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

3.1

3.2

I. Einführung
II. Aspekte der Algorithmenkonstruktion
III. Programmierung im Kleinen — Namen und Dinge
IV. Programmierung im Kleinen — Steuerung des Programmablaufs
V. Zeigervariable, Arrays und Iterationen
VI. Programmieren im Großen — Strukturierter Entwurf und Unterprogramme
VII. Einführung in die objektorientierte Programmierung (OOP)
VIII. Dynamische Datenstrukturen
IX. Weiterführende Konzepte der objektorientierten Programmierung
X. Rekursive Algorithmen
XI. Algorithmen und Zeitkomplexität
XII. Suchen und Sortieren

3.3

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

3.4

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

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

4.1 What can we say about an Algorithm?

What can we say about an Algorithm?

Totale Korrektheit

What can we say about an Algorithm?

Totale Korrektheit

1. **Termination**
   Der Algorithmus endet nach endlich vielen Schritten für jede Eingabe.

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.

5.1 But what is an Algorithm?

5.2 But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

5.3 But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

- Endlich viele wohldefinierte Einzelschritte.

But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

- Endlich viele wohldefinierte Einzelschritte.
- Kann grafisch, textuell, … erfolgen.

But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

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

6.1 Discussing an Algorithm

6.2 Discussing an Algorithm

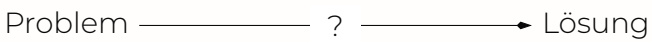
Problem

6.3 Discussing an Algorithm

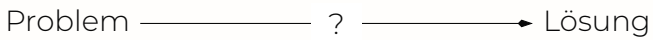
Problem

Lösung

6.4 Discussing an Algorithm



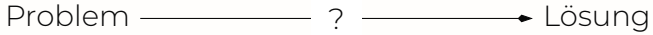
6.5 Discussing an Algorithm



- Problemspezifikation

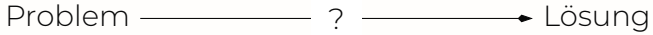
Was meinen Sie mit „schnell“?

Discussing an Algorithm



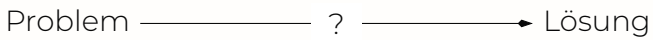
- Problemspezifikation — Was meinen Sie mit „schnell“?
- Problemabstraktion — Was ist gegeben, was ist gesucht?

Discussing an Algorithm



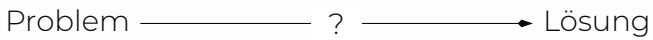
- Problemspezifikation — Was meinen Sie mit „schnell“?
- Problemabstraktion — Was ist gegeben, was ist gesucht?
- Algorithmenentwurf — Wie kommen wir von gegeben zu gesucht?

6.8 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?

6.9 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?

7.1 Kurzgesagt

- Totale Korrektheit

- Totale Korrektheit
  - *Terminiertheit:* Endliche Schritte für jede Eingabe.

