Begriffsklärung
Ein Algorithmus ist ein allgemeines, eindeutiges, endliches Verfahren zur Lösung einer Klasse gleichartiger Problem.
Voraussetzung: Basisfähigkeiten

Beispiel Basisfähigkeiten sind z.B.: Addieren von Dezimalziffern, Merken von Zwischenergebnissen, Befehle hintereinander ausführen

Frage: Gibt es zu jeder Problemklasse immer einen Algorithmus zur Lösung? -> Antwort: NEIN!!

Beispiel: Gegeben sei eine Menge M von Zahlen $z \in \mathbb{R}$. Berechne das Maximum von M. Sei $M = \mathbb{N}$. Annahme a ist Lösung. $a + 1 \in \mathbb{N}$ und $a < a + 1$

Gegeben sei endliche , erfüllbare Menge mit einer Ordnungsrelation $\leq$.
Es gibt ein Algorithmus zur Bestimmung des Maximums von M
Beweis: M erfüllbare es gibt eine berechenbare Funktion mit $M = \{p(x)\}, p(x)$ bei |M| = n
Algorithmus:

1. Setze den Platz i auf 1 (i Zähler für p)
2. Berechne $p(1)$ und setze den neuen Merkplatz v auf $p(1)$
3. Wenn $i = n$ gilt, dann ist das Ergebnis in v. STOP.
4. Erhöre i um 1 und berechne $p(i)$
5. Wenn $v < p(i)$ dann setze v auf den Wert $p(i)$
6. Fahre fort mit Schritt 3

Funktional: M nichtleere, endlich Menge und $\leq$

```
maxi :: Ord t => [t] -> t
maxi [x] = a
maxi (a:l) = if a > maxi l then a else maxi l
maxi [] = error"Maximum der leeren Liste nicht definiert"
```

1 Grundlagen der Berechenbarkeit

- Kern jeder imperativen (objektorientierten) Programmiersprache: $WHILE$
- Minimalistisches Maschinenmodell: URM - Universelle Registermaschine

Einschub: bedeutet Befehlsorientiert!

1.1 Die Sprache $WHILE$

Annahme: Eine unendliche abzählbare Menge von Variablen,zB. kleine lateinische Buchstaben mit oder ohne Indizes aus $\mathbb{N}$.

Syntax	Semantik
x := 0;	Die Variable x erhält den Wert 0
x := x + 1;	Der Wert der Variable wird um eins erhöht
x := x - 1;	Falls der Wert von x $\neq$ 0 ist, wird der Wert der Variable wird um eins erniedrigt
while x $\neq$ y do <anweisung> end;	Falls der Wert von x ungleich dem Wert von y ist, werden <anweisungen> ausgeführt und anschließend die ganze Schleife nochmal

Definition 1 ($WHILE$-programmierbare Funktionen). *Eine Funktion $f : \mathbb{N}^n \to \mathbb{N}$ hießt $WHILE$ programmierbar, genau dann wenn $WHILE$ Programm P_f gibt, so dass für alle $(w_1, \ldots, w_n) \in \mathbb{N}^n$ gibt $f(w_1, \ldots, w_n) = w$ genau dann wenn die Ausführung von P_f auf dem Zustand a_1 mit Wert w_1 terminiert mit dem Wert w in der Variable a_0.*

Gemische Befehlsformen In $WHILE$
h ist Hilfsvariable

x := y:	x := 0; while x $\neq$ y do x := x + 1; end
x := y:	x := 0; x := x +1; ... ; x := x + 1; (mit $n \in \mathbb{N}$)
x := x + y:	h = 0; while h $\neq$ y do x := x + 1; h := h + 1; end;
usw. z.B. weitere arithmetische Operationen; wahr $\triangleq$ 1 und falsch $\triangleq$ 0	
x $\wedge$ y:	x * y
x $\vee$ y:	x + y - x * y
$\neq$ x:	1 - x
b = x < y:	x < y genau dann wenn y - x $\neq$ 0 (Idee)
	b := 0; g := y - x; while h $\neq$ 0 do b := 1; h := 0; end;
if <expr> then <anweisung> end;	h = <expr> while h $\neq$ 0 then <anweisung>; h := 0; end;

1.2 URM

Die URM (Universelle Registermachiene) besteht aus einem Datenspeicher mit beliebig vielen Registern, die natürliche Zahlen enthalten (n_0, n_1 ...) und einem Befehlsspeicher mit beliebig vielen Registern für Befehle (b_0, b_1 ...). Dazu kommt noch der Programmzähler, der die Befehle zuordnet.

Befehlssatz der URM

Syntax	Semantik
zero i;	r_i wird auf 0 gesetzt
succ i;	r_i wird um 1 erhöht
pred i;	r_i wird um 1 erniedrigt
je x y n	Wenn Inhalt von $r_x = r_y$ dann setzte pc auf n
goto n	Setze pc auf n

1 Der imperative Kern von Java

1.1 Wiederholung

- $WHILE$ ist Kern jeder Sprache. Die Syntax haben wir uns angeguckt. Die Semantik ist die Transformation der Zustandsbla. Als Zustand wird die Zuordnung von Variable zu Wert bezeichnet.

- Registermaschiene

1.2 Übersetzung $WHILE$-Programm in Registermaschienen-Programm

- Anlegen einer Symboltabelle. Es gibt zu jedem $WHILE$-Programm eine andere Tabelle st, in der jede Variable eindeutig auf einen Registerplatz abgebildet wird..

WHILE	Registermaschienenbefehl
x = 0	zero st(x)
x := x+1	succ st(x)
x := x-1	pred st(x)
while x $\neq$ y do <Anweisung> end;	je st(x) st(y), k <Anweisung> goto m

Beim Laden des Programms wird m auf die Befehlsnummer des Beginns der Schleife gesetzt und k auf den ersten Befehl nach der Schleife.
Symboltabelle:

a	0
b	1
c	2
h	3

Beispiel (Berechnen von Audrücken)
Berechne $f : \mathbb{N} \to \mathbb{N}$ mit $f(a, b) = a + b - 3$

1. Teil: c := a;		Registermaschienen-Programm	Java

c := 0;
while c != a do
c := c+1;
end;
	0
	1
	2
	3
		zero 2
		je 2 0 4
		succ 2
		goto 1
			c = 0;
			while (c != 0) {
			c = c+1;
			}

2. Teil: c := c + b;			

h := 0;
while h != b do
h := h + 1;
c := c+1;
end;
	4
	5
	6
	7
	8
		zero 4
		je 3 1 9
		succ 3
		succ 2
		goto 5
			h = 0;
			while (h != b) {
			h = h + 1;
			c = c+1;
			}

3. Teil c := c - 3			
c := c - 1; c := c - 1; c := c - 1;	9 10 11	pred 2 pred 2 pred 2	c = c - 1; c = c - 1; c = c - 1;

Ein Java-Programm besteht aus einer Sammlung von *Klassen*. Eine Klasse ist eine Zusammenfassung von *Attributen*, das können Variablen und Methoden (??) unter einem Namen sein. Wenigsten eine dieser Klassen muss eine Methode *main* als Hauptprogramm (bla) enthalten.

