Einführung in die Programmierung:

Signatur einer Methode:

Public void int Methode(Int parametername){}
Die Signatur einer Methode besteht aus ihrem Namen und den Parametertypen in der vorgegebenen Reihenfolge.Die Signaturen von Methoden müssen unterschiedlich sein. Der Rückgabetyp spielt in der Signatur keine Rolle.

Unterschied: Klassenattribute und Attribute einer Klassendeklaration:
Attribute einer Klassendeklaration sind Instanzattribute. Jedes Attribut gehört zu einem Objekt. Klassenattribute sind statisch definiert. Daher enthält jedes erzeugte Objekt dieses Attribut.

```
private int y ; //Instanzattribut
private static int y ;  //Klassenattribut
```

Klassenmethoden sind wie Klassenattribute unabhängig von einem Objekt. Diese sollten auch als static gekennzeichnet werden. Sie haben keinen Zugriff auf Objektattribute.

Void ist ein Rückgabetyp einer Methode, der besagt, dass die Methode nichts zurückgibt.

Konstruktoren sind die am häufigsten überladenen Methoden. Diese Methoden haben den gleichen Namen, wie die Klasse und keinen Rückgabetyp.

```
public Klassenname(int parameter1 , int parameter2){
this.parameter1 = parameter1 ;
this.parameter2 = parameter2 ;
y++ ; // Auf Klassenattribute wird ohne this zugegriffen
}
```

this gibt eine Referenz auf die aktuell bearbeitete Instanz einer Methode.

Ein solcher Konstruktor erzeugt ein Objekt. Falls kein Konstruktor vom Programmierer persönlich erzeugt wurde, stellt Java automatisch einen Default Konstruktor zur Verfügung.

```
public Klassenname(){  // eigener Konstruktor
this.parameter1 = 0 ;
this.parameter2= null ;
}

public Klassenname(){} //Parameterloser Konstruktor
```

Der Default-Konstruktor dient lediglich ein Objekt der Klasse mit plausiblen Werten zu instanziieren
Die **null**-Referenz besagt, dass Parameter ohne einen Wert keinen willkürlichen Wert bekommen sollen. Daher bekommen sie den vordefinierten Wert null, der einen Wert als keinen Wert darstellt.

```
public Klassenname(typ name) {
this.parameter = name.parameter ;
}
```

Der Copy-Konstruktor übernimmt die Werte eines übergebenen Objekts vom selben Typ
Objekte anlegen:

```
Objekt objektname = new Objekt() ;
```

<Klassenname><Variablenname> = new <Konstruktor>();

,,**New**'' reserviert Speicher im Heap und gibt Referenz auf den Speicher zurück.

1.Referenzvariable anlegen
2. Speicher reservieren und Referenz in Variable speichern
3. Objekt mit Attributen initialisieren

Call by Value – Call by Reference
Werte werden für eigene Variablen kopiert. Änderungen der Methode haben keine Auswirkungen auf das Attribut außerhalb der Methode. Änderungen müssen mit return übergeben werden. – Call by Value (Der Speicher von atomaren Datentypen wird auf dem Programmstack reserviert)

Es wird nur eine Referenz an das Objekt übergeben, keinen Wert. Änderungen an Objektattributen sind überall sichtbar. Objekte sind Referenzdatentypen und enthalten meist mehrere Attribute.– Call by Reference (Der Speicher für Referenzdatentypen wird im Heap reserviert)

Verstecken von Information
,,public'' = ausserhalb der Klasse sichtbar
,,protected'' = Nur in der Klasse und abgeleiteter Subklassen sichtbar
,,private'' = Nur innerhalb der Klasse sichtbar

Bei versteckten Attributen(Information hiding) ist nur Zugriff mit speziellen getter-/setter Methoden möglich. Vorteilhaft daran ist:

- bessere Wartbarkeit
- kontrollierter Zugriff
- Sicherheit
- Modularisierung

Abstrakte Klassen:
Eine Klasse ist abstrakt, wenn sie mit ,,abstract'' qualifiziert wurde. Sie enthält mindestens eine abstrakte Methode und kann nicht instanziiert werden. Abstrakte Klassen versorgen abgeleitete Klassen mit gemeinsamen Methoden und Attributen.

