# Object-Oriented Programming in Java

## Lecture 3 - Classes and Objects

Emily Lucia Antosch

HAW Hamburg

13.08.2025

## Contents

# 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. 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

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

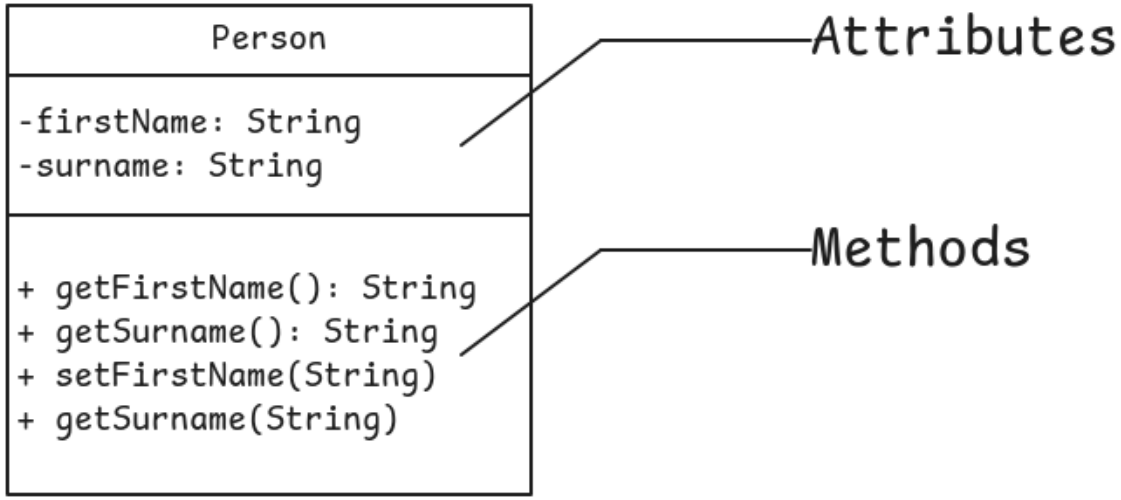


Figure 1: UML notation of a Person class

# 2.1 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**

# 2.1 Class

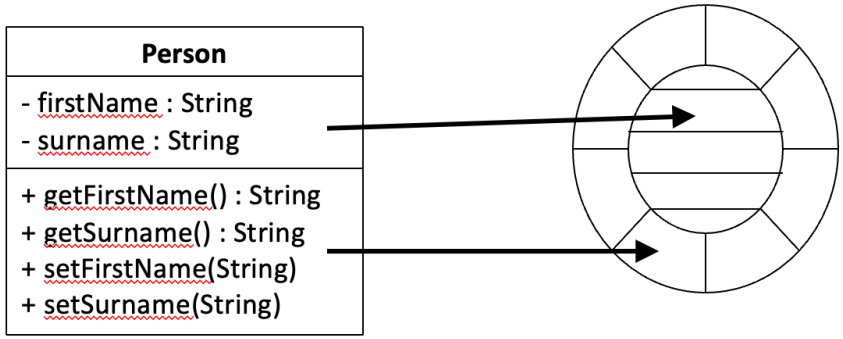


Figure 2: Division of methods and attributes

# 2.2 Relationship Between Class and Object

- Class: Description (“blueprint”) of a data type
  - Object of a class: Created element of the data type
  - Any number of objects of a class can be created.

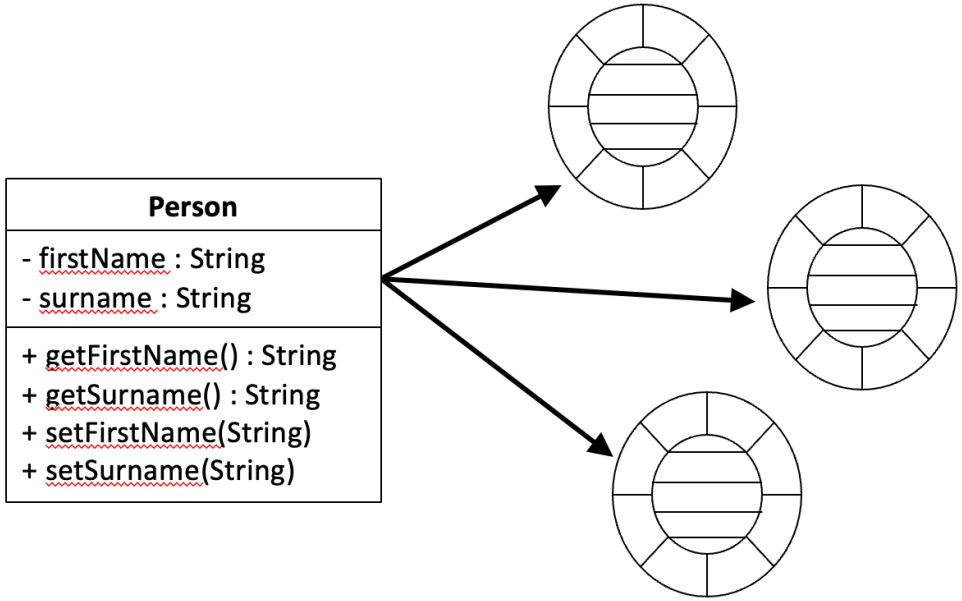


Figure 3: Multiple objects from one class

# 2.3 Classes in Java

- Classes can be declared using the following code:

```java
class ClassName {
    Attributes
    Methods
    }
```

Java

🔥 **Tip**

Create each class in its own file!

# 2.4 Example: Simple Class



## Task 1

Let us create this simple class:

- Class Student, described by name, student number and year of enrollment

# 2.4 Example: Simple Class

# 2.4 Example: Simple Class

## Task 2

Let us create this simple class:

- Class Student, described by name, student number and year of enrollment

Java

```java
class Student {
    String name;
    int matrNumber;
    int enrolledYear;
  }
```

# 2.4 Example: Simple Class

- The class has neither methods nor data encapsulation against external influence.

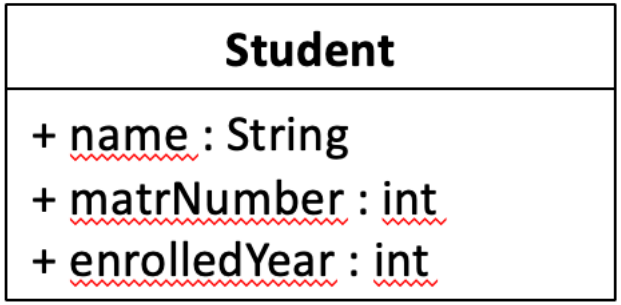


Figure 4: UML representation of the class we just created

# 2.5 Example: One Class, Many Objects

- Class ("One class for all students"):
  - The class is a new data type.
  - Defines what data describes students
- Objects ("A separate object for each student"):
  - Objects are instances in memory.
  - Have the structure of the class, but are filled with data
  - Any number of objects can be created.

