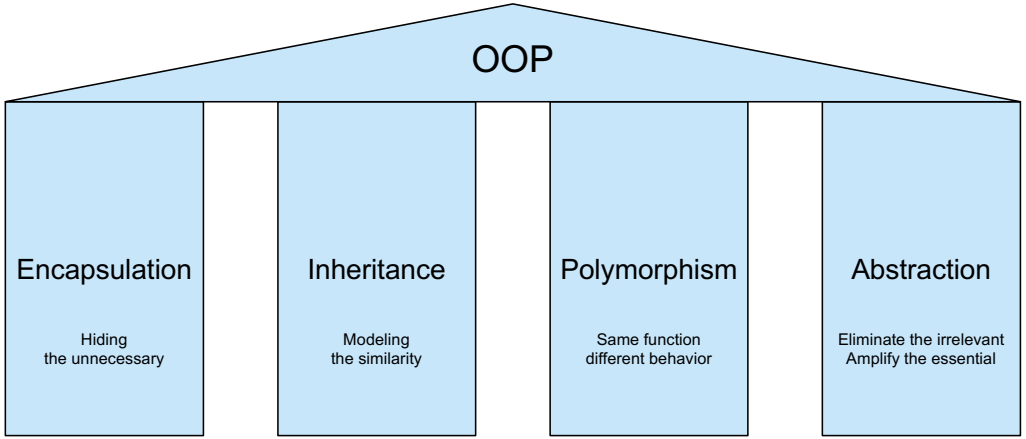


RUHR-UNIVERSITÄT BOCHUM

Vorkurs Informatik

Principles of Object-Oriented Programming

Prof. Dr. Yannic Noller
Software Quality group

Agenda

- Software Quality research @ RUB
- Recap: Classes, Objects, Attributes, Methods
- UML Modelling
- Encapsulation and Composition
- Four Principles of OOP
  - #1 Encapsulation
  - #2 Inheritance
  - #3 Polymorphism
  - (#4 Abstraction)

Software Quality research at RUB

About Me

- since **July 2024**: Professor for Computer Science, Ruhr University Bochum
- **Before**:
  - 2023 – 2024: **Singapore** University of Technology and Design (Assistant Professor)
  - 2020 – 2023: National University of **Singapore** (PostDoc, Research Assist. Prof.)
  - 2016 – 2020: PhD student at HU **Berlin**
  - 2010 – 2016: Bachelor and Master in Software Engineering at University of **Stuttgart**
- **Research Interests:**
  - automated software engineering
  - software testing & verification (e.g., symbolic execution and fuzzing)
  - software repair (e.g., semantic-based)

Software Quality @ RUB (Team)

Prof. Dr. Yannic Noller

Samra Mehboob, M.Phil.
(PhD Student)

Thiago Santos de Moura, M.Sc.
(PhD Student)

* Sophie Poschmann (Team Assistant)
* many student research/teaching assistants

https://informatik.rub.de/en/sq/

Software Quality Research @ RUB

developing new repair techniques to aid developers in fixing program bugs

Automated Program Repair

Software Testing

exploring and designing (hybrid) testing techniques to systematically generate test inputs that expose incorrect program behavior

how to help CS students learn programming by applying concepts from automated testing and repair to guide the students toward the right solution

Intelligent Tutoring Systems

Machine Learning Analysis

automated analysis, testing, and repairing of machine learning models

Human Factors in SE

studying developer needs and requirements for successful deployment of testing and repair techniques in development practice

https://informatik.rub.de/en/sq/

Software Quality @ RUB (Teaching)

