

Object-Oriented Programming in Java

Lecture 3 - Classes and Objects

Emily Lucia Antosch

HAW Hamburg

12.08.2025

Contents

1. Introduction	2
2. Classes and Objects	6
3. Variablen und Speicher	22
4. Methoden	38
5. Konstruktoren	73
6. Klassenvariablen & Klassenmethoden	93
7. License Notice	105

1. Introduction

1.1 Where Are We Now?

- Last time we dealt with the imperative concepts of the Java programming language.
- You can now
 - ▶ use simple data types in Java,
 - ▶ control program flow with control structures and loops, and
 - ▶ convert data types.
- Today we'll cover **Classes and Objects**.

1.1 Where Are We Now?

1. Introduction

1. Imperative Concepts
2. **Classes and Objects**
3. Class Library
4. Inheritance
5. Interfaces
6. Graphical User Interfaces
7. Exception Handling
8. Input and Output
9. Multithreading (Parallel Computing)

1.2 The Goal of This Chapter

- You will implement classes and objects in Java to model real things.
- You will create objects of a class and change their state through operations.
- You will apply additional programming guidelines to improve the quality and maintainability of your code.

2. Classes and Objects

2.1 Class

2. Classes and Objects

- A **class** is a blueprint for objects. It contains
 - ▶ **Attributes** (data fields) and
 - ▶ **Methods** (operations).
- Together, attributes and methods are called **members**.

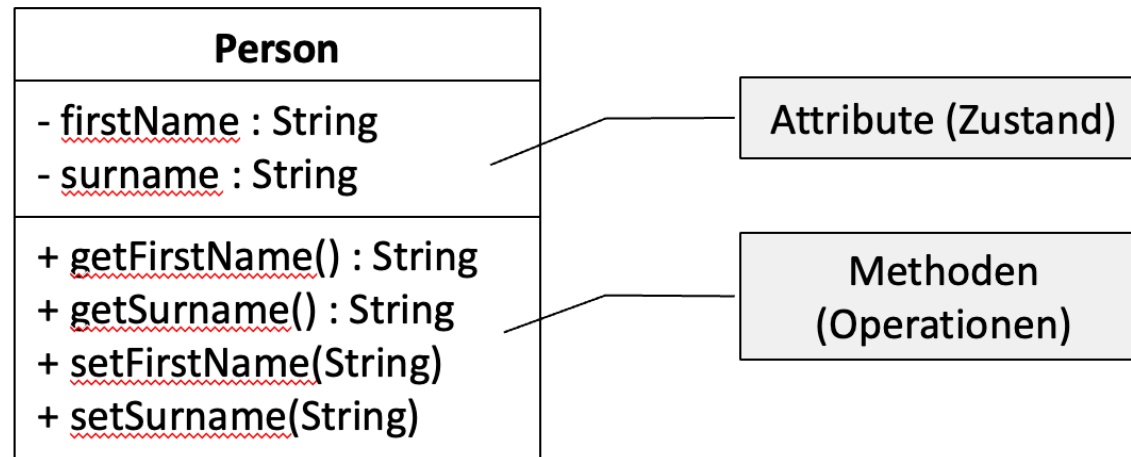


Figure 1: UML notation of a Person class

- Data record of a class created in memory at runtime
- Variables describe the **state** of the object
- Methods describe the **capabilities** of the object
- Terms for variables: **attributes**, **object variables**, **instance variables**

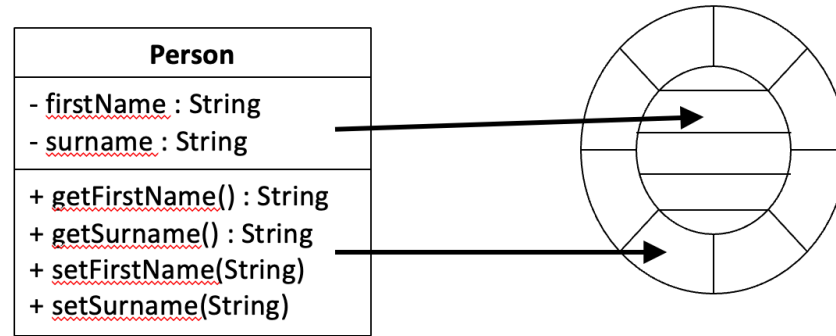


Figure 2: Division of methods and attributes

2.2 Relationship Between Class and Object

2. Classes and Objects

Klasse: Beschreibung („Bauplan“) eines Datentyps

- Objekt einer Klasse: Erzeugtes Element des Datentyps
- Es können beliebig viele Objekte einer Klasse erzeugt werden.

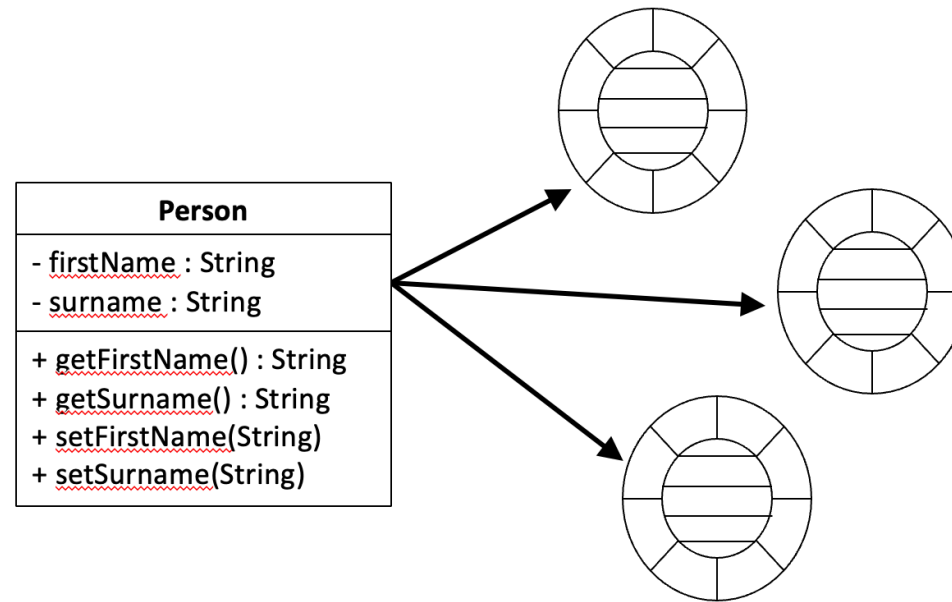


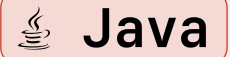
Figure 3: Mehrere Objekte aus einer Klasse

2.3 Classes in Java

2. Classes and Objects

- Klassen können über den folgenden Code deklariert werden:

```
1 class Klassenname {  
2     Attribute  
3     Methoden  
4 }
```



Tip

Legen Sie jede Klasse in einer eigenen Datei an!

Task 1

Lassen Sie uns diese einfache Klasse erstellen:

- Klasse Student, beschrieben durch Name, Matrikelnummer und Studienbeginn (in Jahren)

2.4 Example: Simple Class

Task 2

Lassen Sie uns diese einfache Klasse erstellen:

- Klasse Student, beschrieben durch Name, Matrikelnummer und Studienbeginn (in Jahren)

```
1 class Student {  
2     String name;  
3     int matrNumber;  
4     int enrolledYear;  
5 }
```



2.4 Example: Simple Class

- Die Klasse hat weder Methoden noch eine Datenkapselung gegen Einfluss von außen.

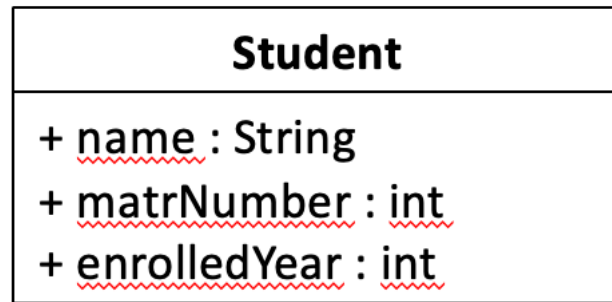


Figure 4: UML-Darstellung der Klasse, die wir eben erstellt haben

2.5 Example: One Class, Many Objects

2. Classes and Objects

- Klasse („Eine Klasse für alle Studierenden“):
 - ▶ Die Klasse ist ein neuer Datentyp.
 - ▶ Legt fest, durch welche Daten Studierende beschrieben werden
- Objekte („Für jede/n Studierende/n ein eigenes Objekt“):
 - ▶ Objekte sind Instanzen im Speicher.
 - ▶ Besitzen Struktur der Klasse, sind aber mit Daten gefüllt
 - ▶ Es können beliebig viele Objekte erzeugt werden.

