize; i++){
            TreeNode poll = deque.pollFirst();
            levelSum += poll.val;
            if(poll.left != null){
                deque.addLast(poll.left);
            }
            if(poll.right != null){
                deque.addLast(poll.right);
            }
        }
        list.add(levelSum / levelSize);
    }
    return list;
}
```

Invert Binary Tree

```java
public TreeNode invertTree(TreeNode root) {
    if(root == null){
        return null;
    }
    invertTree(root.left);
    invertTree(root.right);
    swapChildren(root);
    return root;
}
private void swapChildren(TreeNode root){
    TreeNode tmp = root.left;
    root.left = root.right;
    root.right = tmp;
}
```

Maximum Depth

```java
public int maxDepth(TreeNode root) {
    if(root == null) return 0;
    else {
        int left_height = maxDepth(root.left);
        int right_height = maxDepth(root.right);
        return Math.max(left_height, right_height) + 1;
    }
}
```

Minimum Depth

```java
public int minDepth(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    if(root == null) return 0;
    queue.offer(root);
    int depth = 0;
    while(!queue.isEmpty()){
        int size = queue.size();
        depth++;
        TreeNode curr = null;
        for(int i = 0; i < size; i++){
            curr = queue.poll();
            if(curr.left == null && curr.right == null){
                return depth;
            }
            if(curr.left != null) queue.offer(curr.left);
            if(curr.right != null) queue.offer(curr.right);
        }
    }
    return depth;
}
```

Largest Value in Each Row

```java
public List<Integer> largestValues(TreeNode root) {
    if(root == null){
        return Collections.emptyList();
    }
    List<Integer> result = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while(!queue.isEmpty()){
        int max = Integer.MIN_VALUE;
        for(int i = queue.size(); i > 0; i--){
            TreeNode node = queue.poll();
            max = Math.max(max, node.val);
            if(node.left != null) queue.offer(node.left);
            if(node.right != null) queue.offer(node.right);
        }
        result.add(max);
    }
    return result;
}
```

Balanced Binary Tree

```java
public boolean isBalanced(TreeNode root) {
    if(root == null) return true;
    return Math.abs(height(root.left) - height(root.right)) <= 1
            && isBalanced(root.left)
            && isBalanced(root.right);
}
private int height(TreeNode root){
    if(root == null) return -1;
    return 1 + Math.max(height(root.left), height(root.right));
}
```

Validate Binary Search Tree

```java
public boolean isValidBST(TreeNode root) {
    return validate (root, null, null);
}
public boolean validate (TreeNode root, Integer low, Integer high){
    if(root == null) return true;
    if((low != null && root.val <= low) || (high != null && root.val >= high)){
        return false;
    }
    return validate(root.right, root.val, high) && validate(root.left, low, root.val);
}
```

Count Complete Tree Nodes

```java
public int countNodes(TreeNode root) {
    if(root == null) return 0;
    TreeNode left = root.left;
    TreeNode right = root.right;
    int leftDepth = 0;
    int rightDepth = 0;
    while(left != null){
        left = left.left;
        leftDepth++;
    }
    while(right != null){
        right = right.right;
        rightDepth++;
    }
    if(leftDepth == rightDepth){
        return (int)(Math.pow(2,leftDepth+1) - 1);
    }
    return countNodes(root.left) + countNodes(root.right) + 1;
    }
}
```

Binary Tree Paths

```java
public List<String> binaryTreePaths(TreeNode root) {
    LinkedList<String> paths = new LinkedList<>();
    if(root == null) return paths;

    LinkedList<TreeNode> node_stack = new LinkedList<>();
    LinkedList<String> path_stack = new LinkedList<>();
    node_stack.add(root);
    path_stack.add(Integer.toString(root.val));
    TreeNode node;
    String path;
    while(!node_stack.isEmpty()){
        node = node_stack.pollLast();
        path = path_stack.pollLast();
        if((node.left == null) && (node.right == null)){
            paths.add(path);
        }
        if(node.left != null){
            node_stack.add(node.left);
            path_stack.add(path + "->" + Integer.toString(node.left.val));
        }
        if(node.right != null){
            node_stack.add(node.right);
            path_stack.add(path + "->" + Integer.toString(node.right.val));
        }
    }
    return paths;
}
```

Symmetric Tree

```java
public boolean isSymmetric(TreeNode root) {
    return isMirror(root, root);
}
public boolean isMirror(TreeNode t1, TreeNode t2){
    if(t1 == null && t2 == null) return true;
    if(t1 == null || t2 == null) return false;
    return (t1.val == t2.val)
            && isMirror(t1.right, t2.left)
            && isMirror(t1.left, t2.right);
}
```

```java
public abstract class Node<E extends Comparable<E>> {
    protected final E elem;

    protected Node(E content) {
        super(); // überflüssig, da implizit (Konstruktoraufruf der Oberklasse)
        if (content == null) // Versuch, Knoten mit Inhalt null zu erzeugen
            throw new IllegalArgumentException("must not be null");
        elem = content;
    }

    public boolean contains(E e) { // alternativ auch abstract möglich *1
        return elem.compareTo(e) == 0; // alternativ: elem.equals(e)
    }

    // Abstrakte Methoden (müssen von den Unterklassen implementiert werden):
    public abstract Node<E> append(E e);
    public abstract boolean equals(Object o);

    public String toString() { // alternativ abstract mögl. (oder weglassen)
        return elem.toString() + ", ";
    }
}
```

```java
public class Leaf<E extends Comparable<E>> extends Node<E> {
    public Leaf(E elem) {
        super(elem);  // explizit notwendig, da super() (ohne Param.) unmöglich
    }

    @Override  // @Override-Annotation ist bei Überschreibung immer optional
    public Node<E> append(E e) {
        Leaf<E> newNode = new Leaf<>(e); // neu einzufügender Blattknoten
        if (elem.compareTo(e) > 0)  // *2 (siehe Ende der Aufgabe)
            return new InnerNode<>(elem, newNode, null);
        else
            return new InnerNode<>(elem, null, newNode);
    }

    @Override
    public boolean equals(Object o) {  // *3
        if (o == null || !(o instanceof Leaf))
            return false;
        Leaf<E> oCasted = (Leaf<E>) o;
        return this.elem.compareTo(oCasted.elem) == 0;
    }
}
```

```java
public class InnerNode<E extends Comparable<E>> extends Node<E> {

    private Node<E> left, right;

    public InnerNode(E elem, Node<E> left, Node<E> right) {
        super(elem);
        this.left = left;
        this.right = right;
    }

    public boolean contains(E e) {
        if (super.contains(e)) // alternativ elem.compareTo(e) == 0
            return true;
        else if (elem.compareTo(e) > 0) // e < elem  *1
            return (left != null) ? left.contains(e) : false;
        else                                    // e > elem
            return (right != null) ? right.contains(e) : false;
    }

    public Node<E> append(E e) {  // *2 (siehe Ende der Aufgabe)
        if (elem.compareTo(e) > 0) // e < elem
            left = (left == null) ? new Leaf<>(e) : left.append(e);
        else                                    // e >= elem
            right = (right == null) ? new Leaf<>(e) : right.append(e);
        return this;  // „ich“ bleibe Nachfolger des Vorgängers
    }

    public boolean equals(Object o) {  // *3
        if (o == null || !(o instanceof InnerNode))
            return false;

        InnerNode<E> casted = (InnerNode<E>) o;

        if (this.elem.compareTo(casted.elem) != 0) // Inhalt gleich?
            return false;

        // Linke Nachfolger auf Gleichheit prüfen (könnten auch beide null sein!):
        if (this.left != null)
            if (!this.left.equals(casted.left))
                return false;  // this.left und casted.left ungleich
        else if (casted.left != null)
            return false;  // entweder left oder casted.left ist null

        // Rechte Nachfolger auf Gleichheit prüfen (gleich wie bei den linken, s. o.):
        if (this.right != null)
            if (!this.right.equals(casted.right))
                return false;
        else if (casted.right != null)
            return false;

        return true;  // nie false returnt → alles hat gepasst → Knoten gleich
    }
```

```java
    public String toString() {
        String result = "";
        if (left != null)
            result += left; // identisch zu result += left.toString() ...
        result += super.toString(); // result += elem.toString() + ", "
        if (right != null)
            result += right; // ... wegen String-Konkatenation
        return result;
    }
}
```

```java
public class BinaryTree<T extends Comparable<T>> {
    private Node<T> root;