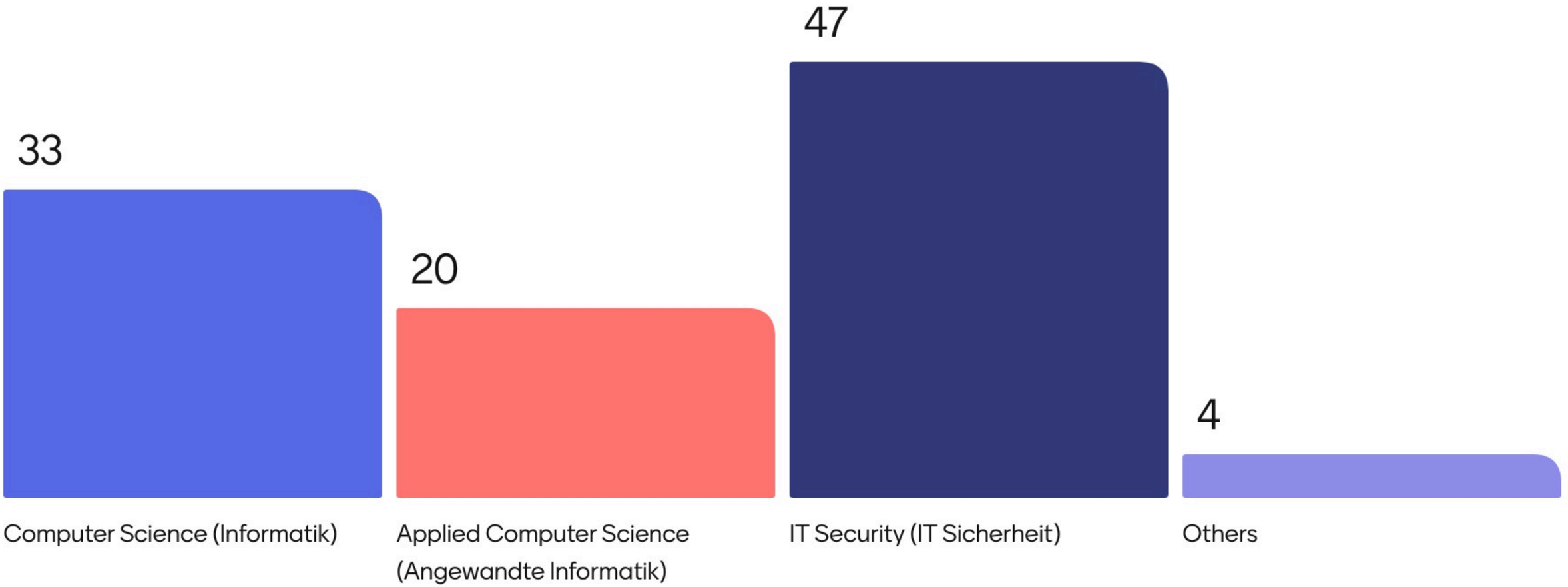
Winter Semester
- Lecture: Software Engineering

Summer Semester
- (new!) Lecture: Automated Debugging and Repair
- Lecture: Advanced Automatic Testing (with Prof. Toffalini)

Every Semester
- Seminar: Automated Software Engineering
- various study projects

https://informatik.rub.de/en/sq/

What's your **study program**?



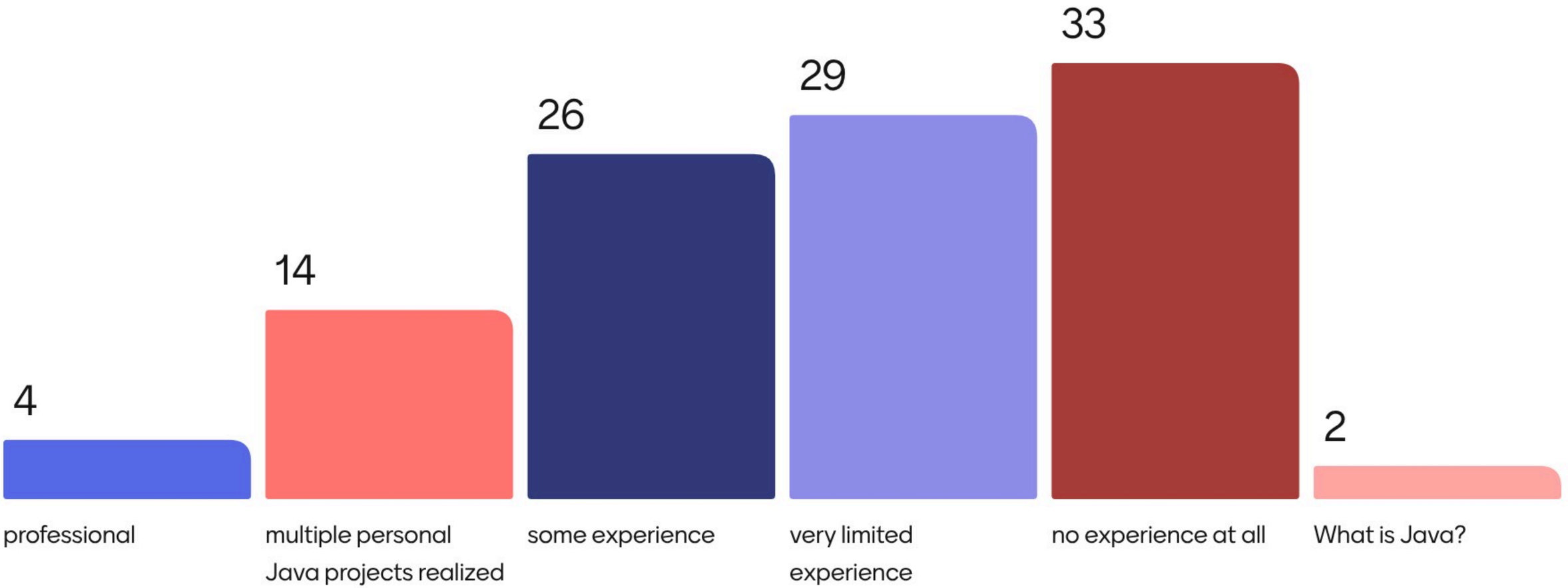
Why do you study?

