Se greit Eidgenossen recap

Hat jemand schon die Pingu-Gang informiert?

Dieses ***Recap*** liefert ***keine Garantie auf Vollständigkeit***. Ich konzentriere mich bewusst auf einige wenige aber wichtige ***Kernthemen***, die in der Hinsicht — zumindest auf den Folien — auch leicht simplifiziert dargestellt sind.

Fröhliche Weihnachten
Flo

1. Algorithmenkonstruktion
2. Programmkonstrukte (Namen & Programmfluss)
3. Arrays und Iterationen
4. Unterprogramme
5. Objektorientierte Programmierung

6. Dynamische Datenstrukturen
7. Weiterführende Konzepte OOP
8. Rekursion
9. Laufzeitkomplexität
10. Suchen und Sortieren

What can we say about an Algorithm?

Totale Korrektheit

1. **Termination**
   Der Algorithmus endet nach endlich vielen Schritten für jede Eingabe.
2. **Partielle Korrektheit**
   Wenn der Algorithmus terminiert, ist er korrekt.

5 Algorithmusanalyse

$$L(0) = 1,\ L(1) = 1 \qquad L(n) = L(n-1) + L(n-2) + 1$$

Aufgabe 1

```
public static long L(int n) {
   if(n == 0 || n == 1) return 1;
   int l1 = 1, l2 = 1;
   for(int i = 0; i < n - 1; i++) {
      int tmp = l1;
      l1 = l2;
      l2 = tmp + l2 + 1;
   }
   return l2;
}
```

6 Algorithmusanalyse

```java
public static long L(int n) {
   if(n == 0 || n == 1) return 1;
   int l1 = 1, l2 = 1;
   for(int i = 0; i < n - 1; i++) {
      int tmp = l1; l1 = l2;
      l2 = tmp + l2 + 1;
   }
   return l2;
}
```

Auflösung 1

1. Terminiertheit
   - `n` ist konstant.
   - Für $n = 0$, bzw. $n = 1$ trivial.
   - Für $n > 1$ wächst i streng monoton an und wird irgendwann $i < n - 1$ verletzen.

2. Partielle Korrektheit

Algorithmusanalyse

```
public static long L(int n) {
   if(n == 0 || n == 1) return 1;
   int l1 = 1, l2 = 1;
   for(int i = 0; i < n - 1; i++) {
      int tmp = l1; l1 = l2;
      l2 = tmp + l2 + 1;
   }
   return l2;
}
```

1. Terminiertheit
2. Partielle Korrektheit
   - Für $n = 0$, bzw. $n = 1$ per Definition.
   - Für $n > 1$ per vollständiger Induktion.

   IA: Für $n = 2$. Mit $i < 1$ haben wir einen Zyklus. Das Ergebnis: $\texttt{l2} = 1 + 1 + 1 = 3 = L(2)$ und $\texttt{l1} = 1 = L(1)$. ✔

   IH: Für $n \geq 0$ ist `l2` die n-te und `l1` die $n - 1$-te Leonardo-Zahl.

   IS: Mit $n \to n + 1$ wird durch `l1 = l2` nämlich nach der IH $\texttt{l1} = L(n) = L(n + 1 - 1)$ und mit `l2 = tmp + l2 + 1` wird $L(n + 1) = L(n + 1 - 1) + L(n + 2 - 1) + 1$.

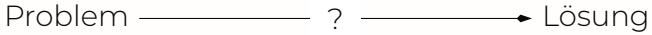
Auflösung 1

But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

- Endlich viele wohldefinierte Einzelschritte.
- Kann grafisch, textuell, … erfolgen.
- **Fordert gemeinsames Sprachverständnis.**

Discussing an Algorithm



- Problemspezifikation — Was meinen Sie mit „schnell“?
- Problemabstraktion — Was ist gegeben, was ist gesucht?
- Algorithmenentwurf — Wie kommen wir von gegeben zu gesucht?
- Korrektheitsnachweis — Löst unser Ansatz das Problem?
- Aufwandsanalyse — Wie verhält er sich?

Bauen Sie mir das

- Spezifikation
- Abstraktion
- Algorithmus
- Korrektheit
- Analyse

Aufgabe 2

Guten Tag ich heiße Günther und ich bin Besitzer eines Plätzchen-ladens. Dafür verwende ich verschiedene Mehle (Weizen oder Dinkel), verwende Zucker oder Süßungsmittel, sowie stets eine weitere Komponente (Kokos oder Zimt).
Da ich alle Plätzchen gleich gern habe, werde ich sie alle immer zum selben Preis verkaufen. Geld an sich ist auch gar nicht das Problem, leider können mir meine Zulieferer immer nur eine begrenzte Menge (in Gramm) an den jeweiligen Stoffen zur Verfügung stellen — die kann ich ihnen aber geben. Die habe ich. Weiter habe ich pro Jahr immer eine handvoll an Rezepten, die immer eine bestimmte Anzahl eines Mehls, eines Süßungsstoffes und einer weiteren Komponente pro Plätzchen benötigen.
Auf der Basis hätte ich jetzt gerne eine Möglichkeit herauszufinden, wie ich insgesamt immer maximal viele Plätzchen aus den gegebenen Ressourcen backen kann.

