Informatik II
Prüfungsvorbereitungskurs

Tag 2, 7.6.2017

Giuseppe Accaputo

g@accaputo.ch

Aufbau des PVK

- **Tag 1**: Java Teil 1
- **Tag 2**: Java Teil 2
- **Tag 3**: Algorithmen & Komplexität
- **Tag 4**: Dynamische Datenstrukturen, Datenbanksysteme

Programm für heute

- Repetition
- Self-Assessment Test 1
- Java Teil 2
  - Mehrdimensionale Arrays
  - Methoden
  - Klassen

Repetition Tag 1

Variablen

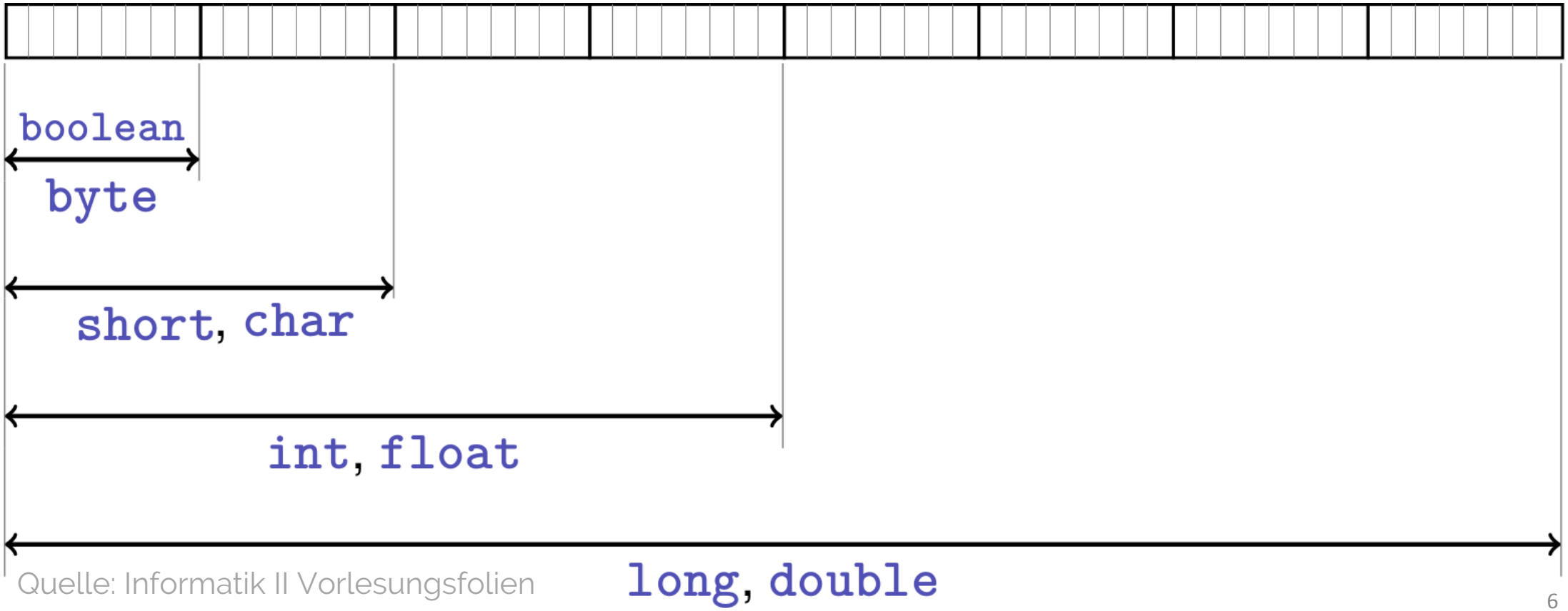
- *Behälter* für einen Wert
- Haben *Datentyp* und eine *Namen*
- Datentyp bestimmt, welche Art von Werten in der Variable erlaubt sind

```
int a = 10, b = 20;
char c = 'd';
float e = 1.2f;
```

- Datentypen:
  `int`, `char`, `float`
- Werte:
  `10`, `20`, `1.2f`, `'d'`
- Namen:
  `a`, `b`, `c`, `e`

Standardtypen und Speicherbelegung

- 1 Byte = 8 Bit



Quelle: Informatik II Vorlesungsfolien

Typkonvertierung

- **Implizit**: Zieltyp ist grösser als Ursprungstyp

  `byte < short < int < long < float < double`

  - `float a = 3;` → Zieltyp = `float`, Ursprungstyp = `int`
    - **Funktioniert**, da Zieltyp `float` > Ursprungstyp `int`
- **Explizit**: Zieltyp ist kleiner als Ursprungstyp
  - `int pi = 3.14;` → Zieltyp = `int`, Ursprungstyp = `float`
    - **Funktioniert so nicht**, da Zieltyp `int` < Ursprungstyp `float`
    - Expliziter Typecast: `int pi = (int)3.14;` → **Funktioniert!**

Präzedenz und Assoziativität

Faustregel Präzedenz:

1. Unäre Operatoren
2. Explizite Klammern
3. Punkt vor Strich
4. Arithmetisch vor Vergleich
5. Vergleich vor Logisch
6. ! vor && vor ||

Faustregel Assoziativität:

- Zuweisungsoperatoren sind rechtsassoziativ
- Alle anderen Operatoren sind linkssassoziativ

```
int b = 0;
int c = 0;
int a = (b = (c = 100));
```

Regeln für binäre Operanden

Regeln für op = {+, -, /, *, %}

1. Ganzzahl **op** Ganzzahl = **Ganzzahl**
2. Ganzzahl **op** Fliesskommazahl= **Fliesskommazahl**
3. Fliesskommazahl **op** Ganzzahl = **Fliesskommazahl**
4. Fliesskommazahl **op** Fliesskom. = **Fliesskommazahl**

Inkrement und Dekrement Operatoren

- **Prä**-Inkrement:

```
y = ++x;      ⇔   x = x + 1; y = x;
```

- **Post**-Inkrement:

```
y = x++;      ⇔   y = x; x = x + 1;
```

- Analog für Dekrement: `--x, x--`
- **Wichtig**: Inkrement und Dekrement Operatoren nur auf Variablen anwenden! Folgendes geht nicht: `--42`

Datentypen

- Primitive Datentypen:
  `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`
- Nicht-primitive Datentypen:
  Alle anderen Datentypen, z.B. `String`, `int[]`, etc.

Datentypen

- Variable mit primitivem Datentyp:
  Konkreter, einzelner Wert

```
int i = 10, j = 11;
```

- Variable mit nicht-primitivem Datentyp:
  Referenz auf Objekt (Speicherblock)

```
int[] arr = new int[2];
```

Vergleiche mittels == Operator

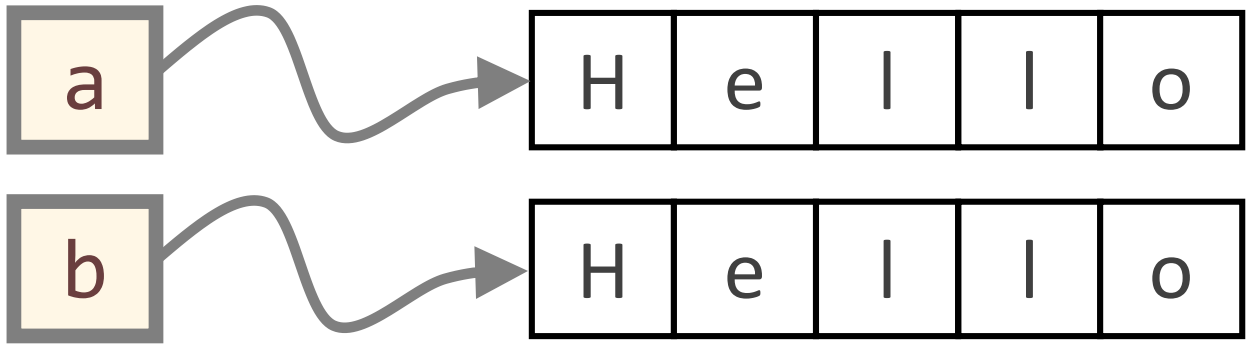
- Operanden mit primitivem Datentyp: == vergleicht Werte der Operanden

```
int i = 10;
int j = 10;
```


```
i == j → true
a == b → false
```

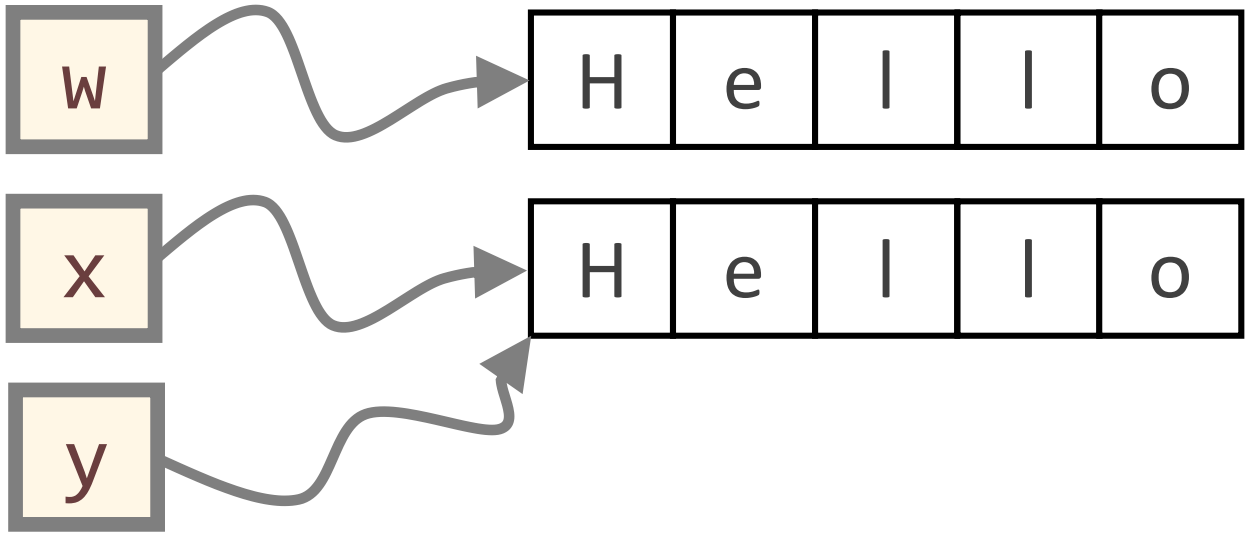
- Operanden mit nicht-primitivem Datentyp: == vergleicht Referenzen

```
String a = "Hello";
String b = "Hello";
```