15 95

Which **programming languages** are you proficient with? (i.e., >= 1 year of experience)

20 99

How are your **Java** skills? (pre prep course)



Recap
Classes, Objects, Attributes, Methods

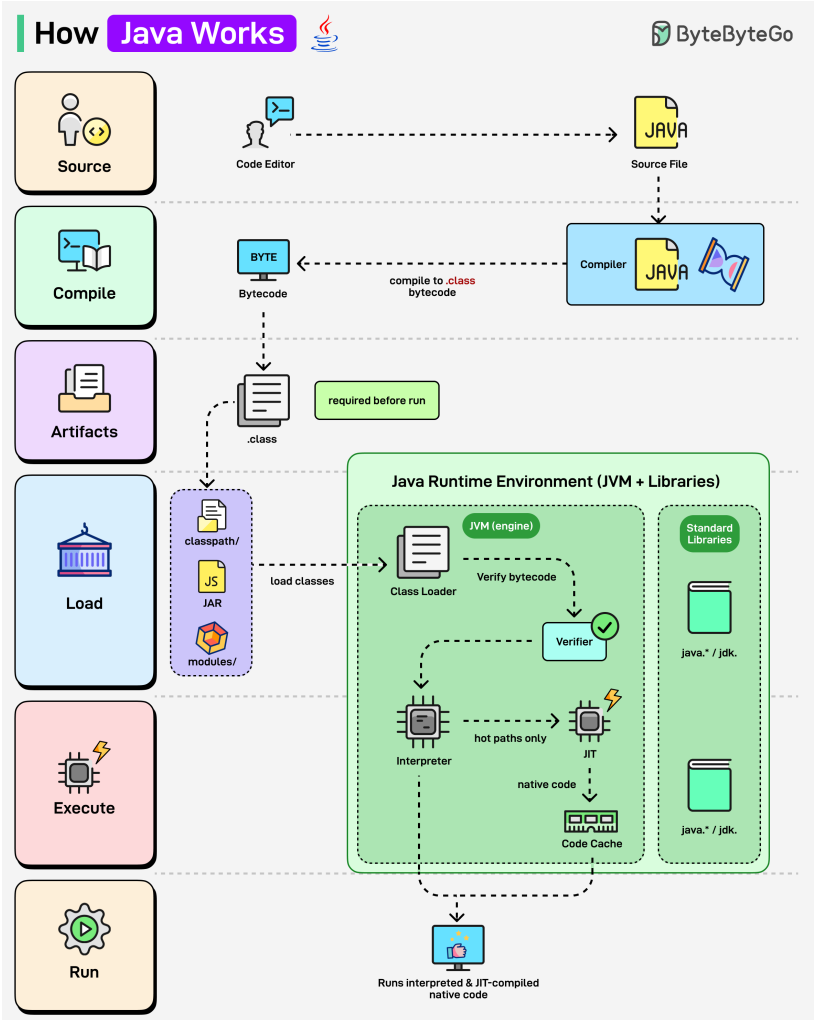
Role of Programming

- **Programming**: Essential form of expression for a computer scientist
- **Programming Languages** (PL) determine what algorithms and ideas you can express
- **Learning** about Programming Languages
  - **choose right PL** for a specific purpose
  - make best use of tools (e.g., debuggers, IDEs, analysis tools)

→ **Learning the concepts**

Programming Languages

- **Broad Classification**
  - **Declarative** (“what to compute“): e.g., Haskell, SQL, spreadsheets
  - **Imperative** (”how to compute it”): e.g., C, Java, Perl
- **Various PL paradigms:**
  - Functional, Logic
  - Sequential, Shared-memory parallel, Distributed-memory parallel
  - Statically typed, Dynamically typed
- Most languages combine **multiple** paradigms
- **Types!**



https://blog.bytebytego.com/p/ep181-how-java-works

Types in Java

What is the other category of types?