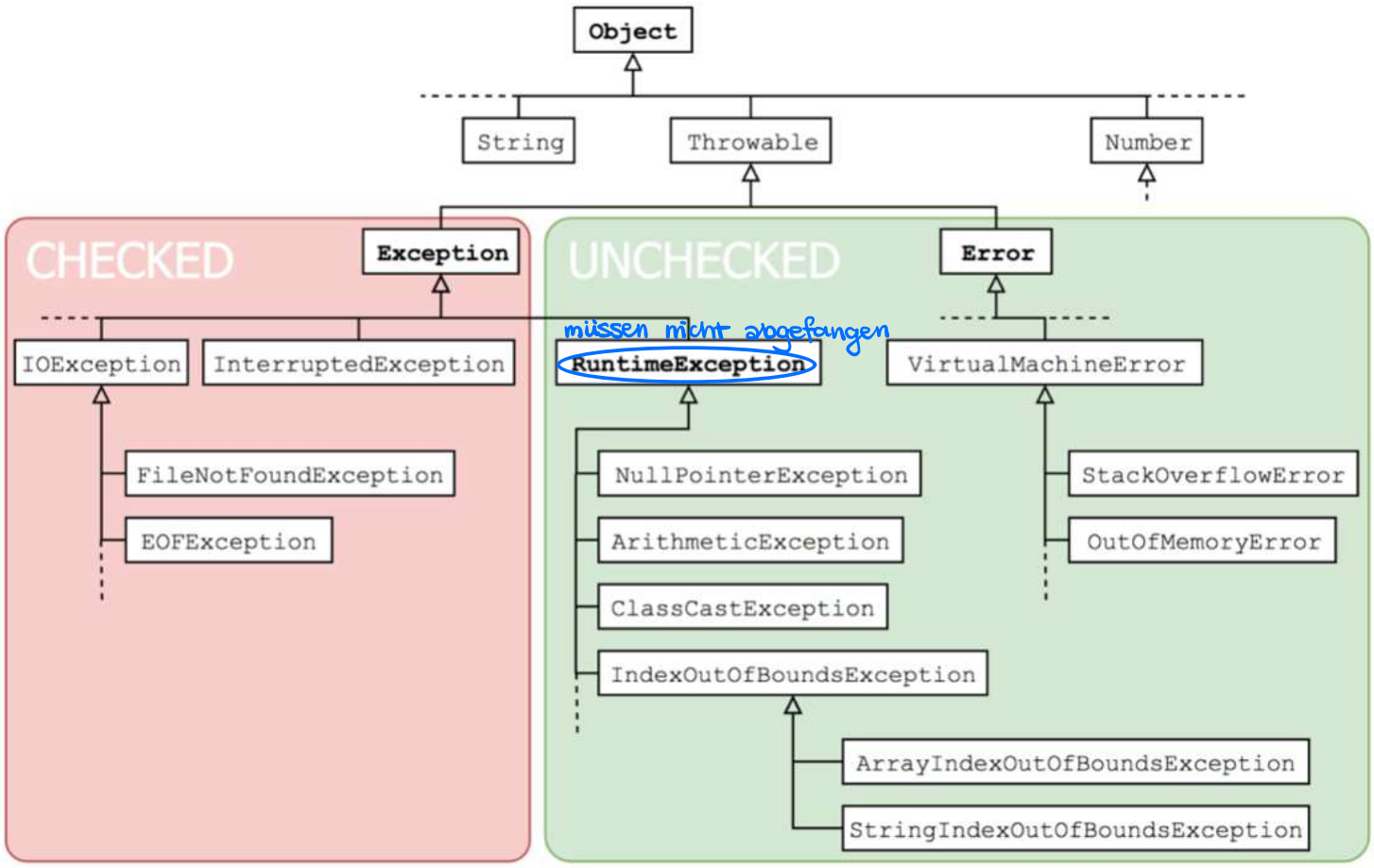
    public boolean contains(T element) {
        if (root == null)
            return false;
        return root.contains(element);
    }

    public void append(T element) {
        if (root == null)
            root = new Leaf<>(element);
        else
            root = root.append(element); // *2
    }

    @Override
    public boolean equals(Object o) {
        if (o == null)  // nicht unbedingt nötig (instanceof tut das auch)
            return false;
        if (!(o instanceof BinaryTree))
            return false;
        BinaryTree<T> casted = (BinaryTree<T>) o;
        if (this.root == null && casted.root == null) // beide sind leer
            return true;
        return root.equals(casted);  // delegiere an equals in Node
    }

    @Override
    public String toString() {
        if (root == null)
            return "[]";
        else
            return "[" + root.toString() + "]";
    }
}
```

Ausnahmen & Fehlerbehandlung



`StackOverFlowError` (Stack-Überlauf) entsteht bei zu vielen Methodenaufruf, da für jeden ein Speicherplatz auf dem Stack benötigt wird

`OutOfMemoryError` (Heap-Überlauf) entsteht bei der Erzeugung von zu vielen Objekten

`ClassCastException` (Laufzeitsfehler): entsteht bei TypeCasting `(A) b` , wenn der Parameter b nicht wirklich ein Objekt der Klasse A ist

Bei Integer-Overflow wird überhaupt keine Exception geworfen, sondern einfach fortgefahren - z.B. `Integer.MAX_ VALUE + 1 == Integer MIN_VALUE`

1
```
void anotherMethod() throws MyExc1, MyExc2 { ... }
```

2
```
try {                          // Tue etwas, wo eine Exception auftreten könnte:
    ...
    anotherMethod();
    ...
} catch (MyExc1 ex) {          // Wenn MyExc1 auftritt wird das hier ausgeführt:
    ...
} catch (MyExc2 ex) {          // Wenn MyExc2 auftritt wird das hier ausgeführt:
    ...                        // z. B. ex.printStackTrace();
} finally {                    // Optional! Wird immer ausgeführt (auch vor return):
    ...
}
```

```
1  public void foobar() throws FileNotFoundException {
2      FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
3  }
4
```

```
5  public void foobar(){
6      try{
7          FileReader fr = new FileReader("C:\\PinguDirectory\\fish.txt");
8      }catch(FileNotFoundException e){
9          System.out.println("Ups!");
10     }
11 }
```

```
public static String readFirstLine(String filePath) {
    // BufferedReader deklarieren, damit später sichtbar (Scope):
    BufferedReader reader;

    // BufferedReader öffnen:
    try {
        reader = new BufferedReader(new FileReader(filePath));
    } catch (FileNotFoundException e) {
        return null;   // Datei existiert nicht
    }