Kurzgesagt

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

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

8.1 Primitive Datentypen

8.2 Primitive Datentypen

boolean

8.3 Primitive Datentypen

boolean

byte **short** **int** **long** **float** **double**

8.4 Primitive Datentypen

boolean

byte **short** **int** **long** **float** **double**

char

8.5 Primitive Datentypen

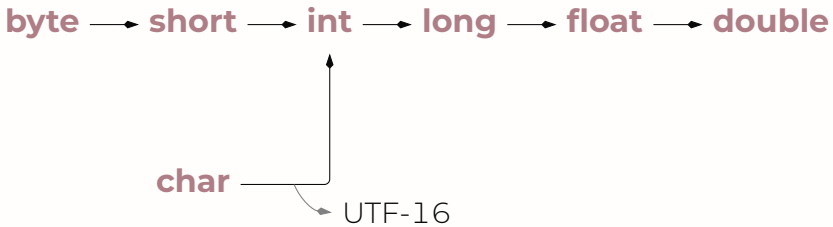
boolean

byte → **short** → **int** → **long** → **float** → **double**

char

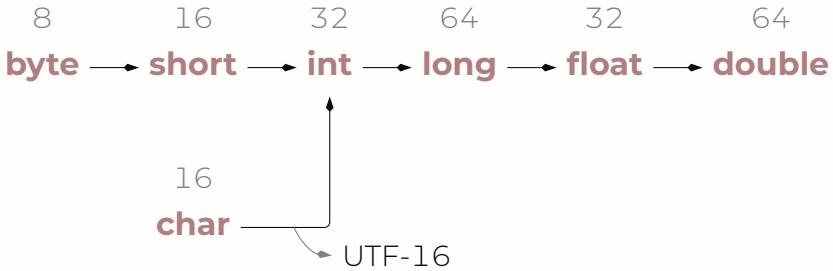
8.6 Primitive Datentypen

boolean



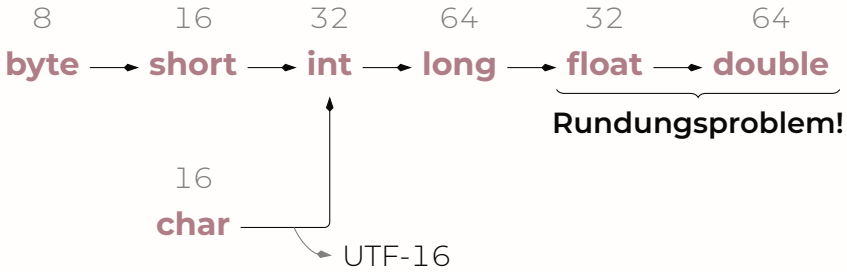
8.7 Primitive Datentypen

boolean



8.8 Primitive Datentypen

boolean



8.9 Primitive Datentypen

boolean

8	16	32	64	32	64
byte	**short**	**int**	**long**	**float**	**double**

+, /, ... ⟶ int

Rundungsproblem!

16

char → UTF-16

9.1 Präzedenzregeln

9.2 Präzedenzregeln

- Eine Auswahl:

9.3 Präzedenzregeln

- Eine Auswahl:

  a++, **a**- -

9.4 Präzedenzregeln

- Eine Auswahl:

9.5 Präzedenzregeln

- Eine Auswahl:

9.6 Präzedenzregeln

- Eine Auswahl:

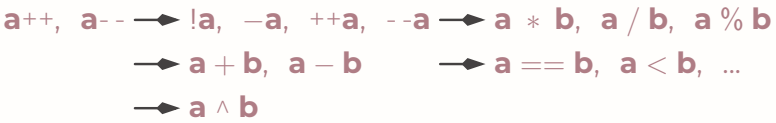
9.7 Präzedenzregeln

- Eine Auswahl:

$$a{+}{+},\ a{-}{-} \rightarrow\ !a,\ -a,\ {+}{+}a,\ {-}{-}a \rightarrow\ a * b,\ a / b,\ a \,\%\, b$$
$$\rightarrow\ a + b,\ a - b \qquad \rightarrow\ a == b,\ a < b,\ \ldots$$

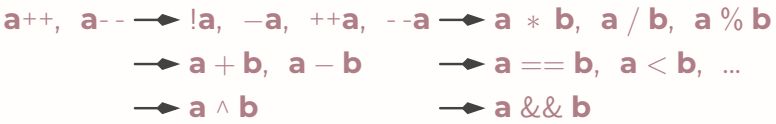
9.8 Präzedenzregeln

- Eine Auswahl:



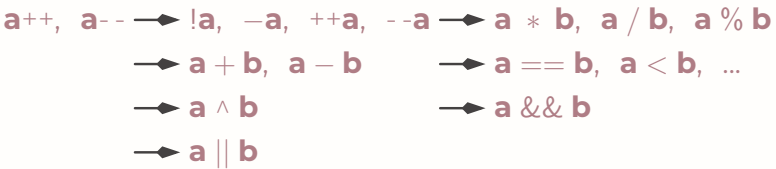
9.9 Präzedenzregeln

- Eine Auswahl:



9.10 Präzedenzregeln

- Eine Auswahl:



10.1 Variablendefinition

10.2 Variablendefinition

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

10.3 Variablendefinition

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

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

10.4 Variablendefinition

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

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

Zuweisung: Wertzuweisung, bzw. Ersetzung des Wertes:

10.5 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';
```

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:

10.7 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';
```

10.8 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.