2.5 Example: One Class, Many Objects

2. Classes and Objects

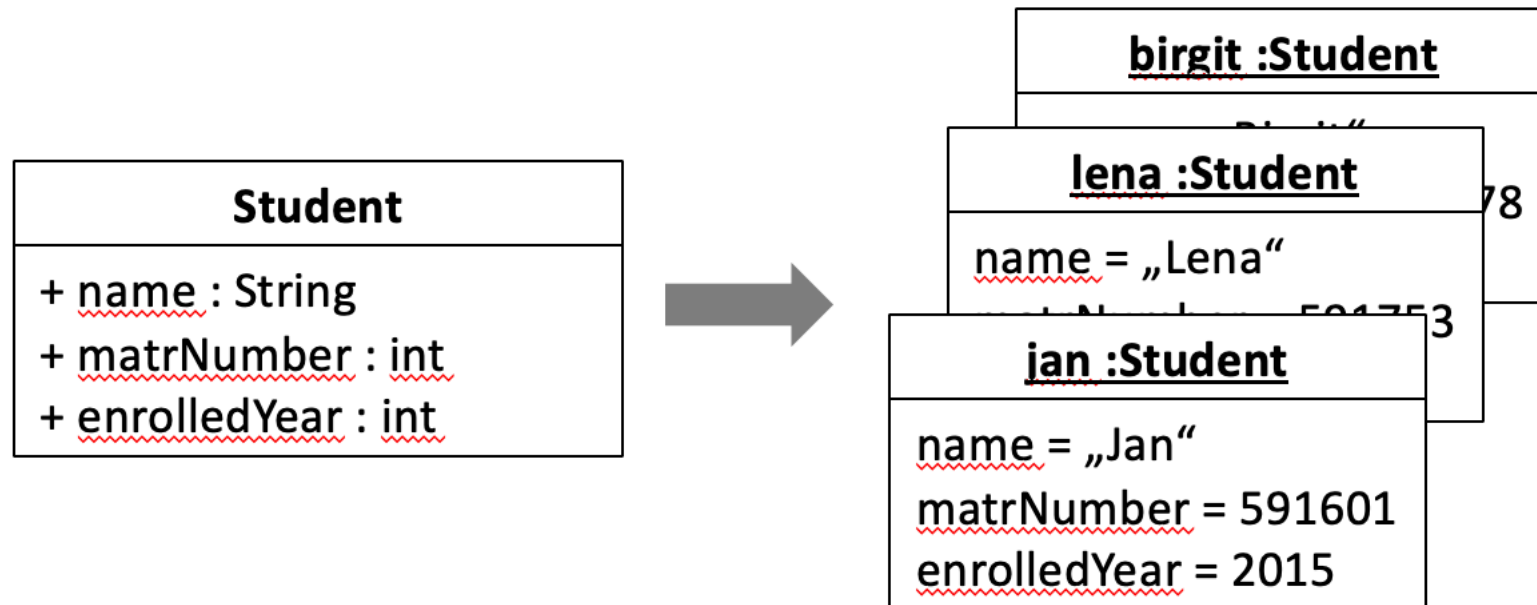
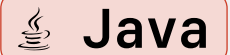


Figure 5: Aus einer Klasse lassen sich mehrere Objekte erstellen

? Question

Welche Werte haben die Variablen count, jan und lena?

```
1 public class StudentDemo {  
2     public static void main(String[] args) {  
3         int count;  
4         Student lena, jan;  
5     }  
6 }
```



? Question

Welche Werte haben die Variablen `count`, `jan` und `lena`?

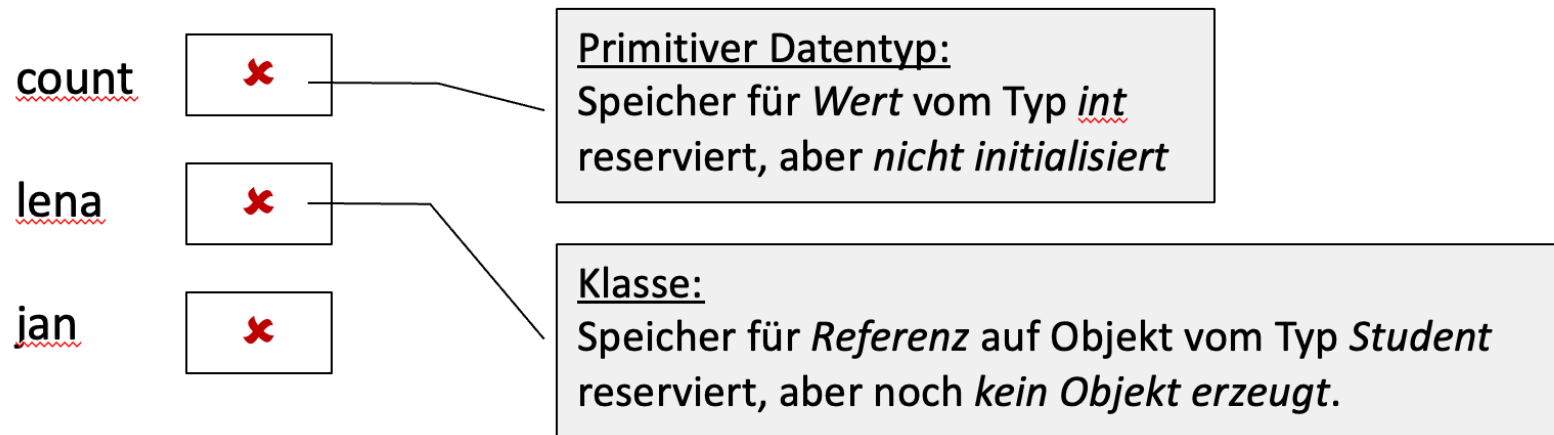
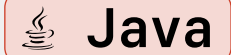


Figure 6: Primitive Datentypen vs. Objekte

2.7 Example: new Operator

- Objekte werden durch den new-Operator erzeugt.



```
1 public class StudentDemo {  
2     public static void main(String[] args) {  
3         int count;  
4         Student lena, jan;  
5         lena = new Student();  
6     }  
7 }
```

new-Operator

2.7 Example: new Operator

- Schritt 1: new-Operator erzeugt Objekt.
 - ▶ Speicherplatz für Objekt (mit Objektvariablen) reservieren.
 - ▶ Objektvariablen mit Standardwerten initialisieren (mehr dazu gleich).

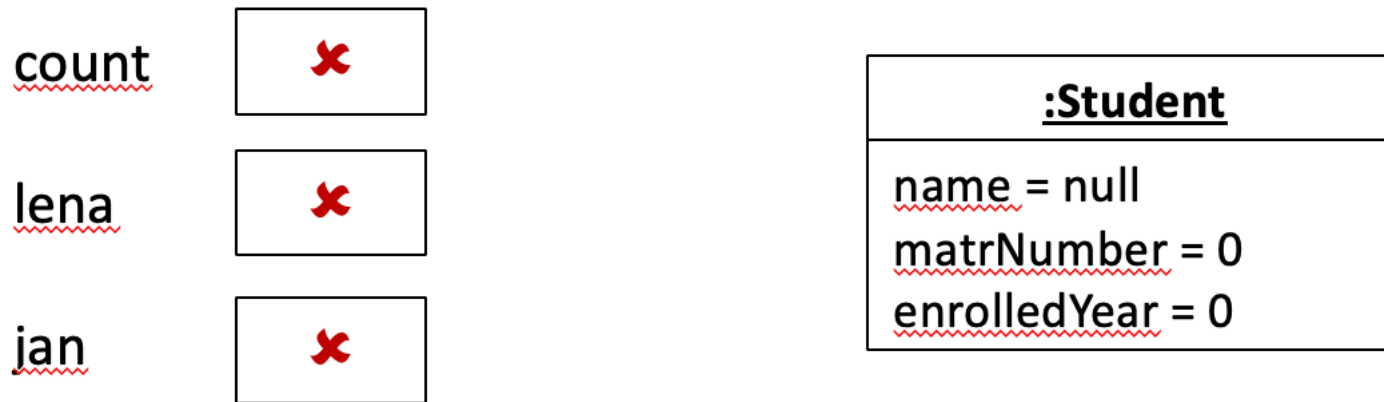


Figure 7: Erstellen von Referenz mit `new`

2.7 Example: new Operator

- Schritt 2: Zuweisung
 - ▶ Schreibt Referenz (“Adresse”) des neuen Objekts in Variable `lena`.
 - ▶ Ist unabhängig vom `new`-Operator und der Erzeugung des Objekts