    // Erste Zeile lesen:
    try {
        return reader.readLine();
    } catch (IOException e) {
        return null;   // nicht lesbar
    } finally {
        // Den BufferedReader in jedem Fall (→ finally) (auch bei
        // IOException) wieder schließen. Dazu muss er geöffnet sein,
        // daher kann man dieses try-catch nicht mit dem erstem try-catch
        // zusammenfassen (weil er dort nicht geschlossen werden kann).
        try {
            reader.close();
        } catch (IOException e) {
            // Hier sind wir leider wirklich machtlos...
            // Wir müssen sie aber fangen (alternativ throws-Klausel)...
        }
    }
}
```

Intermediäre Methoden (für einen `Stream<T>`):

- `Stream<T>` **`filter`**`(`*`myPredicate`*`)` *mit* `myPredicate` *ist Funktion* `boolean test(T t)`
  → Der Ergebnisstream enthält alle Elemente des Streams, für die *`myPredicate`* zu `true` auswertet.
- `Stream<R>` **`map`**`(`*`myFunction`*`)` *mit* `myFunction` *ist Funktion* `R apply(T t)`
  → Der Ergebnisstream enthält die Ergebnisse der Anwendung von *`myFunction`* auf die Elemente.

  Bei primitiven Streams (`IntStream`, `LongStream`, `DoubleStream`) am Beispiel von `IntStream`:
  - `Stream<R>` **`mapToObj`**`(`*`myIntFunction`*`)` *mit* `myIntFunction` *ist Funktion* `R apply(int v)`
  - `IntStream` **`map`**`(`*`myIntFunction`*`)` *mit* `myIntFunction` *ist Funktion* `int applyAsInt(int v)`
- `IntStream` **`mapToInt`**`(`*`myIntFunction`*`)` *mit* `myIntFunction` *ist Funktion* `int applyAsInt(T t)`
  analog: **`mapToDouble`**, **`mapToLong`**
- `Stream<T>` **`peek`**`(`*`myConsumer`*`)` *mit* `myConsumer` *ist Funktion* `void accept(T t)`
  → Wie `forEach`, allerdings wird der Stream hier nicht abgeschlossen.
- `Stream<T>` **`distinct`**`()` → Eliminiert doppelte Elemente (Vergleich mittels `equals()`).
- `Stream<T>` **`limit`**`(long` *`maxSize`*`)` → Beschränkt den Stream auf die ersten *`maxSize`* Elemente.
- `Stream<T>` **`skip`**`(long` *`n`*`)` → Beschränkt den Stream auf *alle* Elemente *ohne* die ersten *`n`* vielen.
- `Stream<T>` **`sorted`**`(`*`myComparator`*`)` *mit* `myComparator` *ist Funktion* `int compare(T o1, T o2)`
  → Ordnet den Stream entsprechend der Vergleichsfunktion (negative Rückgabe, wenn `o1` vor `o2`) an.
  Die Vgl.fkt. `myComparator` ist optional, falls `T` das Interface `Comparable<T>` implementiert.
  Bei primitiven Streams entfällt der Parameter `myComparator` immer (natürliche Sortierung).
- `Stream<T>` **`parallel`**`()` → Verteilt die nachfolgenden Stream-Operationen auf mehrere Threads.

nur für primitive Streams:

- `Stream<...>` **`boxed`**`()` → Wandelt z. B. `LongStream` in einen `Stream<Long>` um.
- `DoubleStream` **`asDoubleStream`**`()` *und analog* **`asLongStream`** int → long → double

Abschließende Methoden (für einen `Stream<T>`):

- `void` **`forEach`**`(`*`myConsumer`*`)` *mit* `myConsumer` *ist Funktion* `void accept(T t)`
  → Führt die übergebene Funktion *`myConsumer`* für jedes Element des Streams aus.
- `boolean` **`anyMatch`**`(`*`myPredicate`*`)` *mit* `myPredicate` *ist Funktion* `boolean test(T t)`
  → Gibt `true` zurück, wenn *`myPredicate`* für mind. ein Element des Streams wahr ist.
  analog: **`allMatch`**, **`noneMatch`**
- `R` **`collect`**`(`*`someCollector`*`)` *mit* `someCollector` *ist z. B.* `Collectors.toList()` *bzw.* `toSet()`, `toMap(`*`myKeysMapper`*`,` *`myValuesMapper`*`)` *oder* `joining(String)` *(`joining` nur auf `Stream<String>`)*
  → Packt die Elemente des Streams in eine neue Datenstruktur (je nach *`someCollector`*).
- `A[]` **`toArray`**`(`*`arrayConstr`*`)` *mit* `arrayConstr` *ist z. B.* `String[]::new`, `Spieler[]::new`, ...
  Der Parameter `arrayConstr` entfällt bei primitiven Streams (wie `DoubleStream`).
- `T` **`reduce`**`(T` *`firstAccVal`*`,` *`myAccFunc`*`)` *mit* `myAccFunc` *ist Funktion* `T apply(T acc, T elem)`
  → Wendet *`myAccFunc`* nacheinander auf alle Stream-Elemente an und baut das Ergebnis über `acc` auf.
- `Optional<T>` **`min`**`(`*`myComparator`*`)` *mit* `myComparator` *ist Funktion* `int compare(T o1, T o2)`
  Bei primitiven Streams entfällt der Param., Rückgabe dann z. B. `OptionalInt` bei `IntStream`
  analog: **`max`**

nur für primitive Streams:

- `int` **`sum`**`()` → summiert die Elemente auf (zu `int` bei `IntStream`, sonst `double` bzw. `long`)
- `OptionalDouble` **`average`**`()` Ergebnis dann mit `<...>.getAsDouble()` zu `double` entpacken

Streams

Nenne mind. vier Beispiele zur Erzeugung eines *IntStreams* über 3, 4, 5, 6, 7.

```
IntStream.of(3, 4, 5, 6, 7)
IntStream.range(3, 8)
Arrays.stream(new int[] {3, 4, 5, 6, 7})
IntStream.iterate(3, i -> i + 1).limit(5)
Stream.iterate(3, i -> i+1) .limit(5).mapToInt(i -> i)
```

Nenne Beispiele zur Erzeugung eines String-Streams über a, aa, aaa, aaaa, aaaaa.

```
Stream.of("a", "aa", "aaa", "aaaa", "aaaaa")
Stream.iterate("a", str -> str + "a").limit(5)
Stream.iterate("", str -> str + "a").skip(1).limit(5)
IntStream.range(1, 6).mapToObj(i -> new String("a").repeat(i))
```

Erzeuge mittels `Stream.iterate` einen Stream über alle Großbuchstaben (A bis Z).

```
Stream.iterate('A', c -> (char) (c + 1)).limit(26)
Stream.iterate('A', c -> c <= 'Z', c -> (char) (c + 1))
Stream.iterate(0, i -> i + 1).limit(26).map(i -> (char)(int)i)
```

Stream API

- **无存储**。*stream*不是一种数据结构，它只是某种数据源的一个视图，数据源可以是一个数组，Java容器或I/O channel等。
- **为函数式编程而生**。对*stream*的任何修改都不会修改背后的数据源，比如对*stream*执行过滤操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新*stream*。
- **惰式执行**。*stream*上的操作并不会立即执行，只有等到用户真正需要结果的时候才会执行。
- **可消费性**。*stream*只能被"消费"一次，一旦遍历过就会失效，就像容器的迭代器那样，想要再次遍历必须重新生成。

中间操作 (Intermediate operations): 总是会惰式执行，调用中间操作只会生成一个标记了该操作的新 stream，仅此而已

a) 无状态的 (Stateless): 元素的处理不受前面元素的影响 `unordered()` `filter()` `map()` `mapToInt()` `mapToLong()` `mapToDouble()` `flatMap()` `flatMapToInt()` `flatMapToLong()` `flatMapToDouble()` `peek()`

b) 有状态的 (Stateful): 必须等到所有元素处理之后才知道最终结果，比如排序是有状态操作，在读取所有元素之前并不能确定排序结果 `distinct()` `sorted()` `sorted(comparator c)` `limit()` `skip()`

结束操作 (Terminal operations): 会触发实际计算，计算发生时会把所有中间操作积攒的操作以*pipeline*的方式执行，这样可以减少迭代次数。计算完成之后stream就会失效

a) 短路操作 (short-circuiting): 不用处理全部元素就可以返回结果，比如*找到第一个满足条件的元素* `anyMatch()` `allMatch()` `noneMatch()` `findFirst()` `findAny()`

b) 非短路操作: 处理完全部元素就可以返回结果 `forEach()` `forEachOrdered()` `toArray()` `reduce()` `collect()` `max()` `min()` `count()`

方法引用

如果Lambda表达式的全部内容就是调用一个已有的方法，那么可以用方法引用来替代Lambda表达式

`Klassenname::methodenname`	(bei statischen Methoden)
`objekt::methodenname`	(bei nicht-statischen Methoden (selten))

Klassenname :: new **(bei Konstruktur)**

forEach()

`void` **`forEach`**`(myConsumer)` *mit* `myConsumer` *ist Funktion* `void accept(T t)`

→ Führt die übergebene Funktion `myConsumer` für jedes Element des Streams aus.

```
// 使用Stream.forEach()迭代
Stream<String> stream = Stream.of("I", "love", "you", "too");
stream.forEach(str -> System.out.println(str));
```

1. Beispiel: Gibt die Strings des Arrays in Großschreibung aus.

```
String[] lang = {"de", "En", "es"};
Arrays.stream(lang).forEach(s -> System.out.println(s.toUpperCase()));
```

2. Beispiel: Setzt die Anzahl an Toren aller Spieler, die 2 Tore haben, auf 0 (hier für Max und Ed).

```
Arrays.stream(mannschaft).forEach(spieler -> {
    if (spieler.tore == 2)
        spieler.tore = 0;
});
```

filter()

`Stream<T>` **`filter`**`(myPredicate)` *mit* `myPredicate` *ist Funktion* `boolean test(T t)`

→ Der Ergebnisstream enthält alle Elemente des Streams, für die `myPredicate` zu `true` auswertet.

```
// 保留长度等于3的字符串
Stream<String> stream= Stream.of("I", "love", "you", "too");
stream.filter(str -> str.length()==3)
    .forEach(str -> System.out.println(str));
```

sort()

`Stream<T>` **`sorted`**`(myComparator)` *mit* `myComparator` *ist Funktion* `int compare(T o1, T o2)`

→ Ordnet den Stream entsprechend der Vergleichsfunktion (negative Rückgabe, wenn `o1` vor `o2`) an.