Der Konstruktor einer abstrakten Klasse bzw. einer Superklasse wird in einer Subklasse durch das Schlüsselwort **super** aufgerufen, der die Parameter des Konstruktors der Superklasse enthält. Zudem können mit ,,super'' Methoden der Superklasse in die Subklasse überschrieben werden.

Interfaces:
Vertrag von über eine Klasse zu implementierende Methoden. Das heisst, dass die Klasse, die das Interface implementiert alle Methoden überschreiben muss (@overwrite). Interfaces sind auch abstrakt, aber nicht mit dem Schlüsselwort abstract qualifizierbar. Interfaces weichen das Prinzip der Mehrfachvererbung auf, da eine Klasse mehrere Interfaces implementieren kann. Es können keine Objekte von Interfaces selbst erstellt werden, genau wie bei abstrakten Klassen.

Vererbung:
Java unterstützt keine Mehrfachvererbung. Das heisst eine Subklasse kann nur von einer Superklasse erben und nicht mehreren. Dafür kann eine Superklasse aber an mehrere Subklassen vererben.

Casten und Polymorphie:

Manche Methoden arbeiten mit Superklassen, wodurch aber auch Subklassen verwendet werden können. Es ist möglich den Typ eines Objektes zu prüfen und diesen dann in den Objekttyp zu ,,casten'' . Ein Cast erfolgt immer in runden Klammern nach dem = Operator. Der Cast sollte mit instanceof überprüft werden. Polymorphismus bedeutet, dass eine Instanz einer Subklasse ein Attribut einer Superklasse referenziert.

Schleifen:
Eine Schleife ist eine Möglichkeit das Programm zu steuern, indem ein Ablauf mehrmals iteriert (wiederholt) wird. In Schleifen können weitere Schleifen , aber auch If-else-Strukturen vorhanden sein.
Eine Schleife besteht aus drei Schritten:

- Setup: Startwert festlegen
- Anpassung:
- Abbruchbedingung

Bei der while-Schleife wird der Code in der Schleife in jedem Schritt ausgeführt und (meistens) nach der Abfrage wird die Bedingung geändert.

```
int i = 0 ;
while ( i < Array.length ){
i++ ;
//weitere Anweisungen ;
}
```

Bei der For-Schleife wird der Code in jedem Schritt in der Schleife ausgeführt und i um eins erhöht.

```
for ( int i = 0 ; i < Array.length ; i++ ){
//Anweisungen ;
}
```

Bei der Do-While Schleife wird der Code mindestens einmal ausgeführt, da die Überprüfung erst am Ende stattfindet.

```
boolean bed = true ;
Do { //Setup
//Anweisung ;
} while(test) ; //Abbruchbedingung
```

„break“ unterbricht die aktuelle Schleife komplett.

Felder (Arrays) :

Felder können mehrere Werte vom gleichen Typ verwalten. Ein Feld ist eine Speicherstelle von mehreren Variablen. Der Index eines Feldes beginnt bei Null.

```
int[] array = new int[5] ;
```

Da ein Feld mit dem Index 0 beginnt, geht dieser hier bis Index 4 bei einer Länge von 5. Deklaration vor dem = Zeichen, danach Initialisierung. Array hat danach feste Größe und kann nicht mehr verändert werden. Kleine Felder können wie folgt initialisiert werden:

```
int[] array = new int[3] {3 , 6 , 9 } ;
int[0] = 3 ;
int[1] = 6 ;
int[2] = 9 ;
```