11.1 Standardwerte

11.2 Standardwerte

- Standardwerte für:

11.3 Standardwerte

- Standardwerte für:
  - Klassen-Variablen (`static`)

11.4 Standardwerte

- Standardwerte für:
  - Klassen-Variablen (`static`)
  - Instanz-Variablen

11.5 Standardwerte

- Standardwerte für:
  - Klassen-Variablen (`static`)
  - Instanz-Variablen
  - Array-Komponenten (wie bei `new int[]`)

11.6 Standardwerte

- Standardwerte für:
  - Klassen-Variablen (`static`)
  - Instanz-Variablen
  - Array-Komponenten (wie bei `new int[]`)
- Der Standardwert ist „Null“:

11.7 Standardwerte

- Standardwerte für:
  - Klassen-Variablen (`static`)
  - Instanz-Variablen
  - Array-Komponenten (wie bei `new int[]`)
- Der Standardwert ist „Null“:
  - `byte`, `short`, `int`, `long` ⟶ `0`

11.8 Standardwerte

- 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`

11.9 Standardwerte

- 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“)

Standardwerte

- 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`

Standardwerte

- 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`

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`

12.1 Blöcke

12.2 Blöcke

- Methodenaufrufe erzeugen „Stack-Frames“.

12.3 Blöcke

- Methodenaufrufe erzeugen „Stack-Frames“.
- Code-Blöcke arbeiten vergleichbar.

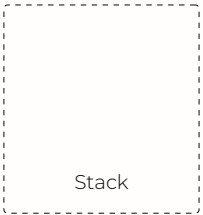
12.4 Blöcke

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

12.5 Blöcke

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

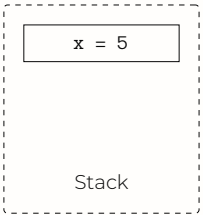
```
int x = 5;
{
   double b = 12.5;
   x = x + (int) b;
}
int y = x;
```



Blöcke

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

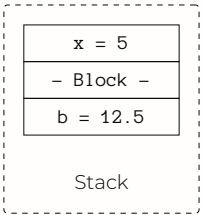
```
int x = 5;
{
   double b = 12.5;
   x = x + (int) b;
}
int y = x;
```



12.7 Blöcke

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

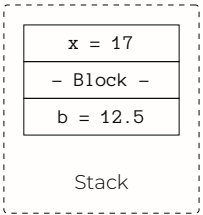
```
int x = 5;
{
   double b = 12.5;
   x = x + (int) b;
}
int y = x;
```



12.8 Blöcke

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

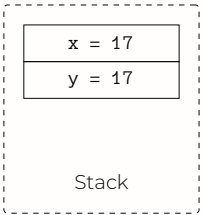
```
int x = 5;
{
   double b = 12.5;
   x = x + (int) b;
}
int y = x;
```



12.9 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;
```



13.1 Überschatten

13.2 Überschatten

13.3 Überschatten

- Instanz-/Klassenvariable mit gleichem Bezeichner wie lokale Variable.

Überschatten

- Instanz-/Klassenvariable mit gleichem Bezeichner wie lokale Variable.
- Der lokale Bezeichner überschattet den „globalen“.

Überschatten

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

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);
  }
}
```

13.7 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
```

14.1 Fallunterscheidungen

14.2 Fallunterscheidungen

- Wenn „X“ dann „Y“ sonst „Z“

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

14.3 Fallunterscheidungen

- Wenn „X“ dann „Y“ sonst „Z“

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

- Der Sonst-Fall (`else Z`) ist optional.

14.4 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.

14.5 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.

15.1 Fallunterscheidungen

Fallunterscheidungen

- Es gibt eine ternäre Kurzform:

```
X ? Y : Z
```

Fallunterscheidungen

- Es gibt eine ternäre Kurzform:

```
X ? Y : Z
```

- Der Sonst-Fall (`: Z`) ist **nicht** optional.

Fallunterscheidungen

- Es gibt eine ternäre Kurzform:

```
X ? Y : Z
```

- Der Sonst-Fall (`: Z`) ist **nicht** optional.
- `Y` und `Z` müssen zu einem Wert evaluieren.

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.

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.

16.1 Kurzgesagt

Kurzgesagt

- *Implizit*: **byte** → **short** → **int** → **long** → **float** → **double**
  Zahlen von klein zu groß, sowie: **char** → **int**.

Kurzgesagt

- *Implizit*: **byte** → **short** → **int** → **long** → **float** → **double**
  Zahlen von klein zu groß, sowie: **char** → **int**.

- *Präzedenzregeln*:
  Post vor Prä, sonst wie Arithmetik & Logik.

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`.

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.

17.1 Arrays

17.2 Arrays

- Sind komplexe Datentypen (→ Heap).

17.3 Arrays

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

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

17.4 Arrays

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

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

- Mehrdimensionale Arrays, sind Arrays von Arrays von…

17.5 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`⟩]

17.6 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`⟩.

17.7 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.

18.1 Schleifen

18.2 Schleifen

- Drei Schleifenarten: `for`, `while`, `do-while`

18.3 Schleifen

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

18.4 Schleifen

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

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

18.5 Schleifen

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

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

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

18.6 Schleifen

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

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

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

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

18.7 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;
}
```

