

Se greit Eidn!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

3.5

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?



4.2

What can we say about an Algorithm?

Totale Korrektheit



4.3

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

Algorithmusanalyse

Aufgabe 1



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



5.2

Algorithmusanalyse

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

Aufgabe 1



5.3

Algorithmusanalyse

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

```
public static long L(int n) {
```

```
}
```

Aufgabe 1



5.4

Algorithmusanalyse

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

```
public static long L(int n) {  
    if(n == 0 || n == 1) return 1;
```

```
}
```

Aufgabe 1



5.5

Algorithmusanalyse

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

```
public static long L(int n) {  
    if(n == 0 || n == 1) return 1;  
    int l1 = 1, l2 = 1;  
  
    }  
}
```

Aufgabe 1



5.6

Algorithmusanalyse

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

```
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++) {  
  
    }  
}
```

Aufgabe 1



5.7

Algorithmusanalyse

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

```
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;  
  
    }  
}
```

Aufgabe 1



5.8

Algorithmusanalyse

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

```
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;  
    }  
}
```

Aufgabe 1



5.9

Algorithmusanalyse

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

```
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;  
    }  
}
```

Aufgabe 1



5.10

Algorithmusanalyse

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

```
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;  
}
```

Aufgabe 1



6.1

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;  
}
```

Auflösung 1

6.2

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

Auflösung 1



6.3

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

Auflösung 1

2. Partielle Korrektheit



6.4

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
 - > n ist konstant.

Auflösung 1

2. Partielle Korrektheit



```
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
 - > n ist konstant.
 - > Für $n = 0$, bzw. $n = 1$ trivial.

2. Partielle Korrektheit

```
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

- › 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

7.1

Algorithmusanalyse

1. Terminiertheit
2. Partielle Korrektheit

```
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



7.2

Algorithmusanalyse

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

```
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

7.3

Algorithmusanalyse

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

```
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

7.4

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 $n = 2$.

Auflösung 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;
}

```

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:
 $l2 = 1 + 1 + 1 = 3 = L(2)$ und $l1 = 1 = L(1)$. ✓

7.6

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:
 $l2 = 1 + 1 + 1 = 3 = L(2)$ und $l1 = 1 = L(1)$. ✓

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

Auflösung 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;
}

```

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:
 $l2 = 1 + 1 + 1 = 3 = L(2)$ und $l1 = 1 = L(1)$. ✓

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

IS: Mit $n \rightarrow n + 1$ wird durch $l1 = l2$ nämlich nach der IH

7.8

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:
 $l2 = 1 + 1 + 1 = 3 = L(2)$ und $l1 = 1 = L(1)$. ✓

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

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

Auflösung 1



8.1

But what is an Algorithm?



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



8.2

But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.



8.3

But what is an Algorithm?

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems.

- › Endlich viele wohldefinierte Einzelschritte.



8.4

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.**

9.1

Discussing an Algorithm



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



9.2

Discussing an Algorithm

Problem



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



9.3

Discussing an Algorithm

Problem

Lösung



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



9.4

Discussing an Algorithm

Problem ————— ? —————> Lösung



9.5

Discussing an Algorithm

Problem ————— ? —————> Lösung

> Problemspezifikation

Was meinen Sie mit „schnell“?



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



9.6

Discussing an Algorithm

Problem ————— ? —————> Lösung

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



9.7

Discussing an Algorithm

Problem ————— ? —————> Lösung

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



9.8

Discussing an Algorithm

Problem ————— ? —————> Lösung

- > 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?



9.9

Discussing an Algorithm

Problem ————— ? —————> Lösung

- > 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?

10.1

Bauen Sie mir das

- › Spezifikation
- › Abstraktion
- › Algorithmus
- › Korrektheit
- › Analyse

Aufgabe 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



10.2

Bauen Sie mir das

- › Spezifikation
- › Abstraktion
- › Algorithmus
- › Korrektheit
- › Analyse

Aufgabe 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 viele Plätzchen aus den gegebenen Ressourcen backen kann.



10.3

Bauen Sie mir das



- › Spezifikation
- › Abstraktion
- › Algorithmus
- › Korrektheit
- › Analyse

Aufgabe 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 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

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.



11.3

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.



11.4

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.



11.5

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.



11.6

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.



11.7

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.



11.8

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 Süßungstoffes** 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.



11.9

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.



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.

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.

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

> **Plätzchen:** e (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.

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

- > **Plätzchen:** (Kokos oder Zimt).
- > **Plätzchenmenge:** 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.

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

- > **Plätzchen:** (Kokos oder Zimt).
- > **Plätzchenmenge:** Ich habe 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.

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

- > **Plätzchen:** (Kokos oder Zimt).
- > **Plätzchenmenge:** Ich 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 (Kokos oder Zimt) 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.

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

- > **Plätzchen:** (Kokos oder Zimt).
- > **Plätzchenmenge:** Ich habe sie alle gleich gern, 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 an den jeweiligen Stoffen zur Verfügung stellen — die ich dann weitergeben. 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.

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

- > **Plätzchen:**
- > **Plätzchenmenge:**
- > **Zulieferer:**
- > **Ressource:**
- > **Ressourcenmenge:**
- > **Rezept:**

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- > **Plätzchen:**
- > **Plätzchenmenge:**
- > **Zulieferer:**
- > **Ressource:**
- > **Ressourcenmenge:**
- > **Rezept:**
- > **Komponente:**

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- > **Plätzchen:**
- > **Plätzchenmenge:**
- > **Zulieferer:**
- > **Ressource:**
- > **Ressourcenmenge:**
- > **Rezept:**
- > **Komponente:**
- > **Maximal viele Plätzchen aus gegebenen Ressourcen:**

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- › **Plätzchen:** Produkt, an dessen Menge wir interessiert sind.
- › **Plätzchenmenge:** Wie viele Plätzchen ich gleich gern habe, werde ich sie alle immer zum selben Preis verkaufen. Geld an sich ist auch gar nicht das Problem, jeder können mir meine Zulieferer immer nur eine begrenzte Menge an den jeweiligen Stoffen zur Verfügung stellen — die ich weitergeben. Die habe ich. Weiter habe ich pro Jahr eine handvoll an Rezepten, die immer eine bestimmte Anzahl eines Mehls, eines Süßungsstoffes und einer weiteren Komponente pro Plätzchen benötigen.
- › **Zulieferer:**
- › **Ressource:**
- › **Ressourcenmenge:**
- › **Rezept:**
- › **Komponente:**
- › **Maximal viele Plätzchen aus gegebenen Ressourcen:** Wie viele Plätzchen ich ausfinden