11.1 Was willer denn?

Auflösung 2

Guten Tag ich heiße Günther und ich bin Besitzer eines Plätzchenladens. Dafür verwende ich verschiedene **Mehle** (Weizen oder Dinkel), verwende **Zucker oder Süßungsmittel,** sowie stets eine **weitere Komponente** (Kokos oder Zimt).
Da ich alle Plätzchen gleich gern habe, werde ich sie **alle immer zum selben Preis verkaufen. Geld an sich ist auch gar nicht das Problem,** leider können mir meine **Zulieferer** immer nur eine **begrenzte Menge (in Gramm)** an den jeweiligen Stoffen zur Verfügung stellen — die kann ich ihnen aber geben. **Die habe ich.** Weiter habe ich pro Jahr immer eine handvoll an **Rezepten,** die immer eine **bestimmte Anzahl eines Mehls, eines Süßungsstoffes** und einer **weiteren Komponente pro Plätzchen** benötigen.
Auf der Basis hätte ich jetzt gerne eine Möglichkeit herauszufinden, wie ich insgesamt immer **maximal viel Plätzchen aus den gegebenen Ressourcen backen** kann.

11.2 Was willer denn?

Auflösung 2

- **Plätzchen:** Produkt, an dessen Menge wir interessiert sind.
- **Plätzchenmenge:** Eine natürliche Zahl $P \geq 0$.
- **Zulieferer:** Stellt Ressourcen für Produktion zur Verfügung.
- **Ressource:** Ressource i wird für Rezept benötigt.
- **Ressourcenmenge:** Fließkommazahl $0 \leq r_i \in \mathbb{R}$ in Gramm.
- **Rezept:**
  Tupel $R_\ell = (k_{\ell_1}, k_{\ell_2}, k_{\ell_3})$ für ein Plätzchen. Anwendung: $r_{\ell_1} -= k_{\ell_1}$, $r_{\ell_2} -= k_{\ell_2}$ und $r_{\ell_3} -= k_{\ell_3}$. Sowie $P+=1$. Anwendbar, wenn $k_{\ell_m} \leq r_{\ell_m}$.
- **Komponente:** Eine benötigte $0 < k_\ell \in \mathbb{R}$ Ressourcenmenge.
- **Maximal viele Plätzchen aus gegebenen Ressourcen:**
  Für gegebene r_i und Rezepte R_ℓ, maximale Plätzchenmenge $k \in \mathbb{N}$ aller anwendbaren Rezeptkombinationen: $\bigtimes_{j \in k} R_{\ell_j}$.

Abstraktion

Auflösung 2

- Gegeben:
  - Ressourcenmengen $r_1, r_2, \ldots, r_n$
  - Rezepte $R = \{(k_3, k_7, k_1), (k_4, k_2, k_9), \ldots\}$, sei $|R| = m$
- Gesucht:
  - Maximale Anwendbarkeit k der Rezepte.

Algorithmenkonstruktion

Auflösung 2

- Erkenntnisse: $r_1, \ldots, r_n$ und $R = \{(k_1, k_2, k_3)_1, \ldots, (\ldots)_m\} \longrightarrow k$
  - Es gibt dieses Maximum. Mit $0 \leq r_i \in \mathbb{R}$ und $0 < k_i \in \mathbb{R}$ sind irgendwann keine Rezepte mehr anwendbar.
  - Das Maximum muss nicht eindeutig sein.
  - Die Reihenfolge der Rezeptanwendung spielt keine Rolle.
  - Jede Anwendung verringert Ressourcen.
- Idee:
  - Beginne mit „Anwendungsvektor“ $A = (0, 0, \ldots, 0)$ mit $|A| = m$.
  - Finde maximale Anwendbarkeit $\max a_i = R_{\max,i}$ von Rezept i durch Inkrement bis nicht mehr anwendbar.
  - Probiere alle $A = (a_1, \ldots, a_m)$ mit $0 \leq a_i \leq R_{\max,i}$.
  - Speichere maximale anwendbare Summe $\sum_{i=0}^{m} a_i$.

Algorithmenkonstruktion II

$r_1, \ldots, r_n$ und $R = \{(k_1, k_2, k_3)_1, \ldots, (\ldots)_m\} \longrightarrow k$

- Ist $A = (a_1, \ldots, a_m)$ anwendbar?

```
1 for i ← 0 to m step 1 do
      // Wende R_i genau a_i mal an.
2     r ← (k_a, k_b, k_c) ← R_i;
3     r_a ← r_a − a_i · k_a;
4     r_b ← r_b − a_i · k_b;
5     r_c ← r_c − a_i · k_c;
      // Oder: „Reduce r_a, r_b, r_c by a_i times k_a, k_b, k_c
         respectively“
6 end
7 return ∀_{i∈{1,…,n}} r_i ≥ 0;
```

Auflösung 2

Algorithmenkonstruktion

$r_1, \ldots, r_n$ und $R = \{(k_1, k_2, k_3)_1, \ldots, (\ldots)_m\} \longrightarrow k$

- Was sind die Maxima $R_{\text{max},i}$?
- Wir schreiben $A_{i \leftarrow j}$ für $(0, \ldots, a_{i-1}, j, a_{i+1}, \ldots, 0)$.