Vergleiche mittels == Operator

```
String w = "Hello";
String x = "Hello";
String y = x;
```

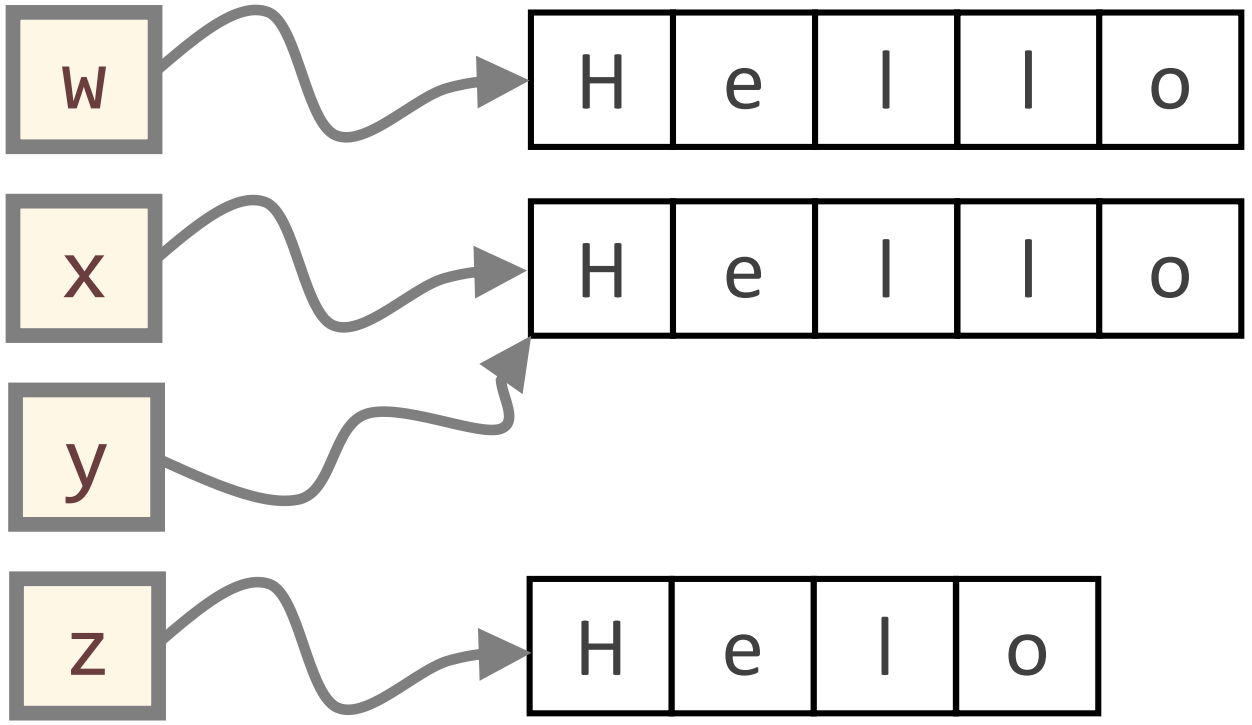



```
w == w → true
w == x → false
x == y → true
```

Merke: Ein Vergleich mittels == mit Referenzen evaluiert nur dann zu `true`, wenn beide Referenzen auf das gleiche Objekt (Speicherblock) zeigen!

Zeichenkettenvergleiche bei Strings

```
String w = "Hello";
String x = "Hello";
String y = x;
String z = "Helo";
```




```
w.equals(x) → true
x.equals(y) → true
y.Equals(z) → false
```

Self-Assessment Test 1

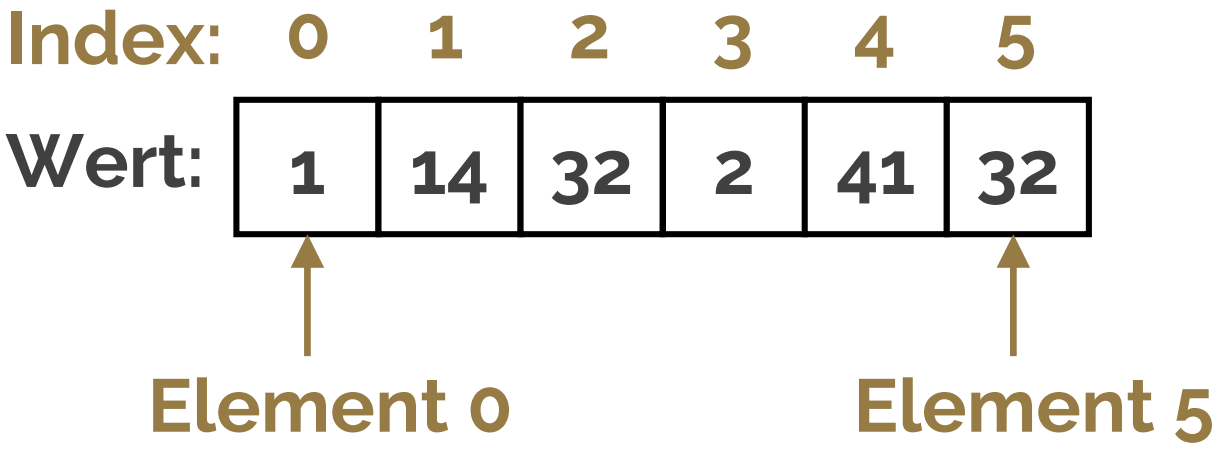
Java Teil 2

Übersicht

- Mehrdimensionale Arrays
- Methoden
- Klassen

Arrays

- **Array**: Objekt, welches mehrere Werte desselben Typs speichert
- **Element**: Ein Wert an einem Index des Arrays
- **Index**: Position eines Elementes im Array. Startet bei 0

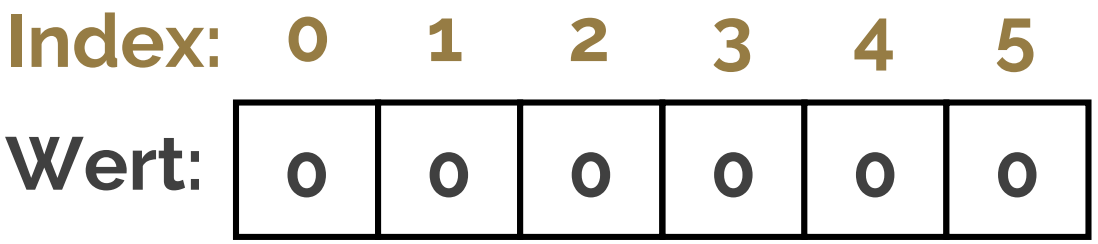


Arrays: Deklaration und Initialisierung

```
Typ[] name = new Typ[Länge];
```

Beispiel

```
int[] arr = new int[6];
```

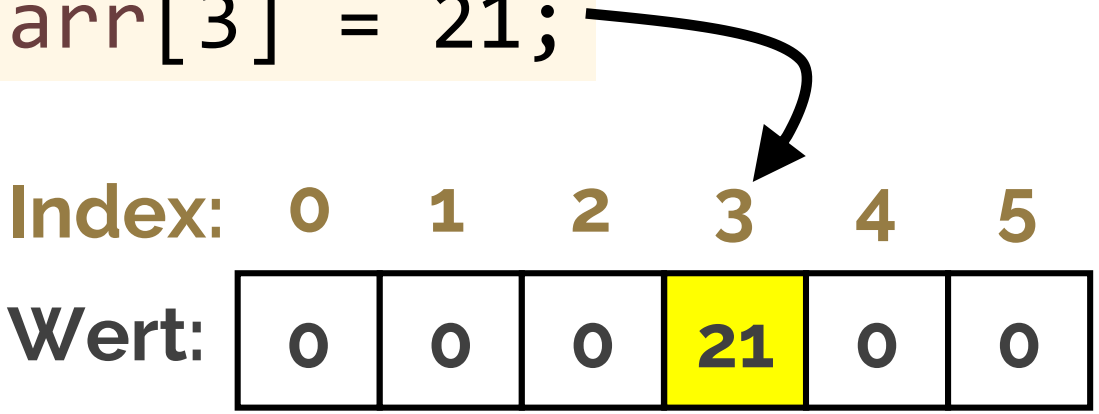


Arrays: Elementzugriff

- Element lesen:
- Element speichern:

```
int a = arr[index];
```

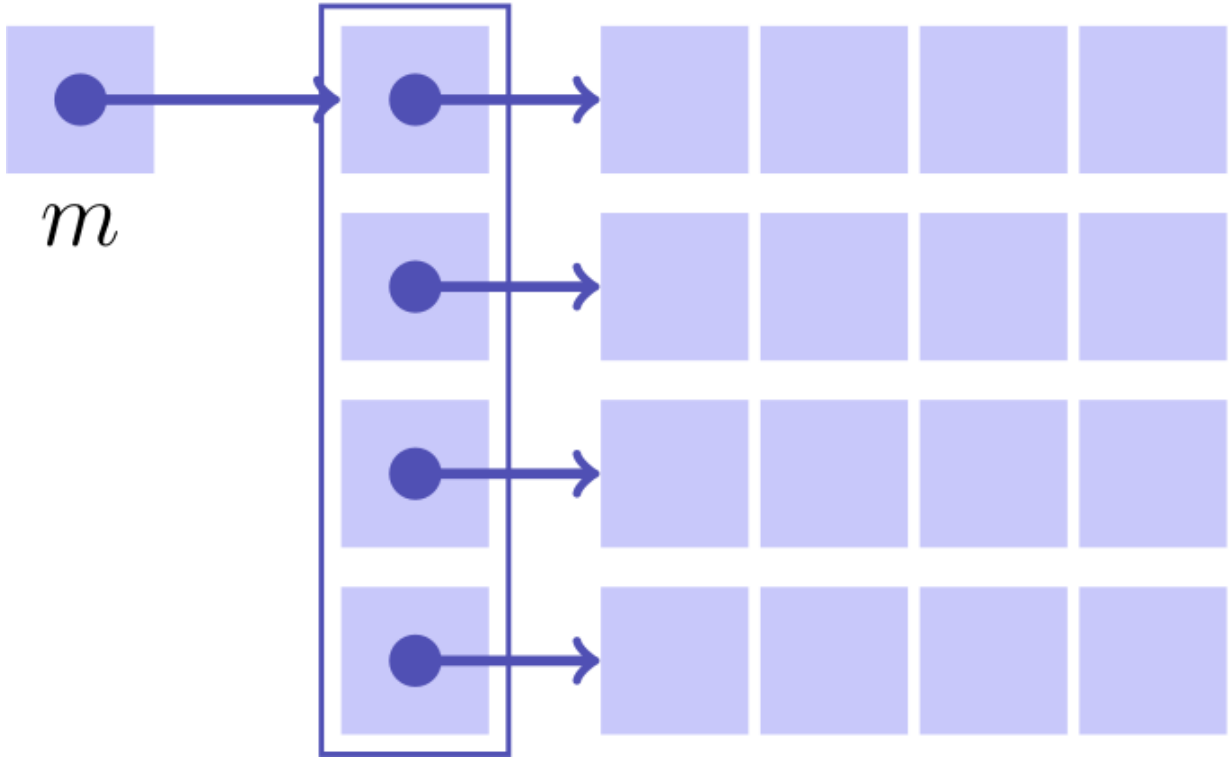
```
arr[3] = 21;
```



- Erste Position auf Index `0`, letzte auf `arr.length - 1`
- **Wichtig**: Exception wird geworfen bei Fehlzugriff!