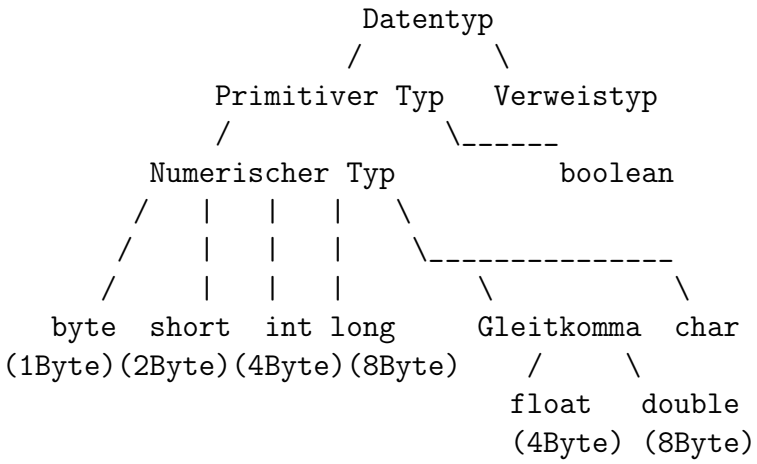
Und jetzt richtig in Java:

```
public class Ausdruck // Berechnung von a + b -3
{
  public static void main (String [] args)
  {
    int a,b,c,h;
    a = Integer.parseInt(args [0]); // Aufruf Ausdruck 10
        20
    b = Integer.parseInt(args [1)];
    c = 0;
    ... // siehe Tabelle
    c = c -1;
    System.out.println("Der␣Wert␣des␣Ausdrucks␣a␣+␣b␣+␣3␣
       ist␣" + c);
  }
}
```

1.3 Java Programm ausführen

- Schreibe Ausdruck in eine Datei mit dem Namen *Ausdruck.java*
- *javac Ausdurck.java*. Dies ruft in der Konsole den Übersetzer auf. Dieser kompiliert java in eine Java Bytecode. Das Ergebnis wird in *Ausdruck.class* gespeichert.
- in der Konsole wird über *java Ausdruck 10 20* dann das Programm ausgeführt.

1 Datentypen von Java



1.1 Operationen auf int, long, float, double und boolean

Die Operationen sind nach absteigender Bindungsstärke sortiert:

- Monadische Operatoren: $-$, $+$, $++$ (Erhöht den Wert um 1), $--$ (Vermindert den Wert um 1), ! (Negation), $typecast$ (Typumwandler)
- Binäre Operatoren: $*$, $/$, $\%$ (Modulo), $+$, $-$
- Vergleichsoperatoren: $<$, $>$, $<=$, $>=$
- Test auf (Un)Gleichheit: $==$, $!=$.
- logische Operatoren: $\&\&$ (und), $||$ (oder)

1.2 Wie werden Ausdrücke gebildet?

- Atomare Ausdrücke
  - Variablen (nach Deklaration mit Typangabe)
  - Konstanten (Literals)
- Zusammengesetzte Ausdrücke
  - $-$<Ausdruck>, $+$<Ausdruck> ...
  - (<Ausdruck>)
  - <Ausdruck>$*$<Ausdruck>, <Ausdruck>/<Ausdruck>

1.2.1 Konstanten

`c = #Zahl` Deklaration innerhalb des Wertebereichs

```
byte b = 3;
Integer.MAX_VALUE //Konstante, deren Wert die
    größtmögliche von int angibt
Integer.MIN_VALUE //Äquivalent zu MAX_VALUE
```

Gleitkommazahlen werden mit einem (.) statt dem (,) angegeben.

```
double d = .00001; \\ist äquivalent zu
d = 1 E -5;
```

1.3 Typumwandlung

Auf der Basis der Zahlypen ist eine Ordnung erklärt.
byte => short => int => long => float => double

Die explizite Umwandlung mit monadischen Operator für jeden Typ t ist definiert.

Beispiel (int) 3.14 (natürlich mit Präzisionsverlust.)
4 + 3.14 funktioniert implizit, es wird in float umgewandelt.

```
byte b = 4;
b = b +1; // ist nicht zulässig, da + erst ab int
    definiert ist
b = (byte) (b+1)
```

Unter dem Aspekt der iterativen Programmierung werden die Begriffe Algorithmus und Programmierung neu betrachtet: Ein Algorithmus ist ein Verfahren zur Lösung eines Problems, und zwar nach dem Schema `Eingabe -> Algorithmus -> Ausgabe`.
Dazu kommen folgende Fragen auf:

- Ist die Ausgabe ok?
  - Liefert die Ausgabe das richtige Ergebnis? "Korrektheit" ist hier das Stichwort.
  - Bekommen wir überhaupt eine Ausgabe, d.h. terminiert der Algorithmus?

  Mit dem "Hoare-Kalkül" bekommen wir eine Antwort
- Wann kommt die Ausgabe?
  Komplexität des Algorithmus, dazu wird die "O-Notation" verwendet.

Ein Programm ist die Realisierung eines Algorithmus. Auch dazu stellen sich Fragen:

- Was ist ein funktionales Programm?
  Die ist eine Menge von Funktionen, die sich gegenseitig aufrufen.
- Was ist ein iteratives Programm?
  Man bezeichnet so eine Folge von Anweisungen.

Demzufolge ist dann die Programmierung die Abarbeitung der Anweisungsfolge. Daraus folge eine Änderung der Zustände, der "Variablen", d.h. diese können ersetzt werden und zugewiesen werden.

Es gibt ein Typsystem, durch das die Werte der Variablen eindeutig definiert werden.
Es gibt:

- primitive Typen: Der Wert ist unteilbar.
- zusammengesetzte Typen: Der Wert ist aus mehreren einzeln manipulierbaren Teilwerten zusammengesetzt, die wiederum primitiv oder zusammengesetzt sein können.

Wir nehmen uns den Typ AB. Die Struktur von A wird als "Datenstruktur" bezeichnet, es ist auch möglich, rekursive Datenstrukturen aufzubauen.
Es stellen sich wieder Fragen:

- Wie kann ich meine Daten entwerfen?
- Wie kann ich die Daten bearbeiten?

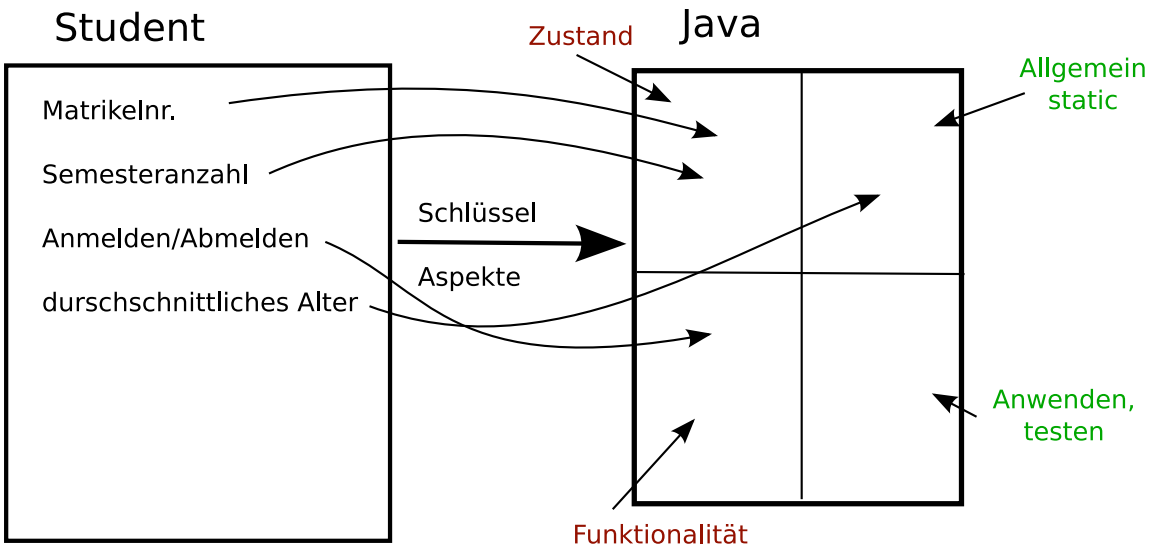
- Was ist die Quelle der Daten? -> OO

Um ein Problem umzusetzen, muss es auf ein z.B. mathematisches Modell abstrahiert werden. Daraus wird dann versucht eine Datenstruktur zu erstellen und diese in einen Algorithmus zu verpacken. Der Algorithmus wird in ein Programm eingebunden, welches die Datenstruktur bearbeitet und eine Ausgabe, z.B. als Datenstruktur, liefert..
Beispiel:
`Problem: Matrikelnumer -> math. Modell: Menge -> Datenstruktur: Array, Vektor`

Definition 1. *Ein Objekt ist die Modellierung / Abstraktion einer physisch oder gedanklich existierenden Entität durch Zustand (Eigenschaften, Attribute) und Verhalten (Funktionalität, Objektmethoden)*

Die Funktionalität von Objekten stellt einen Mechanismus zur Verfügung um den Zustand reguliert lesen bzw. manipulieren zu können.
Wie sieht das in Java aus? Dazu ein konkretes Beispiel ...