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- > **Plätzchen:** Produkt, an dessen Menge wir interessiert sind.
- > **Plätzchenmenge:** Eine natürliche Zahl $P \geq 0$.
- > **Zulieferer:**
- > **Ressource:**
- > **Ressourcenmenge:**
- > **Rezept:**
- > **Komponente:**
- > **Maximal viele Plätzchen aus gegebenen Ressourcen:**

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- › **Plätzchen:** Produkt, an dessen Menge wir interessiert sind.
- › **Plätzchenmenge:** Eine natürliche Zahl $P \geq 0$.
- › **Zulieferer:** Stellt Ressourcen für Produktion zur Verfügung.
- › **Ressource:**
- › **Ressourcenmenge:**
- › **Rezept:**
- › **Komponente:**
- › **Maximal viele Plätzchen aus gegebenen Ressourcen:**

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- > **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:** Die habe ich. Weiter habe ich pro Jahr
- > **Rezept:** eine handvoll an Rezepten, die immer eine bestimmte Anzahl
- > **Komponente:** eines Mehls, eines Süßungsstoffes und einer weiteren Komponente pro
- > **Maximal viele Plätzchen aus gegebenen Ressourcen:** Plätzchen benötigen.

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

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

- > **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:**
- > **Komponente:**
- > **Maximal viele Plätzchen aus gegebenen Ressourcen:**

wie ich insgesamt immer maximal viel Plätzchen aus den gegebenen Ressourcen backen kann.

Guten Tag ich heiße Günther und ich bin Besitzer eines Plätzchenladens. Dafür verwende ich verschiedene Mehle (Weizen oder Dinkel).

- › **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:**
- › **Maximal viele Plätzchen aus gegebenen Ressourcen:**

Ressourcen backen kann.

Guten Tag ich heiße Günther und ich bin Besitzer eines Plätzchenladens. Dafür verwende ich verschiedene Mehle (Weizen oder Dinkel).

- > **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:**

Ressourcen backen kann.

- › **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: $\bigvee_{j \in k} R_{\ell_j}$.

12.1

Abstraktion

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



12.2

Abstraktion

> Gegeben:

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



12.3

Abstraktion

- › Gegeben:
 - › Ressourcenmengen r_1, r_2, \dots, r_n

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



- › Gegeben:
 - › Ressourcenmengen r_1, r_2, \dots, r_n
 - › Rezepte $R = \{(k_3, k_7, k_1), (k_4, k_2, k_9), \dots\}$, sei $|R| = m$

12.5

Abstraktion

- › Gegeben:
 - › Ressourcenmengen r_1, r_2, \dots, r_n
 - › Rezepte $R = \{(k_3, k_7, k_1), (k_4, k_2, k_9), \dots\}$, sei $|R| = m$
- › Gesucht:



12.6

Abstraktion

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



13.1

Algorithmenkonstruktion

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

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



13.2

Algorithmenkonstruktion

› Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$

Auflösung 2



13.3

Algorithmenkonstruktion

- > Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - > 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.

Auflösung 2



13.4

Algorithmenkonstruktion

- > Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - > 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.

Auflösung 2



13.5

Algorithmenkonstruktion

- Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - 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.

Auflösung 2



13.6

Algorithmenkonstruktion

- Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - 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.

13.7

Algorithmenkonstruktion

- > Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - > 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:

13.8

Algorithmenkonstruktion

- > Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - > 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“ $A = (0, 0, \dots, 0)$ mit $|A| = m$.

13.9

Algorithmenkonstruktion

- › Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - › 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“ $A = (0, 0, \dots, 0)$ mit $|A| = m$.
 - › Finde maximale Anwendbarkeit $\max a_i = R_{\max, i}$ von Rezept i durch Inkrement bis nicht mehr anwendbar.

Algorithmenkonstruktion

- > Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - > 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“ $A = (0, 0, \dots, 0)$ mit $|A| = m$.
 - > Finde maximale Anwendbarkeit $\max a_i = R_{\max, i}$ von Rezept i durch Inkrement bis nicht mehr anwendbar.
 - > Probiere alle $A = (a_1, \dots, a_m)$ mit $0 \leq a_i \leq R_{\max, i}$.

- > Erkenntnisse: r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \rightarrow k$
 - > 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“ $A = (0, 0, \dots, 0)$ mit $|A| = m$.
 - > Finde maximale Anwendbarkeit $\max a_i = R_{\max, i}$ von Rezept i durch Inkrement bis nicht mehr anwendbar.
 - > Probiere alle $A = (a_1, \dots, a_m)$ mit $0 \leq a_i \leq R_{\max, i}$.
 - > Speichere maximale anwendbare Summe $\sum_{i=0}^m a_i$.

14.1

Algorithmenkonstruktion



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

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



14.2

Algorithmenkonstruktion



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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

Auflösung 2



Algorithmen



> Konstrukte

> Arrays & Iteration

> Unterprogramme

> OOP



14.3

Algorithmenkonstruktion



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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

1 for $i \leftarrow 0$ **to** m **step 1 do**

6 end

Auflösung 2



Algorithmen



> Konstrukte



Arrays & Iteration



Unterprogramme



OOP



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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

1 for $i \leftarrow 0$ **to** m **step** 1 **do**

// Wende R_i genau a_i mal an.

6 end

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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

1 for $i \leftarrow 0$ **to** m **step** 1 **do**

// Wende R_i genau a_i mal an.

2 $r \leftarrow (k_a, k_b, k_c) \leftarrow R_i$

6 end

14.6

Algorithmenkonstruktion



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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

1 for $i \leftarrow 0$ **to** m **step** 1 **do**

// Wende R_i genau a_i mal an.

2 $r \leftarrow (k_a, k_b, k_c) \leftarrow R_i;$

3 $r_a \leftarrow r_a - a_i \cdot k_a;$

6 end

Auflösung 2



Algorithmen



> Konstrukte



Arrays & Iteration



Unterprogramme



OOP



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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Wende  $R_i$  genau  $a_i$  mal an.
2    $r \leftarrow (k_a, k_b, k_c) \leftarrow R_i$ 
3    $r_a \leftarrow r_a - a_i \cdot k_a$ ;
4    $r_b \leftarrow r_b - a_i \cdot k_b$ ;

6 end

```

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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

```
1 for  $i \leftarrow 0$  to  $m$  step  $1$  do  
    // Wende  $R_i$  genau  $a_i$  mal an.  
2     $r \leftarrow (k_a, k_b, k_c) \leftarrow R_i$   
3     $r_a \leftarrow r_a - a_i \cdot k_a$   
4     $r_b \leftarrow r_b - a_i \cdot k_b$   
5     $r_c \leftarrow r_c - a_i \cdot k_c$   
6 end
```


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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Wende  $R_i$  genau  $a_i$  mal an.
2    $r \leftarrow (k_a, k_b, k_c) \leftarrow R_i$ 
3    $r_a \leftarrow r_a - a_i \cdot k_a$ ;
4    $r_b \leftarrow r_b - a_i \cdot k_b$ ;
5    $r_c \leftarrow r_c - a_i \cdot k_c$ ;
    // Oder: „Reduce  $r_a, r_b, r_c$  by  $a_i$  times  $k_a, k_b, k_c$ 
    // respectively“
6 end

```

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

> Ist $A = (a_1, \dots, a_m)$ anwendbar?