Zugriff auf die Länge des Feldes mit array.length. Arrays können mehrere Dimensionen besitzen. Diese müssen nicht zwingend wie eine N x N – Matrix aussehen:

```
int[][] doppelArray = new int[3][] ;
```

Darauf könnte man wie folgt zugreifen:

```
for(int i = 0 ; i < doppelArray.length ; i++ ) {
        doppelArray[i] = new int[i ++] ;
        for (int j = 0 ; j < doppelArray[i].length ; j++) {
                doppelArray[i][j] = i * j ;
                //weitere Anweisungen ;
        }
}
```

Mit der for-each Schleife ist es möglich auf Kopien der Elemente zu arbeiten. Man kann aber nicht auf die Feldelemente zugreifen.

Listen:

Listen dienen der Verwaltung einer Menge von Objekten. Eine dieser Listen ist ArrayList<T> . Die Besonderheit an der Liste ist , dass sie bereits vordefinierte Methoden hat und bei Bedarf automatisch erweitert wird. Primitive Datentypen werden bei ArrayList nicht unterstützt.
Das T in dem Diamantoperator steht, als Platzhalter, für einen noch nicht definierten Typ. Das heisst sie ist generisch und damit typsicher. mit isEmpty() wird geprüft, ob die Liste leer ist

Konstanten:

Konstanten sind feste Werte, die nicht mehr verändert werden können. Sie werden mit dem Schlüsselwort final qualifiziert und in GROßBUCHSTABEN geschrieben. Enums sind Aufzählungen, in denen Werte vergeben werden, welche auch konstant sind und üblicherweise GROß geschrieben werden. Enums werden in einer eigenen Klasse angelegt, aber ohne final qualifiziert. Zudem sind Enums typsicher.

```
enum Automarken{
VW, BMW , AUDI, PORSCHE
}
Automarken marke1 = Automarken.VW ;
Automarken marke2 = Automarken.AUDI ;
if ( marke2 = Automarken.AUDI) {
//Anweisung
}
```

Gleichheit:

Zahlenwerte lassen sich mit == vergleichen. Dabei wird jedoch nur die Speicheradresse verglichen, nicht der Wert. Daher sollte bei Objekten mit unterschiedlichen Referenzen die equals-Methode implementiert werden.

```
boolean equals(Object o) {  //Objekt, mit dem verglichen werden soll
if (this == o ) {  //Bei gleicher Referenz handelt es sich um dasselbe Objekt
        return true ;
        }
if (this.getClass = o.getClass){ //Sind die Klassen der Objekte unterschiedlich, können sie nicht gleich sein
        return false ;
        }
}
```

Man kann auch mit instanceof vergleichen.

Rekursion:

Rekursion besteht aus dem Teile und Herrsche Prinzip. Rekursion bedeutet das Zerkleinern der Probleme bis zum kleinsten, indem sich Methoden immer wieder selbst aufrufen. Wichtig ist dabei eine Abbruchbedingung einzuführen, da ansonsten die Methoden sich immer weiter selbst aufrufen. Rekursion findet oft Anwendung bei Such- und Sortieralgorithmen und Backtracking ;

Code:

Fakultät rekursiv:

```
public int fakultaet(int n){
        if ( n == 0 ) return 1 ;
        else return n * fakultaet(n-1);
}
```

Fakultät iterativ:

```
public int fakultaet(int n){
        int result = 1 ;
        for(int i = 1 ; i <= n ; i++){
                result *= i ;
        }
        return result ;
}
```

Methode, um Reihenfolge umzukehren:

```
String[] s = { “eins“,“zwei“,“drei“} ;
reverse(s);
for ( x : s ){
        System.out.println(x + ““) ;
        }
}

void reverse(String[] z) {
        int start = 0 ;
        int end = z.length – 1  ;
        while ( start < end){
                Sring temp = z[start] ;
                z[start] = z[end] ;
                z[end] = temp ;
                start++ ;
                end-- ;
        }
}
```

Arten der Rekursion:

- linear (Ein Aufruf führt zum nächsten)
- nicht-linear (Ein Aufruf führt zu mehreren)
- direkt (Eine Methode verweist direkt wieder auf sich)
- indirekt (Selbstaufruf erfolgt über Zwischenschritte)

Fakultätsberechnung → direkt linear

Such- und Sortieralgorithmen

Lineare Suche:

Elemente werden schrittweise gesucht.