1 Modellierung eines Studenten



Bereiche eines Java Programms -> Klasse, Methode, Schleife, Kommentare, Variable, Ausgabe, Objekte ...

1.1 Klasse

Eine Klasse kann aus den vier Teilen (siehe Grafik) bestehen.

```
public class Student
{
  public static int durchschnittsAlter;
  public int matrNr;
  public int semester;
  public boolean anmelden(Vorlesung v)
  {
    ...
  }
```

```
  public Student(int a, int b)
  {
    this.matNr = a;
    this.semester = b;
  }
  public static void main(String[] args)
  {
    Student otto = new Student(1101,1);
    System.out.println("otto ist Student");
  }
}
```

1.1.1 Konventionen einer Klasse

- Befindet sich in einer .java-Datei eine öffentliche Klasse public, dann muss der Dateiname und der Klassenname übereinstimmen.
- Eine Java-Detei kann mehrere Klassendeklarationen enthalten, wobei keine davon dann $public$ ist, d.h. alle müssen $private$ sein. Am besten ist sowieso immer nur eine Klasse in einer Datei ...

1.2 Methode

- Eine Methode ist eine Sammlung von Anweisungen mit einem gemeinsamen Ziel.
- Jede Methode hat einen Name, mit dem sie aufgerufen wird.

Beispiel

```
public static int sum(int a, int b)
{
  return a+b;
}
```

$public$ ist dabei für die Sichtbarkeit zuständig, $static$ gibt an, ob es sich um eine objektgebundene Methode hatndelt. int gibt den Rückgabetyp an, das verlangt eine $return$-Anweisung am Ende der Methode; wenn dort $void$ steht, bedeutet das, dass die Methode keine Rückgabe hat. sum ist der Name. Die Paramter in den Klammern bezeichnet man als Signatur.

```
public boolesn anmelden (Vorlesung v)
{
  wenn v, dann return true;
```

```
  sonst return false;
}

public void anmeldung (Vorlesung v)
{
  wenn ok, dann System.out.println(...);
}
```

1 Testen und Anwenden

Eine Methode, die besonders ist, ist die $main()$-Methode.

Jede Applikation besteht aus mindestens einer Klasse, die die $main$-Methode enthält. Diese Klasse wird auch $Mainklasse$ genannt.
Applikationen sind Java-Programme, die vom Interpreter ausgeführt werden.
Beim Ausführen sucht das Javasysem nach der $main()$-Methode und führt die in ihr stehenden Anweisungen aus.

Ein paar Konventionen zu dieser Methode

- Die Mainklasse muss als $public$ deklariert sein. Einige Compiler arbeiten dennoch mit einer $main$-Methode, die nicht $public$ ist. Das bedeutet trotzdem, dass als Regel $public$ angenommen wird.
- Die $main$-Methode ist $static$, d.h. sie kann aufgerufen und benutzt werden, ohne dass eine Instanz bzw. ein Objekt der Klasse erzeugt wird.
- Die Methode ist $void$, d.h. sie liefert keinen Wert zurück
- Die Methode erwartet immer ein Stringarray `String[] args`

Jetzt führen wir die Studentenanwendung fort

```
class Student
{
 public static int studentenzahl; // ist nicht
    objektgebunden, beschreibt also nicht den Zustand
    eines Studentenobjektes
 public int matNr;
 public int semester;
 public Student(int a, int b) // Solche Methoden werden
    als Konstruktor bezeichnet, die Objekte erzeugen
 {
  matNr = a;
  semester = b;
 }
 public static void main(String[] args) // Dies ist der
    Teil Anwenden und Testen
 {
  Student otto = new Student(555,1);
  System.out.println("Otto␣ist␣Student") // Debugging
```

```
    }
    Lalala, ich hab nicht aufgepasst. Kam da noch was?
  }
```

Man unterscheidet

- Variablen
  Eine Variable ist Behälter für einen Wert. Der enthaltene Wert ist ersetzbar während der Ausführung. Eine Variable ist auch immer benannt und getypt.
  Default-Werte:$int = 0$, $boolean = false$, $char = \backslash u0000$, $String = null$
  Java kennt drei Typen von Variablen:

  1. Klassenvariablen, diese sind $static$ und werden über den Klassennamen aufgerufen: `Klassenname.variablenName`
  2. Lokale Variablen befinden sich innerhalb von Methoden(block). Sie sind nur in dieser Methode sichtbar. Nach dem Verlassen des Blocks werden sie gelöscht bzw. ungültig, d.h. sie werden im Hauptspeicher freigegeben
  3. Objektvariablen werden wie die Klassenvariablen automatisch initialisiert. Sie werden über den Objektnamen zugegriffen.

```
public class Test
{
  public int x;
  public int y;
  public String z;
  public Test(int x, int y)
  {
   x = x; // die linke Seite wird immer in
       Objektvariable interpretiert
   y = y;
  }
  public Test(int a,Sring s)
  {
   x = a;
   z = s;
  }
  Test t1 = new Test(4,3); // -> x= 4, y= 3, z =
     null
  Test t2 = new Test(4, "hallo") // -> x= 4, y=
     0, z = "hallo"
}
```

- Typen, und zwar Referenz- und Grundtypen

Was heute gemacht werden soll: Nachschlag zu Variablen Eigenschaften, Unterschiede, Anwendung Objekte/ Referenzen erzeugen und verwalten

Wie kann ich testen?

```
public class Test
{
 public String Name;
 public Test(String s)
 {
  this.name = s;
 }

 //Hier wird jetzt getestet, irgendwie so sollen wir das
    auch immer machen, wenn in den Aufgaben steht, dass
    getestet werden soll ...
 public static void main(String[] args)
 {
  Test a = new Test("Hallo");
  System.out.println(a.name); // Lässt name von a
     ausgeben
  System.out.println(args[0]);
 }
}
```

Lalelu, ich hab null aufgepasst und TI gemacht ...

Irgendwie geht es um Punkte und Verweise, Referenzen und was weiß ich.

```
Point p1 = new Point(2,3);
Point p2 = p1;
```

Und jetzt ist es total krass und aussergewöhnlich, dass wenn man P1 verschiebt, auch P2 verschoben wird. Die verweisen beide nämlich ganz krass auf die selbe Instanz. Jede Änderung, die über eine der beiden Variablen an der Instanz (Objekt) ausgeführt wird, ist über die andere Variable sichtbar.

1 Die unsichtbare Referenz

Oder auch die "allgegenwärtige" Referenz.