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Wende  $R_i$  genau  $a_i$  mal an.
2    $r \leftarrow (k_a, k_b, k_c) \leftarrow R_i$ 
3    $r_a \leftarrow r_a - a_i \cdot k_a$ ;
4    $r_b \leftarrow r_b - a_i \cdot k_b$ ;
5    $r_c \leftarrow r_c - a_i \cdot k_c$ ;
    // Oder: „Reduce  $r_a, r_b, r_c$  by  $a_i$  times  $k_a, k_b, k_c$ 
    // respectively“
6 end
7 return  $\forall_{i \in \{1, \dots, n\}} r_i \geq 0$ ;
  
```

15.1

Algorithmenkonstruktion



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

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



15.2

Algorithmenkonstruktion



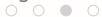
r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \longrightarrow k$

> Was sind die Maxima $R_{\max, i}$?

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



15.3

Algorithmenkonstruktion



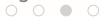
r_1, \dots, r_n und $R = \{(k_1, k_2, k_3)_1, \dots, (\dots)_m\} \longrightarrow k$

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

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



15.4

Algorithmenkonstruktion



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

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

1 for $i \leftarrow 0$ to m step 1 do

|

6 end

Auflösung 2



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

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

1 for $i \leftarrow 0$ to m step 1 do

// Inkrementiere a_i bis nicht mehr anwendbar.

6 end

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

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

1 for $i \leftarrow 0$ **to** m **step** 1 **do**

// Inkrementiere a_i bis nicht mehr anwendbar.

2 $A \leftarrow (0, \dots, 0);$

3 $j \leftarrow 1;$

6 end

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

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

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Inkrementiere  $a_i$  bis nicht mehr anwendbar.
2    $A \leftarrow (0, \dots, 0)$ ;
3    $j \leftarrow 1$ ;
4   while  $\text{anwendbar}(A_{i \leftarrow j})$  do  $j \leftarrow j + 1$ ;

6 end

```

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

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

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Inkrementiere  $a_i$  bis nicht mehr anwendbar.
2    $A \leftarrow (0, \dots, 0)$ ;
3    $j \leftarrow 1$ ;
4   while  $\text{anwendbar}(A_{i \leftarrow j})$  do  $j \leftarrow j + 1$ ;
    //  $j$  ist eins höher als das anwendbare Maximum.
6 end

```

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

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

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Inkrementiere  $a_i$  bis nicht mehr anwendbar.
2    $A \leftarrow (0, \dots, 0)$ ;
3    $j \leftarrow 1$ ;
4   while  $\text{anwendbar}(A_{i \leftarrow j})$  do  $j \leftarrow j + 1$ ;
    //  $j$  ist eins höher als das anwendbare Maximum.
5    $R_{\max, i} \leftarrow j - 1$ ;
6 end

```

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

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

```

1 for  $i \leftarrow 0$  to  $m$  step  $1$  do
    // Inkrementiere  $a_i$  bis nicht mehr anwendbar.
2    $A \leftarrow (0, \dots, 0)$ ;
3    $j \leftarrow 1$ ;
4   while  $\text{anwendbar}(A_{i \leftarrow j})$  do  $j \leftarrow j + 1$ ;
    //  $j$  ist eins höher als das anwendbare Maximum.
5    $R_{\max,i} \leftarrow j - 1$ ;
6 end
7 return  $\forall_{i \in \{1, \dots, n\}} r_i \geq 0$ ;
  
```

16.1

Algorithmenkonstruktion

IV

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

Auflösung 2



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



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

- › Und nun das Testen aller möglichen Anwendungen:

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

- › Und nun das Testen aller möglichen Anwendungen:
 - 1 guatzla \leftarrow 0;

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

> Und nun das Testen aller möglichen Anwendungen:

1 guatzla $\leftarrow 0$;

2 **for** $A \leftarrow (0, \dots, 0)$ **to** $(R_{\max,1}, \dots, R_{\max,m})$ **do**

7 **end**

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

> Und nun das Testen aller möglichen Anwendungen:

```
1 guatzla  $\leftarrow$  0;  
2 for  $A \leftarrow (0, \dots, 0)$  to  $(R_{\max,1}, \dots, R_{\max,m})$  do  
3   if anwendbar( $A$ ) then  
6   end  
7 end
```

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

> Und nun das Testen aller möglichen Anwendungen:

```
1 guatzla  $\leftarrow$  0;  
2 for  $A \leftarrow (0, \dots, 0)$  to  $(R_{\max,1}, \dots, R_{\max,m})$  do  
3   if anwendbar( $A$ ) then  
4     tmp  $\leftarrow \sum_{i \leftarrow 0}^m a_i$   
6   end  
7 end
```

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

> Und nun das Testen aller möglichen Anwendungen:

```
1 guatzla  $\leftarrow$  0;  
2 for  $A \leftarrow (0, \dots, 0)$  to  $(R_{\max,1}, \dots, R_{\max,m})$  do  
3   if anwendbar( $A$ ) then  
4     tmp  $\leftarrow \sum_{i \leftarrow 0}^m a_i$ ;  
5     guatzla  $\leftarrow \max\{a_i, \text{guatzla}\}$ ;  
6   end  
7 end
```

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

> Und nun das Testen aller möglichen Anwendungen:

```

1 guatzla  $\leftarrow$  0;
2 for  $A \leftarrow (0, \dots, 0)$  to  $(R_{\max,1}, \dots, R_{\max,m})$  do
3   if anwendbar( $A$ ) then
4     |   tmp  $\leftarrow \sum_{i \leftarrow 0}^m a_i$ ;
5     |   guatzla  $\leftarrow \max\{a_i, \text{guatzla}\}$ ;
6   end
7 end
8 return guatzla;

```

17.1

Kurzgesagt

Algorithmen



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



- › Totale Korrektheit

- › Totale Korrektheit
 - › *Terminiertheit*:

Endliche Schritte für jede Eingabe.