# 2.5 Example: One Class, Many Objects

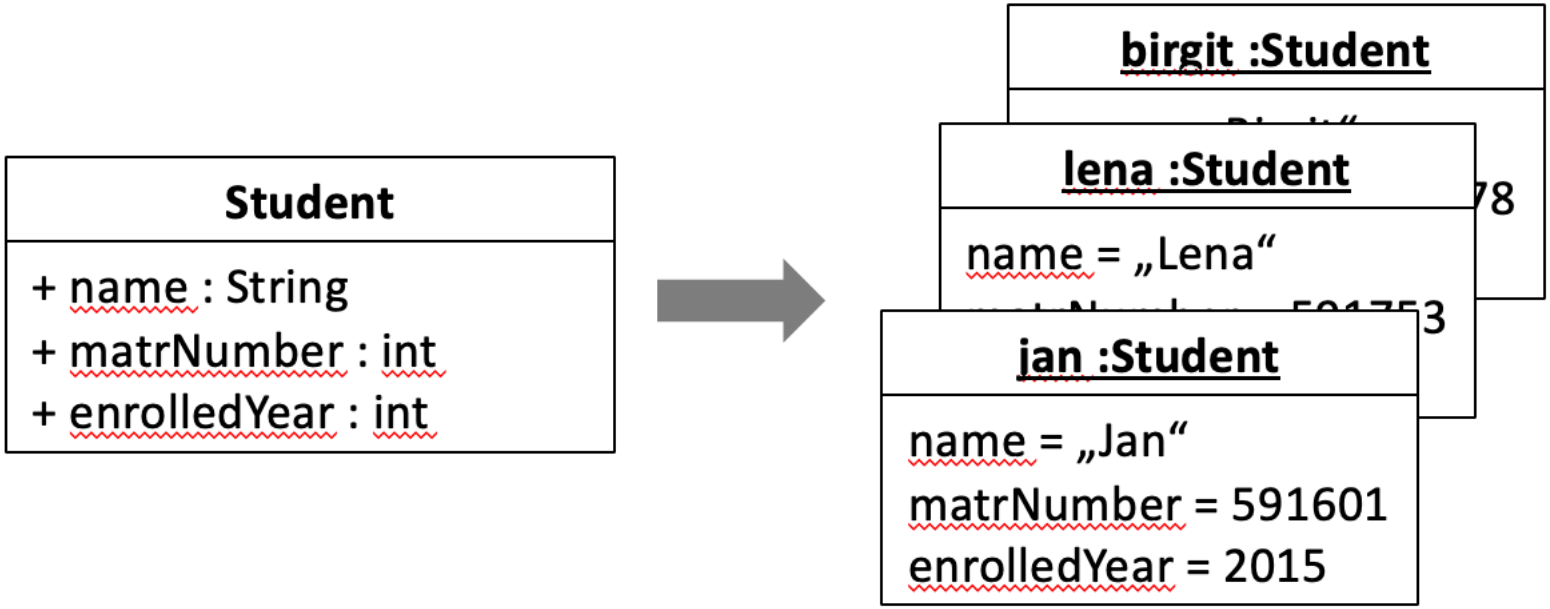


Figure 5: Multiple objects can be created from one class

# 2.6 Example: Local Variables



## ? Question

What values do the variables `count`, `jan` and `lena` have?

Java

```java
public class StudentDemo {
    public static void main(String[] args) {
        int count;
        Student lena, jan;
    }
}
```

# 2.6 Example: Local Variables

## ? Question

What values do the variables `count`, `jan` and `lena` have?

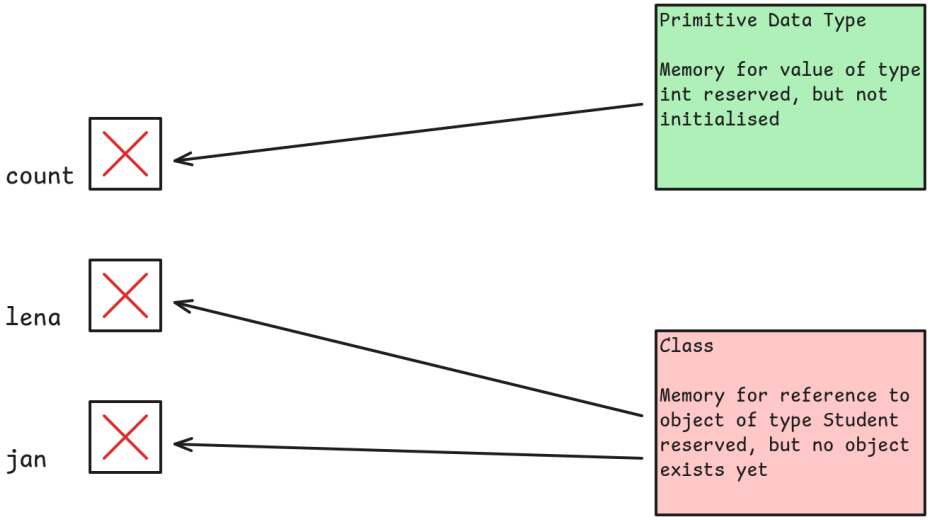


Figure 6: Primitive data types vs. Objects

# 2.7 Example: new Operator

- Objects are created using the new operator.

Java

```java
public class StudentDemo {
    public static void main(String[] args) {
        int count;
        Student lena, jan;
        lena = new Student();
    }
}
```

new-Operator

# 2.7 Example: `new` Operator

- Step 1: `new` operator creates object.
  - Reserve memory space for object (with object variables).
  - Initialize object variables with default values (more on this soon).

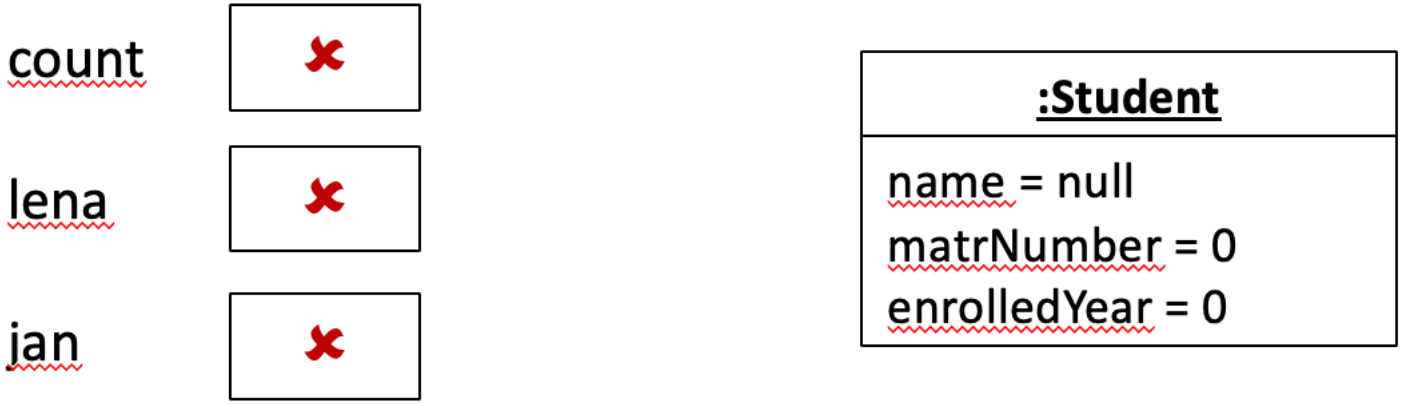


Figure 7: Creating reference with `new`

# 2.7 Example: `new` Operator

- Step 2: Assignment
  - Writes reference ("address") of the new object to variable `lena`.
  - Is independent of the `new` operator and the creation of the object