- In jedem Objekt wird vom Compiler automatisch eine unsichtbare Referenz auf das eigene Objekt mit dem Namen $this$ erzeugt.

- $this$ wird in allen Methoden als verborgender zusätzlicher Parameter übergeben.
- Auf statische Methoden kann durch $this$ nicht zugegriffen werden, da sie nicht objektgebunden sind.

1 Konstruktoren

Ein Beispiel zu Konstruktoren

```
public class Wert
{
 int item;
 Wert(int a)
 {
  item = a;
 }
 public static Wert addiere10(Wert x)
 {
  return x.item=x.item +10;
 }
 public static int addiere10(int x)
 {
  return x = x+10;
 }

 int w = 5;
 int w1 = addiere(w);
 Wert w = new Wert(5);
 Wert w1 = addiere(w);
}
```

Noch ein Beispiel ...

```
public class Test
{
 public int x;
 public int y;
 public String a;

 public Test(int x, int y)
 {
  this.x = x;
  thix.y = y;
 }
 public Test(int y, String a)
 {
  this.y = y;
  this.a = a;
```

```
  }
}
```

Man kann mehrere Konstruktoren mit dem selben Namen angeben, diese müssen sich allerdings durch die Parameter unterscheiden, z.B. verschiedene Typen, unterschiedliche Anzahl etc.. Dies nennt man auch, "Überladen".

1.1 Konventionen

- Ein Konstruktor hat immer den gleichen Namen wie die Klasse.
- Er ist nicht $static$ und keine Objektmethode.
- Der Konstruktor hat keine Rückgabe.
- Eine Klasse kann mehrere Konstruktoren enthalten.
  Diese Konstruktoren können sich gegenseitig aufrufen.
- Java definiert einen Default-Konstruktor automatisch.

```
public class Konstruktor
{
 int a;
 int b = 3;
 {
  a=b+1; // Initialisierungsblock
 }
 public Konstruktor(int x, inb y)
 {
  this(x); // ruft den Konstruktor auf, der nur einen
     Parameter erwartet. this muss immer 1. Anweisung
     sein
  b = y;
 }
 public Konstruktor(int x)
 {
  a = x;
 }
 public Konstruktor()
 {
 }
}
```

Wie werden die Konstruktoren bearbeitet? Die Reihenfolge ist hierbei wichtig!

1. Die Attribute werden deklariert und zugewiesen.
2. Initisalisierungsblöcke werden bearbeitet.
3. Aufruf des Konstruktor

Es wird jetzt aufgerufen:

```
new Konstruktor()
 a = 4
 b = 3
new Konstruktor(5)
 a = 5
 b = 3
new Konstruktor(5,6)
 a = 5
 b = 6
```

2 Anwendung von $this$

Es gibt mehrere Anwendungen von der unsichtbaren Referent

- Zugriff auf Instanzattribute
- Eine Objektmethode bekommt einen formale Parameter vom Typ des Objektes
- Ein Objekt selbst wird übergeben
- Verkettung / Verschachtelung von Konstruktoren

3 Objekte erzeugen und Verwalten

Wir nehmen das Beispiel vom letzen mal mit $Point\ p$. Das eigene Objekt wird in dieser Anweisung noch nicht erzeugt, d.h. es wird auch kein Speicher für das Objekt reserviert, sondern nur für die Referenz wird Speicher reserviert.
Durch den Aufruf $new\ Point()$ wird Speicher für das Objekt reserviert.

Regel Nicht alle Objekte werden am Anfang erzeugt, sondern kurz vor der Verwendung. Dies tut man aus Performancegründen ...

3.1 Garbage-Collector (GC)

Der Garbage-Collector ist ein Java-Programm, welches im Hintergrund läuft.
Es ist für die Bereinigung von ungültigen Objekten zuständig. Ein Objekt ist ungültig, wenn keine Referenz darauf zeigt.

1 Kontrollstrukturen

Kontrollstrukturen legen fest, dass bestimmte Anweisungen im Programm in Abhängigkeit von bestimmten Bedingungen ausgeführt werden.
Sie kontrollieren den Fluss der Prorammausführung.

1.1 if-Anweisung

Syntax `if (Testbedingung/ Prädikat) <Anweisung>;`
(Ein Prädikat ist eine logische Aussage). Ist die Bedingung $true$, dann wird die Anweisung (der Anweisungsblock) ausgeführt. Im anderen Fall wird die Anweisung (der Anweisungsblock) übersprungen.

```
if(p)
  Anweisung;
else
  Anweisung;

if(p)
{
  Anweisungen
}
else
{
  Anweisungen
}
if(p)
  Anweisung;
else if (q)
  Anweisung;
```

Die Bereiche von if und $else\ if$ müssen disjunkt sein.

```
if(p1)
  if(p2)
   Anweisung;
else
  Anweisung;
```

Bei $p1 == true$ wird die 2. if-Anweisung ausgeführt. Wenn allerdings $p1 == false$ ist, dann wird die nächste Anweiung übersprungen, es wird also erst nach dem $else$ weitergemacht, da if und $else$ einen Block bilden.

1.2 ?-Operator

Zur Verkürzung von if-Anweisungen

```
if (x>y)
 max = x;
else
 max = y;

max = (x>y) ? x:y
```

1.3 $switch$-Anweisung

Die $switch$-Anweisung ist eine Mehrfachverzweigung. Es ist eine Alternative zu einer if-Folge.

Syntax und Konventionen

```
switch(Selektor)
{
  case Kontante1 :
  {
    bla bla
  } break;
  ...
  case KonstanteN :
  {
    bla ...
  } break;
  default :
  {
    bla bla
  }
}
```

Die $switch$-Anweisung wird vom Selektor gesteuert. Der Selektor ist ein Ausdruck - KEIN Prädikat - vom Typ $byte$, $short$, $char$ oder int. Der Selektor kann eine Variable, Konstante oder ein arithmetischer Ausdruck sein.
Stimmt der Wert einer $case$-Konstante mit dem Wert des Selektors überein, werden alle Anweisungen nach diesem $case$ ausgeführt. Stimmt der Wert des Selektors mit keiner $case$-Konstante überein, so werden die Anweisungen des $default$-Zweiges ausgeführt. Der $default$-Zweig kann jedoch entfallen.

Die $break$-Anweisung bricht die Ausführung der gesamten $swicht$ Anweisung ab. Man kann es jedoch auch weglassen.

Beispiel

```
if (monat == 1)
 System.out.println("Januar");
else if (monat == 2)
 System.out.println("Februar");
...
else // Besser hier auch else if, damit bei 13 oder sowas
     nicht auch Dezember ausgegeben wird
 System.out.println("Dezember")
```

Wie kann man das $switch$mäßig machen?

```
switch(monat)
{
 case 1 : System.out.println("Januar");
 break;
 case 2 : System.out.println("Februar");
 break;
 ...
 default : Fehlermeldung oder sowas
}
```

Beispiel Und noch eins ...
Es werden die Noten von 1 bis 4 ausgegeben.

```
int x;
x = (int) (Math.random() * 5)
switch(x)
{
  case 1 : System.out.println("Eins");
  break;
  case 2 : System.out.println("Zwei");
  break;
  ...
  default : System.out.println("Durchgefallen")
}
```

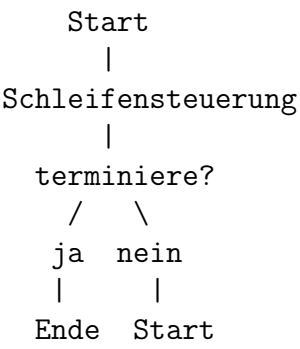
1 Schleifen

Motivation Schleifen werden verwendet um eine bestimmte Folge/ Sequenz von Anweisungen mehrfach auszuüben.
Warum macht man das noch?