Auflösung 2

```
1 for i ← 0 to m step 1 do
      // Inkrementiere a_i bis nicht mehr anwendbar.
2     A ← (0, ..., 0);
3     j ← 1;
4     while anwendbar(A_{i←j}) do j ← j + 1;
      // j ist eins höher als das anwendbare Maximum.
5     R_{max,i} ← j − 1;
6 end
7 return ∀_{i∈{1,...,n}} r_i ≥ 0;
```

$r_1, \ldots, r_n$ und $R = \{(k_1, k_2, k_3)_1, \ldots, (\ldots)_m\} \longrightarrow k$

Auflösung 2

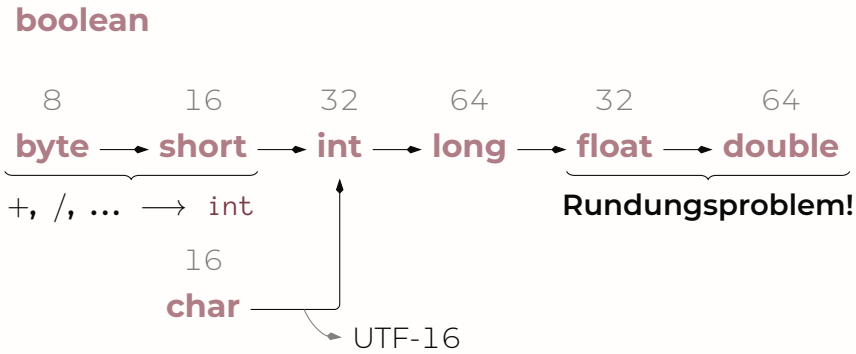
› Und nun das Testen aller möglichen Anwendungen:

```
1 guatzla ← 0;
2 for A ← (0, ..., 0) to (R_max,1, ..., R_max,m) do
3 |   if anwendbar(A) then
4 |   |   tmp ← ∑_{i←0}^{m} a_i;
5 |   |   guatzla ← max{a_i, guatzla};
6 |   end
7 end
8 return guatzla;
```

Kurzgesagt

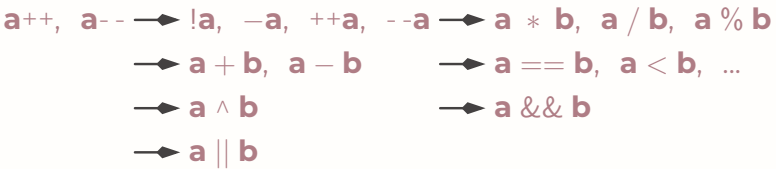
- Totale Korrektheit
  - *Terminiertheit:* Endliche Schritte für jede Eingabe.
  - *Partielle Korrektheit:* Wenn terminiert, dann korrekt.
- Abstraktion ist wichtig.

Primitive Datentypen



Präzedenzregeln

- Eine Auswahl:



Typbestimmung

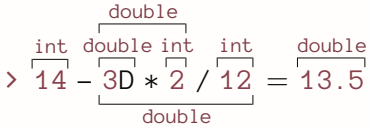
Aufgabe 3

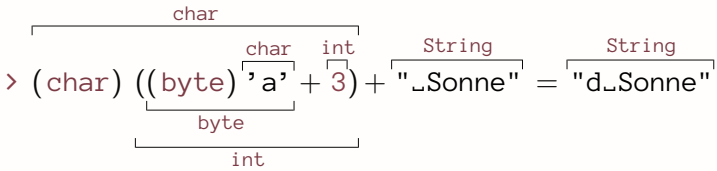
Bestimmen Sie für jeden der folgenden Java-Ausdrücke den Typ *jedes Teilausdrucks*, sowie das Gesamtergebnis. Sie dürfen Rundungsprobleme der Sprache ignorieren:

```
> 14 - 3D * 2 / 12

> (char) ((byte) 'a' + 3) + "␣Sonne"

> 14 - 2 + (3 > 9 || true ^ false ? "Hallo" : "Welt")

> 7/2 - 3 * 2.5 + 2f
```

Typbestimmung

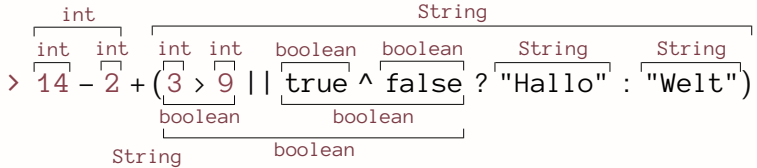
Auflösung 3





Typbestimmung

Auflösung 3



Variablendefinition

Deklaration: Es gibt etwas mit diesem Namen (und Typ):

```
int x;  char w;
```

Zuweisung: Wertzuweisung, bzw. Ersetzung des Wertes:

```
x = 12; x = 9;  w = 'k';
```

Initialisierung: *Erste* Wertzuweisung zu einer Variable:

```
int x = 12;  char w; w = 'x';
```

`final` erzwingt, dass eine Variable nur Initialisiert und dann nicht erneut zugewiesen werden darf.

Standardwerte

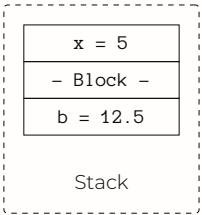
Explicit is betta!

- Standardwerte für:
  - Klassen-Variablen (`static`)
  - Instanz-Variablen
  - Array-Komponenten (wie bei `new int[]`)
- Der Standardwert ist „Null“:
  - `byte`, `short`, `int`, `long` ⟶ `0`
  - `float`, `double` ⟶ `0.0`
  - `char` ⟶ `'\u0000'` („Unicodesymbol mit Wert 0“)
  - `boolean` ⟶ `false`
  - Komplexe Datentypen ⟶ `null`

Blöcke

- Methodenaufrufe erzeugen „Stack-Frames“.
- Code-Blöcke arbeiten vergleichbar.
- Beim Verlassen wird alles davon vom Stack geschmissen.

```
int x = 5;
{
   double b = 12.5;
   x = x + (int) b;
}
int y = x;
```



Überschatten

- Instanz-/Klassenvariable mit gleichem Bezeichner wie lokale Variable.
- Der lokale Bezeichner überschattet den „globalen“.
- So kann auch der Typ verändert werden.

```
class LetThereBeDarkness {
  static int happiness = 7;
  public static void main(String[] args) {
    System.out.println(happiness);
    String happiness = "Maunz";
    System.out.println(happiness);
  }
}
```