Mehrdimensionale Arrays

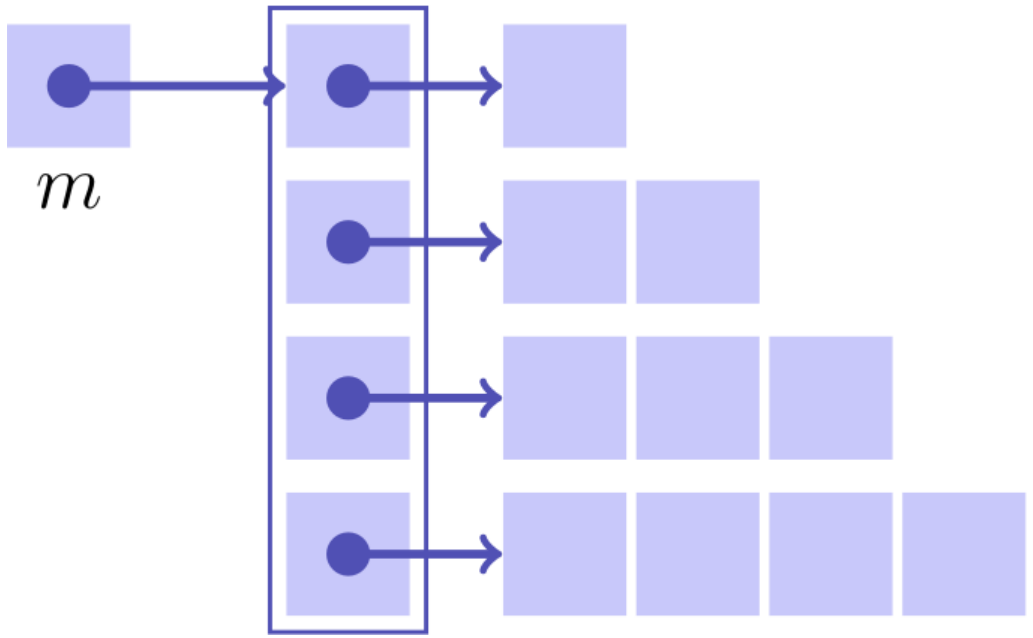
```
double[][] m = new array[4][4];
```



Quelle: Informatik II Vorlesung

Mehrdimensionale Arrays

```
double[][] m = new double[5][];
for (int r = 0; r < m.length; ++r)
  m[r] = new double[r+1];
```



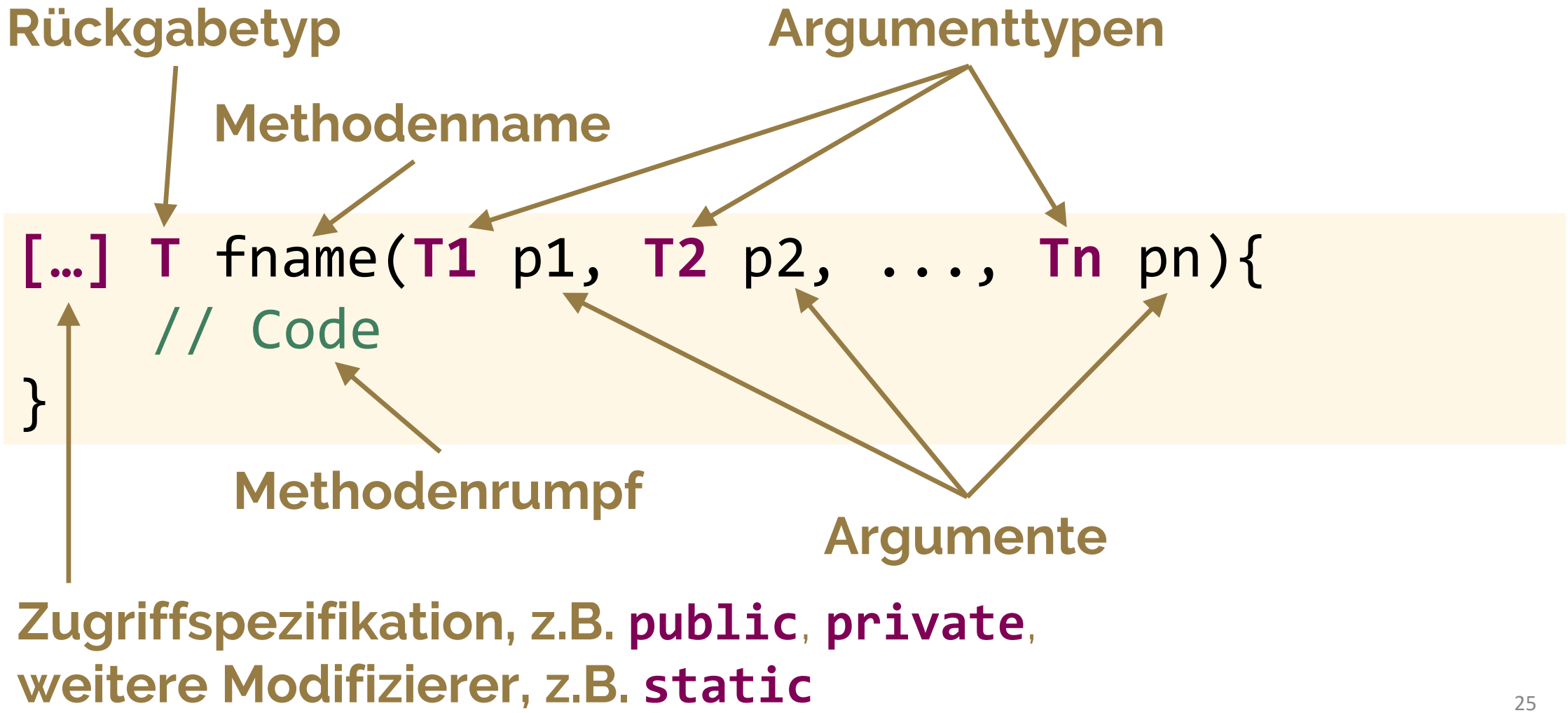
Quelle: Informatik II Vorlesung

Mehrdimensionale Arrays

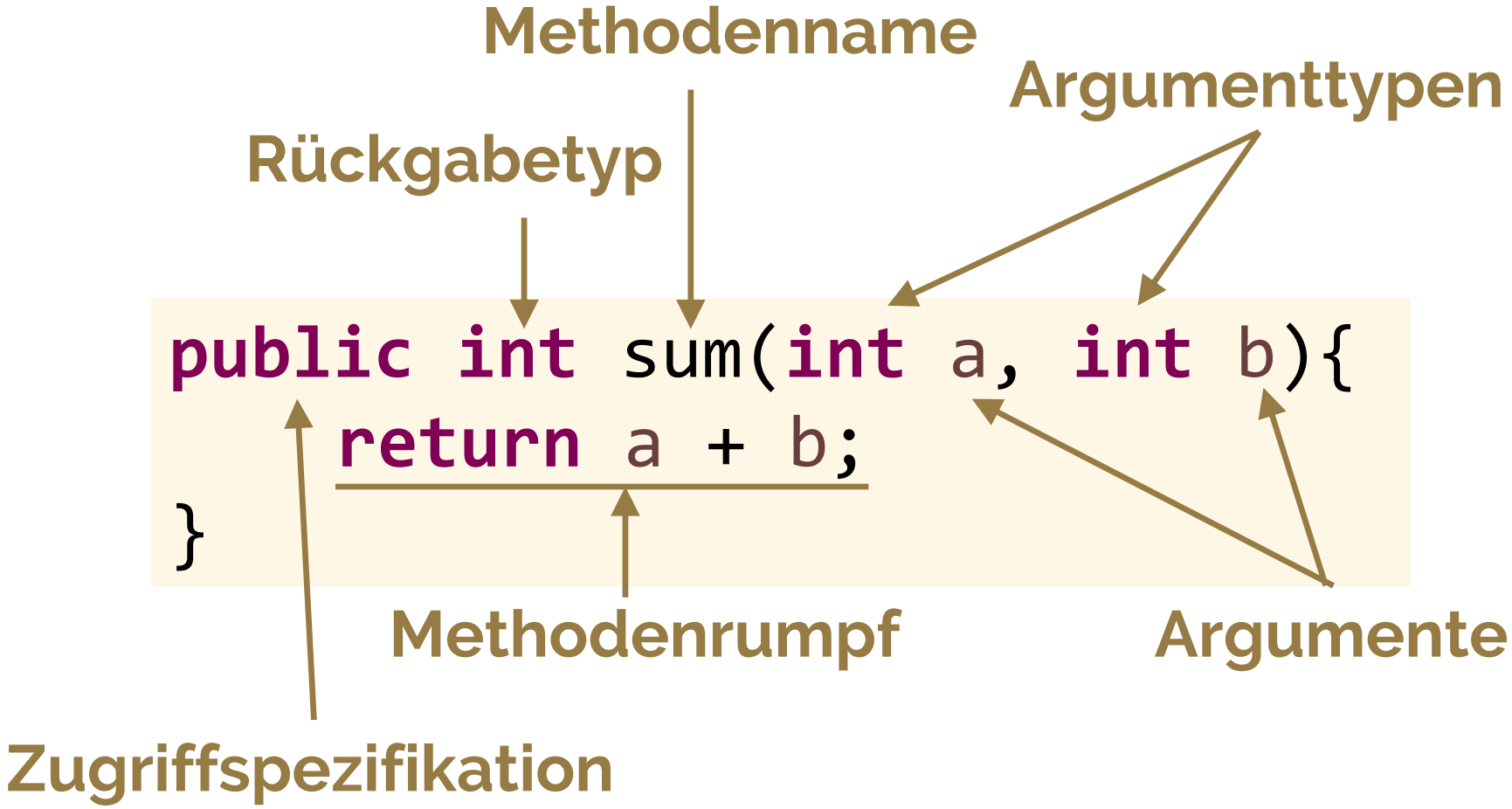
```
double[][] matrix=new array[4][4];

// Einheitsmatrix
for (int r=0; r < matrix.length; ++r){
  for (int c=0; c < matrix[r].length; ++c){
    if (r==c)
        matrix[r][c] = 1;
    else
        matrix[r][c] = 0;
  }
}
```

Quelle: Informatik II Vorlesung

Methoden


Rückgabetyp
Argumenttypen
Methodenname
[…] T fname(T1 p1, T2 p2, ..., Tn pn){
// Code
}
Methodenrumpf
Argumente
Zugriffspezifikation, z.B. public, private,
weitere Modifizierer, z.B. static

Beispiel Summen-Methode



Rückgabewerte von Methoden

1. Rückgabetyp = **`void`**
   Auswertung der Methode kann mittles **`return;`** beendet werden
2. Rückgabetyp != **`void`**:
   Auswertung der Methode muss mittels **`return`** `wert;` beendet werden. Der Wert wird dann an die aufrufende Methode zurückgegeben

- **Wichtig**: Im 2. Fall muss jeder Ausführungspfad eine **`return`** Anweisung enthalten!