› 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.

18.1

Primitive Datentypen

18.2

Primitive Datentypen

boolean



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



18.3 Primitive Datentypen

boolean

byte

short

int

long

float

double



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



18.4 Primitive Datentypen

boolean

byte

short

int

long

float

double

char

18.5 Primitive Datentypen

boolean

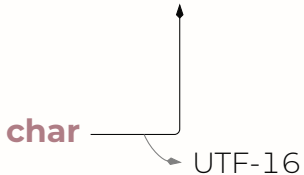
byte → **short** → **int** → **long** → **float** → **double**

char

18.6 Primitive Datentypen

boolean

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



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme

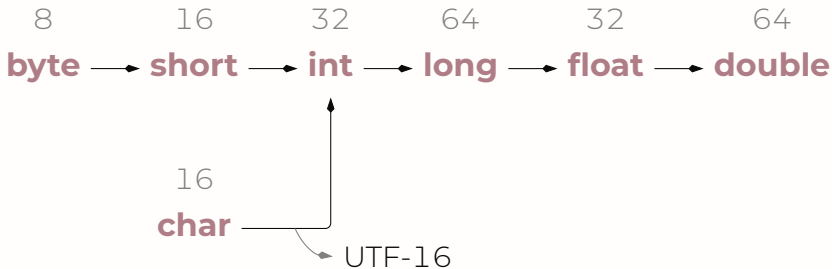


OOP



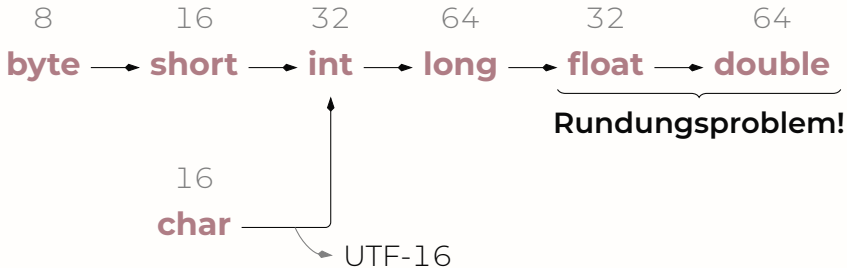
18.7 Primitive Datentypen

boolean



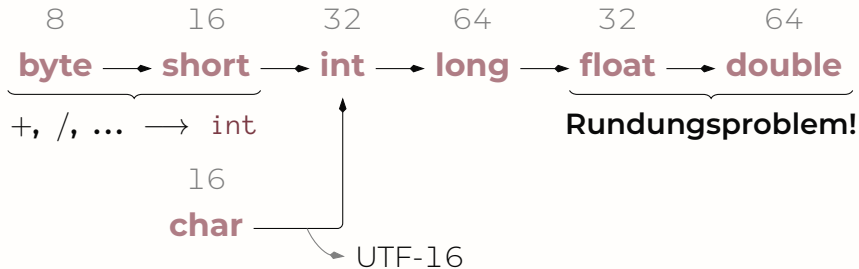
18.8 Primitive Datentypen

boolean



18.9 Primitive Datentypen

boolean



19.1

Präzedenzregeln

19.2

Präzedenzregeln

› Eine Auswahl:

19.3

Präzedenzregeln

› Eine Auswahl:

a++, **a--**



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



19.4

Präzedenzregeln

› Eine Auswahl:

$a++$, $a-- \rightarrow !a, -a, ++a, --a$



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



19.5

Präzedenzregeln

› Eine Auswahl:

$a++$, $a-- \rightarrow !a$, $-a$, $++a$, $--a \rightarrow a * b$, a / b , $a \% b$



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



19.6

Präzedenzregeln

> Eine Auswahl:

$a++, a-- \rightarrow !a, -a, ++a, --a \rightarrow a * b, a / b, a \% b$
 $\rightarrow a + b, a - b$



19.7

Präzedenzregeln

> Eine Auswahl:

$a++, a-- \rightarrow !a, -a, ++a, --a \rightarrow a * b, a / b, a \% b$
 $\rightarrow a + b, a - b \quad \rightarrow a == b, a < b, \dots$



19.8

Präzedenzregeln

› Eine Auswahl:

$a++, a-- \rightarrow !a, -a, ++a, --a \rightarrow a * b, a / b, a \% b$
 $\rightarrow a + b, a - b \quad \rightarrow a == b, a < b, \dots$
 $\rightarrow a \wedge b$



19.9

Präzedenzregeln

› Eine Auswahl:

$a++, a-- \rightarrow !a, -a, ++a, --a \rightarrow a * b, a / b, a \% b$
 $\rightarrow a + b, a - b \quad \rightarrow a == b, a < b, \dots$
 $\rightarrow a \wedge b \quad \rightarrow a \&\& b$



19.10

Präzedenzregeln

› Eine Auswahl:

$a++, a-- \rightarrow !a, -a, ++a, --a \rightarrow a * b, a / b, a \% b$
 $\rightarrow a + b, a - b \quad \rightarrow a == b, a < b, \dots$
 $\rightarrow a \wedge b \quad \rightarrow a \&\& b$
 $\rightarrow a || b$



20.1

Typbestimmung

Aufgabe 3

20.2

Typbestimmung

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

Aufgabe 3



> 14 - 3D * 2 / 12

21.3

Typbestimmung



> $\overbrace{14}^{\text{int}} - \overbrace{3D}^{\text{double}} * \overbrace{2}^{\text{int}} / \overbrace{12}^{\text{int}}$

Auflösung 3



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



double
int double int int
> 14 - 3D * 2 / 12

> $\overbrace{14}^{\text{int}} - \overbrace{3D * 2 / 12}^{\text{double}}$

double

int double int int

Auflösung 3

11

$$\begin{array}{ccccccc} & & \text{double} & & & & \\ & & \text{[]} & & & & \\ \text{int} & \text{double} & \text{int} & & \text{int} & & \text{double} \\ \text{[]} & \text{[]} & \text{[]} & & \text{[]} & & \text{[]} \\ > 14 - 3D * 2 / 12 = 13.5 \\ & & \text{double} & & & & \end{array}$$

> $\overbrace{14}^{\text{int}} - \underbrace{\overbrace{3D}^{\text{double}} * \overbrace{2}^{\text{int}} / \overbrace{12}^{\text{int}}}_{\text{double}}}^{\text{double}} = \overbrace{13.5}^{\text{double}}$

> (char) ((byte) 'a' + 3) + "☪Sonne"