2.7 Example: new Operator

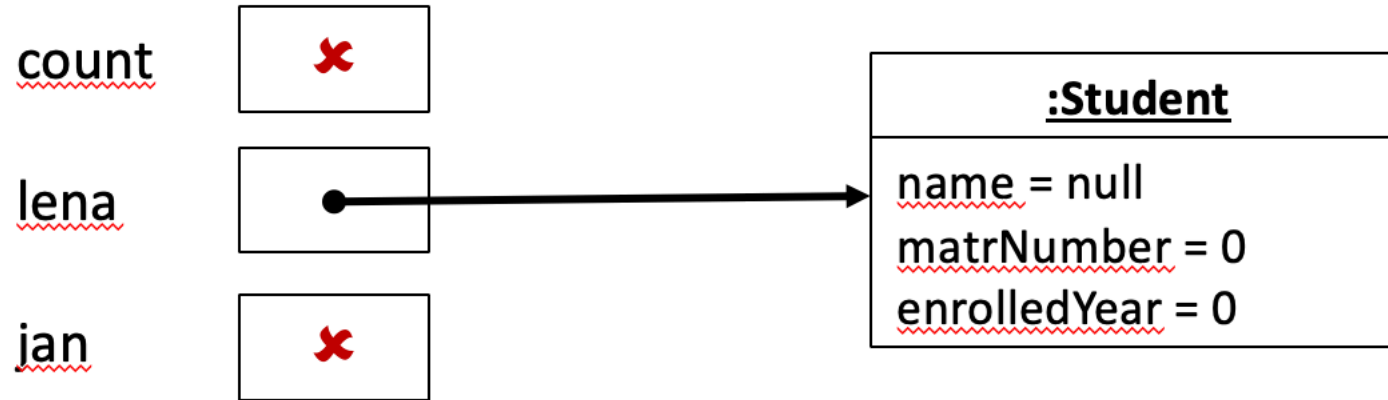


Figure 8: Zuweisung von Referenz an Variable

3. Variablen und Speicher



Conclusion

- Das haben wir uns bereits angeschaut:
 - ▶ Was sind Klassen und Objekte?
 - ▶ Wie deklariert man Klassen?
 - ▶ Wie erzeugt man Objekte?
- Im Folgenden wollen wir uns folgende Aspekte anschauen:
 - ▶ Zugriff auf Objektvariablen
 - ▶ Initialisierung von Objektvariablen
 - ▶ Zuweisung von Referenzen

3.1 Objective

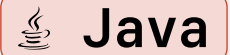
3. Variablen und Speicher

- ▶ Automatische Speicherbereinigung

3.2 Access to Object Variables

- Zugriff auf Objektvariablen erfolgt mittels des Punkt-Operators:

```
1 Objektreferenz.Member
```

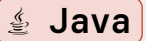


- Dabei ist die `Objektreferenz` eine Referenz auf ein Objekt, die in einer Variable gespeichert ist.
- `Member` ist z.B. ein Attribut/Objektvariable

? Question

Was wird ausgegeben?

```
1  public class StudentDemo1 {
2      public static void main(String[] args) {
3          Student lena = new Student();
4          System.out.println("Enrolled: " + lena.enrolledYear);
5          lena.name = "Lena";
6          lena.matrNumber = 591753;
7          lena.enrolledYear = 2012;
8          System.out.println("Enrolled: " + lena.enrolledYear);
9      }
10 }
```



! Memorize

- **Objekt-/Instanzvariable:** In Klasse als Attribut eines Objektes deklariert.
- **Lokale Variable:** Lokal deklariert (z.B. in Methode oder Schleife).
- **Referenzvariable:** Hat Klasse als Datentyp, kann Referenz auf Objekt speichern.

- Zur Erinnerung:
 - ▶ Lokale Variablen werden nicht automatisch initialisiert.
(Compiler verhindert Zugriff.)
 - ▶ Objektvariablen werden hingegen bei Erzeugung eines Objektes initialisiert.

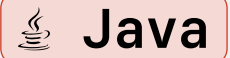
Typ	Datentyp	<u>Initialer Wert</u>
Ganzzahl & Zeichen	<u>byte</u> , <u>short</u> , <u>int</u> , <u>long</u> , <u>char</u>	0
Fließkommazahl	<u>float</u> , double	0.0
Wahrheitswert	<u>boolean</u>	<u>false</u>
Referenzvariable	Beliebige Klasse	null*

Figure 9: Initialwerte von Objektvariablen

3.3 Initialization of Classes

- Initiale Werte können auch in der Klasse selbst festgelegt werden.

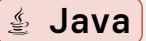
```
1  class Student {  
2      String name = "Unbekannt";  
3      int matrNumber;  
4      int enrolledYear = 2019;  
5  }
```



? Question

Was wird ausgegeben in dem folgenden Code?

```
1  public class StudentDemo {
2      public static void main(String[] args) {
3          Student lena = new Student();
4          System.out.println("Name:      " + lena.name);
5          System.out.println("Number:    " + lena.matrNumber);
6          System.out.println("Enrolled:  " + lena.enrolledYear);
7          lena.name = "Lena";
8          System.out.println("Name:      " + lena.name);
9      }
10 }
```



3.4 Assignment of References

- Nehmen Sie an, wir haben den folgenden Zustand in unserem Code:

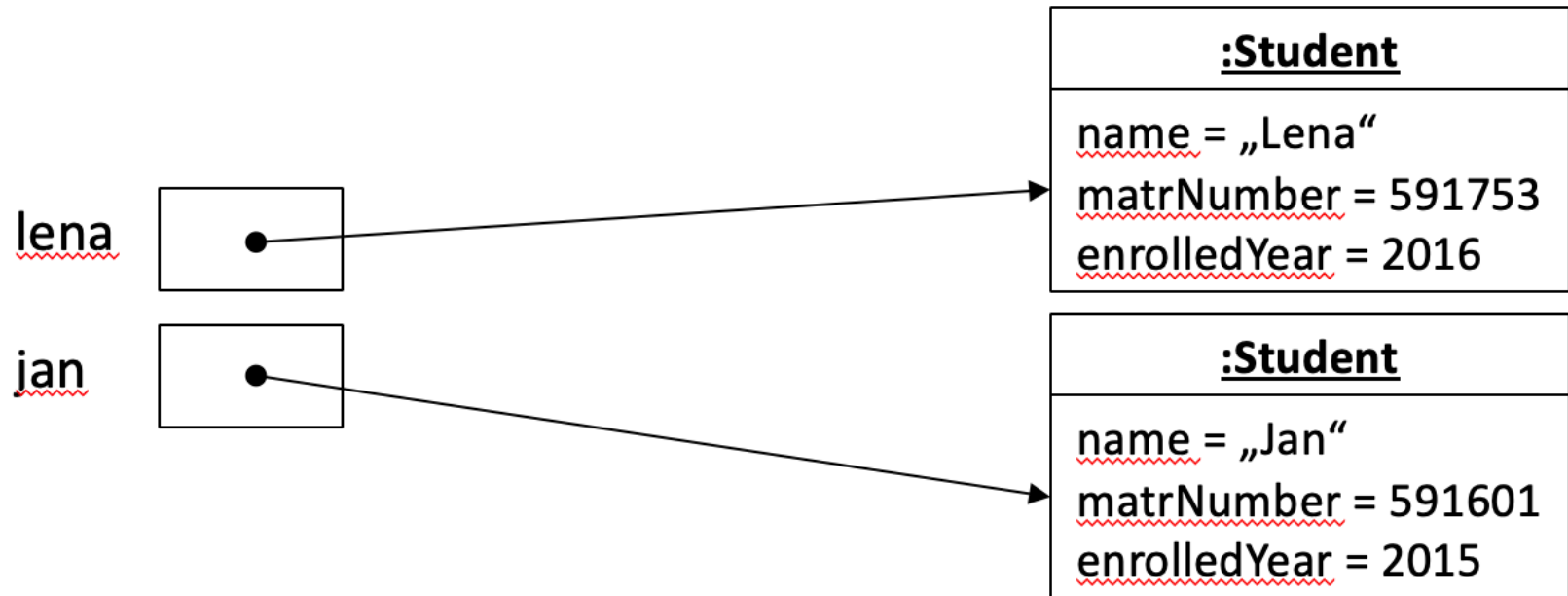
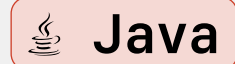