Pass by Value

- Argumentwerte einer Methode werden beim Methodenaufruf in die Parameter kopiert
- Primitive Datentypen:
  Wert wird in Parameter kopiert
- Nicht-primitive Datentypen:
  Referenz wird in Parameter kopiert

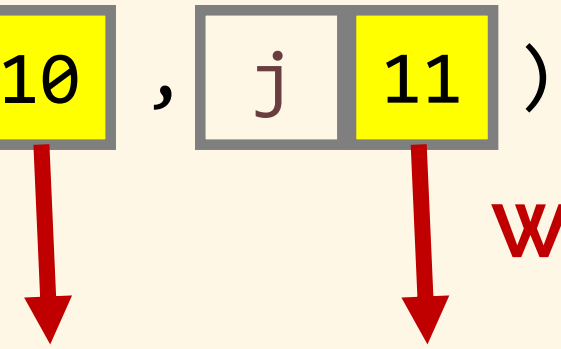
Pass by Value: primitive Datentypen

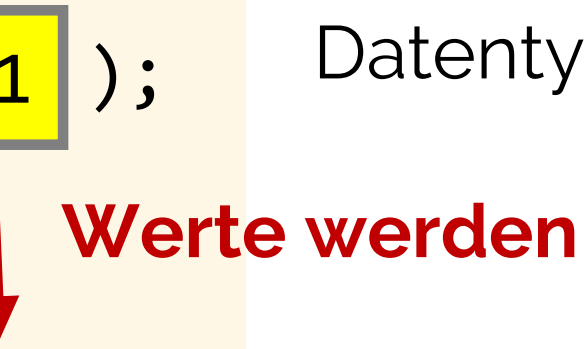
```
void test(){
      int i = 10, j = 11;
      do(i,j);
}

… void do(int a, int b){
      a = 30;
      b = 40;
}
```

- Funktion **do** hat zwei Parameter mit primitiven Datentypen

Pass by Value: primitive Datentypen

```
void test(){
      int i = 10, j = 11;

      do( i  10 , j  11 );

}

void do( a  10 , b  11 )
      …
}
```





Werte werden kopiert

- Funktion **do** hat zwei Parameter mit primitiven Datentypen

Pass by Value: nicht-primitive Datentypen

```
void test(){
     int[] arr = new int[2];
     arr[0] = 10;arr[1] = 11;
     fn(arr);
}
void fn(int x[]){
     x[0] = 30;
     x[1] = 40;
}
```

- Funktion `fn` hat ein Parameter mit nicht-primitivem Datentyp

Pass by Value: nicht-primitive Datentypen

```
void test(){
      int[] arr = new int[2];
      arr[0] = 10;arr[1] = 11;

      fn( arr );

}

void fn( x ){
…}
```

- Funktion `fn` hat ein Parameter mit nicht-primitivem Datentyp

Pass by Value: nicht-primitiven Datentypen

- **Wichtig**: Bei nicht-primitiven Datentypen wird nur die Referenz wird kopiert, nicht der Inhalt des Objektes (Speicherblocks)

Beispiel Pass by Value: Array kopieren

Falsch

```
void test(){
  int x[] = {1,2,3};
  int y[] = {0,0,0};
  wrongCopy(x,y);
}

void wrongCopy(int[] x, int[] y){
  y = x;
}
```

Beispiel Pass by Value: Array kopieren

Korrekt

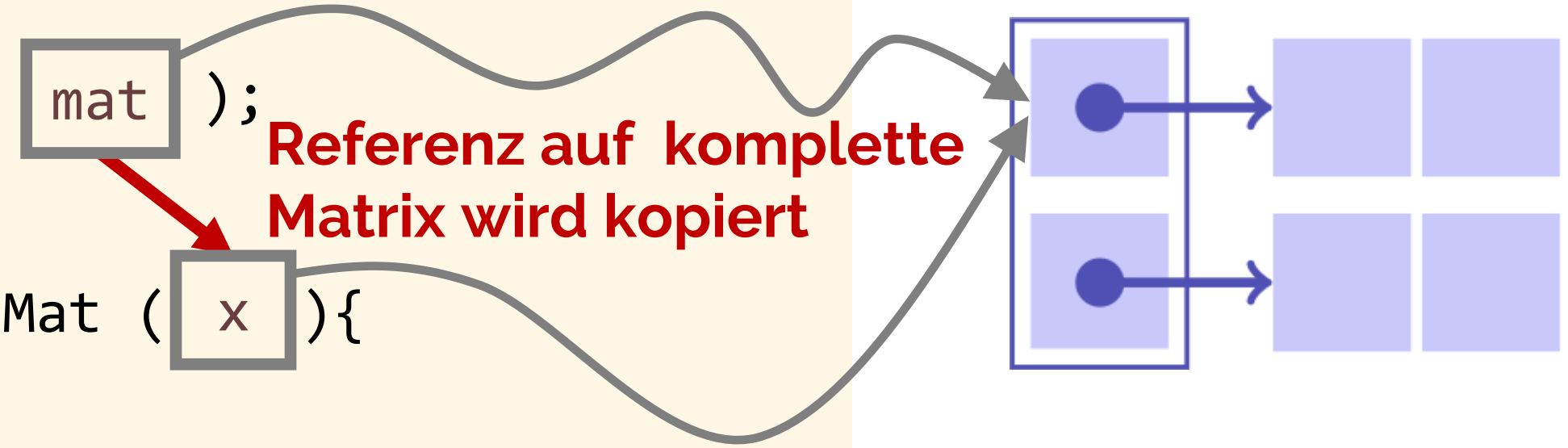
```
void test(){
  int x[] = {1,2,3};
  int y[] = {0,0,0};
  wrongCopy(x,y);
}

void correctCopy(int[] x, int[] y){
  for(int i=0; i<x.length; i++){
    y[i] = x[i];
  }
}
```

Pass by Value: Mehrdimensionales Array

```
void test(){
   int[][] mat = new int[2][2];
   mat[0][0] = 1; mat[0][1] = 2;
   mat[1][0] = 3; mat[1][1] = 4;
   fnMat(mat);
}

void fnMat(int x[][]){
   …
}
```

Pass by Value: Mehrdimensionales Array

```
void test(){
  int[][] mat = new int[2][2];
    …
  fnMat( mat );
}

void fnMat ( x ){
    …
}
```

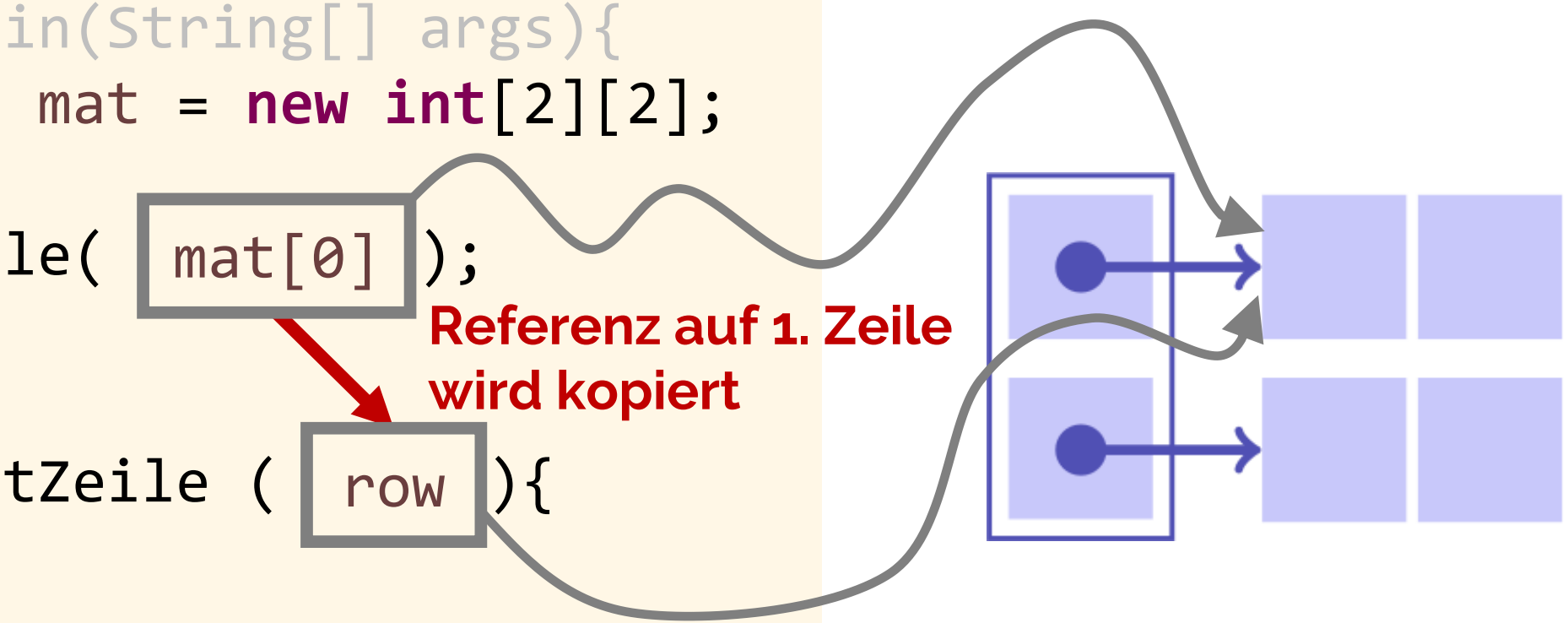


Pass by Value: Mehrdimensionales Array

```
void test(){
   int[][] mat = new int[2][2];
   mat[0][0] = 1; mat[0][1] = 2;
   mat[1][0] = 3; mat[1][1] = 4;
   fnMatZeile(mat[0]);
}

void fnMatZeile (int row[]){
   …
}
```

Pass by Value: Mehrdimensionales Array

```
… void main(String[] args){
   int[][] mat = new int[2][2];
   …
  fnMatZeile( mat[0] );
}

void fnMatZeile ( row ){
   …
}
```



Review: Aufgabe Pass by Value

```
static void I(int x, int y) {
   int temp = x;
   x = y;
   y = temp;
}
static void A(int[] x) {
   x[0] = x[1];
}
static void S (String x, String y) {
   x = y;
}
public static void main(String[] args) {
   int[] x = {1,2};
   String name = "ETH";
   I(x[0],x[1]);
   A(x);
   S(name, "EPFL");
   System.out.println(x[0]+","+x[1]+","+name);
}
```

Quelle: Informatik II Vorlesung

Was wird ausgegeben?