21.8

Typbestimmung



Auflösung 3

> $\overbrace{14}^{\text{int}} - \overbrace{3D * 2}^{\text{double int}} / \overbrace{12}^{\text{int}} = \overbrace{13.5}^{\text{double}}$

> (char) ((byte) $\overbrace{'a'}^{\text{char}} + \overbrace{3}^{\text{int}}$) + $\overbrace{"_Sonne"}^{\text{String}}$



$$> \overbrace{14}^{\text{int}} - \overbrace{3D * 2 / 12}^{\text{double}} = \overbrace{13.5}^{\text{double}}$$

$$> (\text{char}) (\underbrace{((\text{byte}) 'a' + 3)}_{\text{byte}}) + \overbrace{"_Sonne"}^{\text{String}}$$

> $\overbrace{14}^{\text{int}} - \overbrace{\overbrace{3D}^{\text{double}} * \overbrace{2}^{\text{int}} / \overbrace{12}^{\text{int}}}^{\text{double}} = \overbrace{13.5}^{\text{double}}$

> $(\text{char}) \underbrace{((\text{byte}) \overbrace{'a'}^{\text{char}} + \overbrace{3}^{\text{int}})}_{\text{byte}} + \overbrace{"_Sonne"}^{\text{String}}$

int

$$> \overbrace{14}^{\text{int}} - \overbrace{3D}^{\text{double}} * \overbrace{2}^{\text{int}} / \overbrace{12}^{\text{int}} = \overbrace{13.5}^{\text{double}}$$

$$> (\text{char}) \left(\overbrace{((\text{byte}) 'a' + 3)}^{\text{byte}} \right) + \overbrace{"_Sonne"}^{\text{String}}$$

$$> \overbrace{14}^{\text{int}} - \overbrace{3D * 2 / 12}^{\text{double}} = \overbrace{13.5}^{\text{double}}$$

$$> (\text{char}) \underbrace{((\text{byte}) \overbrace{'a'}^{\text{char}} + \overbrace{3}^{\text{int}})}_{\text{int}} + \overbrace{"_Sonne"}^{\text{String}} = \overbrace{"d_Sonne"}^{\text{String}}$$


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

```
> int14 - int2 + (int3 > int9 || booleantrue ^ booleanfalse ? String"Hallo" : String"Welt")
```

> $\overbrace{14}^{\text{int}} - \overbrace{2}^{\text{int}} + (\overbrace{3}^{\text{int}} > \overbrace{9}^{\text{int}}) \parallel \underbrace{\overbrace{\text{true}}^{\text{boolean}} \wedge \overbrace{\text{false}}^{\text{boolean}}}_{\text{boolean}} ? \overbrace{\text{"Hallo"}}^{\text{String}} : \overbrace{\text{"Welt"}}^{\text{String}})$

> $\overbrace{14}^{\text{int}} - \overbrace{2}^{\text{int}} + (\underbrace{\overbrace{3}^{\text{int}} > \overbrace{9}^{\text{int}}}_{\text{boolean}} \parallel \underbrace{\text{true} \wedge \text{false}}_{\text{boolean}} ? \overbrace{\text{"Hallo"}}^{\text{String}} : \overbrace{\text{"Welt"}}^{\text{String}})$

$$\begin{array}{c}
 \text{String} \\
 \hline
 > \overbrace{14}^{\text{int}} - \overbrace{2}^{\text{int}} + \underbrace{\left(\overbrace{3}^{\text{int}} > \overbrace{9}^{\text{int}} \right)}_{\text{boolean}} \parallel \underbrace{\left(\text{true} \wedge \text{false} \right)}_{\text{boolean}} ? \overbrace{\text{"Hallo"}}^{\text{String}} : \overbrace{\text{"Welt"}}^{\text{String}}
 \end{array}$$

$$\begin{array}{c}
 \text{int} \qquad \qquad \qquad \text{String} \\
 \begin{array}{c} \text{int} \quad \text{int} \\ \boxed{14 - 2} \end{array} + \overbrace{\begin{array}{c} \text{int} \quad \text{int} \quad \text{boolean} \quad \text{boolean} \quad \text{String} \quad \text{String} \\ \boxed{3 > 9} \quad || \quad \boxed{\text{true} \wedge \text{false}} \quad ? \quad \boxed{\text{"Hallo"}} : \boxed{\text{"Welt"}} \end{array}}^{\text{String}} \\
 \underbrace{\qquad \qquad \qquad \text{boolean} \qquad \qquad \qquad \text{boolean}}_{\text{boolean}}
 \end{array}$$

```

      int
    [ 14 - 2 ] + ( [ 3 > 9 ] || [ true ^ false ] ? [ "Hallo" ] : [ "Welt" ] )
      int   int   int  int   boolean  boolean   String   String
                [ boolean ]
                [ boolean ]
                [ boolean ]
      String
    = "12Hallo"

```

```

      int                                     String
    [int] [int] [int] [int] [boolean] [boolean] [String] [String]
> 14 - 2 + (3 > 9 || true ^ false ? "Hallo" : "Welt")
      [boolean] [boolean]
      [boolean]
      String
    = "12Hallo"

```

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

```

      int                                     String
    [int] [int] [int] [int] [boolean] [boolean] [String] [String]
> 14 - 2 + (3 > 9 || true ^ false ? "Hallo" : "Welt")
      [boolean] [boolean]
      [boolean]
      String
= "12Hallo"

```

```

    int int int double float
> 7 / 2 - 3 * 2.5 + 2f

```

```

      int                                     String
    [int] [int] [int] [int] [boolean] [boolean] [String] [String]
> 14 - 2 + (3 > 9 || true ^ false ? "Hallo" : "Welt")
      [boolean] [boolean]
      [boolean]
      String
    = "12Hallo"

```

```

      int int int double float
    [int] [int] [int] [double] [float]
> 7 / 2 - 3 * 2.5 + 2f
      [int] [double]

```

```

      int                                     String
    [int] [int] [int] [int] [boolean] [boolean] [String] [String]
> 14 - 2 + (3 > 9 || true ^ false ? "Hallo" : "Welt")
      [boolean] [boolean]
      [boolean]
      String
= "12Hallo"

```

```

      double
    [int] [int] [int] [double] [float]
> 7 / 2 - 3 * 2.5 + 2f
    [int] [double]

```

Auflösung 3

111

$$\begin{array}{ccccccc} & & \text{double} & & & & \\ & \text{int} & \text{int} & \text{int} & \text{double} & \text{float} & \text{double} \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ > & 7 & / & 2 & - & 3 * & 2.5 & + & 2f & = & -2.5 \\ & \uparrow & & & & & & & & & \\ & \text{int} & & & \text{double} & & & & & & \end{array}$$

23.1

Variablendefinition

23.2

Variablendefinition

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



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



23.3

Variablendefinition

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

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



23.4

Variablendefinition

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

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

Zuweisung: Wertzuweisung, bzw. Ersetzung des Wertes:



23.5

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';
```



23.6

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:



23.7

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';
```

23.8

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.