```
Stream<String> stream = Stream.of("I", "love", "you", "too");
//按照长度升序排序 aufsteigende Reihenfolge
stream.sorted((str1, str2) -> str1.length()-str2.length())
    .forEach(str -> System.out.println(str));
//按照长度降序排序 absteigende Reihenfolge
stream.sorted((str1, str2) -> str2.length()-str1.length())
    .forEach(str -> System.out.println(str));
```

limit()

`Stream<T>` **`limit`**`(long maxSize)` → Beschränkt den Stream auf die ersten `maxSize` Elemente.

1. Beispiel: Beschränkt den unendlichen Stream 3, 4, 5, ... auf die Elemente 3 bis 7. ⇒ `3:4:5:6:7:`

```
Stream.iterate(3, x -> x + 1).limit(5).forEach(x -> System.out.print(x + ":"));
```

2. Beispiel: Erzeugt ein Array bestehend aus 5 Zufallszahlen im Bereich von 0 (inkl.) bis 10 (exkl.), wobei keine Zahl doppelt vorkommt (`ints` gibt einen unendlichen *IntStream* von Zahlen).

```
int[] a = new Random().ints(0, 10).distinct().limit(5).toArray();
```

reduce()

`T` **`reduce`**`(T firstAccVal, myAccFunc)` *mit* `myAccFunc` *ist Funktion* `T apply(T acc, T elem)`

→ Wendet `myAccFunc` nacheinander auf alle Stream-Elemente an und baut das Ergebnis über `acc` auf.

```
// 找出最长的单词
Stream<String> stream = Stream.of("I", "love", "you", "too");
Optional<String> longest = stream.reduce((s1, s2) -> s1.length()>=s2.length() ? s1 : s2);
//Optional<String> longest = stream.max((s1, s2) -> s1.length()-s2.length());
System.out.println(longest.get());
```

```
// 求单词长度之和
Stream<String> stream = Stream.of("I", "love", "you", "too");
Integer lengthSum = stream.reduce(0,  // 初始值  // (1)
        (sum, str) -> sum+str.length(), // 累加器 // (2)
        (a, b) -> a+b);  // 部分和拼接器，并行执行时才会用到 // (3)
// int lengthSum = stream.mapToInt(str -> str.length()).sum();
System.out.println(lengthSum);
```

Stream.reduce()用法举例

新元素如何累加

reduce(0, ← 初始值（幺元）

(sum,str)->sum+str.length(),

(a,b)->a+b)

多个部分和如何合并

2. Beispiel: Verknüpft die Wörter, die mit *t* beginnen, zu einem Text (Ergebnis: `"#-ton-tee"`).

```
System.out.println(Stream.of("ton","rot","ab","tee").reduce("#", (acc, str) -> {
    if (str.startsWith("t"))
        return acc + "-" + str;
    return acc;
}));
```

3. Beispiel: Berechnet die Summe der Tore aller Spieler (also hier 8).

```
int anzahlTore = Arrays.stream(mannschaft).map(spieler -> spieler.tore)
        .reduce(0, (acc, elem) -> elem + acc);  // oder .reduce(0, Integer::sum)
```

anyMatch()

`boolean` **`anyMatch`**`(myPredicate)` *mit* `myPredicate` *ist Funktion* `boolean test(T t)`

→ Gibt `true` zurück, wenn `myPredicate` für mind. ein Element des Streams wahr ist.

analog: **`allMatch`**, **`noneMatch`**

1. Beispiel: Prüft, ob *mind. ein* Spieler mehr als 3 Tore geschossen hat (hier nicht, also *keine* Ausgabe).

```
if (Arrays.stream(mannschaft).anyMatch(spieler -> spieler.tore > 3))
    System.out.println("Es gibt einen Spieler mit über 3 Toren");
```

2. Beispiel: Prüft, ob *alle* Elemente aus 4, -6, 2, 8 gerade sind (ja, also folgt die Ausgabe).

```
if (IntStream.of(4, -6, 2, 8).allMatch(x -> x % 2 == 0))
    System.out.println("Alle Elemente sind gerade.");
```

peek()

`Stream<T>` **`peek`**`(myConsumer)` *mit* `myConsumer` *ist Funktion* `void accept(T t)`

→ Wie `forEach`, allerdings wird der Stream hier nicht abgeschlossen. → kann weiter verwendet

Beispiel: Gibt jeden Spieler mittels `println()` (→ `toString`) auf der Konsole aus, bevor die Anzahl seiner Tore auf 0 gesetzt wird. Die Ausgabe ist also *Max[2], Luis[3], Ed[2], Tom[1].* Anschließend haben alle Spieler 0 Tore. Danach: Spieler werden mit `collect` als Menge gespeichert.

```
Set<Spieler> players = Arrays.stream(mannschaft)
                            .peek(System.out::println)
                            .peek(sp -> sp.tore = 0)
                            .collect(Collectors.toSet());
```

distinct()

`Stream<T>` **`distinct()`** → Eliminiert doppelte Elemente (Vergleich mittels `equals()`).

```
Stream<String> stream= Stream.of("I", "love", "you", "too", "too");
stream.distinct()
    .forEach(str -> System.out.println(str));
```

skip()

`Stream<T>` **`skip`**`(long n)` → Beschränkt den Stream auf *alle* Elemente *ohne* die ersten *n* vielen.

Beispiel: Überspringt das Element „`line 1`“, d. h. der Stream enthält nur noch `line 2`, `3` und `4`. Daher wertet `count()` zu `3` aus (Anzahl Stream-Elemente).

```
System.out.println(
   Arrays.stream("line 1\nline 2\nline 3\nline 4".split("\n")).skip(1).count());
```

map()

`Stream<R>` **`map`**`(myFunction)` *mit* `myFunction` *ist Funktion* `R apply(T t)`

→ Der Ergebnisstream enthält die Ergebnisse der Anwendung von `myFunction` auf die Elemente.

```
Stream<String> stream  = Stream.of("I", "love", "you", "too");
stream.map(str -> str.toUpperCase())
    .forEach(str -> System.out.println(str));
```

mapToInt()

`IntStream` **`mapToInt`**`(myIntFunction)` *mit* `myIntFunction` *ist Funktion* `int applyAsInt(T t)`

analog: **`mapToDouble`**, **`mapToLong`**

Beispiel: Berechnet die Summe der Tore aller Spieler durch einen Stream über die Einzel-Tore. Erg.: `8`

```
System.out.println(Arrays.stream(mannschaft).mapToInt(sp -> sp.tore).sum());
```

toArray()

`A[]` **`toArray`**`(arrayConstr)` *mit* `arrayConstr` *ist z. B.* `String[]::new`, `Spieler[]::new`, ...

Der Parameter `arrayConstr` entfällt bei primitiven Streams (wie `DoubleStream`).

1. Beispiel: Packt die Elemente des Streams (1, 6, 3) in ein Integer-Array (`int[]` wäre nicht möglich!).

```
Integer[] arr = Stream.of(1,6,3).toArray(Integer[]::new);
System.out.println(Arrays.toString(arr));  // Ausgabe: [1, 6, 3]
```

2. Beispiel: Erzeugt ein `String`-Array aus den Spielernamen (also `["Max","Luis","Ed","Tom"]`)

```
String[] names = Arrays.stream(mannschaft).map(s->s.name).toArray(String[]::new);
```

3. Beispiel: Erzeugt ein `int`-Array aus den Toren der Spieler (Vorgriff auf primitive Streams).

```
int[] spielerTore = Arrays.stream(mannschaft).mapToInt(s -> s.tore).toArray();
```

min()

`Optional<T>` **`min`**`(myComparator)` *mit* `myComparator` *ist Funktion* `int compare(T o1, T o2)`

Bei primitiven Streams entfällt der Param., Rückgabe dann z. B. `OptionalInt` bei `IntStream`

analog: **`max`**

1. Beispiel: Ermittelt den Spieler mit den meisten Toren und gibt dessen Name aus (hier: Luis).

```
Optional<Spieler> bester = Arrays.stream(mannschaft)
   .max((sp1, sp2) -> sp1.tore > sp2.tore ? 1 : (sp1.tore < sp2.tore ? -1 : 0));
if (bester.isPresent())
    System.out.println("Bester: " + bester.get().name);
else
    System.out.println("Es gibt keine Spieler...");
```

2. Beispiel: Ermittelt das Minimum der gegebenen Zahlen (hier: 4).

```
System.out.println(Stream.of(8, 4, 9, 6).min((a, b) -> a - b).get());
```

3. Beispiel: Ermittelt den kürzesten Namen aller Spielernamen und gibt ihn aus (Ausgabe hier: Ed).

```
System.out.println( Arrays.stream(mannschaft).map(spieler -> spieler.name)
                    .min((str1, str2) -> {
                        if (str1.length() < str2.length())
                            return -1;  // str1 soll vor str2 kommen
                        else if (str1.length() > str2.length())
                            return 1;
                        else return 0;  // Namen sind gleich
                    }).orElse("Es gibt keine Spieler...") );
```

终极武器 collect()

R **collect**(*someCollector*) *mit someCollector ist z. B.* Collectors.toList() *bzw.* toSet(), toMap(*myKeysMapper*, *myValuesMapper*) *oder* joining(String) *(joining nur auf Stream<String>)*
→ Packt die Elemente des Streams in eine neue Datenstruktur (je nach *someCollector*).