Ausgabe:

```
7
Maunz
```

Programmausgaben

Aufgabe 4

```
public class Example {
   public static int main;
   public static String a = "Hallo";
   public static void main(String[] args) {
      int a = 7;
      int b = 15;
      System.out.println(a + ";␣" + main); // i)
      {
         float main = 3 * b++;
         Example.a += b;
         System.out.println(b + ";␣" + main); // ii)
      }
      System.out.println(a + ";␣" + b + ";␣" + main); // iii)
      System.out.println(Example.a + ";␣" + Example.main); // iv)
   }
}
```

Programmausgaben


```java
public class Example {
   public static int main;
   public static String a = "Hallo";
   public static void main(String[] args) {
      int a = 7;
      int b = 15;
      System.out.println(a + ";␣" + main); // i)
      {
         float main = 3 * b++;
         Example.a += b;
         System.out.println(b + ";␣" + main); // ii)
      }
      System.out.println(a + ";␣" + b + ";␣" + main); // iii)
      System.out.println(Example.a + ";␣" + Example.main); // iv)
   }
}
```

Auflösung 4

Programmausgaben

Auflösung 4

```
public class Example {
   public static int main;
   public static String a = "Hallo";
   public static void main(String[] args) {
      int a = 7;
      int b = 15;
      System.out.println(a + ";␣" + main); // i)
      {
         float main = 3 * b++;
         Example.a += b;
         System.out.println(b + ";␣" + main); // ii)
      }
      System.out.println(a + ";␣" + b + ";␣" + main); // iii)
      System.out.println(Example.a + ";␣" + Example.main); // iv)
   }
}
```

Programmausgaben


```java
public class Example {
  public static int main;
  public static String a = "Hallo";
  public static void main(String[] args) {
    int a = 7;
    int b = 15;
    System.out.println(a + ";␣" + main); // i)
    {
      float main = 3 * b++;
      Example.a += b;
      System.out.println(b + ";␣" + main); // ii)
    }
    System.out.println(a + ";␣" + b + ";␣" + main); // iii)
    System.out.println(Example.a + ";␣" + Example.main); // iv)
  }
}
```

i) `7; 0`

ii) `16; 45.0`

iii) `7; 16; 0`

iv) `Hallo16; 0`

Auflösung 4

i) `a = 7`, *da lokales* `a` *das globale* `Example.a` *überschattet.* `main = 0` *durch* `Example.main`, *da Deklaration mit default „*`0`*“ initialisierte.*

ii) `b = 16`, da lokal überschattet und durch `b++` in Initialisierung von lokalem `main`. `main = 45.0`, da `float` und Initialisierung `3 * 15` (Postinkrement).

iii) `a = 7` *mit lokalem* `a`, `b = 16` *durch Postinkrement*, `main = 0` *da Scope von lokalem* `main` *zuende, globales* `Example.main` *nicht mehr überschattet.*

iv) `Example.a = "Hallo16"` *durch Konkatenation* `Example.a += 16`, `Example.main = 0` *wie zuvor.*

Fallunterscheidungen

- Wenn „X“ dann „Y“ sonst „Z“

```
if(X) Y else Z
```

- Der Sonst-Fall (`else Z`) ist optional.
- Wir können `Y` und `Z` durch einen Code-Block aus mehreren Zeilen bestehen lassen.
- Wir können die Ausdrücke verschachteln.

Fallunterscheidungen

- Es gibt eine ternäre Kurzform:

```
X ? Y : Z
```

- Der Sonst-Fall (`: Z`) ist **nicht** optional.
- `Y` und `Z` müssen zu einem Wert evaluieren.
- Wir können die Ausdrücke verschachteln.
- Anders als `if-else` ist der Operator kein Statement! Wir müssen ihn also eingebettet verwenden.

Kurzgesagt

- *Implizit*: **byte** → **short** → **int** → **long** → **float** → **double**
  Zahlen von klein zu groß, sowie: **char** → **int**.
- *Präzedenzregeln*:
  Post vor Prä, sonst wie Arithmetik & Logik.
- *Default-Werte*:
  Zahlen und Zeichen `0`, Boolean `false`, Rest `null`.
- *Überschatten*: Lokal über Global.

Arrays

- Sind komplexe Datentypen (→ Heap).
- Jeder Datentyp kann ein (eindimensionales) Array werden:

```
int[] = new int[12];
```

- Mehrdimensionale Arrays, sind Arrays von Arrays von…
- Zugriff: ⟨`array`⟩[⟨`index`⟩]
  - Liefert ⟨`index`⟩-Element in ⟨`array`⟩.
  - Wenn Array von Array, ist ⟨`array`⟩[⟨`index`⟩] wieder ein Array.

34 Kachelland

Aufgabe 5

Sie befinden sich in Kachelland, welches auf einem schachbrettartigen Feld geplant wurde. Für diese Aufgabe betrachten wir die Fortbewegungskosten, die ein Charakter auf den jeweiligen Feldern hat, wobei wir zwei Formen des Transports unterscheiden: ohne Hilfsmittel und im Boot. Auf diese Weise definieren wir für jede Kachel des zweidimensionalen Landes wie teuer es ist sie ohne Hilfsmittel, oder mit einem Boot, zu überqueren und geben eine Zeit in Minuten an. Dafür verwenden wir ein Array mit zwei Elementen: `cell = {⟨foot⟩, ⟨boat⟩}`. Eine negative Zahl kennzeichnet, dass es unmöglich ist, das Feld mit dem Transportmittel zu überqueren. Konstruieren Sie die folgenden drei **`static`**-Methoden in Java:

1. `int[][][] newLand(int width, int height)`
2. `boolean set(int[][][] board, int x, int y, int foot, int boat)`
3. `int costs(int[][][] board, int x, int y, boolean byBoat)`

Kachelland

Auflösung 5

› Erschaffen wir ein neues Land:

```
static int[][][] newLand(int width, int height) {
   int[][][] board = new int[height][width][2];
   for(int y = 0; y < height; y++) {
      for(int x = 0; x < width; x++) {
         board[y][x] = new int[]{-1, -1};
      }
   }
   return board;
}
```

TileCountry.java

Kachelland

Auflösung 5

› Setzen wir ein Feld:

```
static boolean set(int[][][] board, int x, int y,
                   int foot, int boat) {
  if(y < 0 || x < 0 || // zu klein oder zu groß?
      y >= board.length || x >= board[y].length)
    return false;
  board[y][x] = new int[]{foot, boat};
  return true;
}
```

Kachelland

Auflösung 5

› Kosten abfragen:

```
static int costs(int[][][] board, int x, int y,
       boolean byBoat) {
   if(y < 0 || x < 0 || // zu klein oder zu groß?
       y >= board.length || x >= board[y].length)
     return -1;
   int[] cell = board[y][x];
   return byBoat ? cell[1] : cell[0];
}
```

Schleifen

- Drei Schleifenarten: **`for`**, **`while`**, **`do-while`**
- Sie sind alle gleichmächtig

```
for(int i = 3; i < 8; i++) {
   // Miau
}
```

Anzahl bekannt

```
int i = 3;
while(i < 8) {
   // Miau
   i += 1;
}
```

Kein Maximum

```
int i = 3;
do {
   // Miau
   i += 1;
} while(i < 8);
```

Mindestens 1

- Bei **`for`** sind alle drei Blöcke optional.
- Bei **`do-while`** aufpassen:
  - Semikolon!
  - Wird es wirklich mindestens ein mal durchlaufen?

Schleifen umwandeln

Aufgabe 6

Im Folgenden finden Sie zwei Code-Ausschnitte. Schreiben Sie diese so um, dass sie nur noch Schleifen der angegebenen Art verwenden, ohne dass sich das Verhalten des Codes verändert. Dabei repräsentiert `k` eine natürliche Zahl (exklusive 0), welche Sie nicht kennen.

```java
// do-while
int[] x = new int[]{17, 22, 13, 0};
for(int i = 0; i < x.length - k; i += 1) {
    System.out.println(x[i]);
}
```

```java
// for
int i = 1;
do {
    i *= k;
    k -= i;
} while(k >= 5);
System.out.println(i);
```

Schleifen umwandeln

- Aus **for** machen wir **do-while**
- Wird es überhaupt einmal ausgeführt?

```
int[] x = new int[]{17, 22, 13, 0};
for(int i = 0; i < x.length - k; i += 1)
    System.out.println(x[i]);
```

Auflösung 6

```
int[] x = new int[]{17, 22, 13, 0};
if(x.length - k <= 0) // Sonst keine Ausführung
   return;
int i = 0; // Initialisierung
do { // normale Schleife
   System.out.println(x[i]);
   i++;
} while(i < x.length - k);
```

Schleifen umwandeln

- Aus **do-while** machen wir **for**
- Wird es doch mindestens ein mal ausgeführt?
- Brauchen wir die Variablen danach noch?

Auflösung 6

```
int i = 1;
do {
  i *= k;
  k -= i;
} while(k >= 5);
System.out.println(i);
```

```
int i = 1; // Bedarf nach Schleife
if(k < 5) { // Trotzdem ausführen?
   i *= k;
   // Wie Lebenserfahrung:
   k -= i; // theoretisch unnötig
} else {
   for(; k >= 5; k -= i)
      i *= k; // oder hier k -= i
}
System.out.println(i);
```

Kurzgesagt

- Arrays sind komplexe Datentypen.
- Mehrdimensionale Arrays sind eindimensionale Arrays von eindimensionalen Arrays von…
- Die drei Schleifenarten sind gleichmächtig.
  - Maximum bekannt: `for`
  - Mindestens ein mal: `do-while`
  - Sonst: `while`

Unterprogramme

- Ein Unterprogramm lagert einen Teil des Programms aus.
- In Java: **Methoden**

```
⟨Modifikatoren⟩ ⟨Rückgabetyp⟩ ⟨Name⟩(⟨Parameterliste⟩) {
   ⟨Inhalt⟩
}
```

- Modifikatoren beeinflussen beispielsweise die Sichtbarkeit.

Ein Beispiel

```
⟨Modifikatoren⟩ ⟨Rückgabetyp⟩ ⟨Name⟩(⟨Parameterliste⟩) {
  ⟨Inhalt⟩
}
```

Modifikatoren R-Typ Name Parameterliste

```
public static String giveMeTheMiau(int times, double cuteness) {
  String out = "";
  for(int i = 0; i < times * cuteness; i++) {
    out += "Miau ";
  }
  return out;
}
```

Inhalt

Signatur: Name & Parametertypen
`giveMeTheMiau(int, double)`

Rückgabewert

- `void` kennzeichnet, dass die Methode nichts zurückgibt.
- Sonst: jeder Ausführungspfad muss einen Wert vom angegebenen Typ liefern.

```
public static int iHaveCatPics(String where) {
   if(where.equals("MIAFF"))
      return 42;
}
```

- **Verboten**, da nicht jeder Ausführungspfad `int` liefert.