(1) 1,1,ETH

(2) 1,2,ETH

(3) 2,1,ETH

(4) 2,2,ETH

(5) 1,1,EPFL

(6) 1,2,EPFL

(7) 2,1,EPFL

(8) 2,2,EPFL

Aufgabe Methoden Definition

- Definiere eine Funktion, welche ein `String s` auf die Konsole ausgibt
- Definiere eine Funktion, welche überprüft, ob 3 Integer gleich sind und bei Erfolg `true` zurückgibt
- Definiere eine Funktion, die alle ganzen Zahlen im Intervall [0,100] ausgibt, welche durch 2, 6, und 8 teilbar sind

Lösung Methoden Definition

- Definiere eine Funktion, welche überprüft, ob 3 Integer gleich sind und bei Erfolg `true` zurückgibt

```
boolean eq3(int a, int b, int c){
    return (a == b) && (a == b);
}
```

Lösung Methoden Definition

- Definiere eine Funktion, die alle ganzen Zahlen im Intervall [0,100] ausgibt, welche durch 2, 6, und 8 teilbar sind

```
public void func2(){
  for(int i = 0; i <= 100; i++){
    if((i % 2 == 0)
          && (i % 6 == 0) && (i % 8 == 0)){
       System.out.println(i);
    }
  }
}
```

Fairen Würfel simulieren

- **Fair**: Jede Zahl kann mit gleicher Wahrscheinlichkeit gewürfelt werden
- `Math.random()` gibt eine Pseudozufallszahl $\mathrm{u} \in [0, 1)$ zurück
- Funktion `wuerfle()` soll ein $Y \in \{1, 2, \ldots, 6\}$ mit gleicher Wahrscheinlichkeit zurückgeben, also:

$$\mathbb{P}(Y = k) = 1/6 \text{ für jedes } k \in \{1, 2, \ldots, 6\}$$

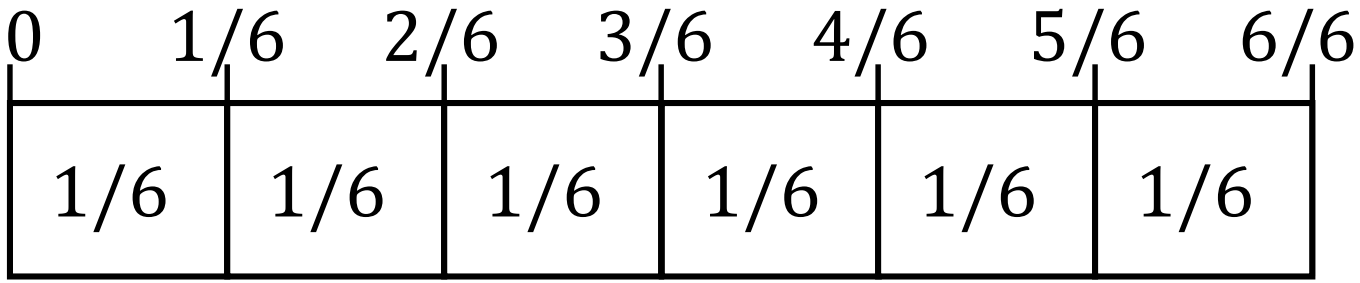
Fairen Würfel simulieren: Algorithmus

1. Generiere eine Pseudozufallszahl $u \in [0,1)$
2. Falls $u \in [0, 1/6)$, dann gib **1** zurück
3. Falls $u \in [1/6\ , 2/6)$, dann gib **2** zurück
4. …
5. Falls $u \in [5/6\ , 6/6)$, dann gib **6** zurück

Info: Wir möchten also herausfinden, in welchem Intervall u landet und geben dementsprechend die Würfelzahl zurück

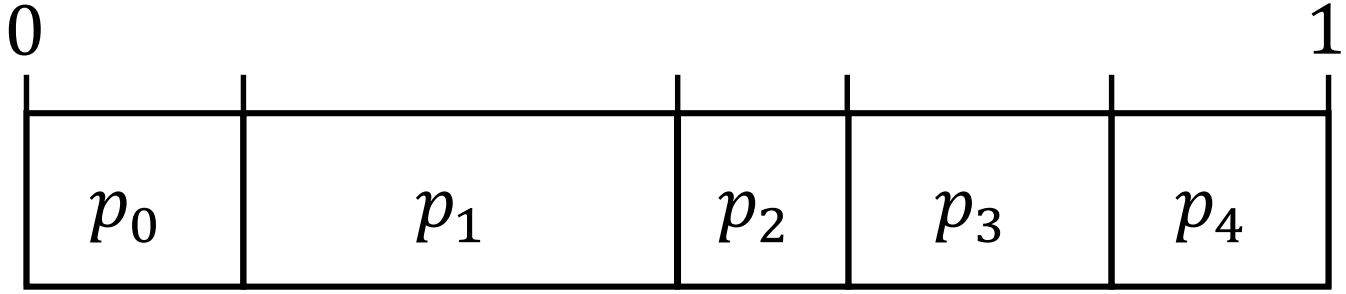
Fairen Würfel simulieren: Algorithmus

- Finde heraus, in welchem Intervall u landet:



Unfairen Würfel simulieren

- **Unfair**: Zahlen können mit verschiedenen Wahrscheinlichkeiten gewürfelt werden
- **Gegeben**: W-keitsvektor $p = (p_0, p_1, \dots, p_{n-1})$ mit $\sum_{i=0}^{n-1} p_i = 1$



- **Gesucht**: `sample(p)` soll ein $j \in [1, n]$ zurückgeben mit Wahrscheinlichkeit p_j

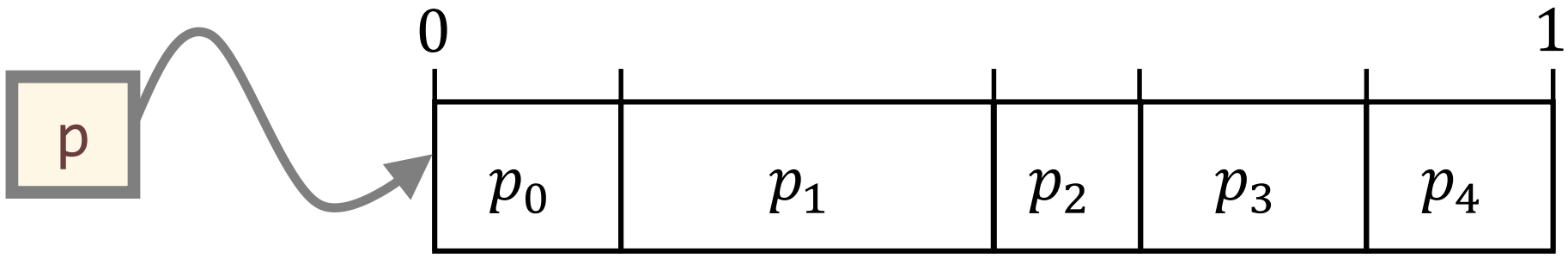
Unfairen Würfel simulieren: Algorithmus

1. Generiere eine Pseudozufallszahl $\mathbf{u} \in [0,1)$
2. Falls $\mathbf{u} \in [0, p_0)$, dann gib **1** zurück
3. Falls $\mathbf{u} \in [p_0, p_0 + p_1)$, dann gib **2** zurück
4. Falls $\mathbf{u} \in [p_0 + p_1, p_0 + p_1 + p_2)$, dann gib **3** zurück
5. …

Info: Wir möchten wieder herausfinden, in welchem Intervall $\mathbf{u}$ landet und geben dementsprechend die Würfelzahl zurück