24.1

Standardwerte



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



24.2

Standardwerte

› Standardwerte für:



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



24.3

Standardwerte

- › Standardwerte für:
 - › Klassen-Variablen (**static**)



24.4

Standardwerte

- › Standardwerte für:
 - › Klassen-Variablen (**static**)
 - › Instanz-Variablen



24.5

Standardwerte

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



24.6

Standardwerte

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



24.7

Standardwerte

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



24.8

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



24.9

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 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 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 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**

25.1

Blöcke



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



25.2

Blöcke

- › Methodenaufrufe erzeugen „Stack-Frames“.



25.3

Blöcke

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

25.4

Blöcke

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

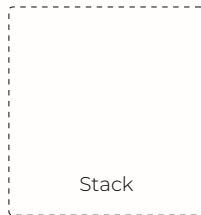


25.5

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;
```

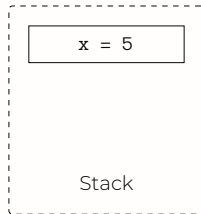


25.6

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;
```

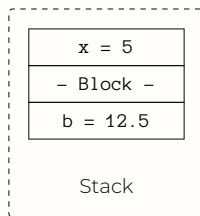


25.7

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;
```

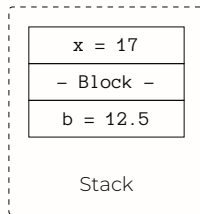


25.8

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;
```

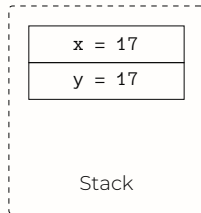


25.9

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;
```



26.1

Überschatten



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme

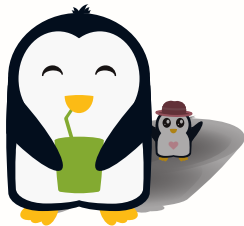


OOP



26.2

Überschatten



26.3

Überschatten

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



26.4

Überschatten

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



26.5

Überschatten

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



26.6

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);  
    }  
}
```

26.7

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
```

27.1

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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```



```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```

```
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)  
    }  
}
```



```
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)  
    }  
}
```

```
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

```
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

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

```
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

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

```
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

- 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).

```
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

- 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).

```
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

- 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.

```
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

- 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.


```
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

- 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 Konkatination Example.a += 16, Example.main = 0 wie zuvor.

30.1

Fallunterscheidungen



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



30.2

Fallunterscheidungen

> Wenn „X“ dann „Y“ sonst „Z“

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



30.3

Fallunterscheidungen

- › Wenn „X“ dann „Y“ sonst „Z“

if(X) Y **else** Z

- › Der Sonst-Fall (**else** Z) ist optional.



30.4

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.

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.

- › Es gibt eine ternäre Kurzform:

$X ? Y : Z$

- › Es gibt eine ternäre Kurzform:

$$X \text{ ? } Y \text{ : } Z$$

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

- › Es gibt eine ternäre Kurzform:

$$X \text{ ? } Y \text{ : } Z$$

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

- › Es gibt eine ternäre Kurzform:

$$X \text{ ? } Y \text{ : } Z$$

- › Der Sonst-Fall (: Z) ist **nicht** optional.
- › Y und Z müssen zu einem Wert evaluieren.
- › Wir können die Ausdrücke verschachteln.

- › Es gibt eine ternäre Kurzform:

$$X \text{ ? } Y \text{ : } Z$$

- › 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.

32.1

Kurzgesagt

Konstrukte

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

- › *Implizit:* **byte** → **short** → **int** → **long** → **float** → **double**
Zahlen von klein zu groß, sowie: **char** → **int**.
- › *Präzedenzregeln:*
Post vor Prä, sonst wie Arithmetik & Logik.

- › *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**.

- › *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.

33.1

Arrays



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



33.2

Arrays

- › Sind komplexe Datentypen (→ Heap).

33.3

Arrays

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

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

33.4

Arrays

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

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

- › Mehrdimensionale Arrays, sind Arrays von Arrays von...

33.5

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>]`

33.6

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>`.

33.7

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.

34.1

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)`

35.1

Kachelland



Auflösung 5

35.2

Kachelland



- › Erschaffen wir ein neues Land:

Auflösung 5

 TileCountry.java

35.3

Kachelland



- Erschaffen wir ein neues Land:

```
static int[][][] newLand(int width, int height) {
```

}

 TileCountry.java

- › Erschaffen wir ein neues Land:

```
static int[][][] newLand(int width, int height) {  
    int[][][] board = new int[height][width][2];  
  
}
```

- › 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++) {  
  
    }  
}
```

- › 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++) {  
  
        }  
    }  
}
```

 TileCountry.java

- › 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};  
        }  
    }  
}
```

TileCountry.java

- › 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

36.1

Kachelland



Auflösung 5

- › Setzen wir ein Feld:

› Setzen wir ein Feld:

```
static boolean set(int[][][] board, int x, int y,  
                  int foot, int boat) {
```

```
}
```

› 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;

}
```

› 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};
}
```

› 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;
}
```


- › Kosten abfragen:

> Kosten abfragen:

```
static int costs(int[][][] board, int x, int y,  
                boolean byBoat) {
```

```
}
```

> 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;  
  
}
```

> 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];
}
```

> 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];
}
```

38.1

Schleifen



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



38.2

Schleifen

- › Drei Schleifenarten: **for**, **while**, **do-while**



38.3

Schleifen

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

38.4

Schleifen

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

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

38.5

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;  
}
```

38.6

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);
```

38.7

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);
```

38.8

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);
```

38.9

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

38.10

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.

38.11

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:

38.12

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!

38.13

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?

39.1

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.

Aufgabe 6

```
// 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]);
}
```

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

40.1

Schleifen umwandeln



Auflösung 6



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



40.2

Schleifen umwandeln



> Aus **for** machen wir **do-while**

Auflösung 6



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



40.3

Schleifen umwandeln



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

Auflösung 6



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



- > 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]);
```



- > 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]);
```



```
int[] x = new int[]{17, 22, 13, 0};
```

- > 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]);
```



```
int[] x = new int[]{17, 22, 13, 0};  
if(x.length - k <= 0) // Sonst keine Ausführung  
    return;
```

- > 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]);
```



```
int[] x = new int[]{17, 22, 13, 0};  
if(x.length - k <= 0) // Sonst keine Ausführung  
    return;  
int i = 0; // Initialisierung
```

- > 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]);
```



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

- > 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]);
```



```
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);
```


> Aus **do-while** machen wir **for**

41.3

Schleifen umwandeln



- > Aus **do-while** machen wir **for**
- > Wird es doch mindestens ein mal ausgeführt?

Auflösung 6



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



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

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

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



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

```
int i = 1; // Bedarf nach Schleife
```

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



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

```
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?  
  