Methodenaufruf

- Methoden mit `static` gehören der Klasse.
  - Wir benötigen kein Objekt.
  - In ihnen „gibt“ es *kein* `this`.
- Java sucht beim Aufruf eine Methode mit entsprechender Signatur.
  - Implizite Typkonvertierung!
- **Überladung**: Gleicher Name, unterschiedliche Signatur.

Call-By & Seiteneffekte

You can call me pingu.
Cute pingu!

- Aufruf: „Stack-Einträge“ der Parameter werden *kopiert*.
  - Primitive Datentypen: call-by-*value*
  - Komplexe Datentypen: call-by-*reference*
- Genau genommen hat Java nur „call-by-value“
- **Seiteneffekt**: Wenn nach dem Aufruf mehr als der Rückgabewert verbleibt.
  - Instanzvariable verändert sich.
  - Durch Referenz wird ein Array modifiziert.
  - Eine Nachricht fliegt durchs Internet.
  - …

Varargs

- Der letzte Parameter darf in Java ein „Vararg“ sein.
- Drei Punkte: erlaubt beliebig viele Argumente des Typs.
- Kann in der Methode als Array verwendet werden.

```
static double[] funFunFun(int factor, double... arguments) {
   for(int i = 0; i < arguments.length; i++) {
      arguments[i] *= factor;
   }
   return arguments;
}
```

Methodenausgaben

Aufgabe 7

Betrachten Sie das Java-Programm auf dem Blatt und geben Sie für die kommentierten Zeilen jeweils die Werte der Variablen `a`, `b` und `c` an (tragen Sie diese in die entsprechende Box ☐ ein). Erklären Sie die Ausgaben dabei durch eine kurze Erläuterung der beteiligten Java-Mechanismen. Relevant sind: i) Überladungen, ii) Seiteneffekte, iii) Überschattungen und iv) Standardwerte. Sie dürfen in ihrer Skizzierung die römischen Zahlen verwenden, um auf die Mechanismen hinzuweisen.

Methodenausgaben

Auflösung 7

```
public class MethodExample {
   public static char a;
   public static int b = 5;
   public static int c = 3;

   static int x(int a) { ... }
   static int x(char[] b) { ... }
   static String x() { ... }
   public static void main(...) {
      // 1)
      ...
   }
}
```

Allgemein:

- `x` wird Überladen und präsentiert drei Signaturen: `x(int)`, `x(char[])` und `x()`.
- Nur `x(char[])` hat Seiteneffekte.
  - In `x(int)` überschattet der Parameter das globale `a`.
  - In `x(char[])` wird `a` verändert.
  - In `x()` wird `b` überschattet.

Methodenausgaben

Auflösung 7

```
public class MethodExample {
   public static char a;
   public static int b = 5;
   public static int c = 3;

   static int x(int a) { ... }
   static int x(char[] b) { ... }
   static String x() { ... }
   public static void main(...) {
      // 1)
      ...
   }
}
```

Werte:

- `a` ist `0` (zugehöriges Unicode-Symbol `NULL`) da Java einer Klassenvariable vom Typ `char` diesen Wert zuweist.
- `b` ist `5` und `c` ist `3` da die Klassenvariablen so initialisiert werden.

Auflösung 7

```
public class MethodExample {
   public static char a;
   public static int b = 5;
   public static int c = 3;

   static int x(int a) { ... }
   static int x(char[] b) { ... }
   static String x() {
      String b = "Hallo";
      return b;
   }
   public static void main(...) {
      ...
      String c = x();
      // 2)
      ...
   }
}
```

Werte:

- `a` ist `0` es bleibt durch den Aufruf von `c()` unverändert.
- `b` ist `5` also ebenfalls unverändert. `c()` erzeugt ein neues `b`, welches das globale überschattet.
- `c` ist `"Hallo"`, das lokale `c` überschattet das globale. Der Wert ergibt sich durch `x()` welches das globale `b` überschattet.

Methodenausgaben

Auflösung 7

```
public class MethodExample {
   public static char a;
   public static int b = 5;
   public static int c = 3;

   static int x(int a) { ... }
   static int x(char[] b) {
      a += b[1];
      return b[0];
   }
   static String x() { ... }
   public static void main(...) {
      ...
      x(c.toCharArray());
      // 3)
      ...
   }
}
```

Werte:

- Durch `c.toCharArray()` wird `x(char[])` aufgerufen. `char[] b` überschattet das globale.
- `a` wird durch `a += b[1]` zu `'a'` (97) geändert.
- `b` ist unverändert 5, da es in `x(char[])` überschattet wird.
- `c` ist `"Hallo"`, wie zuvor.

Methodenausgaben

Auflösung 7

```
public class MethodExample {
   public static char a;
   public static int b = 5;
   public static int c = 3;

   static int x(int a) {
      a = a * a;
      return b + 2;
   }
   static int x(char[] b) { ... }
   static String x() { ... }
   public static void main(...) {
      ...
      int b = x(a);
      // 4)
   }
}
```

Werte:

- Es wird `x(int)` aufgerufen.
- `a` bleibt unverändert `'a'` (97), da der Parameter in `x(int)` das globale überschattet.
- `b` ist 7, da das lokale nun das globale überschattet und den Wert `b + 2` erhält.
- `c` ist `"Hallo"`, wie zuvor.

Reverse

Aufgabe 8

Schreiben Sie eine Methode `static void reverse (int[] arr)` welche das übergebene Array umdreht, sodass das erste Elemente nun an letzter Stelle steht, das zweite an vorletzter Stelle, … und das letzte Element nun an erster Stelle steht.