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

18.8 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);
```

18.9 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

18.10 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.

18.11 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:

18.12 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!

18.13 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?

- Arrays sind komplexe Datentypen.

- Arrays sind komplexe Datentypen.
- Mehrdimensionale Arrays sind eindimensionale Arrays von eindimensionalen Arrays von…

Kurzgesagt

- Arrays sind komplexe Datentypen.
- Mehrdimensionale Arrays sind eindimensionale Arrays von eindimensionalen Arrays von…
- Die drei Schleifenarten sind gleichmächtig.

Kurzgesagt

- Arrays sind komplexe Datentypen.
- Mehrdimensionale Arrays sind eindimensionale Arrays von eindimensionalen Arrays von…
- Die drei Schleifenarten sind gleichmächtig.
  - Maximum bekannt: **`for`**

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`

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`

20.1 Unterprogramme

20.2 Unterprogramme

- Ein Unterprogramm lagert einen Teil des Programms aus.

20.3 Unterprogramme

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

20.4 Unterprogramme

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

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

20.5 Unterprogramme

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

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

- Modifikatoren beeinflussen beispielsweise die Sichtbarkeit.

21.1 Ein Beispiel

21.2 Ein Beispiel

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

21.3 Ein Beispiel

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

```
public static String giveMeTheMiau(int times, double cuteness) {



}
```

21.4 Ein Beispiel

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

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

21.5 Ein Beispiel

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

Modifikatoren

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

21.6 Ein Beispiel

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

Modifikatoren R-Typ

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

21.7 Ein Beispiel

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

Modifikatoren R-Typ Name

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

21.8 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;
}
```

21.9 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

21.10 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)`

22.1 Rückgabewert

22.2 Rückgabewert

- `void` kennzeichnet, dass die Methode nichts zurückgibt.

22.3 Rückgabewert

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

22.4 Rückgabewert

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

```
public static int iHaveCatPics(String where) {


}
```

22.5 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;
}
```

22.6 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.

23.1 Methodenaufruf

23.2 Methodenaufruf

- Methoden mit **`static`** gehören der Klasse.

23.3 Methodenaufruf

- Methoden mit `static` gehören der Klasse.
  - Wir benötigen kein Objekt.

23.4 Methodenaufruf

- Methoden mit `static` gehören der Klasse.
  - Wir benötigen kein Objekt.
  - In ihnen „gibt“ es *kein* `this`.

23.5 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.

23.6 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!

23.7 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.

24.1 Call-By & Seiteneffekte

24.2 Call-By & Seiteneffekte

You can call me pingu.
Cute pingu!

Call-By & Seiteneffekte

You can call me pingu.
Cute pingu!

- Aufruf: „Stack-Einträge“ der Parameter werden *kopiert*.

Call-By & Seiteneffekte

You can call me pingu.
Cute pingu!

- Aufruf: „Stack-Einträge“ der Parameter werden *kopiert*.
  - Primitive Datentypen: call-by-*value*

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*

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“

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.

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.

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.

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.

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.
  - …

25.1 Varargs

25.2 Varargs

- Der letzte Parameter darf in Java ein „Vararg“ sein.

25.3 Varargs

- Der letzte Parameter darf in Java ein „Vararg“ sein.
- Drei Punkte: erlaubt beliebig viele Argumente des Typs.

25.4 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.

25.5 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) {



}
```

25.6 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;
   }

}
```

25.7 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;
}
```

26.1 Kurzgesagt

- *Überladung:* Gleicher Name, andere Signatur.

- *Überladung:* Gleicher Name, andere Signatur.
  - *Signatur:* Name & Parametertypliste

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

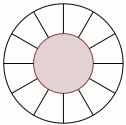
- *Überladung:* Gleicher Name, andere Signatur.
  - *Signatur:* Name & Parametertypliste
  - Müssen zudem in selber Klasse sein (später: Vererbung)

- Beim Aufruf macht Java call-by-value:

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

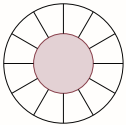
- Beim Aufruf macht Java call-by-value:
  - Alle Parameter werden kopiert (Stack).