Komplexität:
O(1) = Im besten Fall ist das gesuchte Element das erste
O(N) = Im schlechtesten Fall das Letzte
O(N/2) = Durchschnitt

```
public int linearSearch(int[] elements, int key){
        for (int i = 0 ; i < elements.length ; i++ ) {
                if (elements[i] == key ) return i ;
        }
        return -1 ;
}
```

Binäre Suche:

Vorraussetzung: Elemente sind bereits sortiert
Idee : Teile und Herrsche

- Kontrolle des mittleren Elements
  - falls Element nicht in der Mitte liegt, entscheiden, ob links oder rechts weitergesucht wird
    - Jeder Schritt halbiert zu durchsuchende Liste

Komplexität: O(log (N)) → meist besser als lineare Suche

```
public int binarySearch(int[] elements, int key){
        int left = 0 ;
        int right = elements.length – 1 ;
        while ( left < right ) {
                int middle = ( left + right ) / 2 ;
            if( elements[middle] == key ) return middle;
            if (elements[middle] < key ) return middle +1;
            else right = middle – 1 ;
        }
        return -1 ;
}
```

Sortierenalgorithmen:

- Insertion-Sort
- Bubble-Sort
- Quick-Sort
- Merge-Sort
- Bucket-Sort

Insertion-Sort

An jedem Durchlauf wird ein Element an die richtige Stelle der bereits sortierten Daten eingefügt

Vorteile	Nachteile
Kein zusätzlicher Speicher Stabil	Komplexität $O(n^2)$ Verschiebeoperatoren notwendig

Bubble-Sort

Iteriere durch die Liste und sortiere direkt nebeneinander Elemente. In jedem Durchlauf wird das größte Elemente nach hinten geschoben.

Vorteile	Nachteile
Stabil	$O(n^2)$

Quick-Sort

1. Wähle ein zufälliges pivot Element
2. Verschiebe alle Elemente so, dass alle Elemente <= links oder > rechts von einer Position stehen
3. Füge das pivot Element an der Position ein (jetzt steht dieses richtig)
4. Wiederhole die Schritte auf die Teillisten links und rechts

Vorteile	Nachteile
best case O(N log N) kein zusätzlicher Speicher	worst case $O(n^2)$

Merge-Sort

Bei Merge-Sort wird das Feld in möglichst gleich große Teile gesplittet, bis jeder Teil nur noch ein Element enthält. Im Anschluss wird alles wieder zusammengefügt (Es wird das jeweils kleinste Element angefügt, wobei die Elemente dabei sortiert werden.

Vorteile	Nachteile
O(N log N)	

Bucket-Sort

Kann nur eingesetzt werden, wenn eine geringe Menge an Elementen sortiert werden soll. Hierbei werden die Elemente jeweils an das Feld ihres Bucketwertes angefügt. Danach werden die Bucketfelder der Reihe nach durchlaufen. Aus jedem der Bucketfelder werden die enthaltenen Elemente der Reihe nach in das ursprüngliche Eingabefeld zurück kopiert.

Vorteile	Nachteile
sehr schnell O(N)	geringe Menge Elemente

Alle Komplexitätsbereiche:

- O(1) = konstant
- O(log N) = logarithmisch
- O(N) = linear
- O(N log N) = quasi linear
- $O(N^2)$ = quadratisch
- O(n^k) = polynomial
- O(2^N) = exponential

Greedy-Algorithmen

Greedy-Algorithmen werden dazu verwendet eine optimale Konfiguration zu finden, bei denen ein Minimal- oder Maximalwert nicht überschritten werden darf. Dafür müssen aber bereits 2 Bedingungen erfüllt sein. Greedy-Algorithmen arbeiten rekursiv und sind daher meist von exponentieller Komplexität.