Reverse

Auflösung 8

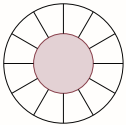
```
static void swap(int[] arr, int i, int j) {
   int tmp = arr[i];
   arr[i] = arr[j];
   arr[j] = tmp;
}

static void reverse(int[] arr) {
   for(int i = 0; i < arr.length / 2; i++) {
      swap(arr, i, arr.length - i - 1);
   }
}
```

- *Überladung:* Gleicher Name, andere Signatur.
  - *Signatur:* Name & Parametertypliste
  - Müssen zudem in selber Klasse sein (später: Vererbung)

- Beim Aufruf macht Java call-by-value:
  - Alle Parameter werden kopiert (Stack).

Allgemein

- Objekte abstrahieren Daten und Verhalten.
- Ermöglicht Wiederverwendung von Komponenten.
- Klassen definieren eine Blaupause:
  - Welche Eigenschaften haben die Daten? **Attribute**
  - Was kann mit den Daten gemacht werden? **Methoden**
- Die Attribute definieren den Zustand
- Die Methoden definieren das Verhalten



Klasse vs. Objekt

Jedes Objekt erzeugt alle Instanzvariablen „für sich“ allein.

- Klasse ist eine Blaupause
- Objekte weisen Attribute konkrete Werte zu

```
class Penguin {
   double cute;
   int age;
   String name;

   void walk(int snow) { ... }
   void peep() { ... }
   ...
}
```

Penguin udbert

cute	5.7
age	12
name	„Udi“

Penguin josie

cute	6.1
age	10
name	„Josie“

Penguin peter

cute	6
age	10
name	„Petaa“

Penguin saphira

cute	8
age	11
name	„Saph“

Der Konstruktor

Instanziierung: Erzeugen eines neuen Objektes.

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- Ein Konstruktor ist *keine* Methode.
- Unterschiede:
  - Muss heißen wie die Klasse
  - Hat keinen Rückgabetyp
  - Kann nichts zurückgeben
  - Benötigt `new` für Aufruf
  - …
- Gemeinsamkeiten:
  - Hat Parameter
  - Kann überladen werden
  - Kann Methoden aufrufen
  - Kann Objekte erzeugen
  - …
- Java erzeugt *genau dann* einen leeren Konstruktor, wenn kein expliziter Konstruktor existiert.

Der Standardkonstruktor

```
public class Supa {
   int x;
}

public class Mega {
   String name;
   public Mega(String name) {
     this.name = name;
   }
}
```

- `Supa s = new Supa();`
  Erlaubt, durch den leeren Standardkonstruktor. `s.x` hat den Standardwert `0`.
- `Mega m = new Mega();`
  Nicht erlaubt. Existiert ein expliziter Konstruktor, erstellt Java keinen leeren mehr!
- `Mega x = new Mega("Huhu");`
  Erlaubt. `x.name` ist `"Huhu"`.

Einen Konstruktor überladen

- Konstruktoren können einander mit `this` aufrufen. Dies muss dann das *erste* Statement sein. Keine Rekursion.

```
public class Car {
   int topSpeed;
   final boolean pengu;
   public Car(int top, boolean p) {
      this.topSpeed = top;
      this.pengu = p;
   }
```

```
   public Car(int top) {
      this(top, true);
      System.out.println("Piep");
   }

   public Car() { this(20); }
}
```

Car(int)

- `new Car()` : topSpeed ist 20, pengu ist `true`. Mit Piep!
- `new Car(14)` : topSpeed ist 14, pengu ist `true`. Mit Piep!
- `new Car(2, false)` : topSpeed ist 2, pengu ist `false`. Kein Piep!

Static vs. Non-Static

- Statische Variablen & Methoden sind an die Klasse gebunden.
  - Kein Objekt notwendig um Sie aufzurufen/abzugreifen.
  - Keine Instanz- sondern Klassenvariablen.
- Die Frage lautet: Ist diese Eigenschaft allen Objekten gemein?

Kameraszenario

Aufgabe 9

Sie sehen das Grundgerüst einer Kameraklasse. Bearbeiten Sie die folgenden beiden Teilaufgaben, welche aufeinander aufbauen. Sie dürfen keine (Java-)Ausnahmen verwenden.

1. Schreiben Sie einen leeren Konstruktor, welcher die von Java zugewiesenen Standardwerte explizit initialisiert. Dieser Konstruktor soll nicht von Code abseits der `Camera`-Klasse aufrufbar sein und so verhindern, dass weitere Kamera-Objekte erstellt werden können.
2. Erweitern Sie die Klasse nun in sofern, dass es immer maximal fünf Kamera-Objekte gibt, welche durch `get(int)` durch eine Zahl $x \in \{0, 1, 2, 3, 4\}$ abgefragt werden können. Dabei soll ein Kamera-Objekt erst (mit ihrem vorher erzeugten Konstruktor) erzeugt werden, wenn es durch die Zahl angefragt wird. Weiter sollen Folgeanfragen mit der Zahl dasselbe Objekt zurückliefern. Geben Sie für $x < 0$ und $x \geq 5$ den Wert `null` zurück.

Der Konstruktor

Camera.java

```
private Camera() {
   x = y = z = 0;
   recording = false;
}
```

Auflösung 9

- **private**: Damit von außen nicht zugreifbar.
- x = y = z = 0 als Kurzform für x = 0, y = 0 und z = 0.

Nur fünf Kameras

Auflösung 9

```
private static Camera[] cameras = new Camera[5];

public static Camera get(int n) {
   if(n < 0 || n >= 5) return null;
   if(cameras[n] == null)
      cameras[n] = new Camera();
   return cameras[n];
}
```

- Das Array kontrolliert den Pool verfügbarer Kameras.
- Statische Erzeuger-Methoden erlauben mehr Kontrolle.