} else {  
  
}  
System.out.println(i);
```

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

```
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;
} else {
    // ...
}
System.out.println(i);
```

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

```
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 {
}
System.out.println(i);
```

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

```
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)
    }
System.out.println(i);
```


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

```
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);
```

42.1

Kurzgesagt

Arrays & Iteration

- › Arrays sind komplexe Datentypen.

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

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

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

- › 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**

- › 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**

43.1

Unterprogramme

43.2

Unterprogramme

- › Ein Unterprogramm lagert einen Teil des Programms aus.

43.3

Unterprogramme

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

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

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

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

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

- › Modifikatoren beeinflussen beispielsweise die Sichtbarkeit.

44.1

Ein Beispiel



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



44.2

Ein Beispiel

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

44.3

Ein Beispiel

```
<Modifikatoren> <Rückgabetyp> <Name>(<Parameterliste>) {  
    <Inhalt>  
}
```

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


44.4

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;  
}
```

44.5

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;  
}
```

44.6

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;  
}
```

44.7

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;  
}
```

44.8

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;  
}
```

44.9

Ein Beispiel

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

	Modifikatoren	R-Typ	Name	Parameterliste
	<code>public static</code>	<code>String</code>	<code>giveMeTheMiau</code>	<code>(int times, double cuteness)</code>
Inhalt {	<code>String out = "";</code>			
	<code>for(int i = 0; i < times * cuteness; i++) {</code>			
	<code> out += "Miau_";</code>			
	<code>}</code>			
	<code>return out;</code>			
	<code>}</code>			

44.10

Ein Beispiel

```

<Modifikatoren> <Rückgabetyp> <Name>(<Parameterliste>) {
    <Inhalt>
}

```

	Modifikatoren	R-Typ	Name	Parameterliste
	<code>public static</code>	<code>String</code>	<code>giveMeTheMiau</code>	<code>(int times, double cuteness)</code>
Inhalt	{			
	<code>String out = "";</code>			
	<code>for(int i = 0; i < times * cuteness; i++) {</code>			
	<code>out += "Miau_";</code>			
			<code>}</code>	
	<code>return</code>	<code>out;</code>		
	<code>}</code>			

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

45.1

Rückgabewert

45.2

Rückgabewert

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

45.3

Rückgabewert

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

45.4

Rückgabewert

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

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

45.5

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;  
}
```

45.6

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.

46.1

Methodenaufruf

46.2

Methodenaufruf

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

46.3

Methodenaufruf

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

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

- 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.

- › 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!

- › 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.

47.1

Call-By & Seiteneffekte



Call-By & Seiteneffekte

You can call me pingu.
Cute pingu!



47.3

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“

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.

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.



- › 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.



- › 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.



- › 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.
 - › ...

48.1

Varargs



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



48.2

Varargs

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



48.3

Varargs

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



48.4

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.

48.5

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) {  
  
  
  
  
  
  
  
  
  
}
```

48.6

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;  
    }  
}
```

48.7

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;  
}
```

49.1

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.


```
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:

```
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)  
        ...  
    }  
}
```

```
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:

```
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()**.

```
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.

```
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**.

```
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.


```
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.

```
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:

```
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)  
        ...  
    }  
}
```

```
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.

```
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.

```
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:

```
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)  
        ...  
    }  
}
```

```
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.


```
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.

```
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.

```
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)  
        ...  
    }  
}
```

```
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:

```
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.

```
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.

```
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.

```
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.


```
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:

```
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)  
    }  
}
```

```
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.

```
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.

```
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.

```
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.

55.1

Reverse

Aufgabe 8

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

56.1

Reverse



Auflösung 8

```
static void reverse(int[] arr) {  
  
}
```

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

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

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

```
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);  
    }  
}
```

57.1

Kurzgesagt

Unterprogramme

› *Überladung*: Gleicher Name, andere Signatur.

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

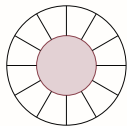
- › *Überladung*: Gleicher Name, andere Signatur.
 - › *Signatur*: Name & Parametertypliste
 - › Müssen zudem in selber Klasse sein (später: Vererbung)

- › *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).

58.1

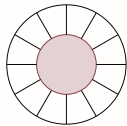
Allgemein



58.2

Allgemein

- › Objekte abstrahieren Daten und Verhalten.



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



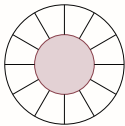
OOP



58.3

Allgemein

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



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



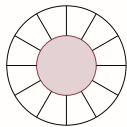
OOP



58.4

Allgemein

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



Algorithmen



Konstrukte



Arrays & Iteration



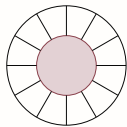
Unterprogramme



OOP



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

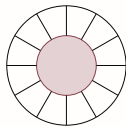
Attribute

58.6

Allgemein

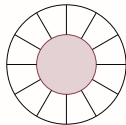
- › Objekte abstrahieren Daten und Verhalten.
- › Ermöglicht Wiederverwendung von Komponenten.
- › Klassen definieren eine Blaupause:
 - › Welche Eigenschaften haben die Daten?
 - › Was kann mit den Daten gemacht werden?