Figure 10: Zuweisung von Referenzen an Variablen

? Question

Was passiert jetzt, wenn wir den folgenden Code hinzufügen:

```
1 jan = lena;
```



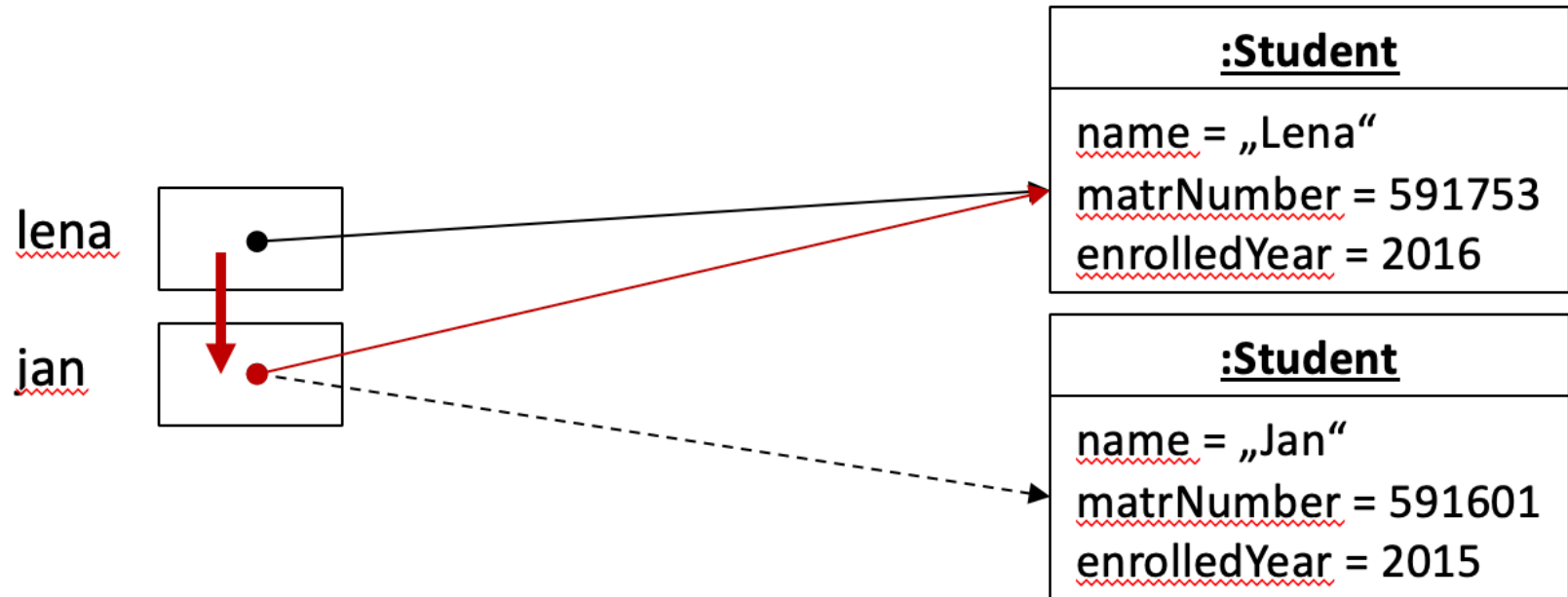


Figure 11: Verschieben von Referenzen

? Question

Was wird ausgegeben, wenn Sie den folgenden Code danach ausführen?

```
1 lena.name = "Birgit";  
2 jan.name = "Kai";  
3 System.out.println(lena.name);  
4 System.out.println(jan.name);
```

**Java**

3.4 Assignment of References

- Jan und Lena, referenzieren nun **dasselbe** Objekt. Änderungen von Werten über jan betreffen dann auch lena.

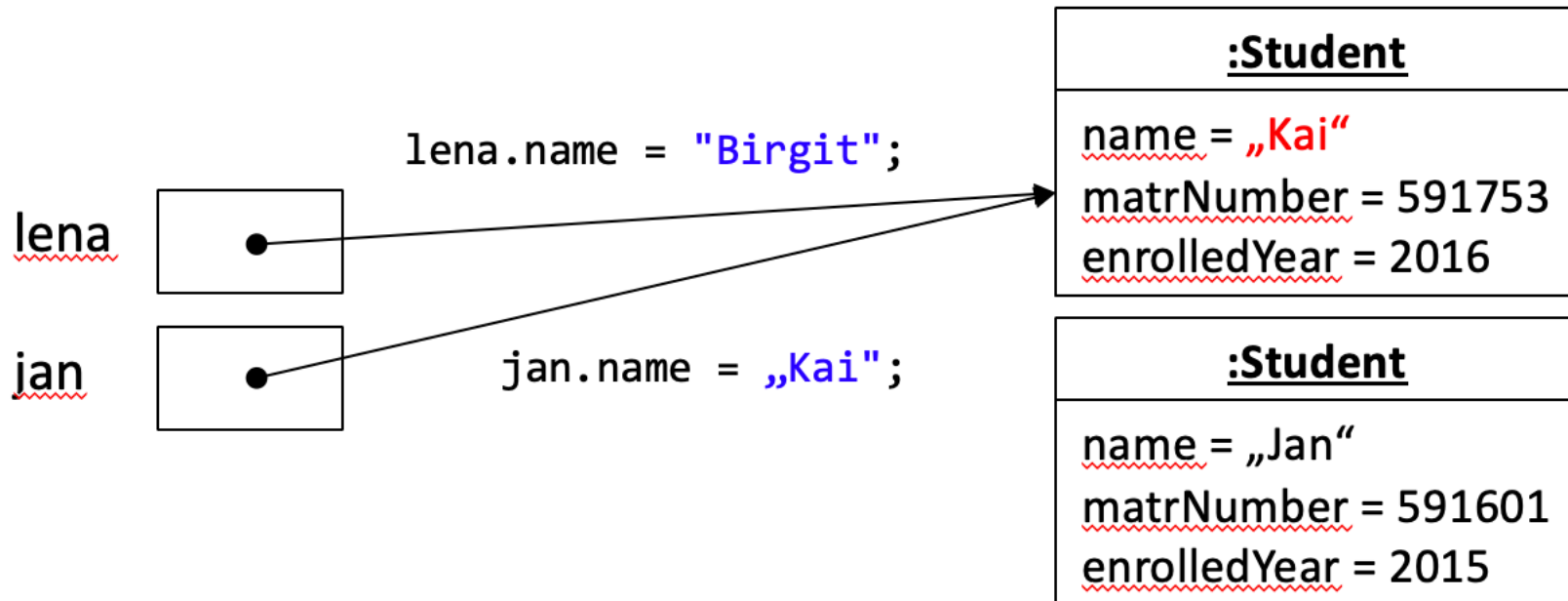


Figure 12: Beide Variablen zeigen zur gleichen Referenz

3.4 Assignment of References

- Das Objekt, was vorher über `j` an referenziert wurde, hat nun keine Referenz mehr.
- Damit gibt es **keine** Möglichkeit mehr, auf das Objekt zuzugreifen.
- Der **Garbage Collector** wird mittels **Reference Counting** den Speicher wieder freigeben.
- Es gibt kein `free` oder `delete` wie in C!