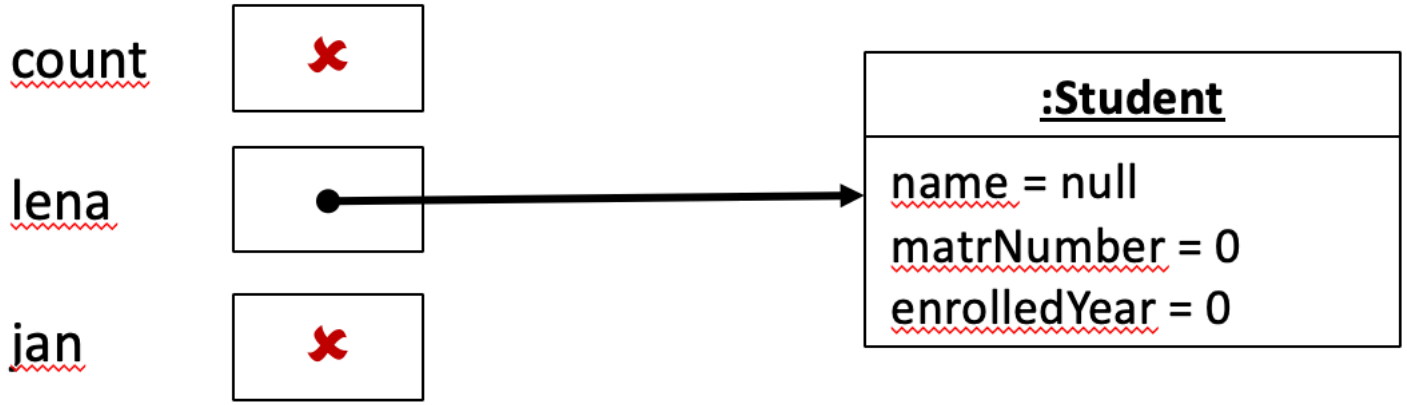


Figure 8: Assignment of reference to variable

# 3. Variables and Memory

# 3.1 Objective

## Conclusion

- We have already looked at this:
  - What are classes and objects?
  - How do you declare classes?
  - How do you create objects?

- In the following we want to look at the following aspects:
  - Access to object variables
  - Initialization of object variables
  - Assignment of references

# 3.1 Objective

- Automatic memory management

# 3.2 Access to Object Variables

- Access to object variables is done using the dot operator:

```java
ObjectReference.Member
```

- The `ObjectReference` is a reference to an object that is stored in a variable.
- `Member` is, for example, an attribute/object variable

# 3.2 Access to Object Variables

## ? Question

What will be output?

Java

```java
public class StudentDemo1 {
    public static void main(String[] args) {
      Student lena = new Student();
      System.out.println("Enrolled: " + lena.enrolledYear);
      lena.name = "Lena";
      lena.matrNumber = 591753;
      lena.enrolledYear = 2012;
      System.out.println("Enrolled: " + lena.enrolledYear);
    }
  }
```

# 3.3 Initialization of Classes

**! Memorize**

- **Object/Instance variable**: Declared in class as an attribute of an object.
- **Local variable**: Declared locally (e.g. in method or loop).
- **Reference variable**: Has class as data type, can store reference to object.

# 3.3 Initialization of Classes

- As a reminder:
  - Local variables are not automatically initialized. (Compiler prevents access.)
  - Object variables, however, are initialized when an object is created.

| Type | Data Type | Initial Value |
|---|---|---|
| Integer and Character | `byte, short, int, long, char` | 0 |
| Floating Point | `float, double` | 0.0 |
| Truth | `boolean` | `false` |
| Reference | Any Class | `null` |

Table 1: Value ranges of data types

# 3.3 Initialization of Classes

- Initial values can also be set in the class itself.

Java

```java
class Student {
  String name = "Unknown";
  int matrNumber;
  int enrolledYear = 2019;
}
```

# 3.3 Initialization of Classes

## ? Question

What will be output in the following code?

```java
public class StudentDemo {
    public static void main(String[] args) {
      Student lena = new Student();
      System.out.println("Name:     " + lena.name);
      System.out.println("Number:   " + lena.matrNumber);
      System.out.println("Enrolled: " + lena.enrolledYear);
      lena.name = "Lena";
      System.out.println("Name:     " + lena.name);
    }
  }
```

# 3.4 Assignment of References

- Assume we have the following state in our code:

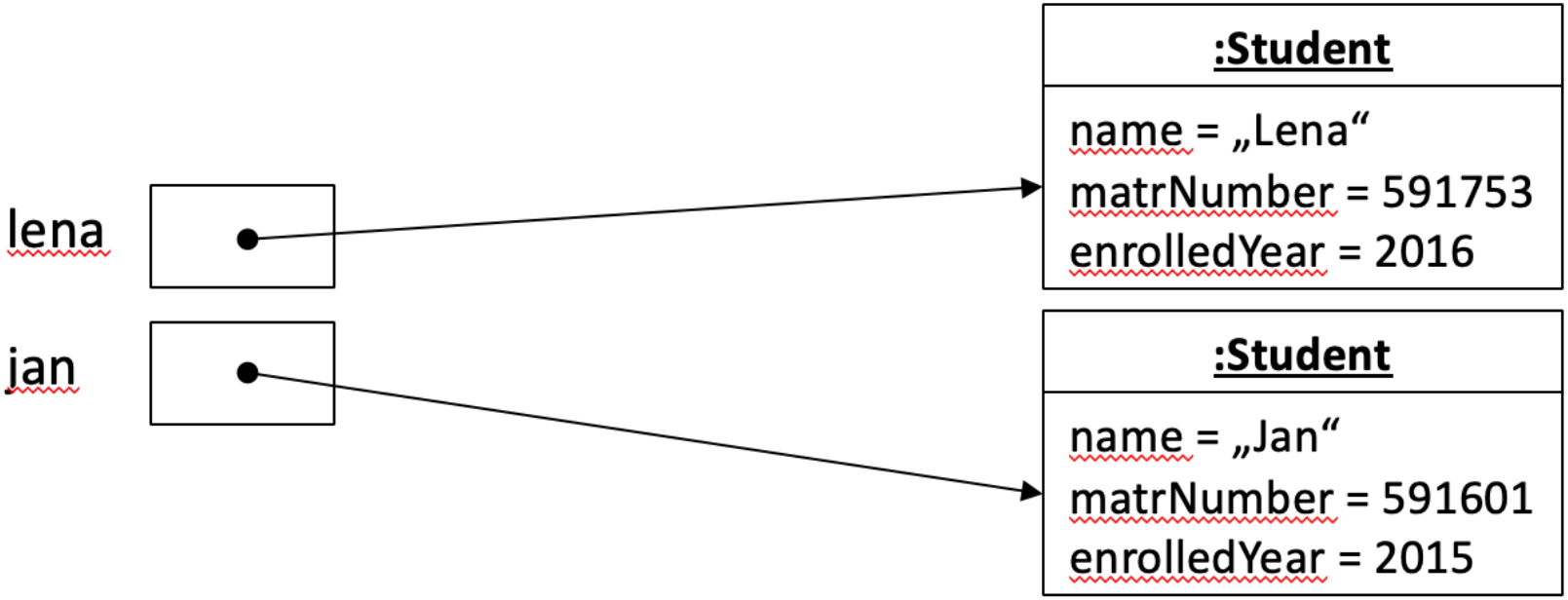


Figure 9: Assignment of references to variables

# 3.4 Assignment of References

## ? Question

What happens now if we add the following code:

```java
jan = lena;
```

# 3.4 Assignment of References

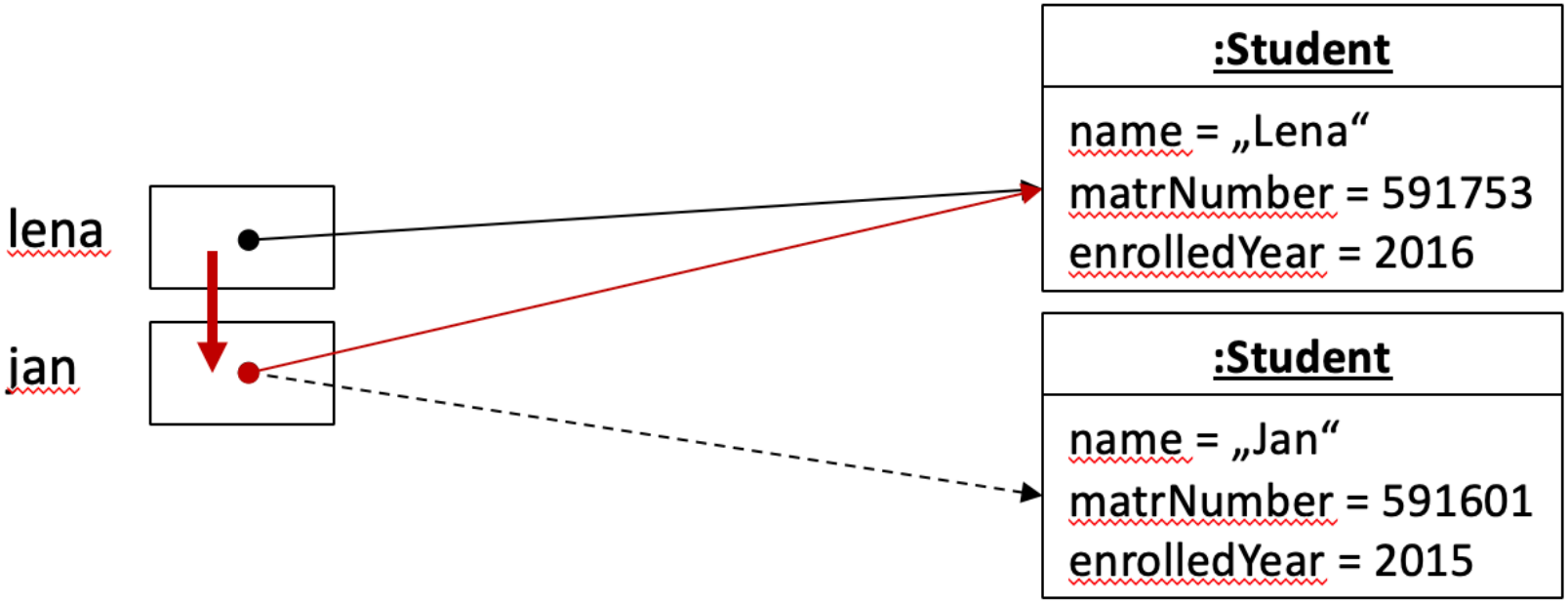


Figure 10: Moving references

# 3.4 Assignment of References

## ? Question

What will be output when you execute the following code afterwards?

```java
lena.name = "Birgit";
jan.name = "Kai";
System.out.println(lena.name);
System.out.println(jan.name);
```

# 3.4 Assignment of References

- Jan and Lena now reference the **same** object. Changes to values via `jan` also affect `lena`.

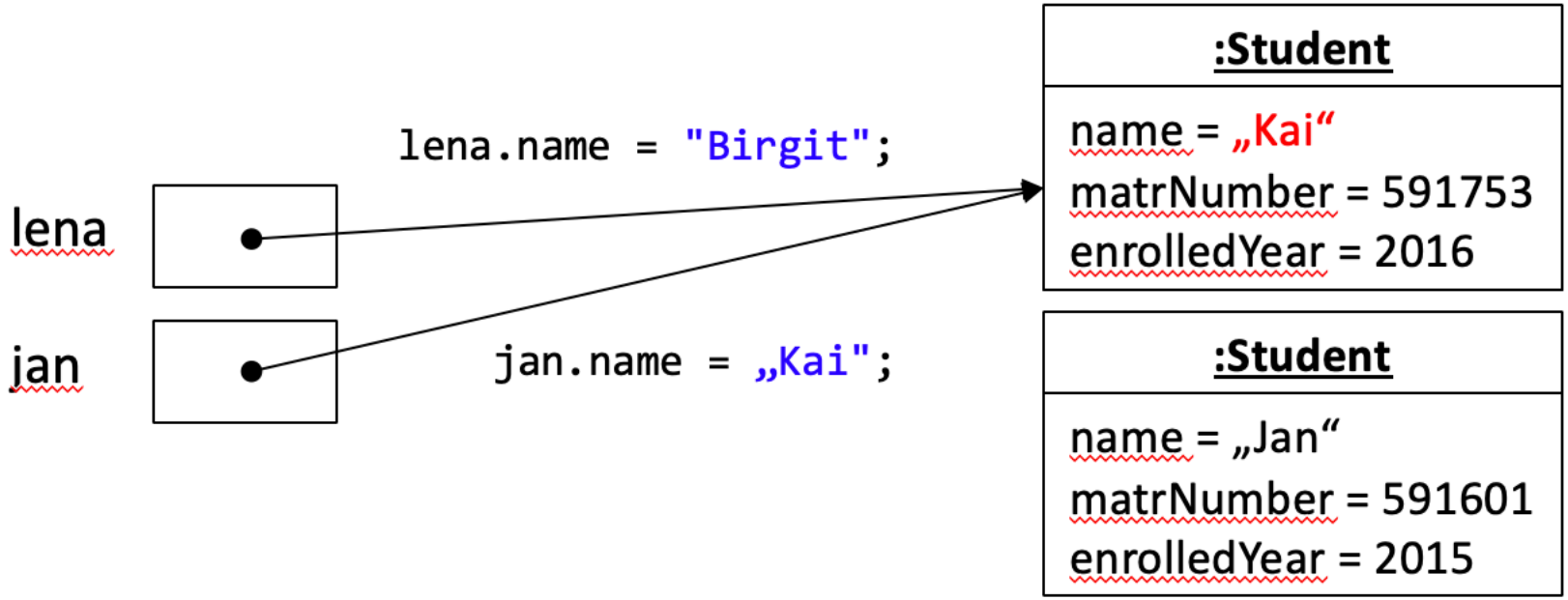


Figure 11: Both variables point to the same reference

# 3.4 Assignment of References

- The object that was previously referenced via `jan` now has no reference anymore.
- Thus there is **no** way to access the object anymore.
- The **Garbage Collector** will free the memory again using **Reference Counting**.
- There is no `free` or `delete` like in C!