```
//  将Stream规约成List
Stream<String> stream = Stream.of("I", "love", "you", "too");
List<String> list = stream.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);// 方式1
//List<String> list = stream.collect(Collectors.toList());// 方式2
System.out.println(list);
```



Collectors.toList() 和 Collectors.toSet() -> 使用 collect() 生成 Collection

前面已经提到通过`collect()`方法将*Stream*转换成容器的方法，这里再汇总一下。将*Stream*转换成*List*或*Set*是比较常见的操作，所以*Collectors*工具已经为我们提供了对应的收集器，通过如下代码即可完成：

```
// 将Stream转换成List或Set
Stream<String> stream = Stream.of("I", "love", "you", "too");
List<String> list = stream.collect(Collectors.toList()); // (1)
Set<String> set = stream.collect(Collectors.toSet()); // (2)
```

上述代码能够满足大部分需求，但由于返回结果是接口类型，我们并不知道类库实际选择的容器类型是什么，有时候我们可能会想要人为指定容器的实际类型，这个需求可通过`Collectors.toCollection(Supplier<C> collectionFactory)`方法完成。

```
// 使用toCollection()指定规约容器的类型
ArrayList<String> arrayList = stream.collect(Collectors.toCollection(ArrayList::new));// (3)
HashSet<String> hashSet = stream.collect(Collectors.toCollection(HashSet::new));// (4)
```

Collectors.joining() -> 使用 collect() 做字符串 joining

这个肯定是大家喜闻乐见的功能，字符串拼接时使用`Collectors.joining()`生成的收集器，从此告别*for*循环。
`Collectors.joining()`方法有三种重写形式，分别对应三种不同的拼接方式。无需多言，代码过目难忘。

```
// 使用Collectors.joining()拼接字符串
Stream<String> stream = Stream.of("I", "love", "you");
//String joined = stream.collect(Collectors.joining());// "Iloveyou"
//String joined = stream.collect(Collectors.joining(","));// "I,love,you"
String joined = stream.collect(Collectors.joining(",", "{", "}"));// "{I,love,you}"
```

1. Beispiel: Wandelt den Stream von Spielern mittels `collect` in eine `java.util.List` um.

```
List<Spieler> spieler = Arrays.stream(mannschaft).collect(Collectors.toList());
```

2. Beispiel: Verknüpft die Elemente des String-Streams zu einem String (`"rotblaugelb"`).

```
String s = Stream.of("rot", "blau", "gelb").collect(Collectors.joining());
```

3. Beispiel: Verknüpft die Spielernamen zu dem kommagetrennten String `"Max, Luis, Ed, Tom"`.

```
String s = Arrays.stream(mannschaft).map(sp -> sp.name) // map kommt noch...
                                    .collect(Collectors.joining(", "));
```

4. Beispiel: Erzeugt für die Zahlen 1 bis 5 eine Abbildung der Zahl als String auf ihre Quadratzahl, d. h. das Ergebnis `myMap` ist die `Map` {`"1"` → 1, `"2"` → 4, `"3"` → 9, `"4"` → 16, `"5"` → 25 }. Anschließend wird die `Map` mittels `forEach` ausgegeben (Stream über Schlüssel-Menge). `boxed()` ist nötig, um den `IntStream` (vgl. Kap. 19.4.3) zu einem `Stream<Integer>` umzuwandeln, weil `collect` nicht für primitive Streams existiert (zum. nicht in dieser Form).

```
Map<String,Integer> myMap = IntStream.range(1, 6).boxed()
                                     .collect(Collectors.toMap(i -> ""+i, i -> i*i));
myMap.keySet().stream().forEach(k -> System.out.println(k + ": " + myMap.get(k)));
```

5. Beispiel: Gruppiert die Spieler nach der Anzahl ihrer geschossenen Tore in Listen, d. h. das Ergebnis `grouped` ist die `Map` { 1 → [*Tom*], 2 → [*Max*, *Ed*], 3 → [*Luis*] }.

```
Map<Integer, List<Spieler>> grouped = Arrays.stream(mannschaft)
                                    .collect(Collectors.groupingBy(sp -> sp.tore));
```

Collectors.toMap() -> 使用 collect() 生成 Map

前面已经说过*Stream*背后依赖于某种数据源，数据源可以是数组、容器等，但不能是*Map*。反过来从*Stream*生成*Map*是可以的，但我们要想清楚*Map*的*key*和*value*分别代表什么，根本原因是我们要想清楚要干什么。通常在三种情况下`collect()`的结果会是*Map*：

Schlüssel muss eindeutig -> keine Duplikate erlaubt

1. 使用`Collectors.toMap()`生成的收集器，用户需要指定如何生成*Map*的*key*和*value*。
2. 使用`Collectors.partitioningBy()`生成的收集器，对元素进行二分区操作时用到。
3. 使用`Collectors.groupingBy()`生成的收集器，对元素做*group*操作时用到。

mehrere Elemente dürfen auf den selben Schlüssel in Liste gruppiert

情况1：使用`toMap()`生成的收集器，这种情况是最直接的，前面例子中已提到，这是和`Collectors.toCollection()`并列的方法。如下代码展示将学生列表转换成由<学生，GPA>组成的*Map*。非常直观，无需多言。

```
// 使用toMap()统计学生GPA
Map<Student, Double> studentToGPA =
     students.stream().collect(Collectors.toMap(Function.identity(),// 如何生成key
                                        student -> computeGPA(student)));// 如何生成value
```

情况2：使用`partitioningBy()`生成的收集器，这种情况适用于将`Stream`中的元素依据某个二值逻辑（满足条件，或不满足）分成互补相交的两部分，比如男女性别、成绩及格与否等。下列代码展示将学生分成成绩及格或不及格的两部分。

```
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
         .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

情况3：使用`groupingBy()`生成的收集器，这是比较灵活的一种情况。跟SQL中的*group by*语句类似，这里的*groupingBy()也是按照某个属性对数据进行分组，属性相同的元素会被对应到Map*的同一个*key*上。下列代码展示将员工按照部门进行分组：

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
            .collect(Collectors.groupingBy(Employee::getDepartment));
```

以上只是分组的最基本用法，有些时候仅仅分组是不够的。在SQL中使用*group by*是为了协助其他查询，比如*1. 先将员工按照部门分组，2. 然后统计每个部门员工的人数*。Java类库设计者也考虑到了这种情况，增强版的`groupingBy()`能够满足这种需求。增强版的`groupingBy()`允许我们对元素分组之后再执行某种运算，比如求和、计数、平均值、类型转换等。这种先将元素分组的收集器叫做**上游收集器**，之后执行其他运算的收集器叫做**下游收集器**(*downstream Collector*)。

```
// 使用下游收集器统计每个部门的人数
Map<Department, Integer> totalByDept = employees.stream()
                    .collect(Collectors.groupingBy(Employee::getDepartment,
                                                   Collectors.counting()));// 下游收集器
```

上面代码的逻辑是不是越看越像SQL？高度非结构化。还有更狠的，下游收集器还可以包含更下游的收集器，这绝不是为了炫技而增加的把戏，而是实际场景需要。考虑将员工按照部门分组的场景，如果*我们想得到每个员工的名字（字符串），而不是一个个*Employee*对象*，可通过如下方式做到：

```
// 按照部门对员工分布组，并只保留员工的名字
Map<Department, List<String>> byDept = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                        Collectors.mapping(Employee::getName,// 下游收集器
                                Collectors.toList())));// 更下游的收集器
```

sum()

```java
public long sum(long n) {
    return LongStream.range(1, n+1).sum();
}
```

average Temperatur

```java
public static double avgTemp(double[] t) {
    return Arrays.stream(t)          // DoubleStream über alle doubles
                 .average()          // Durchschnitt als OptionalDouble
                 .getAsDouble();     // OptionalDouble zu double
```

remove Duplicates

```java
public static int[] removeDuplicates(int[] arr) {
    return Arrays.stream(arr)        // int-Array zu IntStream
                 .distinct()         // Duplikate entfernen
                 .toArray();         // IntStream zu int-Array
}
```

Vielfache von num

```java
public static int[] extractMultiplesOf(int[] array, int num) {
    return Arrays.stream(array)
                 .filter(x -> x % num == 0)  // beschränkt den Stream auf
                 .distinct()                 // Elemente, für die x%num = 0
                 .toArray();
}
```

reverse