3.4 Assignment of References

3. Variablen und Speicher

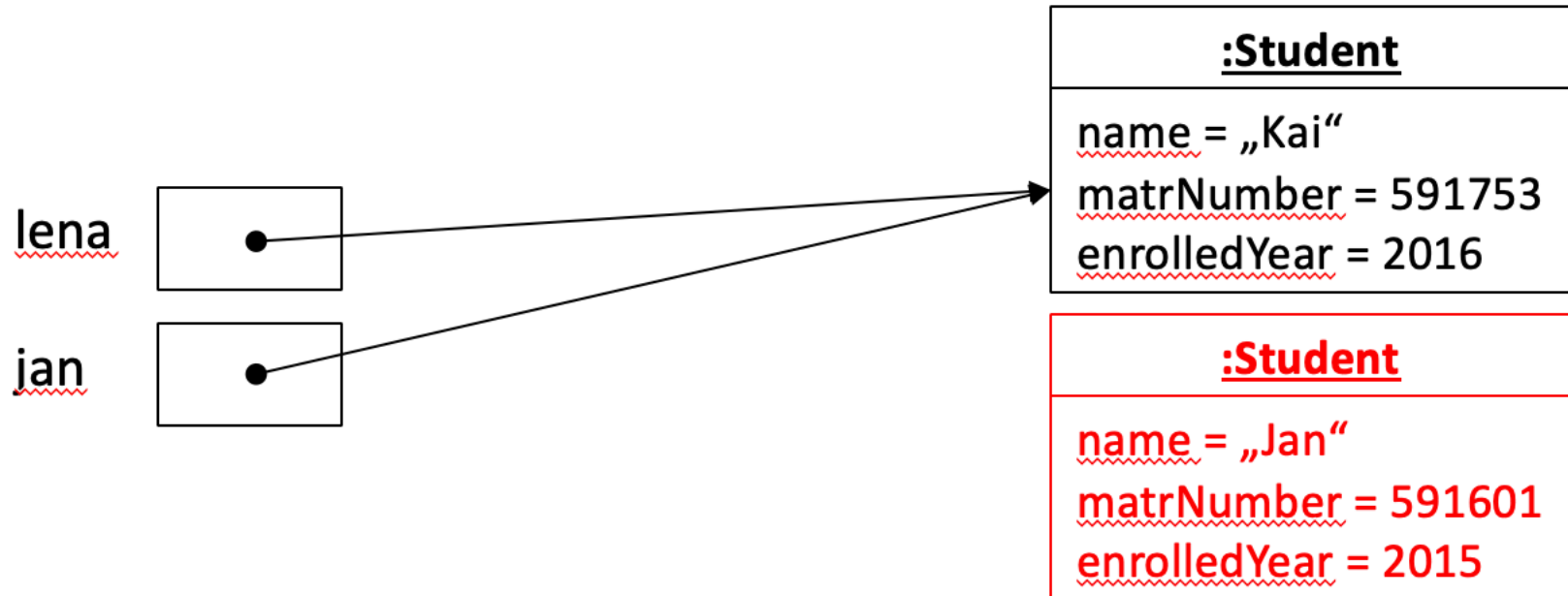


Figure 13: Der Garbage Collector gibt Speicher frei

4. Methoden

- Methoden entsprechen Funktionen aus C, die Sie ja bereits kennen.

```
1 Rückgabetyp Methodenname(Parameter) {  
2     Anweisung;  
3 }
```



- Rückgabetyp
 - ▶ Primitiver Datentyp, Klasse eines Objekts oder `void`.
 - ▶ Rückgabe erfolgt wie in C mittels `return`.
- Methodenname
 - ▶ Beliebiger gültiger Bezeichner (siehe Kapitel 2)

- ▶ Von unserem Coding Style: **camelCase** (ich erlaube auch **snake_case**)

- Methoden entsprechen Funktionen aus C, die Sie ja bereits kennen.

```
1 Rückgabetyp Methodenname(Parameter) {  
2     Anweisung;  
3 }
```



- Parameter
 - ▶ Leer oder durch Komma getrennte Parameter
 - ▶ Jeder Parameter ist in der Form: datentyp bezeichner
- Aufruf
 - ▶ Methodenname gefolgt von Klammern

- ▶ Argumente in den Klammern
- ▶ Ausdruck wird durch Rückgabewert ersetzt

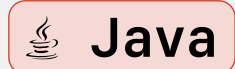
Task 3

Berechnen Sie den Mittelwert von zwei Fließkommazahlen.

Task 4

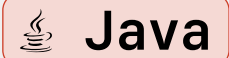
Berechnen Sie den Mittelwert von zwei Fließkommazahlen.

```
1 public class MathUtils {  
2     double average(double a, double b) {  
3         return (a + b) / 2.0;  
4     }  
5 }
```



4.2 Methods: Examples

```
1  public class MathUtilsDemo {
2      public static void main(String[] args) {
3          MathUtils math = new MathUtils();
4          double a1 = 3.5, a2 = 7;
5          double mean = math.average(a1, a2);
6          System.out.println(mean);
7          System.out.println(math.average(1.5, 3.2));
8      }
9  }
```



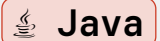
Task 5

Berechnen Sie die Summer aller Ziffern einer Ganzzahl.

Task 6

Berechnen Sie die Summer aller Ziffern einer Ganzzahl.

```
1  public class MathUtils {
2      int sumOfDigits(int number) {
3          int sum = 0;
4          while(number > 0) {
5              sum += number % 10;
6              number /= 10;
7          }
8          return sum;
9      }
10 }
```



4.2 Methods: Examples

```
1  public class MathUtilsDemo {
2      public static void main(String[] args) {
3          MathUtils math = new MathUtils();
4          System.out.println(math.sumOfDigits(0));
5          System.out.println(math.sumOfDigits(2016));
6      }
7  }
```



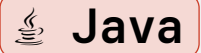
! Memorize

- **Getter**: Methode, die den Wert einer Instanzvariable zurückgibt
- **Setter**: Methode, die einer Instanzvariable einen (zu übergebenen) Wert zuweist

4.3 Getters and Setters

4. Methoden

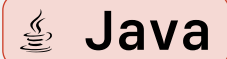
```
1  public class Student {  
2      String name;  
3  
4      void setName(String studentName){  
5          name = studentName;  
6      }  
7  
8      String getName(){  
9          return name;  
10     }  
11  
12 }
```



4.4 Methods: Call-by-Value

- Parameterübergabe bei Aufruf einer Methode:
 - ▶ Grundsätzlich der Wert der Variablen übergeben („Call by value“).
 - ▶ Nicht möglich, eine Art „Zeiger“ auf die Variable zu übergeben.
 - ▶ In Methode kann die beim Methodenaufruf verwendete Variable nicht verändert werden.

```
1 double square(double a) {  
2     a = a * a; // Local, does NOT modify b in main()  
3     return a;  
4 }
```



4.4 Methods: Call-by-Value

4. Methoden

Aufrufende Methode:

```
int a = -7;  
int b = 3;  
  
printAbs(a);  
  
System.out.println(a);  
...
```

Speicher:

a	-7
b	3

Aufgerufene Methode:

```
void printAbs(int a) {  
    if (a < 0) {  
        a = -a;  
    }  
    System.out.println(a);  
}
```

Speicher:

a	7
---	---

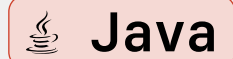
(nur bis Ende der Methode)

Figure 1: Diagramm für Call-by-Value

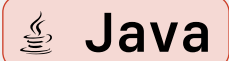
4.5 Methods: Objects as Parameters

- Referenzvariablen als Parameter:
 - ▶ Wert (d.h. gespeicherte Referenz) der übergebenen Variable wird nicht verändert.
 - ▶ Aber: Das referenzierte Objekt kann verändert werden!

```
1 public class CallByValueDemo {  
2     static void setNameBirgit(Student student) {  
3         student.name = "Birgit";  
4         System.out.println(student.name);  
5     }
```




```
1 public static void main(String[] args) {  
2     Student lena = new Student();  
3     lena.name = "Lena";  
4     setNameBirgit(lena);  
5     System.out.println(lena.name);  
6 }  
7 }
```



4.5 Methods: Objects as Parameters

- Ursprünglicher Zustand in `main()`-Methode:



Figure 2: Ausgangszustand

4.5 Methods: Objects as Parameters

- Aufruf der (unsinnigen) Methode `setNameBirgit()`:

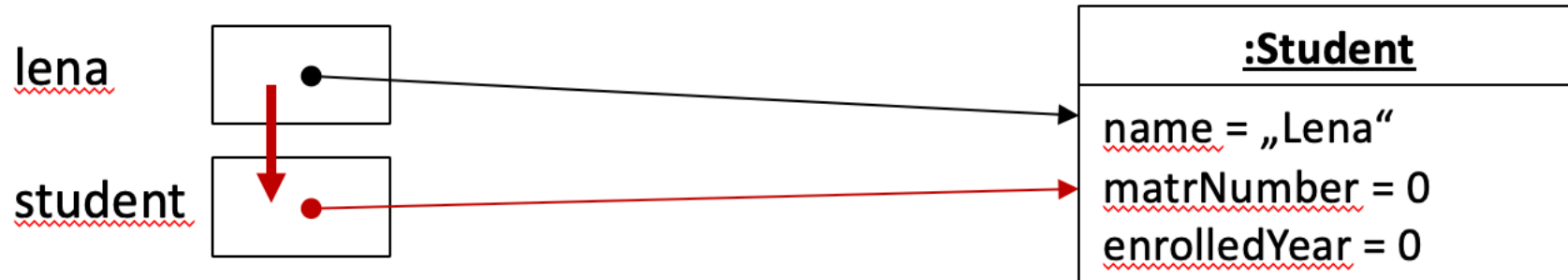


Figure 3: Verschieben der Referenz in die Methode

4.5 Methods: Objects as Parameters

- Änderung des Objektes in der Methode setNameBirgit():

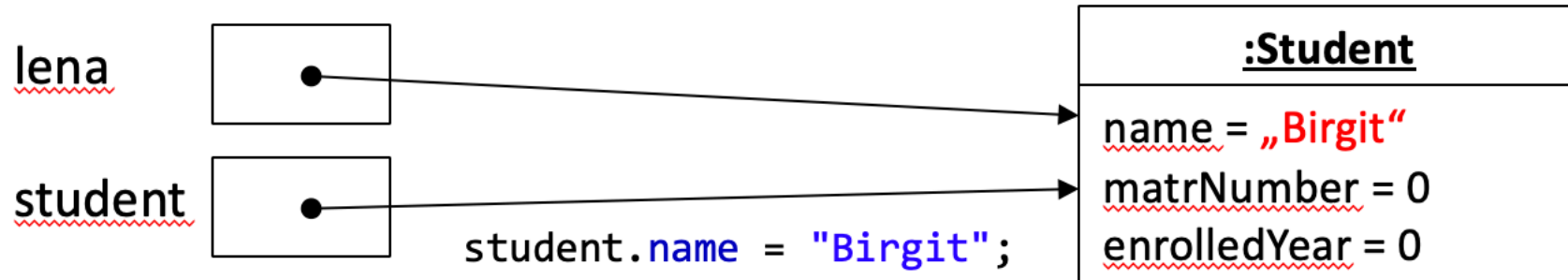
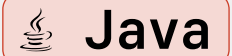


Figure 4: Änderung des Objekts in Methode

? Question

Schauen Sie sich den folgenden Code an. Was ist unschön?

```
1 public class Student {  
2     String name;  
3  
4     void setName(String newName) {  
5         name = newName;  
6     }  
7 }
```



- Wir würden gerne den Parameter des Setters auch `name` nennen!

? Question

Funktioniert der folgende Aufruf?

```
1 void setName(String name) {  
2     name = name;  
3 }
```



- Wir würden gerne den Parameter des Setters auch `name` nennen!

? Question

Funktioniert der folgende Aufruf?

```
1 void setName(String name) {  
2     name = name;  
3 }
```



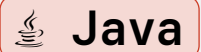
- Nein! Der Compiler würde jeweils auf die lokale Variable verwenden.

- Wie können wir auf die Instanzvariable zugreifen?

! Memorize

- Mittels `this` können wir auf die derzeitige Instanz zugreifen, in der wir uns gerade befinden.

```
1 public class Student {  
2     String name;  
3  
4     void setName(String name) {  
5         this.name = name;  
6     }
```



```
7    }
```

- Also
 - ▶ Wenn Sie auf eine lokale Variable zugreifen wollen, verwenden Sie einfach den Bezeichner der Variable
 - ▶ Wenn Sie auf eine Instanzvariable zugreifen wollen, verwenden Sie die `this`-Referenz mit dem Memberoperator mit `..`

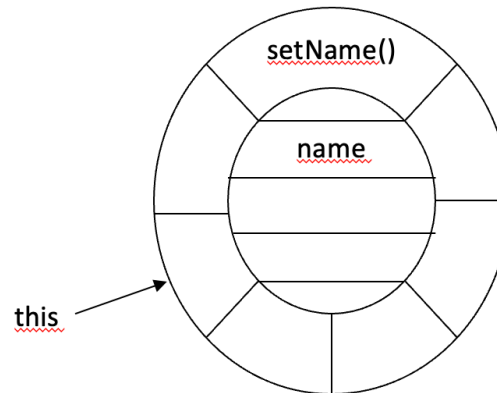
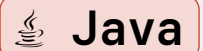


Figure 5: `this`-Referenz

Task 7

Schreiben Sie eine Methode, die das Maximum zweier Integerzahlen berechnet.

```
1 int max(int a, int b) {  
2     if (a > b) {  
3         return a;  
4     } else {  
5         return b;  
6     }
```

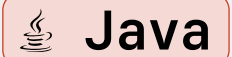


```
7 }
```

Task 8

Schreiben Sie eine weitere Methode in der selben Klasse, die das Maximum zweier `double`-Zahlen berechnet.

```
1  double max(double a, double b) {  
2      return (a > b) ? a : b; // Compact if/else syntax  
3  }
```



Task 9

Schreiben Sie noch eine weitere Methode in der selben Klasse, die das Maximum von drei Integerzahlen zurückgibt.

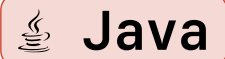
```
1  int max(int a, int b, int c) {  
2      return max(max(a, b), c);  
3  }
```



4.7 Overloading

- Überladene Methoden (overloading):
 - ▶ In einer Klasse existieren mehrere Methoden mit demselben Namen.
 - ▶ Nur möglich, falls Parametertypen unterschiedlich.
 - ▶ Compiler wählt anhand der Parameter die richtige Methode aus.

```
1 int max(int, int)
2 double max(double, double)
3 int max(int, int, int)
```

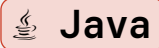


- Beachte:
 - ▶ Signatur: Methodenname und Parametertypen
 - ▶ Nur Datentypen der Parameter relevant (Unterscheidung durch Namen nicht möglich)
 - ▶ Unterscheidung durch Rückgabotyp reicht nicht (Warum?)

! Memorize

- Das sind nicht die gleichen Methoden!

```
1 int max(int a, int b) {  
2     if (a > b) {  
3         return a;  
4     } else {  
5         return b;  
6     }  
7 }
```



```
1 short max(short a, short b) {  
2     System.out.println("Aaaarrgrhhh!");  
3     return 7;  
4 }
```

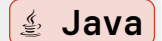


- Leerzeilen sollen die Lesbarkeit des Codes erhöhen.
- Eine Leerzeile steht zwingend zwischen Methoden oder Klassen (1).
- Eine Leerzeile steht in der Regel nach Deklarationen (2).
- Eine Leerzeile steht in der Regel zwischen logischen Abschnitten.



Example

```
1  public class MyClass {
2
3      public static int max(int a, int b) {
4          return (a > b) ? a : b;
5      }
6
7      public static void swap(Object a, Object b) {
8          Object temp;
9
10         temp = a;
11         a = b;
12         b = temp;
13     }
14 }
```

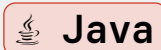


Java

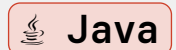
4.8 Methoden: Coding Style

- Leerzeichen sollen die Lesbarkeit des Codes erhöhen.
- Ein Leerzeichen steht zwingend zwischen Ausdrücken.
- Ein Leerzeichen steht in der Regel zwischen Operanden.

```
1 public static void  
  main(String[] args) {  
2     int a = 5;  
3  
4     for (int i = 1; i < 10; i++) {  
5         a *= i;  
6     }  
7 }
```



```
1 public static void  
  main(String[] args) {  
2     int a=5;  
3  
4     for(int i=1;i<10;i++){  
5         a*=i;  
6     }  
7 }
```



- Wir haben in diesem Abschnitt eine Menge geschafft!
 - ▶ Syntax von Methoden
 - ▶ Parameter (call by value)
 - ▶ this-Referenz
 - ▶ Überladen von Methoden
 - ▶ Coding Style

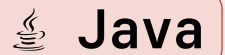
Lassen Sie uns nun betrachten, wie man Objekte gezielt erzeugen und initialisieren kann.