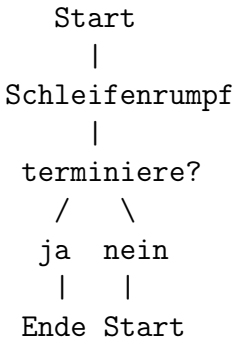
- Es wird Code gespart.
- Der Programmierer legt fest, wie oft die Codesequenz ausgeführt wird.

Aufbau Eine Schleife besteht aus dem Schleifenrumpf, der auszuführenden Codesequenz und der Schleifensteuerung. Diese dient dazu festzulegen, wie oft der Schleifenrumpf wiederholt werden soll.
Es gibt zwei Steuerungsarten von Schleifen:

1. die kopfgesteuerte, d.h. es wird erst das Prädikat geprüft:



2. die fussgesteuerte, hier wird der Rumpf mindestens einmal ausgeführt.



1.1 for-Schleife

Die for-Schleife ist kopfgesteuert.
Wann wird sie verwendet?

- Sie wird eingesetzt, wenn die Anzahl der Iterationen im Voraus bekannt ist.
- Die Anzahl der Iterationen wird in der Schleifensteuerung durch eine oder mehrere Laufvariablen festgelegt

Syntax

```
for([Initialisierung];[Prüfung des Schleifenprädikates];[
    Aktualisierung]) // [] ist Optional

for(;;) // ist dann eine endlose Schleife
```

Initialisierung Hier werden die Laufvariablen initialisiert. Dies sind lokale Variablen. Ihr Wert kann in jeder Iteration (Schleifendurchlauf) geändert werden. Die Initialisierung wird vor der ersten Iteration einmalig durchgeführt.
Mehrere Initialisierungen werden durch Kommata getrennt.

Schleifenprädikat Das Schleifenprädikat wird bei jedem Schleifendurchlauf geprüft. Es können mehrere Bedingungen verknüpft werden. Wird kein Prädikat angegeben bzw. definiert, dann wird $true$ angenommen.

Aktualisierung Die Aktualisierung wird am Ende der Schleife durchgeführt.

```
for(int i = 0;i<2; i++)
  i -= 2;

for(int i = 0;i<2; ++i)
  i -=2;
```

Diese beiden Schleifen sind äquivalent!

```
int i = 4;
for(; i <= 5;)
{
 System.out.println("Blabla");
 i = i+1;
} // kann man machen ist äquivalent zu
```

```
for(int i=4;i<=5;i++)
{...}

for(int i = 2; j=30;(i<=15) && (j<20); i++, j++)
```

1.2 Verschachtelung von *for*-Schleifen

```
int x = 0;
for(int i = 0; i< x; i++)
for(int j = 1; ; j++)
x = x+i+j;
```

2 Arrays

Über Arrays können mehrere Daten eines Typs in einer einzigen Struktur gespeichert werden. Auf die Elemente des Arrays kann über den Index zugegriffen werden. Der Index fängt bei 0 an zu zählen und geht dann bis Länge des Arrays - 1.
Die Länge kann nachträglich nicht geändert werden. In Java sind Arrays übrigens Referenzvariablen. Es sind also Objekte. Weil dem so ist müssen Arrays auch mit *new* initialisiert werden.

```
int[] feld;
int feld[]; // beides gültige Deklarationen für ein Array
    mit int-Werte
int[] feld = new int[10]; // Deklariert und erzeugt ein
   int-Feld mit 10 Plätzen (Länge 10)
int[] feld = {3,10,14}; // Deklaration + Erzeugung +
   Initialisierung
feld[feld.length - 1] => 14
```

Folgendes Array ist in Java möglich:

```
new int {1,2,3}
```

Sowas benutzt man z.B. als Parameter?

1 Verschachtelte Arrays

Oder auch zweidimensionale Arrays genannt.

```
int[][] feld = new int[3][4];
```

Dadurch wird eine Matrix erstellt mit $3 * 4$ Feldern.

```
int[][] feld = new int[2][]; // Hier wird das
   verschachtelte Array noch nicht genauer definiert
feld[0] = new int[3]; // Das erste Eintrag bekommt ein
   Array der länge 3 zugewiesen
feld[1] = new int[2]; // Dem zweiten Eintrag wird ein
   Array der Länge 2 zugewiesen
```

Beispiel

```
for(int i = 0;i<feld.length;i++)
  for(int j = 0;j<feld[i].length;j++)
  {
    ...
  }
```

Arrays sind Referenzen

```
int[] field1 = {23,74};
int[] field2 = {3,101};
field1 = field2; // Dann würde der ursprüngliche Inhalt
   von field1 verlorengehen, da ja nur eine Referenz aufs
    Objekt zeigt.
```

Noch ein megakrasses Beispiel, welches jedes verschachteltes Array in einer neuen Zeile ausgibt.

```
public static void main(String[] args)
{
  String feld[][] = {{"Alle","meine", "Entchen"},{"
     schwimmen","auf","dem","See"}};
  for(int i= 0; i< feld.length;i++)
  {
    for(int j=0;j<feld[i].length,j++)
```

```
      {
        System.out.print(feld[i][j]+" ");
      }
      System.out.println("");
    }
  }
```

Und schon wieder ein irrsinnig sinnvolles Beispiel ...

```
  public stataic void main(String[] args)
  {
    int a[][] = new int [3][4];
    int value;
    for (int i=0;;i<a.length;i++)
    {
      value = (i+1)*100;
      for (int, j = 0; j<a[i].length;j++)
      {
        a[i][j] = value+(j+1);
      }
    }
  }
```

Hier erhalten wir folgende Werte

j \ i	0	1	2
0	101	201	301
1	102	202	302

2 Sortieren

Eine Vorstufe des Sortierens ist das Suchen.

2.1 Lineare Suche

Eine Linearsuche ist eine Suche, bei der ein Feld noch einem Eintrag ab einem Index der Reihe nach durchsucht wird. Die Suche wird abgebrochen, sobald man den gewünschten Eintrag gefunden hat.

```
  publi static int linsuche(int[] a, int Eintrag)
  {
    for (int i = 0; i<a.length;i++)
    {
      if (a[i] == Einrag)
```

```
        return 1;
    }
    return -1;
}
```

2.2 Binäre Suche

Die binäre Suche kann nur bei sortierten Feldern verwendet werden.

Idee Wenn ein Array sortiert ist, steht nach einem Vergleich mit dem mittleren Element fest, ob das gesuchte Element sich in der oberen oder untere Hälfte befindet. Man kann also bereits mit einem Vergleich die Hälfte der Elemente ausschneiden.

1 Binäre Suche

Die binäre Suche kann nur bei sortierten Arrays verwendet werden. Die Idee legt darin, dass wenn ein Array sortiert ist, dann steht nach einem Vergleich mit dem mittleren Element fest, ob das gesuchte Element in der untere oder oberen Hälfte sich befindet.