Unfairen Würfel simulieren

```
int sample(int p[]){
   …
}
```



Unfairen Würfel simulieren: Code

- Wandtafel

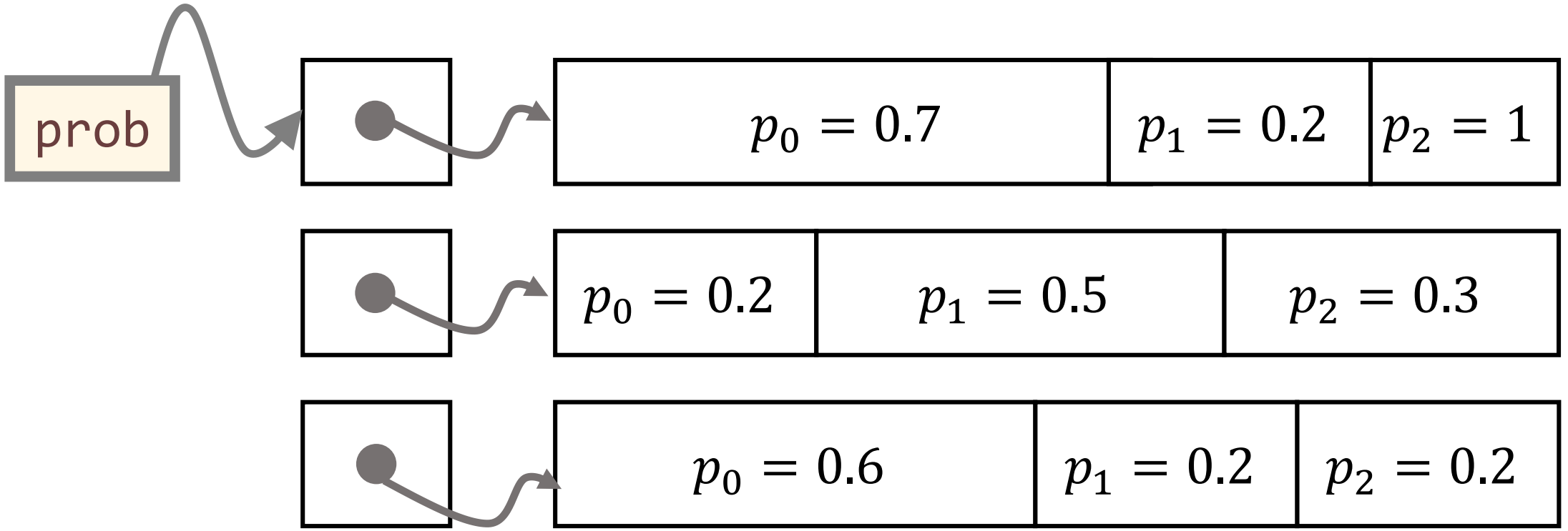
Wahrscheinlichkeitsmatrix

- Modell: Wandtafel

	Sonne	Wolken	Regen
Sonne	0.7	0.2	0.1
Wolken	0.2	0.5	0.3
Regen	0.4	0.2	0.2

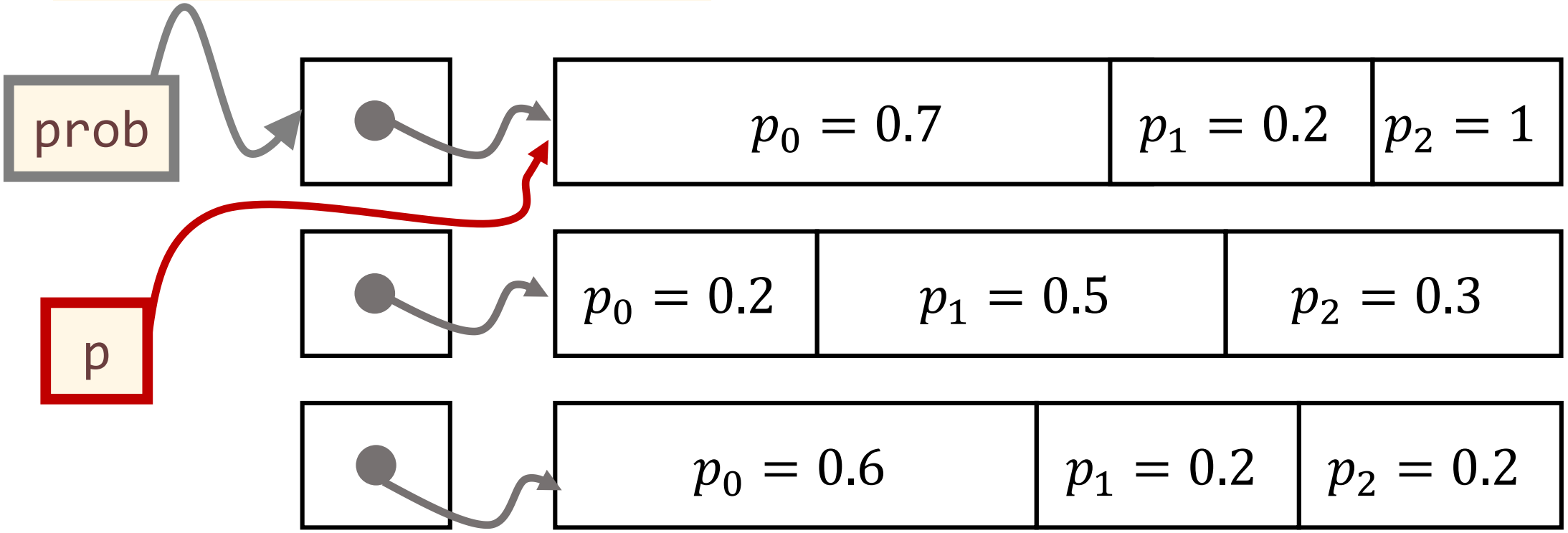
Wahrscheinlichkeitsmatrix

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

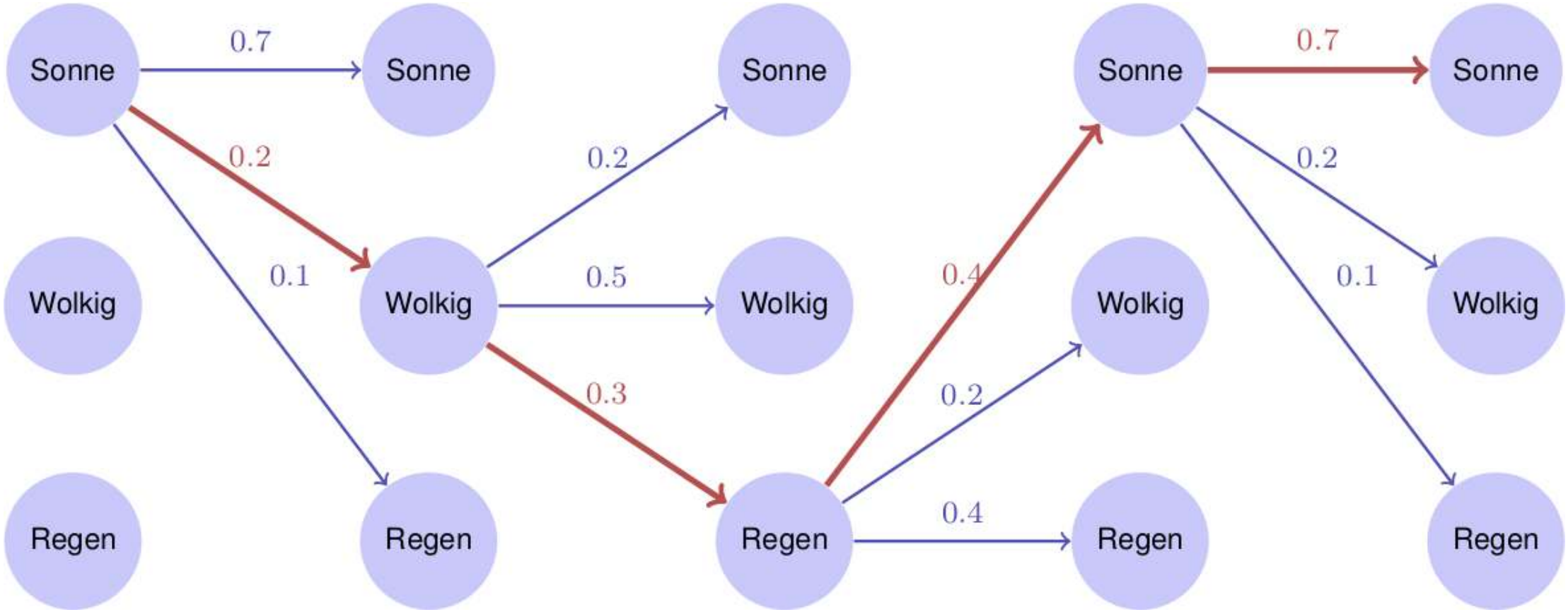


```
int sample(int p[]){
   …
}
```

```
int ind = sample(prob[0]);
```



Simulation des Wetters



Quelle: Informatik II Vorlesung

Beispiel Einfaches Wettermodell

```
public static void main(){
  double[][] P = {{0.9,0.1}, {0.5,0.5}};
  int sonnig = 0;
  int wetter = 0; // sonnig
  for (int i = 0; i<365; ++i){
        wetter = Sample(P[wetter]);
        if (wetter == 0)
              sonnig++;
  }
  System.out.println("Sonnige Tage: " + sonnig);
}
```

Quelle: Informatik II Vorlesung

Self-Assessment Test 2 Aufgabe 5

- Wandtafel
- Evtl. auch Übung

Klassen und Objekte

Pascal	Java
• RECORDs in Pascal sind reine Datenobjekte. Auf ihnen wird mit Prozeduren operiert • Wertsemantik: Instanzen werden *in place* alloziert	• Klassen in Java beherbergen Daten und Code • Referenzsemantik: Instanzen müssen mit **new** alloziert werden • Instanzen heissen auch Objekte.

Java Klassen

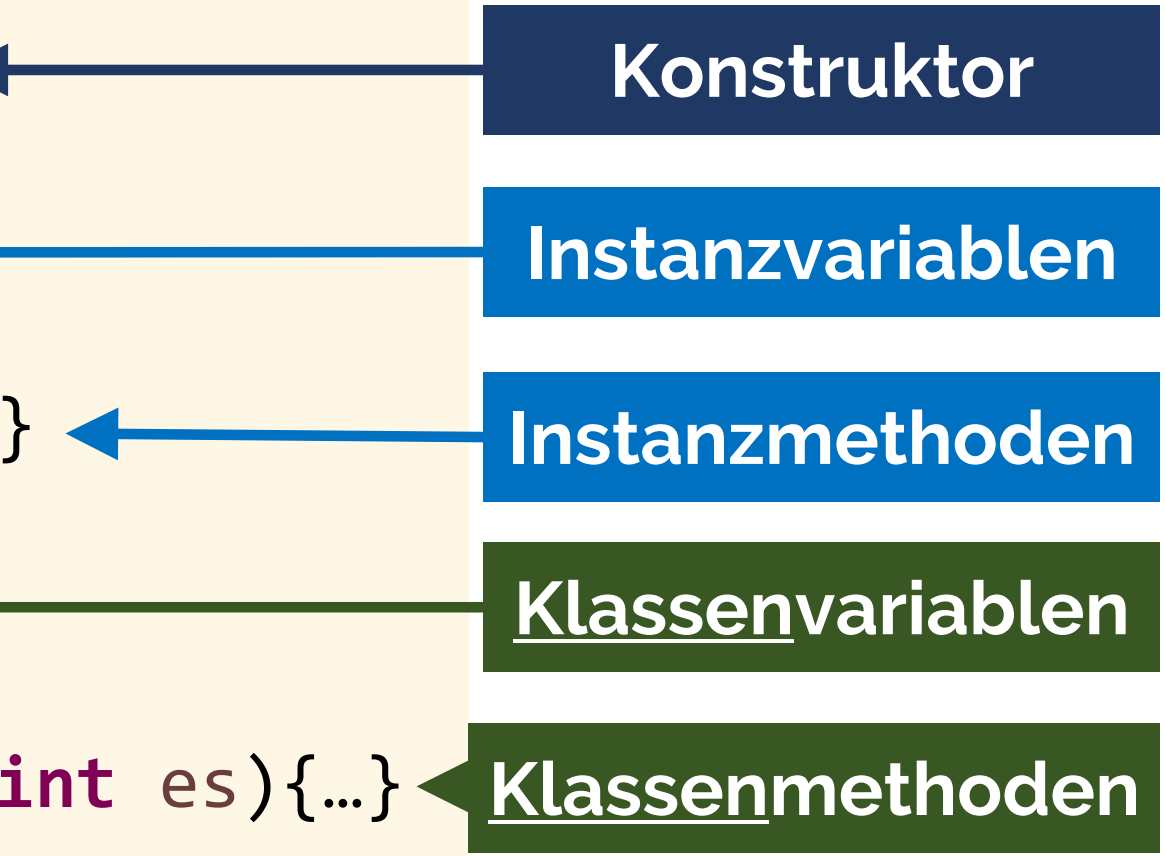
- Java-Programm besteht aus mindestens einer Klasse
- Java-Programm hat eine Klasse mit `main`-Methode

```
public class BeispielKlasse {
   public static void main(String[] args) {
      // Code
   }
}
```