Sichtbarkeit & Gültigkeit

- Klassen, Methoden & Attribute haben **Sichtbarkeiten**.
  - Sie können nur verwendet werden, wo sie sichtbar sind.
  - Diese Sichtbarkeit wird durch `public`, `private`, … kontrolliert.
- Variablen, … haben einen **Gültigkeitsbereich** (Scope).
  - So gibt es den Gültigkeitsbereich der Klasse und
  - Den lokalen Gültigkeitsbereich in Methoden, Blöcken, …
- Eine Variable kann z.B. sichtbar, aber gerade durch Überschattung nicht gültig sein.

Gruppierung

- Java nutzt **Packages** als Namensräume
  - Ordner der dann alle zugehörigen Java-Dateien enthält.
  - Können über `import <package>;` sichtbar gemacht werden.
  - Dabei wird `java.lang` immer importiert.
- Sichtbarkeiten interagieren auch mit Vererbung.

Scopes

- Java erlaubt vier verschiedene Sichtbarkeitsmodifikatoren:
  - *private:* Nur innerhalb der Klasse.
  - *protected:* Im gesamten Paket sowie allen Unterklassen.
  - *public:* Überall sichtbar wo Paket sichtbar.
  - *„default“:* Überall im Paket.

- *Hinweis:* Objekte einer Klasse können auf private Elemente anderer Objekte der gleichen Klasse zugreifen.

Ein Scope Beispiel

› Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

› Die Klasse `Tiger` sei im selben, `Auto` sei in einem anderen Paket, `Felspingu` erbe von `Pinguin` aber sei in einem anderem Paket:

	1	2	3	4	5
`Pinguin`	☑	☑	☑	☑	☑
`Tiger`	☑	☐	☑	☑	☑

	1	2	3	4	5
`Auto`	☑	☐	☐	☑	☐
`Felspingu`	☑	☐	☑	☑	☐

Fehlerbehandlung

- Wenn irgendetwas schief geht, wirft Java eine Ausnahme.
- Wir unterscheiden drei Arten:
  - **Error:** Katastrophe, zum Beispiel kein Speicher. Kaum behandelbar.
  - **RuntimeException:** Vorhersehbar, aber nun schwer behandelbar (wie ein negativer Array-Index, …).
  - **Exception:** Schwer vorhersehbare Ausnahme (wie Datei nicht gefunden, …).
- Normale „Exceptions“ *müssen* in Java:
  - Behandelt werden (`try-catch`), oder
  - weitergeleitet werden (`throws`).

Eskalationsfreude

- Eine Ausnahme unterbricht den Programmfluss.
- Wird sie nicht direkt behandelt, eskaliert die Ausnahme den „Call-Stack“ hinauf.
- Was über die `main`-Methode eskaliert wird von der JVM abgefangen und beendet das Programm.
- Ausnahmen sollten in Java aber nicht zur Werterückgabe verwendet werden.

Fehler finden

Aufgabe 10

Im Code finden sich **10 Fehler**, welche vom Java-Compiler erkannt werden. Identifizieren Sie **acht** dieser Fehler durch

1. die Angabe der Code-Zeilennummer und
2. eine kurze Erklärung, was den Fehler verursacht.

Tragen Sie die Informationen in die Tabelle ein. Wenn Sie mehrere Fehler in einer Code-Zeile finden, tragen sie diese separat ein — die zusätzlichen Zeilen existieren dabei nur, um Ihnen eine Selbstkorrektur zu ermöglichen. Folgefehler zählen dabei nicht als separate Fehler. Wird also beispielsweise ein Bezeichner inkorrekt initialisiert, so zählt nicht jede Verwendung als zusätzlicher Fehler sondern nur die inkorrekte Initialisierung.

Fehlersuche

Auflösung 10

1. *Zeile 2*: `void` ist kein Datentyp.
2. *Zeile 3/10*: Wenn `omega` `final` ist, kann es im Konstruktor nicht geändert werden.
3. *Zeile 6*: das Produkt aus `char` und `double` ist `double` und muss explizit zu `int` konvertiert werden.
4. *Zeile 9*: Ein Konstruktor muss genau so heißen wie die Klasse. Hier ist das `I` aber groß. Alternativ fehlt mindestens der Rückgabetyp.
5. *Zeile 13*: `double` ist ein Keyword und kann nicht als Bezeichner verwendet werden.

Fehlersuche

Auflösung 10

6. *Zeile 18*: `andromeda` ist in `iDoGood()` keine Variable.
7. *Zeile 23*: Hier fehlt das Semikolon, es ist auskommentiert.
8. *Zeile 25*: `System.Err` existiert nur kleingeschrieben als `System.err`.
9. *Zeile 30*: `global` ist in Java kein gültiges Methoden-Präfix.
10. *Zeile 31*: `iDoGood` ist ein Konstruktor und kann nur mit `new` aufgerufen werden (die `boolean`-Methode bräuchte ein Objekt).

Kurzgesagt

- Eine Klasse definiert die Blaupause für Objekte.
  - Attribute definieren den Zustand.
  - Methoden definieren den Verhalten.

- Der Konstruktor baut den initialen Zustand
  - **Instanziierung**: Erzeugen eines neuen Objektes.
  - Wenn keiner: erzeugt Java den leeren Standardkonstruktor.
  - `this` erlaubt Aufruf von Überladungen.

- Klassen, Methoden, …: **Sichtbarkeit** (`public`, …)

- **Gültigkeit**sbereich: Wo die Variablen „deklariert sind“.

Sources & current version
on GitHub

Florian Sihler

22.12.2021