```java
static void reverse(Object[] a) {
    // Folgendes entspricht genau der for-Schleife aus der urspr. Lösung:
    IntStream.range(0, a.length/2).forEach(i -> {
        Object tmp = a[i];
        a[i] = a[a.length-1-i];
        a[a.length-1-i] = tmp;
    });
}
```

Potenz von x (bis max. n)

```java
public static long[] powerOfX(int x, byte n) {
    return IntStream.range(0, n+1)
                    .mapToLong(i -> (long) Math.pow(x, i))
                    .toArray();
}
```

Print stars

```java
public static void printTriangle(int size) {
    IntStream.range(0, size).forEach(x -> {
            IntStream.range(0, x+1).forEach(y ->
                    System.out.print("*")
            );
            System.out.println();
    });
}
```

Palindrome

```java
public boolean isPalindrome(char[] c) {
    return IntStream.range(0, c.length)
                    .allMatch(i -> c[i] == c[c.length-i-1]);
}
```

n Fibonacci Zahlen

```java
static void fibonacci(int n) {
    System.out.println(Stream.iterate(new long[] {0, 1},
                      arr -> new long[] {arr[1], arr[0] + arr[1]})
            .limit(n).map(arr -> arr[1]).map(String::valueOf)
            .collect(Collectors.joining(", ")));
}
```

Summe der 3 drößten geraden Zahlen

```java
public static void main(String[] args) {
    int zahlen[] = {5, 8, 9, 2, -10, 3, 4, 7, 6, 1};
    System.out.println(
```

a)

```java
Arrays.stream(zahlen)
    .filter(i -> i % 2 == 0)
    .map(i -> -i)
    .sorted()
    .map(i -> -i)
    .limit(3)
    .sum());
```

b)

```java
Arrays.stream(zahlen)
        .filter(i -> i % 2 == 0)
        .boxed()
        .sorted((a, b) -> b - a)
        .limit(3)
        .mapToInt(i -> i)
        .sum()
```

Primzahlen (2 bis Max)

```java
int[] primes(int max) {
    return IntStream.range(2, max+1)
                    .filter(num ->
                        IntStream.range(2, num)
                                 .allMatch(div -> num % div != 0)
                    )
                    .toArray();
}
```

erst n viele Primzahlen

```java
long[] morePrimes(int count) {
    return LongStream.iterate(2, x -> x + 1)
                .filter(num -> LongStream.range(2, num/2+1)
                                         .noneMatch(div -> num % div == 0))
                .limit(count)
                .toArray();
}
```

Hex to Dez Zahlen

```java
public static long hexToDez(String hex) {
    if (hex == null || !hex.startsWith("0x"))
        throw new IllegalArgumentException();

    return hex.chars()      // IntStream über alle Zeichen von hex
        .skip(2)            // Überspringe das Präfix (0 und x)
        .map(digit -> {     // Bilde jedes Zeichen digit auf ...
                            // ... den zugehörigen int-Wert ab:
            if (digit >= '0' && digit <= '9') // '0'-'9'
                return digit - '0';           // wird zu 0-9
            if (digit >= 'a' && digit <= 'f') // 'a'-'f'
                return digit - 'a' + 10;      // wird zu 10-15
            // Alle anderen Zeichen sind ungültig:
            throw new IllegalArgumentException();
        })
        .reduce(0, (acc, i) -> acc * 16 + i); // siehe unten
}
```

Most frequent elements

```java
public static Set<Integer> mostFrequentElements(int[] arr) {
    // Wie bei der Lösung mit Schleifen wird zunächst die Map erstellt,
    // welche die Elemente auf ihre Häufigkeiten abbildet.
    Map<Integer, Integer> counts = Arrays.stream(arr)
            .distinct()     // jedes Element nur einmal zählen
            .boxed()        // damit collect möglich ist
            .collect(Collectors.toMap(
                elem -> elem,   // Array-Element als Schlüssel übernehmen
                elem -> (int) Arrays.stream(arr)   // neuer Stream
                        .filter(x -> x == elem)    // nur elem behalten
                        .count()  // Stream-Länge als Value übernehmen
            ));           // Cast zu int nötig, weil count() long zurückgibt

    // Dann bestimmen wir das Maximum:
    int max = counts.values()   // Häufigkeiten
            .stream()   // Stream<Integer>
            .mapToInt(x -> x)   // IntStream (damit max() möglich ist)
            .max()   // OptionalInt
            .orElse(0); // max = Maximum, falls Stream nicht leer, sonst 0

    // Jetzt suchen wir alle Keys, deren Value gleich max ist.
    return counts.keySet()   // alle Elemente
            .stream()   // Stream<Integer>
            .filter(x -> counts.get(x) == max) // Häufigkeit ist maximal
            .collect(Collectors.toSet()); // verbleibende zurückgeben
}
```

Hier ein Beispiel für einen Echo-Server. Dieser erlaubt nur die gleichzeitige Verbindung eines einzigen Clients. Trennt sich dieser, so kann der nächste Client eine Verbindung aufbauen. Der Echo-Server schickt dem Client das zurück, was er vom Client empfängt:

```
public class EchoServer {
   public static void main(String[] args) throws IOException {
      ServerSocket server = null;
      try {
         server = new ServerSocket(8000); // Server-Socket erzeugen
      } catch (BindException e) {
         System.out.println("8000 ist bereits durch einen Server belegt.");
         return;
      }
      try {
         while (true) {   // dauerhaft Clients akzeptieren (jeweils nur einer gleichzeitig)
            Socket client = server.accept();   // neuen Client annehmen

            BufferedReader in = new BufferedReader(new InputStreamReader(
                                                   client.getInputStream()));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(
                                                   client.getOutputStream()));

            // Start: Ab jetzt kann mit dem Client kommuniziert werden (über in und out)
            while (true) {
               String line = in.readLine();   // empfange Daten vom Client
               if (line == null) // Client hat Verbindung geschlossen
                  break;
               out.println(line);   // dieselben Daten an den Client zurückschicken
               out.flush();   // Senden durchführen (println schreibt nur in den Buffer)
            }
            // Ende

            client.close();   // Verbindung zum zuvor angenommenen Client schließen
         }
      } finally {
         server.close(); // am Ende das Server-Socket wieder schließen (in jedem Fall)
      }
   }
}
```

Einfacher Client, der eine Nachricht sendet und die Server-Antwort auf der Konsole ausgibt:

```
public class Client {
   public static void main(String[] args) throws IOException  {
      Socket socket = new Socket("127.0.0.1", 8000);   // zum Server verbinden

      try {
         BufferedReader in = new BufferedReader(new InputStreamReader(
                                                socket.getInputStream()));
         PrintWriter out = new PrintWriter(new OutputStreamWriter(
                                                socket.getOutputStream()));

         out.println("Dieser Text wird an den Server geschickt");
         out.flush();

         System.out.println("Antwort des Servers: " + in.readLine());
      } finally {
         socket.close();
      }
   }
}
```

Erzeugung eines Threads

实现接口 VS 继承 Thread -> 实现接口会更好一些

- Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口
- 类可能只要求可执行就行，继承整个 Thread 类开销过大

Ein Thread darf nur einmal start() werden. Um run() wiederholt auszuführen, muss es explizit in einer Schleife implementiert

1.Als eigene Klasse, die Runnable Interface implementiert

方法1: 实现 Runnable 接口

– 实现接口中的 run() 方法

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // ...
    }
}
```

– 使用 Runnable 实例再创建一个 Thread 实例，然后调用 Thread 实例的 start() 方法来启动线程

```
public static void main(String[] args) {
    MyRunnable instance = new MyRunnable();
    Thread thread = new Thread(instance);
    thread.start();
}
```

2. Als Unterklasse von Thread

方法2: 继承 Thread 类

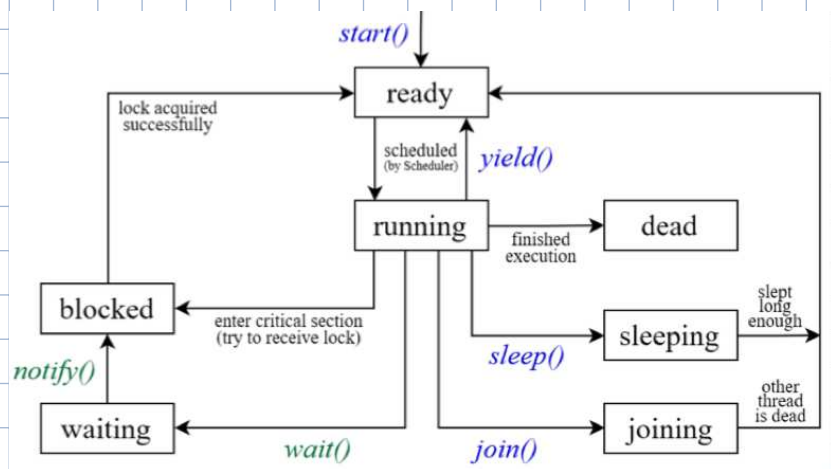
- 需要实现 run() 方法，因为 Thread 类也实现了 Runnable 接口

```
public class MyThread extends Thread {
    public void run() {
        // ...
    }
}
```