# 3.4 Assignment of References

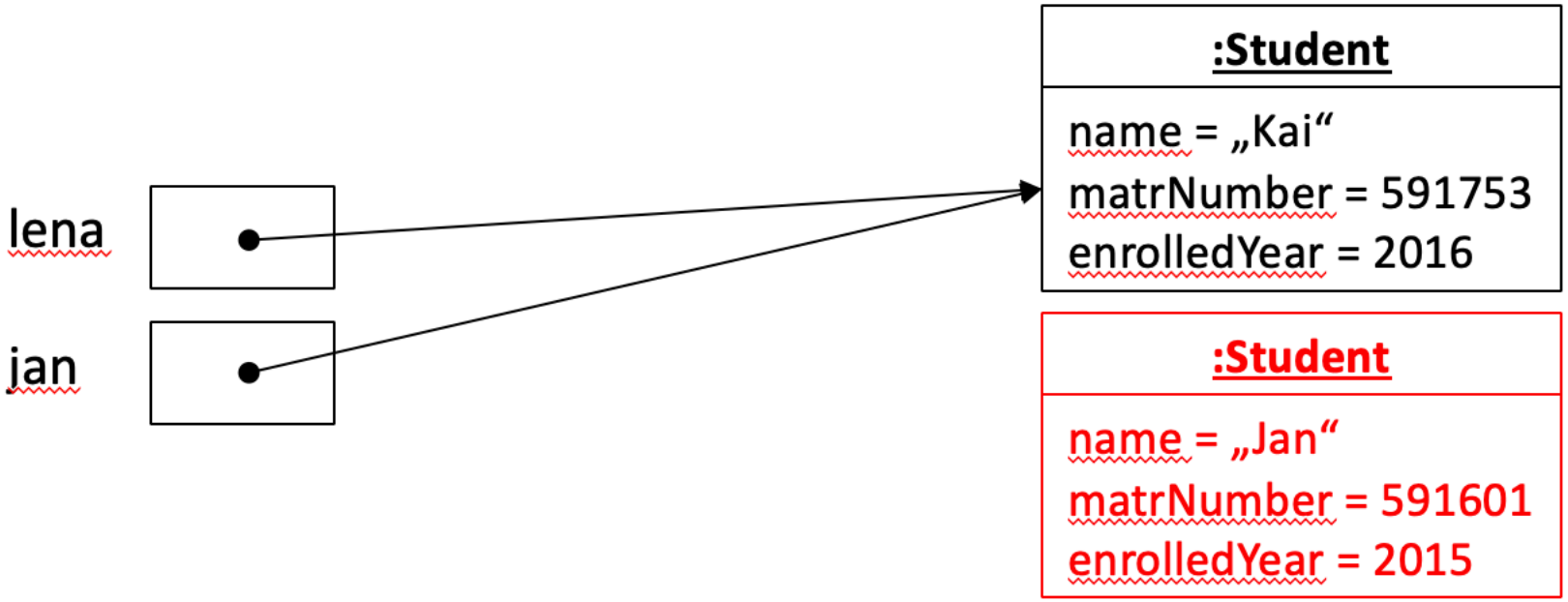


Figure 12: The Garbage Collector frees memory

# 4. Methods

# 4.1 Methods: Syntax

- Methods correspond to functions from C, which you already know.

```java
ReturnType MethodName(Parameters) {
 Statement;
}
```

- Return type
  - Primitive data type, class of an object or `void`.
  - Return is done as in C using `return`.
- Method name
  - Any valid identifier (see Chapter 2)
  - From our coding style: **camelCase** (I also allow **snake_case**)

# 4.1 Methods: Syntax

- Methods correspond to functions from C, which you already know.

```java
ReturnType MethodName(Parameters) {
 Statement;
}
```

- Parameters
  - Empty or comma-separated parameters
  - Each parameter is in the form: `datatype identifier`
- Call
  - Method name followed by parentheses
  - Arguments in the parentheses
  - Expression is replaced by return value

# 4.2 Methods: Examples



## Task 3

Calculate the average of two floating-point numbers.

# 4.2 Methods: Examples

## Task 4

Calculate the average of two floating-point numbers.

Java

```java
public class MathUtils {
        double average(double a, double b) {
            return (a + b) / 2.0;
        }
    }
```

# 4.2 Methods: Examples



Java

```java
public class MathUtilsDemo {
    public static void main(String[] args) {
        MathUtils math = new MathUtils();
        double a1 = 3.5, a2 = 7;
        double mean = math.average(a1, a2);
        System.out.println(mean);
        System.out.println(math.average(1.5, 3.2));
    }
}
```

# 4.2 Methods: Examples



## Task 5

Calculate the sum of all digits of an integer.

# 4.2 Methods: Examples

## Task 6

Calculate the sum of all digits of an integer.

Java

```java
public class MathUtils {
    int sumOfDigits(int number) {
        int sum = 0;
        while(number > 0) {
            sum += number % 10;
            number /= 10;
        }
        return sum;
    }
}
```

# 4.2 Methods: Examples



Java

```java
public class MathUtilsDemo {
    public static void main(String[] args) {
        MathUtils math = new MathUtils();
        System.out.println(math.sumOfDigits(0));
        System.out.println(math.sumOfDigits(2016));
    }
}
```

# 4.3 Getters and Setters

**! Memorize**

- **Getter**: Method that returns the value of an instance variable
- **Setter**: Method that assigns a (passed) value to an instance variable

# 4.3 Getters and Setters



Java

```java
public class Student {
  String name;

  void setName(String studentName){
    name = studentName;
  }

  String getName(){
    return name;
  }

}
```

# 4.4 Methods: Call-by-Value

- Parameter passing when calling a method:
  - Basically the value of the variable is passed ("Call by value").
  - Not possible to pass a kind of "pointer" to the variable.
  - In method, the variable used in the method call cannot be changed.

Java

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

# 4.4 Methods: Call-by-Value

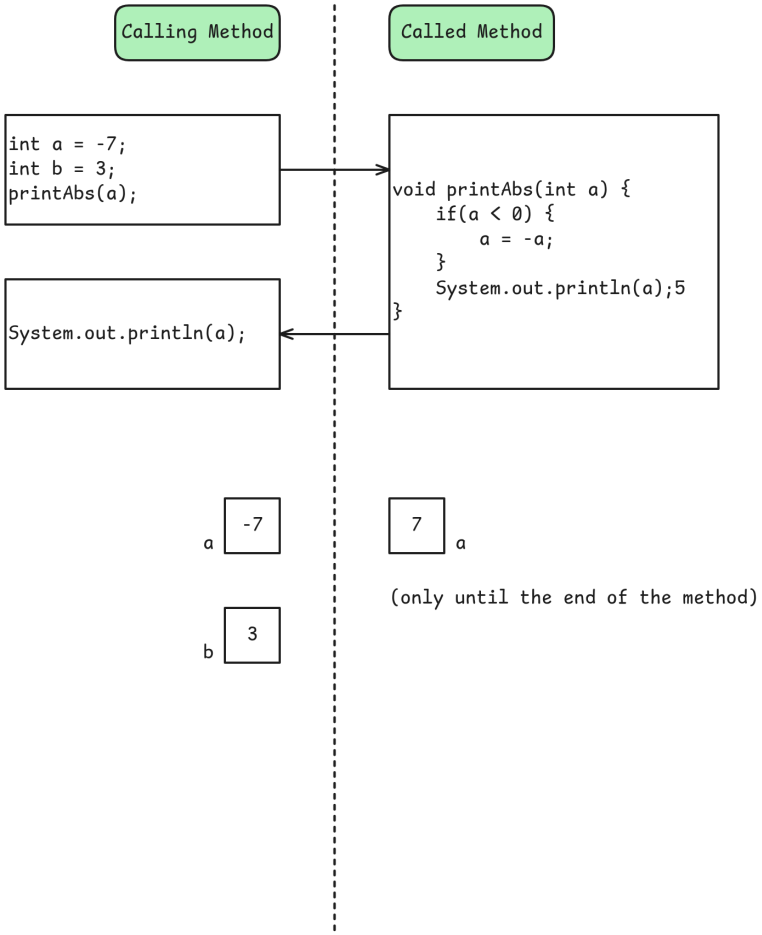


Figure 1: Diagram for Call-by-Value

# 4.5 Methods: Objects as Parameters

- Reference variables as parameters:
  - Value (i.e. stored reference) of the passed variable is not changed.
  - But: The referenced object can be changed!