- Java Virtual Machine führt `main`-Methode bei Programmstart aus

Aufbau einer Klasse

```
public class MeineKlasse {
    public MeineKlasse(){…}

    private int i;

    public int setI(int i){…}

    public static int s;

    public static void setS(int es){…}
}
```



Konstruktor

- Spezielle Methoden, die den Namen der Klasse tragen und keinen Rückgabetyp haben
- Kann Parameter haben und daher auch überladen werden
- Wird beim Aufruf mit `new` wie eine Funktion aufgerufen
- Wird kein passender Konstruktor gefunden, gibt Compiler eine Fehlermeldung aus

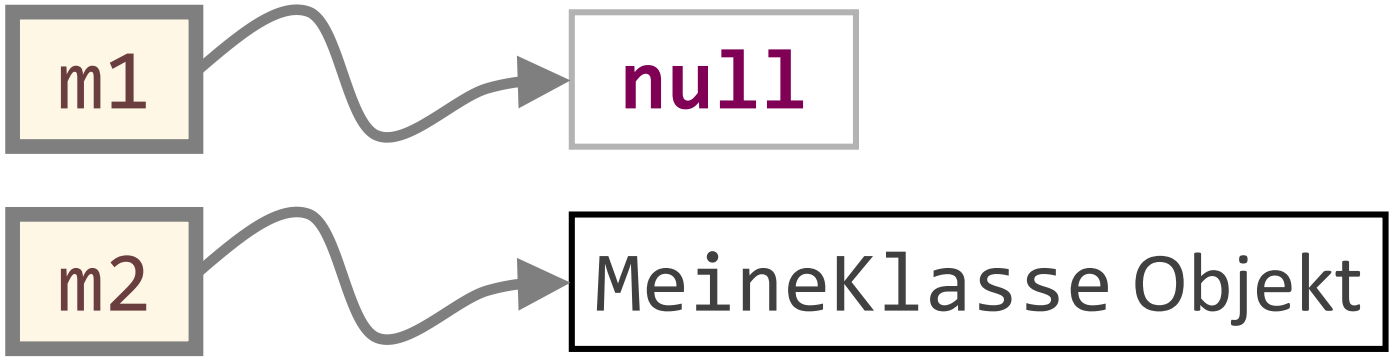
Beispiel Konstruktor

```
public class MeineKlasse {
    private int i;
    public MeineKlasse(){…}
}


MeineKlasse m1 = new MeineKlasse();
```

Speicherallokation mit `new`

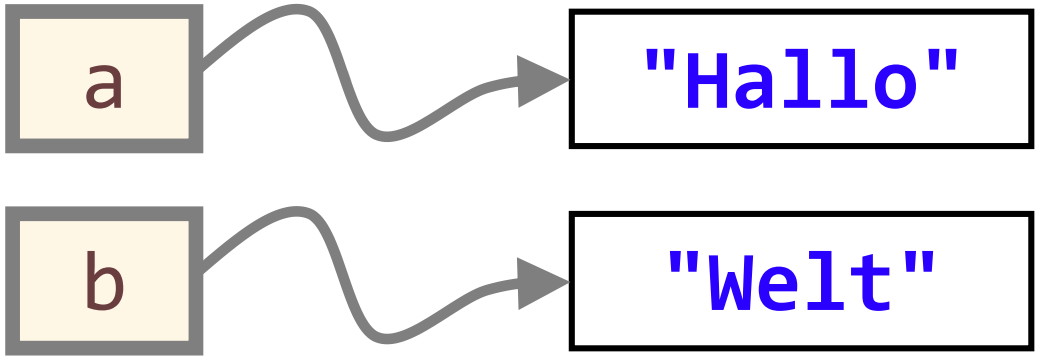
- Für die Nutzung von Klassen benötigt man dynamischen Speicher, also Speicher den man *explizit* anfordern muss
- Speicherallokation in dyn. Speicher erfolgt mittels **`new`**:

```
MeineKlasse m1;
MeineKlasse m2 = new MeineKlasse();
```



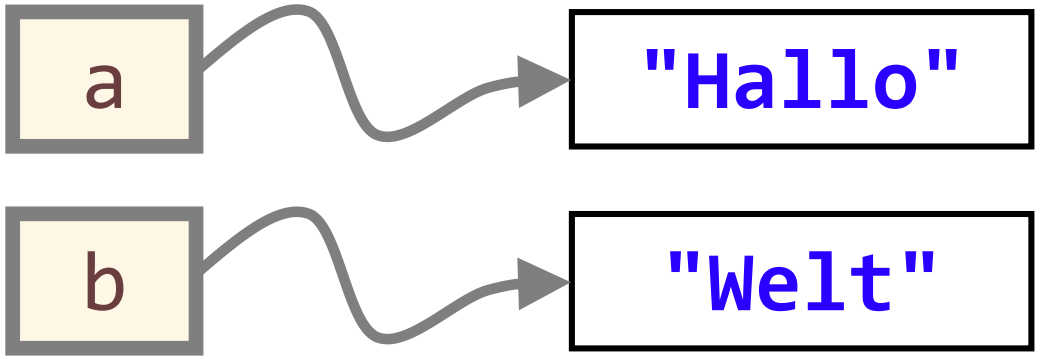
Beispiel Dynamische Speicherallokation

```
String a = new String("Hallo");
String b = new String ("Welt");
```



Beispiel Dynamische Speicherallokation

```
String a = new String("Hallo");
String b = new String ("Welt");
System.out.println(a + " " + b);
```

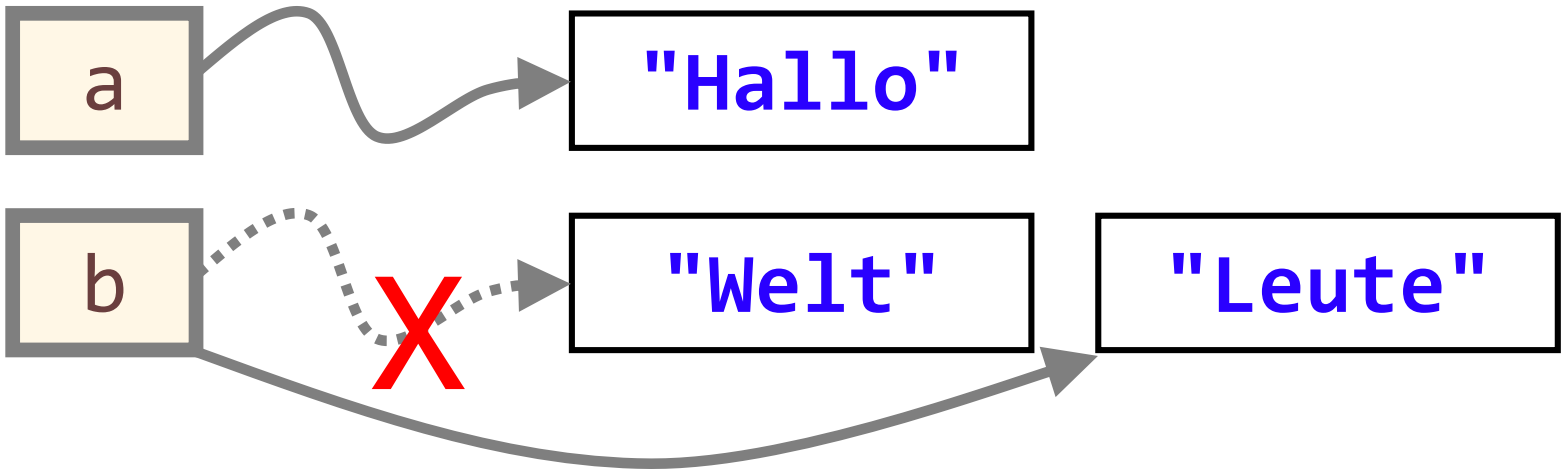


Konsole:

```
Hallo Welt
```

Beispiel Dynamische Speicherallokation

```
String a = new String("Hallo");
String b = new String("Welt");
System.out.println(a + " " + b);
b = new String("Leute");
```

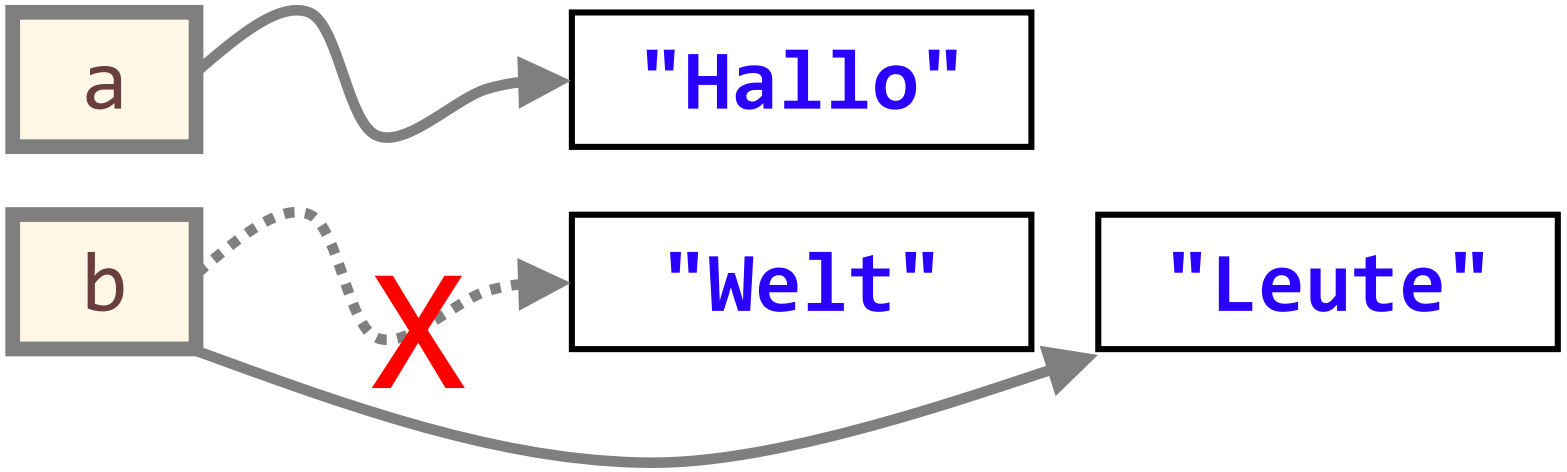


Beispiel Dynamische Speicherallokation

```
String a = new String("Hallo");
String b = new String ("Welt");
System.out.println(a + " " + b);
b = new String("Leute");
System.out.println(a + " " + b);
```