5. Konstruktoren

! Memorize

- Spezielle Methoden zur Initialisierung eines Objektes
 - ▶ Werden bei Erzeugung eines Objektes ausgeführt
 - ▶ Beachte: Konstruktoren haben keinen Rückgabetyp

```
1 Klassenname(Parameter) {  
2 Anweisungen;  
3 }
```



5.1 Konstruktoren

- Mittels des folgenden Codes könnte man einen Konstruktor für die Klasse Student erstellen:

```
1 public class Student {  
2     String name;  
3  
4     Student(String name) { // Achtung: Kein  
    Rückgabetyp  
5         this.name = name;  
6     }  
7 }
```



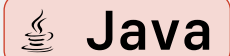
5.1 Konstruktoren

- Standardkonstruktor: Konstruktor mit leerer Parameterliste
- Initialisiert Instanzvariablen je nach Typ mit 0, 0.0, false oder null



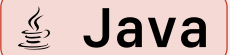
Example

```
1 public class Student {  
2     String name;  
3  
4     Student() {        // Standardkonstruktor  
5     }  
6 }
```



- Das vorige Beispiel könnte man auch so darstellen:

```
1 public class Student {  
2     String name;  
3  
4     Student() {        // Standardkonstruktor  
5         name = null;  
6     }  
7 }
```



5.1 Konstruktoren

- Compiler erzeugt unter bestimmten Bedingungen automatisch den Standardkonstruktor
- Einfache Regeln:
 1. Falls Sie für eine Klasse keinen Konstruktor schreiben:
 - Der Compiler erzeugt automatisch den Standardkonstruktor
 2. Falls Sie für eine Klasse mindestens einen Konstruktor schreiben:
 - Der Compiler erzeugt keinen Konstruktor
 - Es existieren nur die von Ihnen implementierten Konstruktoren

5.2 `this`: Referenz vs. Methode

- `this`-Referenz (zur Erinnerung):
 - ▶ Verwendung innerhalb beliebiger (Instanz-)Methode
 - ▶ Wie eine Variable verwendet
 - ▶ Enthält Referenz auf Objekt, für das die Methode aufgerufen wurde
- `this()`-Methode
 - ▶ Verwendung nur innerhalb eines Konstruktors.
 - ▶ Ist ein Methodenaufruf.
 - ▶ `this(Parameterliste)` ruft Konstruktor mit passender Parameterliste auf.
 - ▶ Darf nur als erste Anweisung im Konstruktor stehen.

5.2 this: Referenz vs. Methode

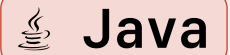
```
1  public class Aircraft {
2      String model, airline;
3      int numberEngines;
4
5      Aircraft() {
6          numberEngines = 1;
7      }
8
9      Aircraft(String model) {
10         this();
11         this.model = model;
12     }
13
14     Aircraft(String model, String airline) {
15         this(model);
16         this.airline = airline;
17     }
18 }
```



? Question

Wird der folgende Code kompilieren? Was denken Sie?

```
1 public class Aircraft {  
2     String model;  
3  
4     public static void main(String[] args) {  
5         Aircraft aircraft = new Aircraft();  
6     }  
7 }
```





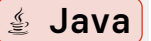
Success

- Ja, wird er!
 - ▶ Der Compiler erzeugt einen Standardkonstruktor, weil die Klasse keinen enthält.
 - ▶ Dieser wird dann in der `main()`-Methode nun aufgerufen!

? Question

Wird der folgende Code kompilieren? Was denken Sie?

```
1  public class Aircraft {  
2      String model;  
3  
4      Aircraft(String model) {  
5          this.model = model;  
6      }  
7  
8      public static void main(String[] args) {  
9          Aircraft aircraft = new Aircraft();  
10     }  
11 }
```



✗ Error

- Nein, wird er nicht!
 - ▶ Da die Klasse einen Konstruktor enthält, erzeugt der Compiler keinen Standardkonstruktor.
 - ▶ Der verwendete Konstruktor in der `main()`-Methode existiert daher nicht.

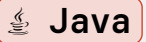
☰ Task 10

- Lassen Sie uns eine Klasse Circle schreiben:
 - ▶ Repräsentiert einen geometrischen Kreis
 - Dargestellt durch
 - x- und y-Koordinate des Mittelpunktes
 - Radius r
 - ▶ Konstruktoren:
 - Standardkonstruktor erzeugt Einheitskreis um den Koordinatenursprung (0 ; 0)
 - Konstruktor mit x, y und Radius als Parametern
 - Konstruktor mit Radius als Parameter

5.4 Aufgabe: Klasse Circle

5. Konstruktoren

```
1  public class Circle {
2      double x, y, radius;
3
4      public Circle(double x, double y, double radius) {
5          this.x = x;
6          this.y = y;
7          this.radius = radius;
8      }
9
10     Circle() {
11         this(0.0, 0.0, 1.0);
12     }
13
14     Circle(double radius) {
15         this(0.0, 0.0, radius);
16     }
17 }
```



5.4 Aufgabe: Klasse Circle

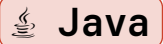
5. Konstruktoren

```
18  Circle(Circle circle) {  
19      this(circle.x, circle.y, circle.radius);  
20  }  
21 }
```

5.4 Aufgabe: Klasse Circle

- Die Klasse, die wir eben geschrieben haben, können wir wie folgt erzeugen:

```
1 public class CircleDemo {  
2     public static void main(String[] args) {  
3         Circle circle1 = new Circle();  
4         Circle circle2 = new Circle(2.5);  
5         Circle circle3 = new Circle(circle2);  
6         Circle circle4 = new Circle(-1.2, 7.1, 3.0);  
7     }  
8 }
```



Tip

Folgender Menüpunkt in IntelliJ IDEA kann Ihnen viel Arbeit ersparen: Code / Generate / Constructor

Task 11

Erweitern Sie die Klasse um Getter und Setter-Methoden.

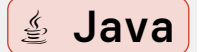
Tip

Über die Menüpunkte Code / Generate / Getter and Setter können Sie sich weitere Zeit sparen!

5.4 Aufgabe: Klasse Circle

5. Konstruktoren

```
1  double getX() {  
2      return x;  
3  }  
4  
5  // getY() und getRadius() entsprechend  
6  
7  void setX(double x) {  
8      this.x = x;  
9  }  
10  
11 void setY(double y) {  
12     this.y = y;  
13 }  
14
```



Java

5.4 Aufgabe: Klasse Circle

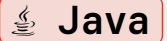
```
15 void setRadius(double radius) {  
16     if (radius >= 0.0) {           // Nicht erlaubte Daten verhindern  
17         this.radius = radius;  
18     }  
19 }
```

6. Klassenvariablen & Klassenmethoden

? Question

Glauben Sie, dass Sie die Variable PI nur einmal im Speicher reservieren?

```
1 public class Circle {  
2     double x, y, radius;  
3     final double PI = 3.141592653589793;  
4  
5     double getArea() {  
6         return PI * radius * radius;  
7     }  
8 }
```



Java

✗ Error

- Antwort: Nein, für jedes Objekt erneut!