- 当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。

```
public static void main(String[] args) {
    MyThread mt = new MyThread();
    mt.start();
}
```

Threads 机制



Grundlegende 基础机制 start() yield() sleep()

- `public void start()`: Führt die Thread-Erzeugung durch, d. h. der Thread bekommt den Zustand „*ready*". Sobald der Scheduler den Thread in den Zustand „*running*" versetzt, führt dieser die `run`-Methode aus. Threads werden niemals mit `t.run()` ausgeführt, da sonst der entscheidende Schritt der Thread-Erzeugung fehlt!
- `public static void yield()`: Der aktuell ausgeführte Thread unterbricht seine Ausführung selbstständig und wird von „*running*" auf „*ready*" zurückgesetzt. Dies ist sinnvoll, wenn ein Thread weiß, dass er aktuell keine Rechenzeit benötigt (z. B. weil er auf etwas warten muss) und diese somit an einen anderen Thread abgeben kann.

```
public void run() {
    Thread.yield();
}
```

- `public static void sleep(int msec) throws InterruptedException`: Legt den aktuell in Ausführung (Zustand „*running*") befindlichen Thread für eine bestimmte Zeit schlafen. Der Thread kehrt frühestens nach der spezifizierten Zeit wieder in den Zustand „*ready*" zurück, beginnt die Ausführung aber nicht unbedingt zur selben Zeit wieder!

```
public void run() {
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Interrupt 中断

一个线程执行完毕之后会自动结束，如果在运行过程中发生异常也会提前结束。

- `public void interrupt()`: Beim Aufruf von `t.interrupt()` wird an den `Thread t` ein *Interrupt*-Signal gesendet. Dieser nimmt dieses Signal wahr, indem dort an manchen Stellen (z. B. im Zustand „*joining*" oder „*sleeping*" ist) eine `InterruptedException` geworfen wird. Der Thread kann darauf reagieren (festgelegt im jeweiligen `catch`-Block), muss aber nicht. Es handelt sich somit mehr um eine „Bitte" als um eine „Aufforderung".

```
public class InterruptExample {

    private static class MyThread2 extends Thread {
        @Override
        public void run() {
            while (!interrupted()) {
                // ..
            }
            System.out.println("Thread end");
        }
    }
}
```

```
public static void main(String[] args) throws InterruptedException {
    Thread thread2 = new MyThread2();
    thread2.start();
    thread2.interrupt();
}
```

```
Thread end
```

InterruptedException

– 通过调用一个线程的 interrupt() 来中断该线程，如果该线程处于阻塞、限期等待或者无限期等待状态，那么就会抛出 InterruptedException，从而提前结束该线程。但是不能中断 I/O 阻塞和 synchronized 锁阻塞

– 对于以下代码，在main() 中启动一个线程之后再中断它，由于线程中调用了 Thread.sleep() 方法，因此会抛出一个 InterruptedException，从而提前结束线程，不执行之后的语句

```
public class InterruptExample {

    private static class MyThread1 extends Thread {
        @Override
        public void run() {
            try {
                Thread.sleep(2000);
                System.out.println("Thread run");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public static void main(String[] args) throws InterruptedException {
    Thread thread1 = new MyThread1();
    thread1.start();
    thread1.interrupt();
    System.out.println("Main run");
}
```

Koordination 线程之间的协调

当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调。

join()

- `public final void join() throws InterruptedException`: Der Thread, der `t.join()` aufruft, wartet anschließend auf Beendigung des Threads `t` und wechselt somit in den Zustand „*joining*“. Er kehrt erst in den Zustand „*ready*“ zurück und fährt mit der Ausführung fort (in der nächsten Codezeile), wenn `t` beendet wurde. Er kann beim Warten unterbrochen werden und sollte in einem `try-catch`-Block spezifizieren, wie er darauf reagieren möchte (ggf. gar nicht, indem einfach wieder `join` aufgerufen wird → Endlosschleife).

在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。

对于以下代码，虽然 b 线程先启动，但是因为在 b 线程中调用了 a 线程的 join() 方法，b 线程会等待 a 线程结束才继续执行，因此最后能够保证 a 线程的输出先于 b 线程的输出。

```
public class JoinExample {

    private class A extends Thread {
        @Override
        public void run() {
            System.out.println("A");
        }
    }

    private class B extends Thread {

        private A a;

        B(A a) {
            this.a = a;
        }

        @Override
        public void run() {
            try {
                a.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("B");
        }
    }

    public void test() {
        A a = new A();
        B b = new B(a);
        b.start();
        a.start();
    }
}
```

```
public static void main(String[] args) {
    JoinExample example = new JoinExample();
    example.test();
}
```

```
A
B
```

await() signal() signalAll()

- java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程。
- 相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活。
- 使用 Lock 来获取一个 Condition 对象。

```
public class AwaitSignalExample {

    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public void before() {
        lock.lock();
        try {
            System.out.println("before");
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public void after() {
        lock.lock();
        try {
            condition.await();
            System.out.println("after");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

```
public static void main(String[] args) {
    ExecutorService executorService = Executors.newCachedThreadPool();
    AwaitSignalExample example = new AwaitSignalExample();
    executorService.execute(() -> example.after());
    executorService.execute(() -> example.before());
}
```

```
before
after
```

wait() notify() notifyAll()

Jedes Objekt verfügt implizit über folgende Member:

Attribute `waitingThreads: ThreadQueue` -> Queue von Threads, die darauf warten, dieses Objekt zu verwenden

`final wait(): void` -> reiht den Thread, in dem der Aufruf geschieht, in die `waitingThreads-Queue` des Objektes ein, auf dem die Methode aufgerufen wird und setzt dessen Zustand auf `WAITING`

`final notify(): void` -> der erste Thread in der `waitingThreads-Queue` wird in den Zustand `BLOCKED` versetzt und in die blockedThreads-Queue eingefügt

`final notifyAll(): void` -> alle Threads in der `waitingThreads-Queue` wird in wird in den Zustand `BLOCKED` versetzt und in die blockedThreads-Queue eingefügt

- 调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程会调用 notify() 或者 notifyAll() 来唤醒挂起的线程。
- 只能用在同步方法或者同步控制块中使用，否则会在运行时抛出IllegalMonitorStateException。
- 使用 wait() 挂起期间，线程会释放锁。这是因为，如果没有释放锁，那么其它线程就无法进入对象的同步方法或者同步控制块中，那么就无法执行 notify() 或者 notifyAll() 来唤醒挂起的线程，造成死锁。

```
public class WaitNotifyExample {

    public synchronized void before() {
        System.out.println("before");
        notifyAll();
    }

    public synchronized void after() {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("after");
    }
}
```

```
public static void main(String[] args) {
    ExecutorService executorService = Executors.newCachedThreadPool();
    WaitNotifyExample example = new WaitNotifyExample();
    executorService.execute(() -> example.after());
    executorService.execute(() -> example.before());
}
```

```
before
after
```

Synchronized 锁机制 -> 可控制多个线程对共享资源的互斥访问 (由JVM实现)

- 当持有synchronized锁的线程长期不释放锁的时候，正在等待的线程不可以选择放弃等待，改为处理其他事情
- synchronized是非公平锁 -> 指多个线程在等待同一个锁时，不必须按照申请锁的时间顺序来依次获得锁

1. 同步一个代码块

```
public void func() {
    synchronized (this) {
        // ...
    }
}
```

它只作用于同一个对象，如果调用两个对象上的同步代码块，就不会进行同步。

对于以下代码，使用 ExecutorService 执行了两个线程，由于调用的是同一个对象的同步代码块，因此这两个线程会进行同步，当一个线程进入同步语句块时，另一个线程就必须等待。

```
public class SynchronizedExample {

    public void func1() {
        synchronized (this) {
            for (int i = 0; i < 10; i++) {
                System.out.print(i + " ");
            }
        }
    }
}
```

```
public static void main(String[] args) {
    SynchronizedExample e1 = new SynchronizedExample();
    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> e1.func1());
    executorService.execute(() -> e1.func1());
}
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

对于以下代码，两个线程调用了不同对象的同步代码块，因此这两个线程就不需要同步。从输出结果可以看出，两个线程交叉执行。