Laufzeit Man kann also bereits mit einem Vergleich die Hälfte der Elemente des Arrays behandeln.

```
public static int binäreSuche(int[] a, int Eintrag)
{
  int u = 0; // untere Schranke
  int x = a.length-1;  // obere Schranke
  while(u<=x))
  {
    int m = (u + x) / 2; // mittleres Element bestimmen
    if(Eintrag < a[m])
      x = m-1;
    else if(Eintrag > a[m])
      u = m+1;
    else
      return 1; // Eintrag gefunden
  }
  return -1;  // Eintrag nicht gefunden
}
```

2 Schleifensteuerung mit *break* und *continue*

Konventionen *break* und *continue* haben die Möglichkeit eine Schleife in Abhängigkeit von einer Bedingung zu verlassen oder fortzusetzen. Für beide Anweisungen gibt es jeweils ein Konstrukt mit und ohne Marke (Label).
Die *break*-Anweisung ohne Marke beendet die aktuelle Schleife. Bei verschachtelten Schleifen kann mit einer *break*-Anweisung ohne Marke immer nur die innere Schleife verlassen werden.

```
for(while (...)
{
  ...
  break; //bricht die Schleife ab
}
```

```
while(...)
{
  for(...)
  {
    ...
    break; // springt in die while Schleife
  }
}
```

Soll mit $break$ auch die äußere Schleife beendet werden, muss vor der Schleife, die mit $break$ beendet werden soll, eine Marke gesetzt werden.
Die Marke muss daher direkt vor einer Schleife bzw. Anweisungsblock liegen.

```
marke:
{
  ...
  for(...)
  {
    ...
    break marke;
    ...
  }
  ...
}
```

Es können mehrere Verweise auf eine Marke existieren. Ein Marke darf aber nicht selbst mehrfach verwendet werden.

Die Anweisung $continue$ beendet den aktuellen Schleifendurchlauf und setzt das Programm mit dem nächsten Durchlauf der Schleife fort.

Beispiel Guckt, ob im Array eine negative Zahl ist und gibt den Index der ersten negativen Zahl zurück.

```
public static void main(String[] args)
{
  int[] zahlen = {123,42,-23,5,-4,-5}
  int pos = -1;
  for(int i = 0, i <zahlen.length; i++)
  {
    if(zahlen[i] < 0)
    {
      pos = i;
```

```
        break;
      }
    }
    System.out.println(pos);
  }
```

1 Fortsetzung verschachtelte Arrays

Ein Beispiel, welches die ersten negativen Zahlen jedes verschachtelten Arrays ausgibt.

```
public static void main(String[] args)
{
  int flag = 0;
  int[][] feld {{36,78},{8,-3,38},{-7,0,-99}};
  Marke:
  for(int i = 0; i<feld.length;i++)
  {
    for(int j=0; j<feld[i].length;j++)
    {
      if(feld[i][j] <0)
      {
        System.out.println(i + "␣" + j);
        flag = 1; //Wir haben eine negative Zahl gefunden
        continue Marke; // Mach bei Marke weiter
      }
    }
  }
  if(flag == 0)
    System.out.println("Keine␣negative␣Zahl␣gefunden")
}
```

Ein nächstes Beispiel. Wir gucken, ob die Arrays dieselben Elemente enthalten (Reihenfolge egal).

```
public static boolean vergleich(int[] a, int[] b)
{
  if(a.length != b.length)
    return false;
  boolean[] temp = new boolean[a.length-1];
  for(int i =0;i<temp.length;i++)
    temp[i] = false;
  else
  {
    Marke:
    for(int i:a) // i repräsentiert ein Element von a
    {
      for(int j =0; i<b.length, j++)
      {
        if(i == b[j] && temp[j] == false)
```

```
          {
            tempt[j] = true;
            continue Marke;
          }
        }
      }
      for(boolean u:tmp)
        if(u==false)
          return u;
      return (1==1);
    }
}
```

2 Wrapper-Klassen

Für jeden primitiven Datentyp wird eine zugehörige Wrapper-Klasse in Java bereitgestellt (Integer für int, Byte für byte ...).
Dadurch können primitive Datentypen als Objekte dargestellt und bearbeitet werden.
Viele Methoden in Java erwarten als Parameter einen $Object$-Typ.

1 Fortsetzung Wrapper-Klassen

1.1 Konstruktoren

```
// Zwei möglichkeiten für Konsturktoren: Integer (int
   value) und Integer(String value)
Integer I = new Integer(3);
Integer I = new Integer("3");
```

Die Methode $intValue()$ liefer den Integerwert zurück (int). $toString()$ liefert einen String zurück, der den gegebenen Wert repräsentiert. Die Methode $parseInt(String\,value)$ versucht einen String in einen Int-Wert zu konvertieren.

```
int i = Integer.parseInt("3");
public static void main(String[] arg)
{
  int i = Integer.parseInt(arg[0]);
  Integer I = new Integer(arg[0]);
  System.out.println(arg[0]); //Liefert 4 als String Wert
  System.out.println(i); // Liefert 4 als int Wert (zur
     Ausgabe natürlich wieder ein String)
  System.out.println(I.intValue()); // selbiges
  System.out.println(I.toString()); // String wird
     zurückgegeben
}
```

2 Exceptions

Was ist eine Exception? Eine Exception kennzeichnet eine besondere Situation, die eingetreten ist. In Java ist eine Exception eine Instanz der Klasse Exception.
Das Auslösen einer Exception wird $throw$ genannt. Exceptions werden immer innerhalb einer Methode geworfen. ($try\ \{...\}$). Das Behandeln/ Auffangen von Exceptions funktioniert über $catch\ \{...\}$

```
methodeA(...) throws ex1 //ex1 = Exceptionklasse
{
  throw new ex1(...);
  ...
}
methodeB()
{
  try
    methodeA(...);
  catch(ex1 e)
```

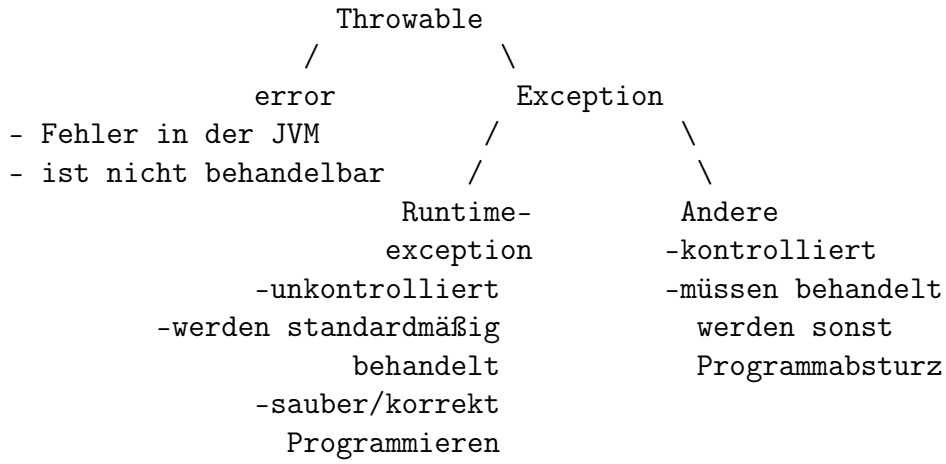
```
    {
      ... //Exception-Behandlung
    }
  }


  try
  {
    Exceptionwerfen
  }
  catch(ex1 e1)
  {
    ... //Behandlung ex1
  }
  catch(exn en)
  {
    ... //Behandlung exn
  }
  finally //optional, sobald irgendein catch zutrifft, wird
      dies ausgeführt
  {
    ...
  }
```

2.1 Wie das in Java intern geht

Exception wird ausgelöst dann wird der try-catch-Block gesucht. Entweder wird dieser gefunden und dann behandelt, oder es wird geguckt, ob es sich um eine kontrollierte Exception handelt. Wenn der Block gefunden ist, dann wird der catch-Block ausgeführt. Wenn es sich nicht um eine kontrollierte Exception handelt, dann wird eine Standard Behandlung ausgeführt. Im anderen Fall bekommt man einen Compiler-Fehler. (Ich find das ja so ein wenig unlogisch, aber Mohammed wird schon recht haben ...)