27.1 Allgemein



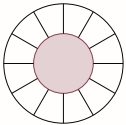
27.2 Allgemein

- Objekte abstrahieren Daten und Verhalten.



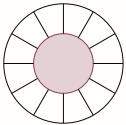
27.3 Allgemein

- Objekte abstrahieren Daten und Verhalten.
- Ermöglicht Wiederverwendung von Komponenten.



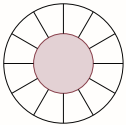
27.4 Allgemein

- Objekte abstrahieren Daten und Verhalten.
- Ermöglicht Wiederverwendung von Komponenten.
- Klassen definieren eine Blaupause:



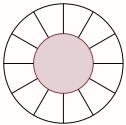
Allgemein

- Objekte abstrahieren Daten und Verhalten.
- Ermöglicht Wiederverwendung von Komponenten.
- Klassen definieren eine Blaupause:
  - Welche Eigenschaften haben die Daten? **Attribute**



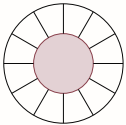
27.6 Allgemein

- 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**



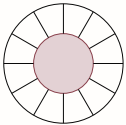
Allgemein

- 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



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



28.1 Klasse vs. Objekt

28.2 Klasse vs. Objekt

- Klasse ist eine Blaupause

28.3 Klasse vs. Objekt

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

28.4 Klasse vs. Objekt

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

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

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

28.5 Klasse vs. Objekt

- 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“

28.6 Klasse vs. Objekt

- 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“

28.7 Klasse vs. Objekt

- 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 saphira

cute	8
age	11
name	„Saph“

28.8 Klasse vs. Objekt

- 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“

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“

29.1 Der Konstruktor

29.2 Der Konstruktor

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).

29.3 Der Konstruktor

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- Ein Konstruktor ist *keine* Methode.

29.4 Der Konstruktor

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- Ein Konstruktor ist *keine* Methode.
- Unterschiede:

Der Konstruktor

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- Ein Konstruktor ist *keine* Methode.
- Unterschiede:
  - Muss heißen wie die Klasse

29.6 Der Konstruktor

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- Ein Konstruktor ist *keine* Methode.
- Unterschiede:
  - Muss heißen wie die Klasse
  - Hat keinen Rückgabetyp

29.7 Der Konstruktor

- Erlaubt für Werte beim Erzeugen („initialer Zustand“).
- Ein Konstruktor ist *keine* Methode.
- Unterschiede:
  - Muss heißen wie die Klasse
  - Hat keinen Rückgabetyp
  - Kann nichts zurückgeben

29.8 Der Konstruktor

- 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

Der Konstruktor

- 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
  - …

Der Konstruktor

- 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:

Der Konstruktor

- 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

Der Konstruktor

- 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

Der Konstruktor

- 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

Der Konstruktor

- 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

Der Konstruktor

- 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
  - …

Der Konstruktor

- 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 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.

30.1 Der Standardkonstruktor

30.2 Der Standardkonstruktor

```
public class Supa {
   int x;
}
```

30.3 Der Standardkonstruktor

```
public class Supa {
   int x;
}
```

```
> Supa s = new Supa();
```

30.4 Der Standardkonstruktor

```
public class Supa {
   int x;
}
```

- `Supa s = new Supa();`
  Erlaubt, durch den leeren Standardkonstruktor. `s.x` hat den Standardwert `0`.

30.5 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`.

30.6 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();`

30.7 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!

30.8 Der Standardkonstruktor

```java
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");`

30.9 Der Standardkonstruktor

```java
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"`.

31.1 Einen Konstruktor überladen

31.2 Einen Konstruktor überladen

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

31.3 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;

                                    }
```

31.4 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;
   }                                          }
```

31.5 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");
   }

}
```

31.6 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); }
}
```

31.7 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)

31.8 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");
   }
                  Car(int)
   public Car() { this(20); }
}
```

31.9 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()`:

31.10 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!

31.11 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)` :

31.12 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");
   }
                  Car(int)
   public Car() { this(20); }
}
```

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

31.13 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");
   }
                 Car(int)
   public Car() { this(20); }
}
```

- `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)` :

31.14 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");
   }
                  Car(int)
   public Car() { this(20); }
}
```

- `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!