Java

```java
public class CallByValueDemo {
        static void setNameBirgit(Student student) {
            student.name = "Birgit";
            System.out.println(student.name);
        }
```

# 4.5 Methods: Objects as Parameters



Java

```java
public static void main(String[] args) {
            Student lena = new Student();
            lena.name = "Lena";
            setNameBirgit(lena);
            System.out.println(lena.name);
        }
    }
```

# 4.5 Methods: Objects as Parameters

- Original state in `main()` method:

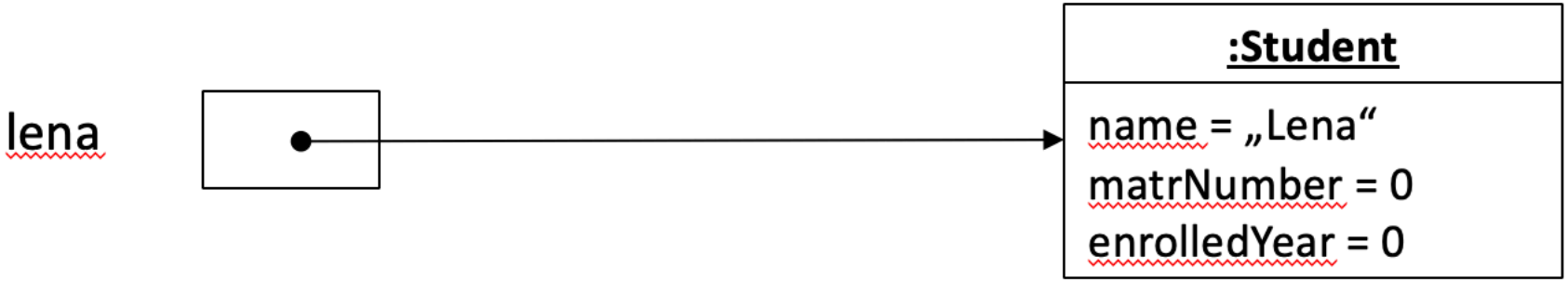


Figure 2: Initial state

# 4.5 Methods: Objects as Parameters

- Call of the (meaningless) method `setNameBirgit()`:

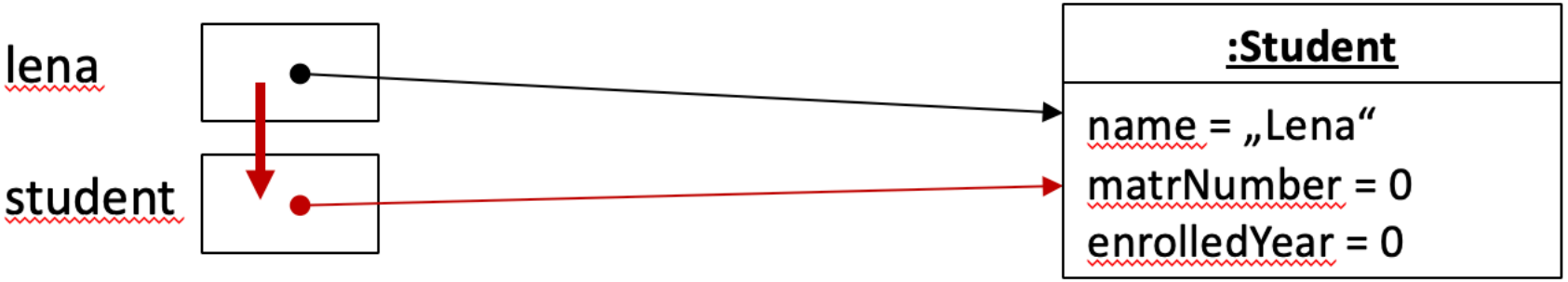


Figure 3: Moving the reference to the method

# 4.5 Methods: Objects as Parameters

- Modification of the object in the method `setNameBirgit()`:

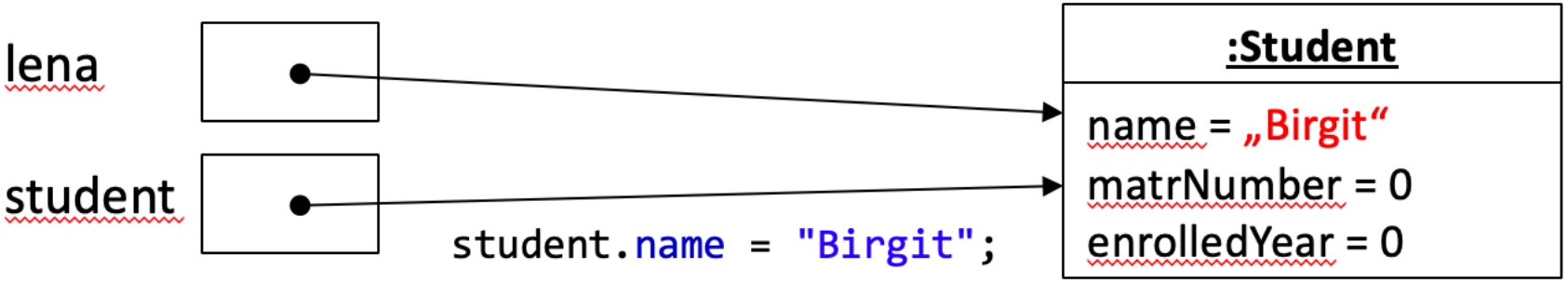


Figure 4: Modification of object in method

# 4.6 this Reference

## ? Question

Look at the following code. What is unsightly?

Java

```java
public class Student {
        String name;

        void setName(String newName) {
            name = newName;
        }
    }
```

# 4.6 this Reference

- We would like to name the parameter of the setter `name` as well!

**? Question**

Does the following call work?

```java
void setName(String name) {
    name = name;
}
```

# 4.6 this Reference

- We would like to name the parameter of the setter `name` as well!

**? Question**

Does the following call work?

```java
void setName(String name) {
    name = name;
}
```

- No! The compiler would use the local variable in each case.
- How can we access the instance variable?

# 4.6 this Reference

## ! Memorize

- Using `this` we can access the current instance we are currently in.

Java

```java
public class Student {
        String name;

        void setName(String name) {
            this.name = name;
        }
    }
```

# 4.6 this Reference

- So
  - If you want to access a local variable, simply use the identifier of the variable
  - If you want to access an instance variable, use the `this` reference with the member operator with `.`.

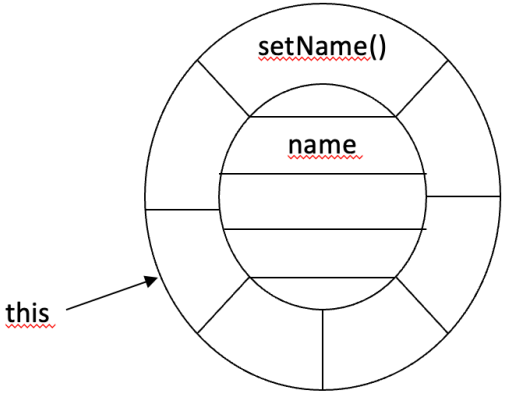


Figure 5: `this` reference

# 4.7 Overloading

## Task 7

Write a method that calculates the maximum of two integers.

Java

```java
int max(int a, int b) {
        if (a > b) {
            return a;
        } else {
            return b;
        }
    }
```

# 4.7 Overloading

## Task 8

Write another method in the same class that calculates the maximum of two `double` numbers.

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

# 4.7 Overloading

## Task 9

Write yet another method in the same class that returns the maximum of three integers.

```java
int max(int a, int b, int c) {
    return max(max(a, b), c);
}
```

# 4.7 Overloading

- Overloaded methods (overloading):
  - Multiple methods with the same name exist in a class.
  - Only possible if parameter types are different.
  - Compiler selects the correct method based on the parameters.

Java

```java
int max(int, int)
double max(double, double)
int max(int, int, int)
```

# 4.7 Overloading

- Note:
  - Signature: Method name and parameter types
  - Only data types of parameters relevant (distinction by names not possible)
  - Distinction by return type is not sufficient (Why?)