1 Fortsetzung Exception



Beispiel Runtimeexception

```
public static void main(String[] arg)
{
  int i = Integer.parseInt(arg[0]); // Was passiert, wenn
      ein a übergeben wird? --> Runtime-Exception
  // besser ist:
  try
  {
    int j = Integer.parseInt(arg[0]);
  }
  catch(RuntimeException e)
  {
  }
}
```

Unsere eigene Exceptionklasse

```
class KontoException extends Exception
{
  static KontoException()
  {
    super(); // Bezieht sich auf auf die Superklasse
  }
  static KontoException(String fehler)
  {
    super(fehler)
  }
```

```
  public String getLocalizedMessage() // überschreibt die
      getLocalizedMessage der Exception-Klasse
  {
    return "Fehler in Berechnung Guthaben"
  }
}

public class Guthaben
{
  double guthaben;
  public Guthaben()
  {
    guthaben = 0; // kann auch weggelassen werden, da
       Instanzvariablen auf default-Werte gesetzt werden
  }
  public void addGuthaben(double a) throws KontoException
  {
    if(a<0)
      throw new KontoException("Schulden nicht erlaubt.")
          ;
    guthaben = guthaben +a;
  }
  public double nenneGuthaben() throws KontoException
  {
    if(this.guthaben <=0)
      return guthaben / throw new KontoException("Konto
         leer");
  }
  public static void main(String[] arg)
  {
    Guthaben g = new Guthaben();
    try
    {
      g.addGuthaben(0);
    }
    catch(KontoException k)
    {
      System.out.println(k.toString()); // -> Fehler in
         Guthaben
```

```
            System.out.println(k.getMessage); // -> Schulden
                nicht erlaubt
        }
    }
}
```

1 Strings

1.1 Konventionen / Besonderheiten

$String$ in Java ist eine Klasse und kein Feld von $char$s. Hier geht es also um Referenztypen. Diese Klasse $String$ ist $final$ deklariert, dass heißt es können keine weiteren Klassen von ihr erben.
Obwohl es sich bei $String$ um eine Klasse handelt, muss man deren Instanzen NICHT durch einen Konstruktor oder new erzeugen. Man kann es aber natürlich.

```
String h = "ALP␣1"; // Referent auf Objekt wird erstellt
String h1 = "ALP␣2";
h = h + h1; // Konkatenation von Strings. Hier kreiert
   Java eine neue Speicherzelle und schreib die
   Konkatenation rein und legt dann die Referenz um. Das
   alte h wird vom Garbage-Collector bereinigt.
```

In Java werden zwar Stringarten unterschieden, und zwar statische und dynamische Strings.
Strings, deren Inhalt zur Compilerzeit feststehen (statische Strings), werden nur einmal erzeugt. Diese werden über einen Stringpool verwaltet.

```
String a = "Hallo"; // a zeigt auf Speicherzelle mit
   Hallo
String b = "Hallo"; // die Referenz b zeigt auf die selbe
    Hallo-Speicherzelle
```

Werden Strings dynamisch erzeugt, d.h. zur Laufzeit des Programmes, so erzeugt Java immer eine neue separate Speicherzelle für diese Stringinstanz.

```
String b1 = new String("ALP␣1"); // b1 -> ALP 1
String b2 = new String("ALP␣1"); // b2 -> ALP 1 sind
   unterschiedliche Speicherzellen
```

1.2 Vergleichen von Strings

Mit $==$-Operator werden keine Inhalte von Speicherzellen verglichen sondern nur die Referenzen. D.h. ob die Variablen auf dieselbe Adresse weisen. Mit $equals$ werden aber die Inhalte von Speicherzellen verglichen, auf die die Referenz zeigt.

```
String x1 = "Hallo";
String x2 = "Hallo";
String x3 = new String("Hallo");
String x4 = "Hi";
x1 == x2 -> true, da Referenzen gleich
```

```
x1 == x3 -> false, da zwei verschiedene Speicherzellen
x1.equals(x2) -> true
x2.equals(x3) -> true
```

1.3 String-Methoden

```
String a = "ALP␣II";
a.length() -> 6
trim() // entfernt alle Leerzeichen am Anfang und am Ende
    eines Strings
String space = "␣␣ALP␣II␣";
a.trim() -> "ALP␣II";
String a = "HALLO";
String b = "Hallo";
// toLowerCase()/ toUpperCase() wandelt die Zeichen eines
    Strings in klein bzw. Großbuchstaben um.
indexOf(String s) // Liefert den Wert innerhalb eines
   Strings, von dem der übergebende Teil beginnt
String s = "Hallo";
int i = indexOf("ll") -> 2
substring(int a, int b) // liefert eine Teilstring von
   Index a bis b-1
s.substring(2,4); -> "ll"
```

1 Hoare-Kalkül

Achtung - Klausurrelevant!

1.1 Motivation

Jeder Allgorithmus erwartet einen Input und liefert einen Output.

- Ist das Ergebnis für jede mögliche Eingabe $e \in Input$ nach der Berechnung von dem Algorithmus korrekt?
- Ist es immer gewährleistet, dass die Berechnung des Algorithmuses terminiert? Kommt es also immer zu einem Output?

Die Antwort auf die Fragen ist äquivalent zur Frage nach der totalen Korrektheit.
Vorstellung des Kunden -> Anforderungsdefinition -> Formale Spezifikation (Modell) -> Programm -> Testen für jede mögliche Eingabe bzw. Verifizieren. Je nach dem, wie das Ergebnis dann ist, muss man wieder zurückkehren.

1.2 Verifikation

Dies ist ein mathematisches Verfahren für die Untersuchung der Korrektheit von Programmen, z.B. das Hoare-Kalkül.
Das zentrale Element des Hoare-Kalküls ist das Hoare-Tripel, das beschreibt, wie ein Programmteil den Zustand einer Berechnung ändert. Das Tripel ist

$$\{P\}\ S\ \{Q\}$$

wobei man P und Q die Zusicherungen nennt und S ist ein Programmteil. P heißt die Vorbedingung und Q ist die Nachbedingung.
Zusicherungen sind Formeln / Ausdrücke der Prädikatenlogik.
Falls P für den Programmzustand vor der Ausführung von S gilt, dann gilt Q danach. Bei einem Algorithmus kann der Input als Anfangszustand und der Output als Endzustand verstanden werden. Dies sind dann also Vor- und Nachbedingung P und Q.

1.3 Verfahren

1. Anwenden von Q auf S_n (wir gehen von rechts nach links) das Resultat ist eine Zwischenbedingung $\alpha_{n-1,n}$.
2. $\{P\}\ S_1, S_2 \ldots S_{n-1}\ \{\alpha_{n-1,n}\}$ Verfahre hiermit rekursiv
3. $\{P\}\ nix\ \{\alpha\}$
4. Zeigen, dass $P \Rightarrow \alpha$ bzw. $P \Leftrightarrow \alpha$

1.4 Hoare-Regeln

Diese Regeln sind Abhängig von der Anweisung S_n.

1. $x := a\{Q\} \rightarrow \alpha \equiv Q[x/a]$ d.h. ersetze jedes Vorkommen von x in Q durch a.

Es ist in folgendem Beispiel die Korrektheit von $swap(a, b)$ zu beweisen.

```
swap(int x, int y)
```

$$\{x = a \wedge y = b\} \;\; tmp := x; \;\; x := y; \;\; \underbrace{y := tmp;}_{S_n} \;\; \underbrace{\{x = b \wedge y = a\}}_{Q}$$

$Q[y/tmp]$

$$\{x = a \wedge y = b\} \;\; tmp := x; \;\; x := y; \;\; \underbrace{\{x = b \wedge tmp = a\}}_{Q}$$

$$\{x = a \wedge y = b\} \;\; tmp := x; \;\; \underbrace{\{y = b \wedge tmp = a\}}_{Q}$$

$$\{x = a \wedge y = b\} \;\; NIX \;\; \underbrace{\{y = b \wedge x = a\}}_{Q}$$

Das diese Beiden Aussagen äauivalent sind, ist ja wohl nicht zu übersehen ...