32.1 Static vs. Non-Static

32.2 Static vs. Non-Static

- Statische Variablen & Methoden sind an die Klasse gebunden.

32.3 Static vs. Non-Static

- Statische Variablen & Methoden sind an die Klasse gebunden.
  - Kein Objekt notwendig um Sie aufzurufen/abzugreifen.

32.4 Static vs. Non-Static

- Statische Variablen & Methoden sind an die Klasse gebunden.
  - Kein Objekt notwendig um Sie aufzurufen/abzugreifen.
  - Keine Instanz- sondern Klassenvariablen.

32.5 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?

33.1 Sichtbarkeit & Gültigkeit

33.2 Sichtbarkeit & Gültigkeit

- Klassen, Methoden & Attribute haben **Sichtbarkeiten**.

33.3 Sichtbarkeit & Gültigkeit

- Klassen, Methoden & Attribute haben **Sichtbarkeiten**.
  - Sie können nur verwendet werden, wo sie sichtbar sind.

33.4 Sichtbarkeit & Gültigkeit

- Klassen, Methoden & Attribute haben **Sichtbarkeiten**.
  - Sie können nur verwendet werden, wo sie sichtbar sind.
  - Diese Sichtbarkeit wird durch `public`, `private`, … kontrolliert.

33.5 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).

33.6 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

33.7 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, …

33.8 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.

34.1 Gruppierung

34.2 Gruppierung

- Java nutzt **Packages** als Namensräume

34.3 Gruppierung

- Java nutzt **Packages** als Namensräume
  - Ordner der dann alle zugehörigen Java-Dateien enthält.

34.4 Gruppierung

- Java nutzt **Packages** als Namensräume
  - Ordner der dann alle zugehörigen Java-Dateien enthält.
  - Können über `import` ‹package› ; sichtbar gemacht werden.

34.5 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.

34.6 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.

35.1 Scopes

35.2 Scopes

- Java erlaubt vier verschiedene Sichtbarkeitsmodifikatoren:

35.3 Scopes

- Java erlaubt vier verschiedene Sichtbarkeitsmodifikatoren:
  - *private:* Nur innerhalb der Klasse.

35.4 Scopes

- Java erlaubt vier verschiedene Sichtbarkeitsmodifikatoren:
  - *private:* Nur innerhalb der Klasse.
  - *protected:* Im gesamten Paket sowie allen Unterklassen.

35.5 Scopes

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

35.6 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.

35.7 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.

36.1 Ein Scope Beispiel

36.2 Ein Scope Beispiel

- Es gibt vier Sichtbarkeitsbereiche für Variablen.

36.3 Ein Scope Beispiel

- Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Pinguin { // 1. global



}
```

36.4 Ein Scope Beispiel

- Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse



}
```

36.5 Ein Scope Beispiel

- Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt


}
```

36.6 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)

}
```

36.7 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
}
```

36.8 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:

36.9 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`	☑	☐	☑	☑	☐

37.1 Fehlerbehandlung

37.2 Fehlerbehandlung

- Wenn irgendetwas schief geht, wirft Java eine Ausnahme.

37.3 Fehlerbehandlung

- Wenn irgendetwas schief geht, wirft Java eine Ausnahme.
- Wir unterscheiden drei Arten:

37.4 Fehlerbehandlung

- Wenn irgendetwas schief geht, wirft Java eine Ausnahme.
- Wir unterscheiden drei Arten:
  - **Error:** Katastrophe, zum Beispiel kein Speicher. Kaum behandelbar.

37.5 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, …).

37.6 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, …).

37.7 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:

37.8 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

37.9 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`).

38.1 Eskalationsfreude

38.2 Eskalationsfreude

- Eine Ausnahme unterbricht den Programmfluss.

38.3 Eskalationsfreude

- Eine Ausnahme unterbricht den Programmfluss.
- Wird sie nicht direkt behandelt, eskaliert die Ausnahme den „Call-Stack“ hinauf.

38.4 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.

38.5 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.

39.1 Kurzgesagt

Kurzgesagt

- Eine Klasse definiert die Blaupause für Objekte.

Kurzgesagt

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

Kurzgesagt

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

Kurzgesagt

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

- Der Konstruktor baut den initialen Zustand

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.

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.

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.

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`, …)

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