Erklärung dynamische Programmierung

Das grundlegende Prinzip der dynamischen Programmierung besagt, dass Zwischenergebnisse gespeichert werden sollen, damit sie nicht später erneut berechnet werden müssen.

Erklärung Backtracking

Backtracking ist ein wichtiges Algorithmenmuster für Such- und Optimierungsprobleme.
Vorraussetzungen für Backtracking:

- Menge von K Konfigurationen
- Es gibt eine Anfangskonfiguration Ko
- Für jede Konfiguration Ki kann die Menge der direkten Erweiterungen bestimmt werden
- Für jede Konfiguration ist entscheidbar, ob es sich um eine Lösung handelt oder nicht.

Der Algorithmus sucht die beste Lösung aus einer Menge berechneter Lösungen aus oder der Algorithmus findet nicht die beste Lösung und bricht nach der ersten Lösung ab.

Stack/-Queue/-Deque-Speicher

Ein Stapel bzw. Kellspeicher(Stack) kann eine beliebige Menge an Elementen speichern. Der Stack arbeitet mit dem LIFO Prinzip . Last in- first out. Man packt ein Element oben an den Stapel und dieses ist das, was als erstes rausgenommen wird.

- Stack.push() //Fügt Elemente an den Stapel an
- Stack.pop() //Nimmt Element weg

Atomare Datentypen und Referenzen werden im Stack gespeichert – Objekte im Heap

Die Queue (Warteschlange) kann ,wie der Stack, eine beliebige Menge an Elementen speichern, nur dass das Prinzip FIFO ist. First in- first out. Das heisst es kann hinten ein Element hinzugefügt werden und vorne wird es wieder weggenommen.

- Queue.offer() // Fügt Element an die Schlange an
- Queue.poll() //Entfernt Element

Die ArrayDeque (Double-Ended-Queue) hat die Besonderheit, dass man Elemente vorne und hinten hinzufügen und entfernen kann.

Wrapper Klassen (Boxing/Unboxing)

Parameter für generische Klassen können nur Klassen oder Interfaces sein. Da es vermieden werden soll für jeden atomaren Datentyp eine neue Klasse zu implementieren , gibt es Wrapper-klassen. Dabei werden Umwandlungen von und in Wrapper-Klassen durchgeführt. Das nennt man Autoboxing und Unboxing. (int = atomar, Integer = Wrapper.) Beim boxing wird das Argument in eine Instanz der Wrapper-Klasse umgewandelt und beim unboxing wird der int-Wert aus der Referenz auf Integer extrahiert

Die Klasse System / Zeitmessungen

Die Klasse System bietet ein paar wichtige Methoden zur Kommunikation mit der Konsole.

- System.out. → stellt Ausgabeverbindung her
- System.in → stellt Eingabeverbindung her
- System.err → dient der Ausgabe von Fehlermeldungen auf die Konsole
- System.currentTimeMillis() → liefert die aktuelle Zeit gemessen in Millisekunden // Nur für Uhrzeitangaben, für Laufzeitmessungen zu ungenau
- System.nanoTime() → liefert hochauflösende Zeit der JVM. Dient für Laufzeitmessungen. Keine absolute Zeit

Äquivalenzrelation und LSP

Dasselbe Vs. das Gleiche

- Bei primitiven Datentypen wird nicht darunter unterschieden
- Bei Referenzdatentypen schon

Bei Vergleichen der Attributwerte der referenzierten Speicher ist die Methode boolean equals(Object o) zu benutzen.

Aus mathematischer Sicht ist Gleichheit eine Äquivalenzrelation.