# 4.7 Overloading

## ! Memorize

- These are not the same methods!

```java
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

```java
short max(short a, short b) {
    System.out.println("Aaaarrrghhh!");
    return 7;
}
```

# 4.8 Methods: Coding Style

- Blank lines should increase code readability.
- A blank line is mandatory between methods or classes (1).
- A blank line is usually placed after declarations (2).
- A blank line is usually placed between logical sections.

# 4.8 Methods: Coding Style

## Example

Java

```java
public class MyClass {

    public static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    public static void swap(Object a, Object b) {
        Object temp;

        temp = a;
        a = b;
        b = temp;
    }
}
```

# 4.8 Methods: Coding Style

- Spaces should increase code readability.
- A space is mandatory between expressions.
- A space is usually placed between operands.

Java

```java
public static void
main(String[] args) {
    int a = 5;

    for (int i = 1; i < 10; i++) {
        a *= i;
    }
}
```

Java

```java
public static void
main(String[] args) {
    int a=5;

    for(int i=1;i<10;i++){
        a*=i;
    }
}
```

# 4.9 What a chapter!

- We have accomplished a lot in this section!
  - Syntax of methods
  - Parameters (call by value)
  - this reference
  - Overloading methods
  - Coding Style

Let us now look at how to specifically create and initialize objects.

# 5. Constructors

# 5.1 Constructors

## ! Memorize

- Special methods for initializing an object
  - Are executed when an object is created
  - Note: Constructors have no return type

Java

```java
ClassName(Parameters){
Statements;
}
```

# 5.1 Constructors

- Using the following code, one could create a constructor for the class `Student`:

```java
public class Student {
        String name;

        Student(String name) {  // Attention: No return type
            this.name = name;
        }
    }
```

# 5.1 Constructors

- Default constructor: Constructor with empty parameter list
- Initializes instance variables depending on type with 0, 0.0, `false` or `null`

# 5.1 Constructors

## Example

Java

```java
public class Student {
        String name;

        Student() {       // Default constructor
        }
    }
```

# 5.1 Constructors

- The previous example could also be represented like this:

Java

```java
public class Student {
        String name;

        Student() {       // Default constructor
            name = null;
        }
    }
```

# 5.1 Constructors

- Compiler automatically generates the default constructor under certain conditions
- Simple rules:

1. If you don’t write a constructor for a class:
   - The compiler automatically generates the default constructor
2. If you write at least one constructor for a class:
   - The compiler does not generate a constructor
   - Only the constructors you implemented exist

# 5.2 `this`: Reference vs. Method

- `this` reference (as a reminder):
  - Use within any (instance) method
  - Used like a variable
  - Contains reference to object for which the method was called
- `this()` method
  - Use only within a constructor.
  - Is a method call.
  - this(parameter list) calls constructor with matching parameter list.
  - May only stand as the first statement in the constructor.

# 5.2 `this`: Reference vs. Method



Java

```java
public class Aircraft {
    String model, airline;
    int numberEngines;

    Aircraft() {
        numberEngines = 1;
    }

    Aircraft(String model) {
        this();
        this.model = model;
    }

    Aircraft(String model, String airline) {
        this(model);
        this.airline = airline;
    }
}
```

# 5.3 Constructors: Examples

## ? Question

Will the following code compile? What do you think?

Java

```java
public class Aircraft {
        String model;

        public static void main(String[] args) {
            Aircraft aircraft = new Aircraft();
        }
    }
```

# 5.3 Constructors: Examples



**Success**

- Yes, it will!
  - The compiler generates a default constructor because the class does not contain one.
  - This is then called in the `main()` method!

# 5.3 Constructors: Examples

## ? Question

Will the following code compile? What do you think?

Java

```java
public class Aircraft {
        String model;

        Aircraft(String model) {
            this.model = model;
        }

        public static void main(String[] args) {
            Aircraft aircraft = new Aircraft();
        }
    }
```

# 5.3 Constructors: Examples



## ✗ Error

- No, it will not!
  - Since the class contains a constructor, the compiler does not generate a default constructor.
  - The constructor used in the `main()` method therefore does not exist.

# 5.4 Task: Class Circle



## Task 10

- Let us write a class Circle:
  - Represents a geometric circle
    - Represented by
    - x and y coordinates of the center point
    - Radius r
  - Constructors:
    - Default constructor creates unit circle around coordinate origin (0 ; 0)
    - Constructor with x, y and radius as parameters
    - Constructor with radius as parameter
    - Constructor with object of class Circle as parameter (creates copy)

# 5.4 Task: Class Circle



Java

```java
public class Circle {
  double x, y, radius;

  public Circle(double x, double y, double radius) {
      this.x = x;
      this.y = y;
      this.radius = radius;
  }

  Circle() {
      this(0.0, 0.0, 1.0);
  }

  Circle(double radius) {
      this(0.0, 0.0, radius);
  }

```

# 5.4 Task: Class Circle



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

# 5.4 Task: Class Circle

- The class we just wrote can be created as follows:

```java
public class CircleDemo {
    public static void main(String[] args) {
        Circle circle1 = new Circle();
        Circle circle2 = new Circle(2.5);
        Circle circle3 = new Circle(circle2);
        Circle circle4 = new Circle(-1.2, 7.1, 3.0);
    }
}
```

**Tip**

The following menu item in IntelliJ IDEA can save you a lot of work: Code / Generate / Constructor

# 5.4 Task: Class Circle

## Task 11

Extend the class with getter and setter methods.

## Tip

Via the menu items Code / Generate / Getter and Setter you can save even more time!

# 5.4 Task: Class Circle



Java

```java
double getX() {
    return x;
}

// getY() and getRadius() accordingly

void setX(double x) {
    this.x = x;
}

void setY(double y) {
    this.y = y;
}

```

# 5.4 Task: Class Circle



```
15  void setRadius(double radius) {
16      if (radius >= 0.0) {        // Prevent disallowed data
17          this.radius = radius;
18      }
19  }
```

# 6. Class Variables & Class Methods

# 6.1 Class Variables

## ? Question

Do you think you reserve the variable `PI` only once in memory?

Java

```java
public class Circle {
        double x, y, radius;
        final double PI = 3.141592653589793;

        double getArea() {
            return PI * radius * radius;
        }
    }
```

# 6.1 Class Variables

## ✗ Error

- Answer: No, again for each object!

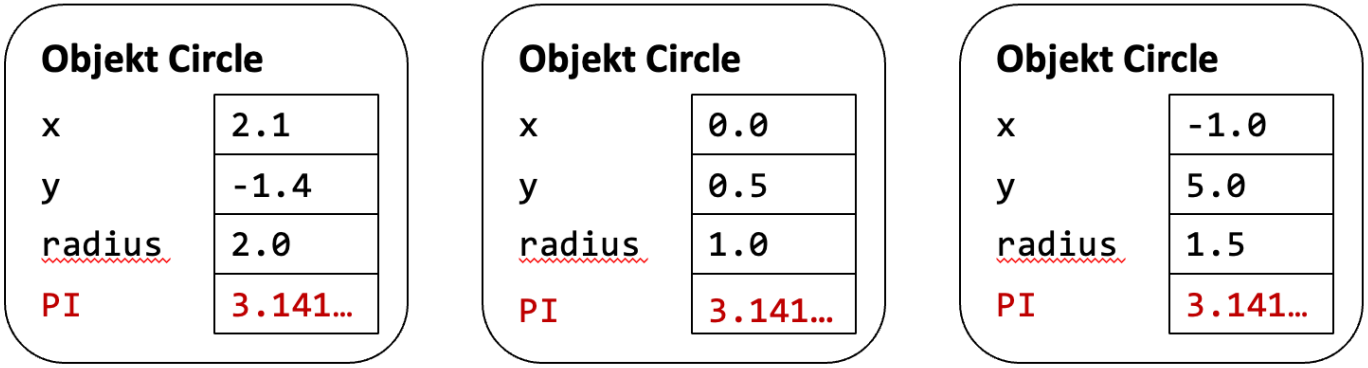


Figure 6: Memory is reserved again for each object.

# 6.1 Class Variables

## ? Question

What do you think, does the variable `count` count the number of objects?