1 WHILE Schleife

1. Finde eine geeignete Schleifeninvariante P. Das ist ein mathematischer Ausdruck der bei jeder Ordnung/ jedem Durchgang im Programm gültig bleibt.
   Wie finde ich die Invariante? -> Hierzu gibt es leider kein ordentliches Verfahren ...
2. Finde eine geeignete Zwischenbedingung α_1 als neue Vorbedingung für die While-Schleife
3. Verifiziere den Erhalt der Schleifeninvariante P

   $$\{P \wedge B\} S \{P\}$$

4. Weise nach, dass die Schleifeninvariante stark genug ist, dass sie die Nachbedingung γ erzeugt.

   $$(P \wedge \neg B) \Rightarrow \gamma$$

5. $\{Q\} c_1, c_2 \ldots c_{n-1} \{\alpha_1\}$

Dazu ein Beispiel

```
(a >= 0) Vorbedingung Alpha 1
y := 0
z := 0
while(y != a)
{
  z = z + 2*y+1;
  y = y+1;
}
(z = a*a) Nachbedingung Gamma
```

Wir müssen jetzt jeden der fünf Schritte überprüfen ...

1. Finde die Schleifeninvariante: $y \leq a \wedge z = y^2$
2. Finde die Zwischenbedingung α_1: $z == 0 \wedge y == 0 \wedge a \geq 0$
3. $\{(y \leq a \wedge z = y^2) \wedge y! = a\}\ z = z + 2 * y + 1; y = y + 1\ \{y \leq a \wedge z = y^2\}$
   Jetzt kommt die tolle Ersetzungsregel:

   $$\{(y \leq a \wedge z = y^2) \wedge y! = a\}\ z = z + 2 * y + 1\ \{y + 1 \leq a \wedge z = (y + 1)^2\}$$

Und noch einmal

$$\{(y \leq a \wedge z = y^2) \wedge y! = a\} NIX \; \{y + 1 \leq a \wedge z + 2 * a + 1 = (y + 1)^2\}$$

Es muss gezeigt werden, dass aus dem ersten Ausdruck der zweite folgt, wir zerlegen also die einzelnen Anweisungen:

$$\underbrace{y \leq a \wedge y! = a}_{=y<a} \Rightarrow y + 1 \leq a$$

$$z = y^2 \Rightarrow \underbrace{z + 2y + 1}_{=y^2+2y+1=(y+1)^2} = (y + 1)^2$$

Wir man sieht, sind also die Ausdrücke äquivalent.

4. $\{y \leq a \wedge z = y^2 \wedge (\overline{y \neq a})\} \Rightarrow \{z = a^2\}$ das entspricht $z = a^2 \wedge y = a$ das gilt also auch.

5. $\{a \geq 0\} \; y = 0; z = 0 \; \{z = 0 \wedge y = 0 \wedge a \geq 0\}$ kann ersetzt werden zu $\{a \geq 0\} NIX \{0 = 0 \wedge 0 = 0 \wedge a \geq 0\}$. Und man sieht, dass $a \geq 0 \Leftrightarrow a \geq 0$ gilt.

Als kleiner Tipp, eine Vorbedingung kann auch einfach nur $true$ sein ...

2 Laufzeitverhalten

Input -> Algorithmus -> Output. Innerhalb des Algorithmus findet eine Berechnung statt. Es stellt sich die Frage, ob die Berechnung korrekt ist. Die kann mit dem Hoare-Kalkül beantwortet werden. Eine andere Frage ist, ob die Berechnung effizient ist. Dafür gibt es zwei Blickwinkel: Einmal die Laufzeit und den Speicherverbrauch.

```
for(int j = 1; u < a.length; j++)
{
  key = a [j];
  // ???
  i = j-1;
  while(i > -1 && a[i] > key)
  {
    a[i+1] = a[i];
    i = i-1;
  }
  a[i+1]  = key;
}
```

Wie ist nun die Laufzeit von diesem Algorithmus? Dazu gucken wir uns erstmal an wieviel uns eine Zeile "kostet". Wir weisen jeder Zeile eine Konstante c_i zu, die die Kosten (Laufzeit) pro Zeile angibt.

Kosten	wie oft
c_1	n
c_2	$n-1$
c_3	- Kommentar
c_4	$n-1$
c_5	$\sum_{j=1}^{n} t_j$, mit t_j Anzahl der Durchläufe für j
c_6	$\sum_{j=1}^{n} t_j - 1$
c_7	$\sum_{j=1}^{n} t_j - 1$
c_5	$n-1$

Die Gesamtkosten ergeben sich als Summe aller Kosten.
Best-Case: Das Array ist bereits sortiert. Daraus folgt

$$T_{min=}c_6 * 0 + c_7 * 0 + c_6(n-1) * 1... = a_n + c.$$

Der Worst-Case: Das Array ist falschrum sortiert. Dann folgt $t_j = \sum_{j=1}^{n} j$

$$T_{max} = ... = an^2 + bn + c$$

Es reicht eine Klasse, Ordnung der Funktions-Laufzeit anzugeben. Das ist das Wachstumsverhalten. Das ist gerade die O-Notation. Ziel der O-Notation ist eine obere Schranke für die Laufzeit $T(n)$ zu bestimmen.

1 Regeln zu O-Notation

1. $O(\log n)$ $f(x) = \log_a n$

   $$log_a n = \underbrace{\log_a b}_{=c} * \log_b n$$

   $$O(c * \log_b n) = O(\log_b n)$$

2. Wir haben ein Problem der Länge n und dieses Problem halbiere sich immer. dann haben wir immer $\log n$ also $O(\log n)$

3. Sequenz zweier Anweisungen mit $T_1(n) = O(f(n))$ und $T_2(n) = O(g(n))$. Dann ist $T(n) = T_1(n) + T_2(n)$ und $O(f(n)) + O(g(n))$
   Allgemein gilt, die Funktion die stärker wächst, bestimmt die Laufzeit (sie ist die obere Schranke).

   $$T(n) = \max(O(f(n)) + O(g(n)))$$

4. Schleifen: Iteration (n-1), wenn die Anzahl der Schleifenausführung ist (z.B. Insertion Sort)

   ```
   Schleife(Pradikat)
   {
     Rumpf o(h(n))
   }
   ```

   Hier gilt immer $T_{Schleife}(n) = o(f(n)) * o(h(n))$ also Anzahl * Kosten.

5. Rekursion:

   ```
   o(p(n)
   {
   c1
   o(p(n/a)
   c2
   }
   ```

   $T_{rekursiv}(n) = c_1 + o(n/a) + c_2$
   $c = c_1 + c_2$ Dann ist $c + c + c + c... = T?$ $O(c * \log n) = O(\log n)$

2 Merge-Sort

Funktioniert nach "Divide and Conquer". Teile das Array in zwei gleich große Teile (Divide). Und sortiere beide Teile rekursiv mit Merge-Sort (Conquer). Füge beide Teile u einem sortiertem Feld zusammen (Combine).

```
mergeSort(a[], links, rechts)
{
  if(links < rechts)
  {
    mitte = links + recht / 2;
    mergeSort(a[], links, mitte);
    mergeSort(a[], mitte, rechts);
    merge(a[], links, mitte, rechts);
  }
}
```