- Reflexivität // alle Elemente (a = a) → a.equals(a) //Ist immer wahr
- Transitivität // wenn a = b, dann b = a → a.equals(b) hat immer denselben Wahrheitswert wie b.equals(a)
- Symmetrie // wenn a = b und b = c, dann a = c → falls a.equals(b) und b.equals(c) wahr sind, dann ist auch c.equals(a) wahr

Die Eigenschaft von equals eine Äquivalenzrelation zu implementieren ist unvereinbar mit dem Liskov'schen Substitutionsprinzip.

Wird equals mit getClass implementiert, dann wird die für Gleichheit geforderte Äquivalenzrelation erfüllt. → LSP nicht

Wird equals mit instanceof implementiert, dann wird das Liskov'sche Substitutionsprinzip erfüllt. → Transitivität und Symmetrie der Äquivalzenzrelation nicht.

Heuristik

Eine Heuristik versucht bei einem Problem , über das es wenige Kenntnisse gibt, das wahrscheinlichste Ergebnis zu generieren.

→ nahezu Gegenteil ist ein Algorithmus

Operatoren:

Zuweisungsoperatoren

x += k entspricht x = x + k
x -= k entspricht x = x – k
x*= k entspricht x = x * k
x /= k entspricht x = x / k
x++ entspricht x+1
x-- entspricht x-1
--x entspricht
++x entspricht

Logische Operatoren

a = x > y größer als
a = x >= y größer/gleich als
a = x < y kleiner als
a = x <= y kleiner/gleich als
a = x ==y gleich
a = x != y ungleich

Logische Verknüpfungen

a = b && c logisches und
a = b || c logisches oder
a = ! b logische Negation

Implementation von Algorithmen und Codemustern:

Binäre Suche (Nicht rekursiv)

```
public int binarySearch( Comparable[] feld , Comparable o){
        int left = 0 ;
        int right = feld.length – 1 ;
        while ( left <= right ) {
                int middle = ( left + right) / 2 ;
                int result = feld[middle].compareTo(o) ;
                if(result == 0) return middle;
                if(result > 0 ) {
                        right= middle -1 ;
                        }
                        else {
                        left = middle + 1 ;
                        }
                }
                return – 1 ;
        }
```

Insertion- Sort

```
public void insertionSort(Comparable[] feld) {
        for ( int i = 0 ; i < feld.length ; i++ ) {
                Comparable wert = feld[i] ;
                for ( int posi = i ; i > 0 ; i-- ) {
                if (feld[posi -1].compareTo(wert)<= 0 ){
                        feld[posi] = wert ;
                        break ;
                }
                feld[posi] = feld[posi – 1 ] ;
        }
        }
}
```

Pascal'sches Dreieck (Speicher reservieren)

```
int[][] reserviereSpeicher(int n){
        int[][] dreieck ;
        dreieck = new int[n + 1][] ;
        for(int i = 0 ; i < n ; i++) {
                dreieck[i] = new int[i + 1] ;
        }
        return dreieck ;
}
```

Bubble-Sort

```
void bubbleSort(int[] array){
        boolean swapped ;
        int n = array.length ;
        do {
        swapped = false ;
        n-- ;
        for (int i = 0 ; i < n ; i++ ){
                if ( array[i] > array[i + 1] ){
                int temp = array[i] ;
                array[i + 1] = temp ;
                swapped = true ;
                }
        }
        }
        while (swapped) ;
}
```

Quick-Sort

```
private static int split(double[] a , int left , int right) {
        int l = left ;
        int r = right ;
        int pivot = a[(l + r) / 2] ;
        while ( l <= r ) {
                while (a[l] < pivot ) l++ ;
                while (a[r] > pivot ) r-- ;
                if ( l <= r ) {
                        double temp = a[l] ;
                        a[l++] = a[r] ;
                        a[r--] = temp ;
                }
        }
        return l ;
}
```