```java
public class Circle {
        double x, y, radius;
        int count;    // Count number of objects created

        Circle() {
            radius = 1.0;
            count++;
        }
    }
```

# 6.1 Class Variables

## ✗ Error

- Answer: No, each object gets a new variable `count`!

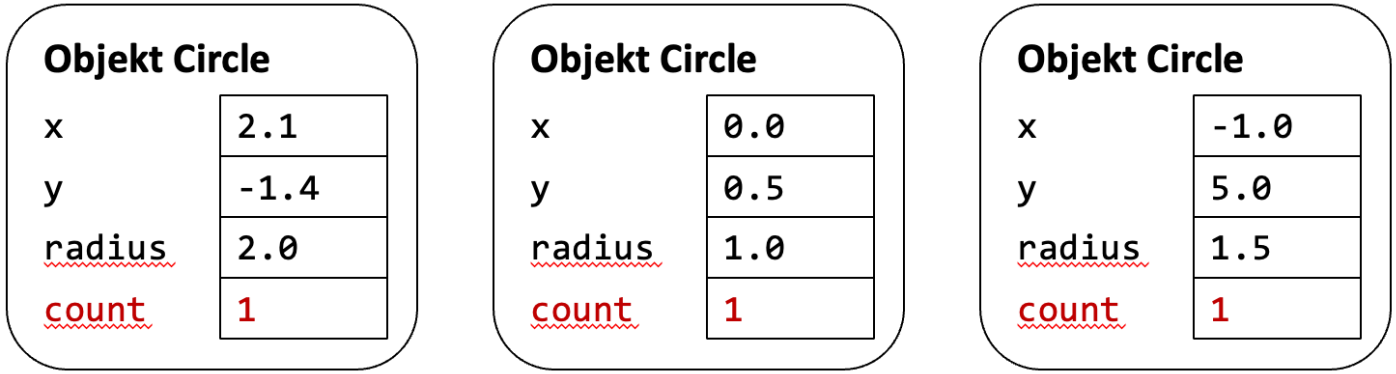


Figure 7: The variable `count` is created again and again.

# 6.1 Class Variables

- One solution is **class variables**!
- Class variables are created only once for the entire class
- Are not created for a specific (not for each object) object
- Are already created at program start (loading the class)
  - They exist even when there is (still) no object of the class.
- Syntax: Variable with keyword static

# 6.1 Class Variables

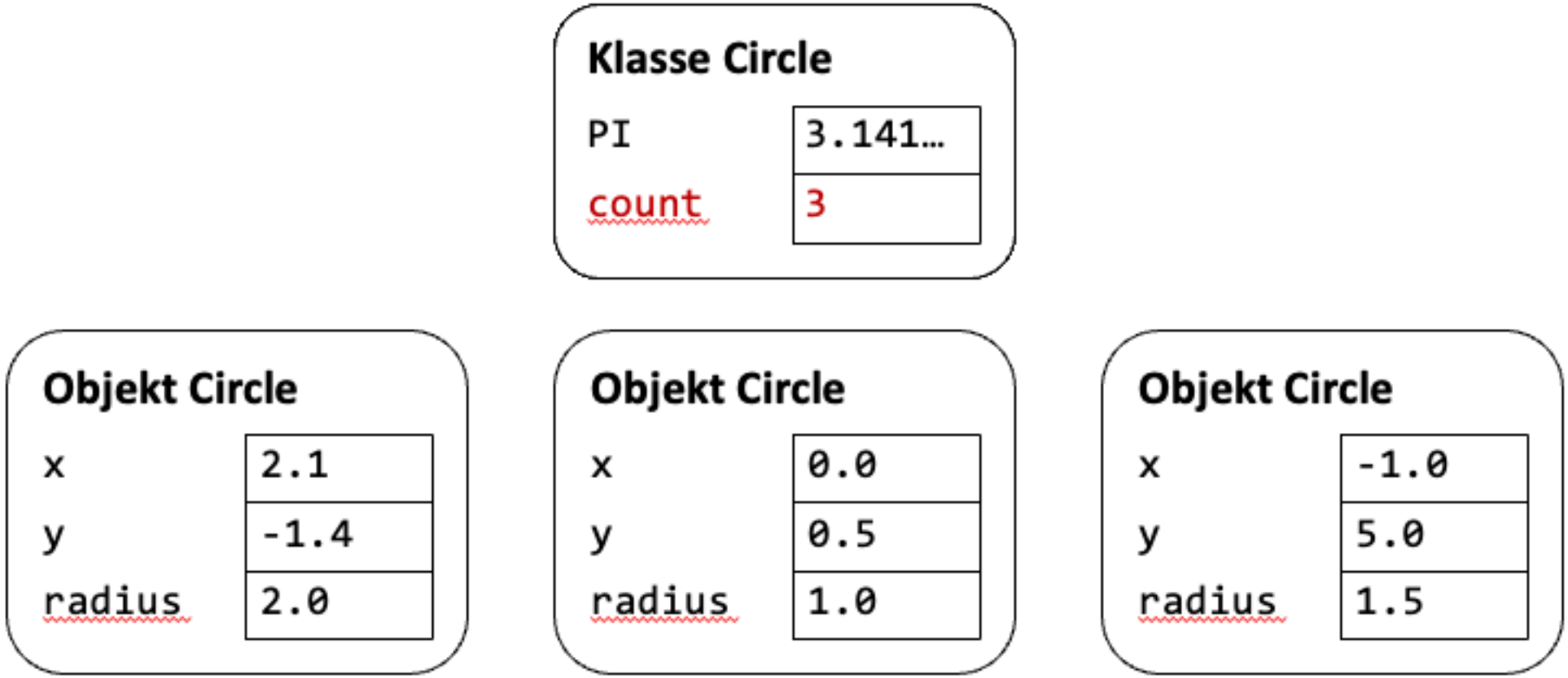


Figure 8: Class attributes in the different objects

# 6.1 Class Variables

- Class `Circle` with class variables:

Java

```java
public class Circle {
        double x, y, radius;
        static final double PI = 3.141592653589793;
        static int count;

        double getArea() {
            return PI * radius * radius;
        }

        Circle() {
            radius = 1.0;
            count++;
        }
    }
```

# 6.1 Class Variables

- Access to class variables
  - Within the method of the class it corresponds to access to instance variables
  - Outside the class `ClassName.VariableName`

```java
public class CircleDemo {
        public static void main(String[] args) {
            Circle circle1 = new Circle();
            Circle circle2 = new Circle();
            Circle circle3 = new Circle();

            System.out.println("Number of objects: " + Circle.count);
        }
    }
```

# 6.2 Class Methods

- Completely analogous to class variables:
  - Class methods are called for a class
  - Are not called for a specific object
  - Method becomes class method through keyword static
  - Can be called without object of the class being created
- Call outside the class:

```
ClassName.MethodName
```

Java

# 6.2 Class Methods

## ! Memorize

- Important consequences:
  - `this` reference does not exist in class methods
  - Instance variables do not exist in class methods

## Example

- In this example there are two class methods!

```java
public class Circle {
        double x, y, radius = 1.0;

        static double getPi() {
            return 3.141592653589793;
        }
    }

    public class CircleDemo {
        public static void main(String[] args) {
            System.out.println("Pi: " + Circle.getPi());
        }
    }
```

# 7. License Notice

# 7.1 Attribution


- 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.