Attribute
Methoden



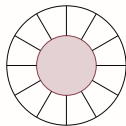
- › Objekte abstrahieren Daten und Verhalten.
- › Ermöglicht Wiederverwendung von Komponenten.
- › Klassen definieren eine Blaupause:
 - › Welche Eigenschaften haben die Daten?
 - › Was kann mit den Daten gemacht werden?
- › Die Attribute definieren den Zustand

Attribute
Methoden



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

Attribute
Methoden



59.1

Klasse vs. Objekt

59.2

Klasse vs. Objekt

- › Klasse ist eine Blaupause

59.3

Klasse vs. Objekt

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

59.4

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() { ... }  
    ...  
}
```



59.5

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“

59.6

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“

59.7

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“

59.8

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“

59.9

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“



60.1

Der Konstruktor

60.2

Der Konstruktor

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

60.3

Der Konstruktor

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

60.4

Der Konstruktor

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

60.5

Der Konstruktor

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

60.6

Der Konstruktor

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

60.7

Der Konstruktor

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

60.8

Der Konstruktor

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

60.9

Der Konstruktor

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

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

61.1

Der Standardkonstruktor

61.2

Der Standardkonstruktor

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


61.3

Der Standardkonstruktor

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

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

61.4

Der Standardkonstruktor

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

> `Supa s = new Supa();`

Erlaubt, durch den leeren Standardkonstruktor. `s.x` hat den Standardwert `0`.

61.5

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`.

61.6

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();`

61.7

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!

61.8

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");`

61.9

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".

62.1

Einen Konstruktor überladen

62.2

Einen Konstruktor überladen

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

62.3

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;
```

```
}
```

62.4

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;  
    }  
}
```

62.5

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");  
}
```

62.6

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); }
```

62.7

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


62.8

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); }  
}
```




62.9

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

A curved arrow points from the `this(20);` line in the `Car()` constructor to the `Car(int)` constructor definition.

› `new Car()`:

62.10

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); }  
}
```

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

62.11

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); }  
}
```

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

62.12

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");
}

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

A curved arrow points from the `Car(20);` line in the `Car()` constructor to the `this(top, true);` line in the `Car(int top)` constructor, indicating a recursive call to the same class's constructor.

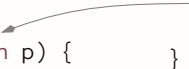
- > `new Car()`: topSpeed ist 20, pengu ist true. Mit Piep!
- > `new Car(14)`: topSpeed ist 14, pengu ist true. Mit Piep!

62.13

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



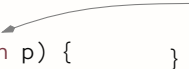
- > `new Car()`: topSpeed ist 20, pengu ist true. Mit Piep!
- > `new Car(14)`: topSpeed ist 14, pengu ist true. Mit Piep!
- > `new Car(2, false)`:

62.14

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

A curved arrow points from the `this(20);` line in the `Car()` constructor to the `this(top, true);` line in the `Car(int top)` constructor, indicating a recursive call.

- > `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!

63.1

Static vs. Non-Static

63.2

Static vs. Non-Static

- › Statische Variablen & Methoden sind an die Klasse gebunden.



63.3

Static vs. Non-Static

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

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

- › 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?

64.1

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.

65.1

Der Konstruktor

Auflösung 9

```
private Camera() {  
  
}
```

65.3

Der Konstruktor

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

Auflösung 9



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



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

> **private**: Damit von außen nicht zugreifbar.


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

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

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

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

66.1

Nur fünf Kameras

Auflösung 9

66.2

Nur fünf Kameras

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

Auflösung 9

66.3

Nur fünf Kameras

```
private static Camera[] cameras = new Camera[5];  
  
public static Camera get(int n) {  
  
  
  
  
}
```

Auflösung 9

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

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

}
```

```
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();
}
```

```
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];
}
```



```
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.

```
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.

67.1

Sichtbarkeit & Gültigkeit

67.2

Sichtbarkeit & Gültigkeit

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



67.3

Sichtbarkeit & Gültigkeit

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



67.4

Sichtbarkeit & Gültigkeit

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



67.5

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).



67.6

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



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, ...

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.

68.1

Gruppierung

68.2

Gruppierung

- › Java nutzt **Packages** als Namensräume



Algorithmen



Konstrukte



Arrays & Iteration



Unterprogramme



OOP



68.3

Gruppierung

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

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

- 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.

- › 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.

69.1

Scopes

- › Java erlaubt vier verschiedene Sichtbarkeitsmodifikatoren:

69.3

Scopes

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

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

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

- 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.

- 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.

70.1

Ein Scope Beispiel

70.2

Ein Scope Beispiel

- › Es gibt vier Sichtbarkeitsbereiche für Variablen.

70.3 Ein Scope Beispiel

Ein Scope Beispiel

- › Es gibt vier Sichtbarkeitsbereiche für Variablen.

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

$$\}$$

70.4

Ein Scope Beispiel

- > Es gibt vier Sichtbarkeitsbereiche für Variablen.

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

70.5

Ein Scope Beispiel

- > Es gibt vier Sichtbarkeitsbereiche für Variablen.

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

70.6

Ein Scope Beispiel

- > Es gibt vier Sichtbarkeitsbereiche für Variablen.

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

70.7

Ein Scope Beispiel

- › Es gibt vier Sichtbarkeitsbereiche für Variablen.

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

70.8

Ein Scope Beispiel

- Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Penguin { // 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 **Penguin** aber sei in einem anderem Paket:

70.9

Ein Scope Beispiel

- > Es gibt vier Sichtbarkeitsbereiche für Variablen.

```
public class Penguin { // 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	✓	□	✓	✓	□

71.1

Fehlerbehandlung

71.2

Fehlerbehandlung

- › Wenn irgendetwas schief geht, wirft Java eine Ausnahme.

71.3

Fehlerbehandlung

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

71.4

Fehlerbehandlung

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

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, ...).

71.6

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, ...).

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:

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

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**).

72.1

Eskalationsfreude

72.2

Eskalationsfreude

- › Eine Ausnahme unterbricht den Programmfluss.

72.3

Eskalationsfreude

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

- › 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.

- › 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.

73.1

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-Zeilenummer 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.

74.1

Fehlersuche

Auflösung 10

Fehlersuche

1. Zeile 2: `void` ist kein Datentyp.

Fehlersuche

1. Zeile 2: `void` ist kein Datentyp.
2. Zeile 3/10: Wenn `omega` `final` ist, kann es im Konstruktor nicht geändert werden.

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.

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 Rückgabotyp.

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 Rückgabotyp.
5. Zeile 13: `double` ist ein Keyword und kann nicht als Bezeichner verwendet werden.

75.1

Fehlersuche



Auflösung 10

6. Zeile 18: `andromeda` ist in `iDoGood()` keine Variable.

- 6. Zeile 18: `andromeda` ist in `iDoGood()` keine Variable.
- 7. Zeile 23: Hier fehlt das Semikolon, es ist auskommentiert.

- 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`.

- 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.

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).

76.1

Kurzgesagt

OOP

- › Eine Klasse definiert die Blaupause für Objekte.

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

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

- › Eine Klasse definiert die Blaupause für Objekte.
 - › Attribute definieren den Zustand.
 - › Methoden definieren den Verhalten.
- › Der Konstruktor baut den initialen Zustand

- › 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.

- › 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.

- › 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.

- › 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`, ...)

- › 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**, ...)
- › **Gültigkeitsbereich**: Wo die Variablen „deklariert sind“.



Sources & current version
on GitHub

Florian Sihler

22.12.2021