Konsole:

```
Hallo Leute
```



Instanzen einer Klasse

```
MeineKlasse m1 = new MeineKlasse();
MeineKlasse m2 = new MeineKlasse(10);
```

- `m1, m2` sind *Instanzen* der Klasse `MeineKlasse` und werden mittels `new`-Operator *instanziert*

Instanzen einer Klasse

```
MeineKlasse m1 = new MeineKlasse();
MeineKlasse m2 = new MeineKlasse(10);
```

Instanz **m1**
Instanzmethoden Instanzvariablen

Instanz **m2**
Instanzmethoden Instanzvariablen

- Jede Instanz hat eigene Kopie von Instanzmethoden und -variablen
- Ändert Instanz Wert der eigenen Instanzvariablen, so ist die Änderung nur für die Instanz geltend

Aufgabe Instanzvariablen ändern

- Was wird auf der Konsole ausgegeben?

```
public class InstVar {
     public int a = 10;
}
…
InstVar s1 = new InstVar();
InstVar s2 = new InstVar();
s2.a = 100;
System.out.println(s1.a);
```

Lösung Instanzvariablen ändern

```
public class InstVar {
     public int a = 10;
}
…
InstVar s1 = new InstVar();
InstVar s2 = new InstVar();
s2.a = 100;
System.out.println(s1.a);
```

Konsole:

```
10
```

Datenkapselung

- Fundamentales Konzept der objektorientierten Programmierung (**Prüfungsrelevant**)
- Wir verbergen interne Daten und Strukturen vor dem Zugriff von aussen
  - Wie die Datenfelder (z.B. Instanzvariablen) einer Klasse repräsentiert werden, sollte für den Benutzer nicht sichtbar sein

Klasse ohne Datenkapselung

```
public class BspKaps {
    public int a;
    public double b;
}
```

- **Fehlende Datenkapselung**: Benutzer können direkt auf die Variablen zugreifen (**`public`**)

Klasse mit Datenkapselung

```
public class BspKaps {
	private int a;
	public int getA() {
		return a;
	}
	public void setA(int a) {
		this.a = a;
	}

	private double b;
	public double getB() {
		return b;
	}
	public void setB(double b) {
		this.b = b;
	}
}
```

- **Datenkapselung**: Getter und Setter Methoden werden verwendet, um Zugriffe auf die Variablen zu kontrollieren

Modifizierer für Datenkapselung

	Klasse	Paket	Sub-Klasse	Global
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	x
(keiner)	✓	✓	x	x
`private`	✓	x	x	x

- Klasse: Zugriff innerhalb Klasse, z.B. Methode auf Variable
- Paket: Zugriff zwischen Klassen innerhalb des gleichen Pakets
- Sub-Klasse: Zugriff von Sub-Klasse auf Basisklasse

Prüfung 08.2014 Aufgabe 2

- Whiteboard

Beispiel Datenkapselung bei `Schloss`

```
public class Schloss {
    public int code = 10;
}
```

- **IST**: jedermann kann Code vom Schloss ändern, da Instanzvariable **`public`** ist
- **SOLL**: Nur wer das Masterpasswort hat, darf Schloss-Code ändern

Beispiel Datenkapselung bei `Schloss`

```
class Schloss {
	private int code = 10;
	private String mpw = "jl1923ikx";

	public void setCode(int code, String pw){
		if(pw.equals(mpw))
			this.code = code;
		else
			System.out.println("Falsches PW");
	}
}
```

Aufgabe Instanzvariablen ändern

- Kompiliert dieser Code?

```
public class InstVar2{
    public int a = 10;
    private double b = 1.12;
}
…
InstVar2 k1 = new InstVar2();
k1.a = (int)100.01023;
k1.b = (int)4.412;
```

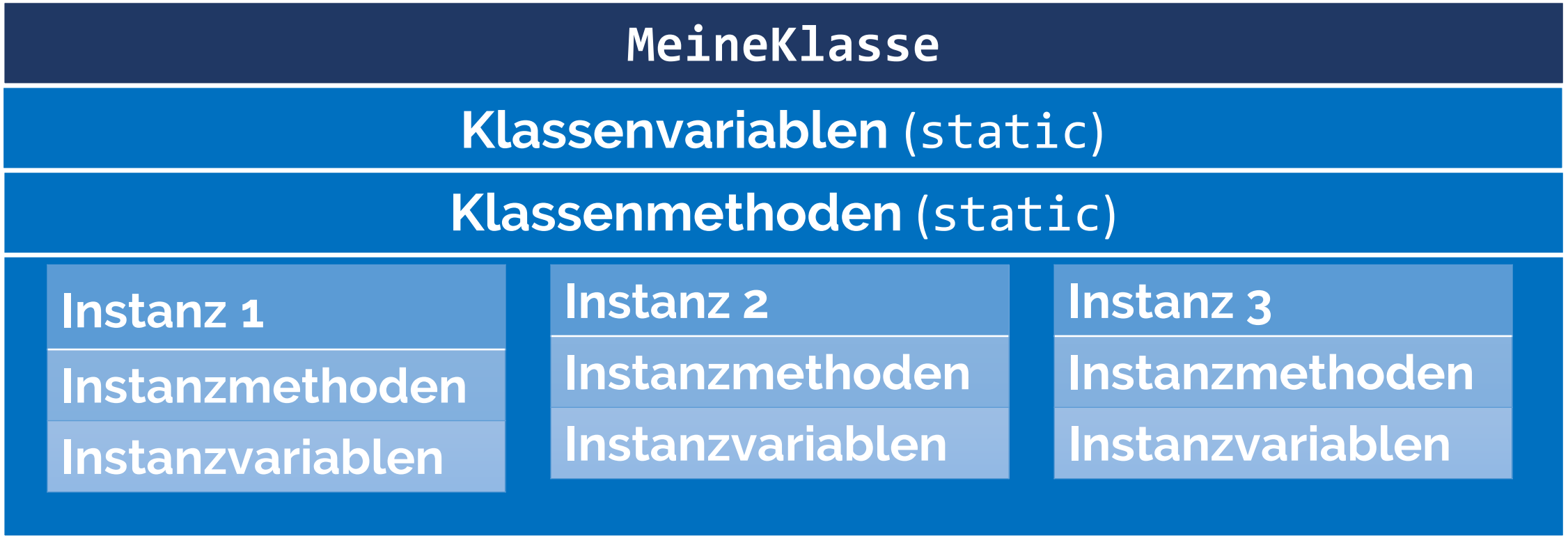
Lösung Instanzvariablen ändern

Kompilierfehler: The field b is not visible

```
public class InstVar2{
    public int a = 10;
    private double b = 1.12;
}
…
InstVar2 k1 = new InstVar2();
k1.a = (int)100.01023;
k1.b = (int)4.412; // Fehler!
```

Klassenvariablen und Klassenmethoden

- Klassenvariablen und Klassenmethoden sind über alle Instanzen verfügbar (sofern Zugriff gewährleistet)



Zugriff auf Klassenvariablen und -methoden

```
public class KlassVar {
    public int i = 1;
    public static int s = 2;
}
…
KlassVar a = new KlassVar();
```

```
a.s = 100;
KlassVar.s = 100;
```

- Zugriff auf Klassenvariablen und -methoden über Instanz oder Klassennamen möglich

Aufgabe Klassenvariablen ändern

- Was wird auf der Konsole ausgegeben?

```
public class KlassVar {
      public int i = 1;
      public static int s = 2;
}
…
KlassVar a = new KlassVar();
KlassVar b = new KlassVar();
a.s = 100; b.s += 10;
b.i -= 10; a.s -= 100;
System.out.println(b.i);
System.out.println(b.s);
```

Lösung: Klassenvariablen ändern

```
public class KlassVar {
      public int i = 1;
      public static int s = 2;
}
…
KlassVar a = new KlassVar();
KlassVar b = new KlassVar();
a.s = 100; b.s += 10;
b.i -= 10; a.s -= 100;
System.out.println(b.i);
System.out.println(b.s);
```

Konsole:

```
-9
10
```

Aufgabe Klassen

```
class TestClass {
   static int calls = 0;
   int count;
   public TestClass(int start){
      count = start;
   }
   int Next() {
      calls++;
      return count++;
   }
   static int Calls() {
      return calls;
   }
}
```

```
TestClass c1 = new TestClass(5);
TestClass c2 = new TestClass(10);
while (c1.Next() < 10);
while (c2.Next() < 20);
System.out.println(
   c1.Next() + " " + c2.Next());
System.out.println(
   c1.Calls() + " " + c2.Calls());
```

Ausgabe ?

(1) 11 21 19 19

(2) 10 20 5 10

(3) 11 21 5 10

(4) 10 20 20 20

(5) 11 21 17 17

(6) 10 20 6 11

(7) 11 21 6 11

(8) 10 20 17 17

Quelle: Informatik II Vorlesung

Zugriffsregeln zwischen Variablen und Methoden einer Klasse

1. Instanzmethoden können auf Instanzmethoden und Instanzvariablen direkt zugreifen
2. Instanzmethoden können auf Klassenmethoden und Klassenvariablen direkt zugreifen
3. Klassenmethoden können auf Klassenmethoden und Klassenvariablen direkt zugreifen
4. Klassenmethoden *können nicht direkt* auf Instanzmethoden und Instanzvariablen zugreifen