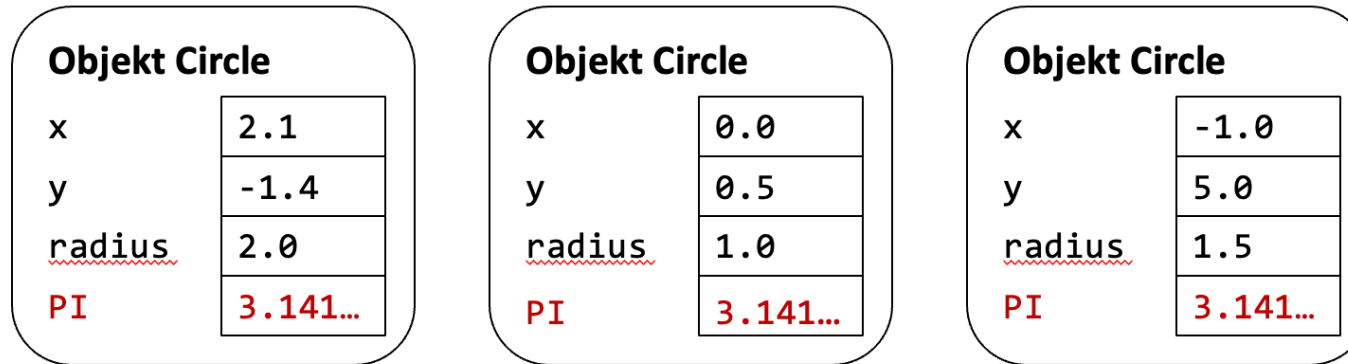
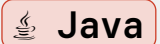


Figure 6: Speicher wird für jedes Objekt erneut reserviert.

? Question

Was meinen Sie, zählt die Variable `count` die Anzahl der Objekte?

```
1 public class Circle {
2     double x, y, radius;
3     int count;    // Count number of objects created
4
5     Circle() {
6         radius = 1.0;
7         count++;
8     }
9 }
```



✗ Error

- Antwort: Nein, jedes Objekt bekommt eine neue Variable count!

Objekt Circle	
x	2.1
y	-1.4
<u>radius</u>	2.0
<u>count</u>	1

Objekt Circle	
x	0.0
y	0.5
<u>radius</u>	1.0
<u>count</u>	1

Objekt Circle	
x	-1.0
y	5.0
<u>radius</u>	1.5
<u>count</u>	1

Figure 7: Die Variable count wird immer wieder neu gemacht.

- Eine Lösung sind **Klassenvariablen**!
- Klassenvariablen werden nur einmal für die gesamte Klasse angelegt
- Werden nicht für ein bestimmtes (nicht für jedes Objekt) angelegt
- Werden bereits bei Programmstart (Laden der Klasse) erzeugt
 - ▶ Sie existieren auch dann, wenn es (noch) kein Objekt der Klasse gibt.
- Syntax: Variable mit Schlüsselwort static

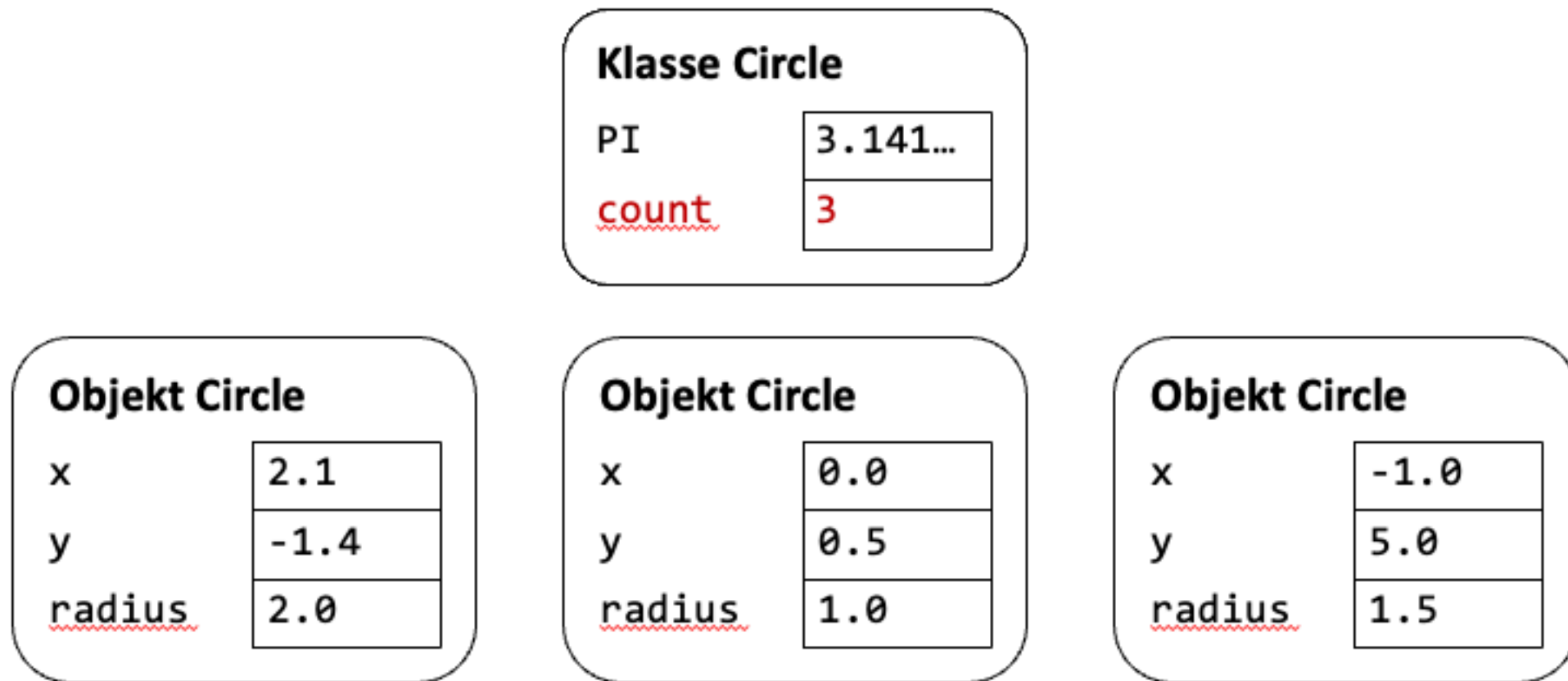


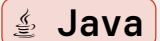
Figure 8: Klassenattribute in den verschiedenen Objekten

6.1 Klassenvariablen

6. Klassenvariablen & Klassenmethoden

- Klasse Circle mit Klassenvariablen:

```
1  public class Circle {  
2      double x, y, radius;  
3      static final double PI = 3.141592653589793;  
4      static int count;  
5  
6      double getArea() {  
7          return PI * radius * radius;  
8      }  
9  
10     Circle() {  
11         radius = 1.0;  
12         count++;  
13     }  
14 }
```



6.1 Klassenvariablen

6. Klassenvariablen & Klassenmethoden

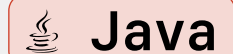
- Zugriff auf Klassenvariablen
 - ▶ Innerhalb der Methode der Klasse entspricht es dem Zugriff auf Instanzvariablen
 - ▶ Außerhalb der Klasse `Klassenname.Variablenname`

```
1  public class CircleDemo {
2      public static void main(String[] args) {
3          Circle circle1 = new Circle();
4          Circle circle2 = new Circle();
5          Circle circle3 = new Circle();
6
7          System.out.println("Anzahl Objekte: " + Circle.count);
8      }
9  }
```



- Vollkommen analog zu Klassenvariablen:
 - ▶ Klassenmethoden werden für eine Klasse aufgerufen
 - ▶ Werden nicht für ein bestimmtes Objekt aufgerufen
 - ▶ Methode wird durch Schlüsselwort `static` zur Klassenmethode
 - ▶ Können aufgerufen werden, ohne dass Objekt der Klasse erzeugt wurde
- Aufruf außerhalb der Klasse:

```
1 Klassenname.Methodenname
```



! Memorize

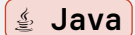
- Wichtige Konsequenzen:
 - ▶ `this`-Referenz existiert nicht in Klassenmethoden
 - ▶ Instanzvariablen existieren nicht in Klassenmethoden



Example

- In diesem Beispiel sind zwei Klassenmethoden!

```
1  public class Circle {
2      double x, y, radius = 1.0;
3
4      static double getPi() {
5          return 3.141592653589793;
6      }
7  }
8
9  public class CircleDemo {
10     public static void main(String[] args) {
11         System.out.println("Pi: " + Circle.getPi());
12     }
13 }
```



7. License Notice

7.1 Attribution

- This work is shared under the CC BY-NC-SA 4.0 License and the respective Public License
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>
- This work is based off of the work Prof. Dr. Marc Hensel.
- Some of the images and texts, as well as the layout were changed.
- The base material was supplied in private, therefore the link to the source cannot be shared with the audience.