```
public static void main(String[] args) {
    SynchronizedExample e1 = new SynchronizedExample();
    SynchronizedExample e2 = new SynchronizedExample();
    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> e1.func1());
    executorService.execute(() -> e2.func1());
}
```

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

2. 同步一个方法

```
public synchronized void func () {
    // ...
}
```

它和同步代码块一样，作用于同一个对象。

3. 同步一个类

```
public void func() {
    synchronized (SynchronizedExample.class) {
        // ...
    }
}
```

作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步。

```
public class SynchronizedExample {

    public void func2() {
        synchronized (SynchronizedExample.class) {
            for (int i = 0; i < 10; i++) {
                System.out.print(i + " ");
            }
        }
    }
}
```

```
public static void main(String[] args) {
    SynchronizedExample e1 = new SynchronizedExample();
    SynchronizedExample e2 = new SynchronizedExample();
    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> e1.func2());
    executorService.execute(() -> e2.func2());
}
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

4. 同步一个静态方法

```
public synchronized static void fun() {
    // ...
}
```

作用于整个类。

Thread-Zustand 线程状态

一个线程只能处于一种状态，并且这里的线程状态特指 Java 虚拟机的线程状态，不能反映线程在特定操作系统下的状态

新建 (NEW): 创建后尚未启动

可运行 (RUNNABLE): 正在 Java 虚拟机中运行。但是在操作系统层面，它可能处于运行状态，也可能等待资源调度（例如处理器资源），资源调度完成就进入运行状态。所以该状态的可运行是指可以被运行，具体有没有运行要看底层操作系统的资源调度

阻塞 (BLOCKED): (被动) 请求获取 monitor lock 从而进入 synchronized 函数或者代码块，但是其它线程已经占用了该 monitor lock，所以出于阻塞状态。要结束该状态进入从而 RUNNABLE 需要其他线程释放 monitor lock

无限期等待 (WAITING): 等待其它线程显式地唤醒 ((主动) 通过调用 Object.wait() 等方法进入)

限期等待 (TIMED_WAITING): 无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒

- 调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“使一个线程睡眠”进行描述。调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“挂起一个线程”进行描述。睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态

死亡 (TERMINATED): 可以是线程结束任务之后自己结束，或者产生了异常而结束

进入方法	退出方法
没有设置 Timeout 参数的 Object.wait() 方法	Object.notify() / Object.notifyAll()
没有设置 Timeout 参数的 Thread.join() 方法	被调用的线程执行完毕
LockSupport.park() 方法	LockSupport.unpark(Thread)

进入方法	退出方法
Thread.sleep() 方法	时间结束
设置了 Timeout 参数的 Object.wait() 方法	时间结束 / Object.notify() / Object.notifyAll()
设置了 Timeout 参数的 Thread.join() 方法	时间结束 / 被调用的线程执行完毕
LockSupport.parkNanos() 方法	LockSupport.unpark(Thread)
LockSupport.parkUntil() 方法	LockSupport.unpark(Thread)

Producer-Consumer-Problem

Ein Semaphor verwaltet eie ganzzahlige Anzahl an Ressourcen - folgende beide Methode sind synchronized

`acquire(): void` -> gibt dem aufrufenden Thread eine der Ressourcen, falls noch vorhanden, sonst muss der Thread warten `wait()`

`release(): void` -> gibt eine der Ressourcen wieder frei, falls nach diesem Aufruf Ressourcen frei sind, wird der erste wartende Thread notifiziert `notify()`

Semaphore 类似于操作系统中的信号量，可以控制对互斥资源的访问线程数。

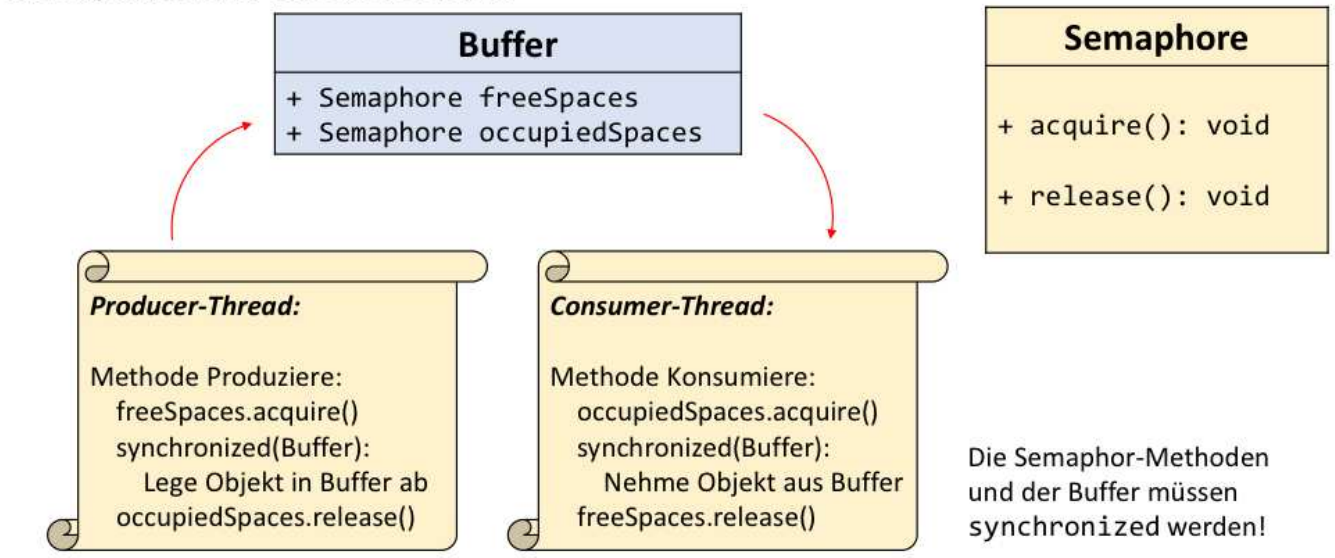
以下代码模拟了对某个服务的并发请求，每次只能有 3 个客户端同时访问，请求总数为 10。

```
public class SemaphoreExample {

    public static void main(String[] args) {
        final int clientCount = 3;
        final int totalRequestCount = 10;
        Semaphore semaphore = new Semaphore(clientCount);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < totalRequestCount; i++) {
            executorService.execute(()->{
                try {
                    semaphore.acquire();
                    System.out.print(semaphore.availablePermits() + " ");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release();
                }
            });
        }
        executorService.shutdown();
    }
}
```

```
2 1 2 2 2 2 2 1 2 2
```

Nutzung im Producer-Consumer-Problem:



BlockingQueue

java.util.concurrent.BlockingQueue 接口有以下阻塞队列的实现：

- **FIFO 队列**： LinkedBlockingQueue、ArrayBlockingQueue（固定长度）
- **优先级队列**： PriorityBlockingQueue

提供了阻塞的 take() 和 put() 方法：如果队列为空 take() 将阻塞，直到队列中有内容；如果队列为满 put() 将阻塞，直到队列有空闲位置。

使用 BlockingQueue 实现生产者消费者问题

```
public class ProducerConsumer {

    private static BlockingQueue<String> queue = new ArrayBlockingQueue<>(5);

    private static class Producer extends Thread {
        @Override
        public void run() {
            try {
                queue.put("product");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.print("produce..");
        }
    }

    private static class Consumer extends Thread {

        @Override
        public void run() {
            try {
                String product = queue.take();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.print("consume..");
        }
    }
}
```

```
public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
        Producer producer = new Producer();
        producer.start();
    }
    for (int i = 0; i < 5; i++) {
        Consumer consumer = new Consumer();
        consumer.start();
    }
    for (int i = 0; i < 3; i++) {
        Producer producer = new Producer();
        producer.start();
    }
}
```

```
produce..produce..consume..consume..produce..consume..produce..consume..produce..consume..
```

Reader-Writer-Szenario

```
public class Reader extends Thread {
    private final Shared share;
    public Reader(Shared s) {
        share = s;
    }
    public void run() {
        synchronized (share) {
            while (!share.changed) { // wird manchmal falsch geweckt
                try {
                    share.wait();
                } catch (InterruptedException e) {
                    return;
                }
            }
            System.out.println(share.value);
            share.changed = false;
            share.notifyAll();  // weckt auch andere Reader auf :/
        }
    }
}
```

```
public class Writer extends Thread {
    private final Shared share;
    private final int n;
    public Writer(Shared s, int n) {
        share = s;
        this.n = n;
    }
    public void run() {
        for (int i = 1; i <= n; i++) {
            synchronized (share) {
                while (share.changed) { // wird manchmal falsch geweckt
                    try {
                        share.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                share.value = i;
                share.changed = true;
                share.notifyAll();  // weckt auch andere Writer auf :/
            }
        }
    }
}
```