two categories of types

- **primitive data types:** 8 essential built-in data types

Object types
(also known as class data types)

Typ	Size (Byte)	Values
`boolean`	1	`true` or `false`
`char`	2	16-Bit Unicode Character
`byte`	1	$-2^7 \ldots 2^7-1$ (i.e., $-128 \ldots 127$)
`short`	2	$-2^{15} \ldots 2^{15}-1$ (i.e., $-32768 \ldots 32767$)
`int`	4	$-2^{31} \ldots 2^{31}-1$
`long`	8	$-2^{63} \ldots 2^{63}-1$
`float`	4	$+/-3.40282347 * 10^{38}$
`double`	8	$+/-1.79769313486231570 * 10^{308}$

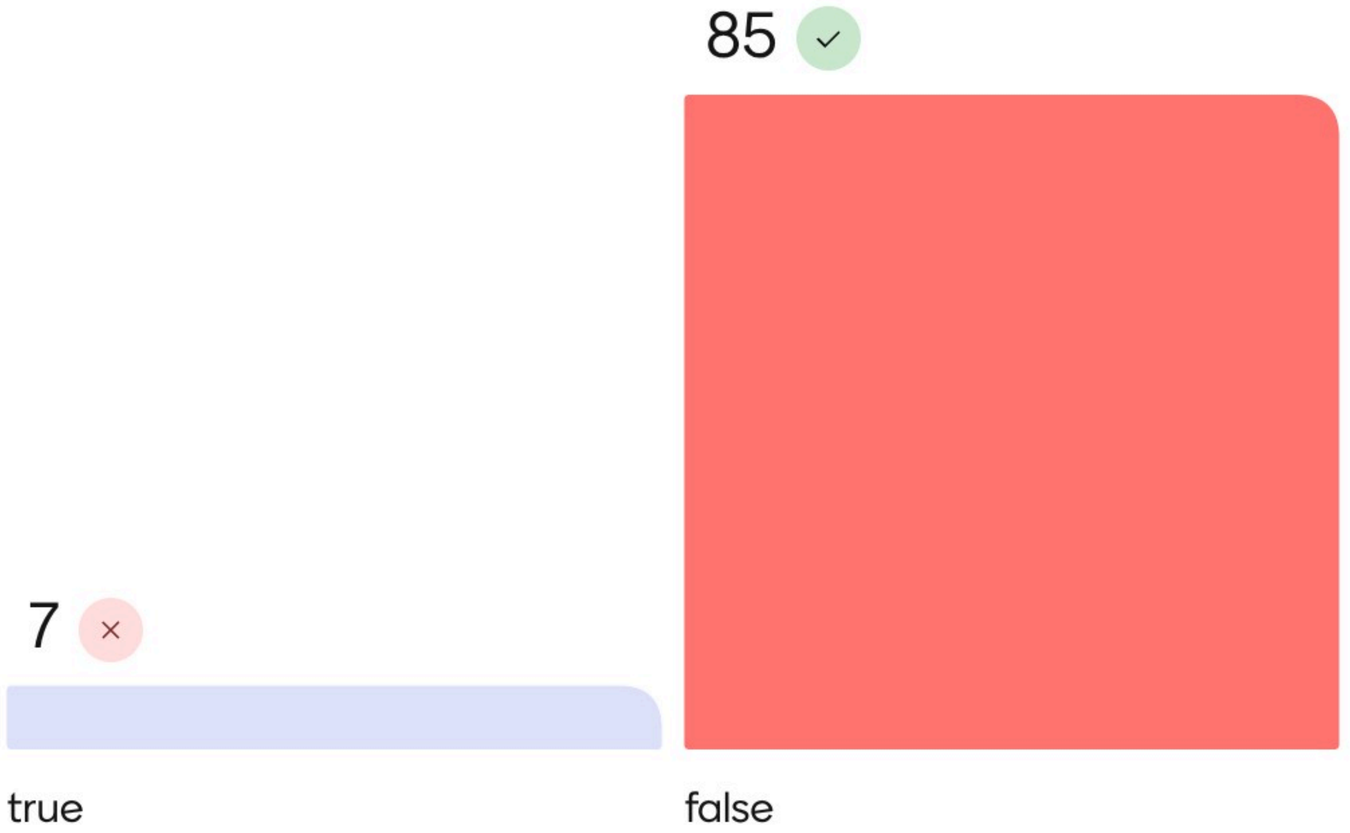
→ Mentimeter

Quiz

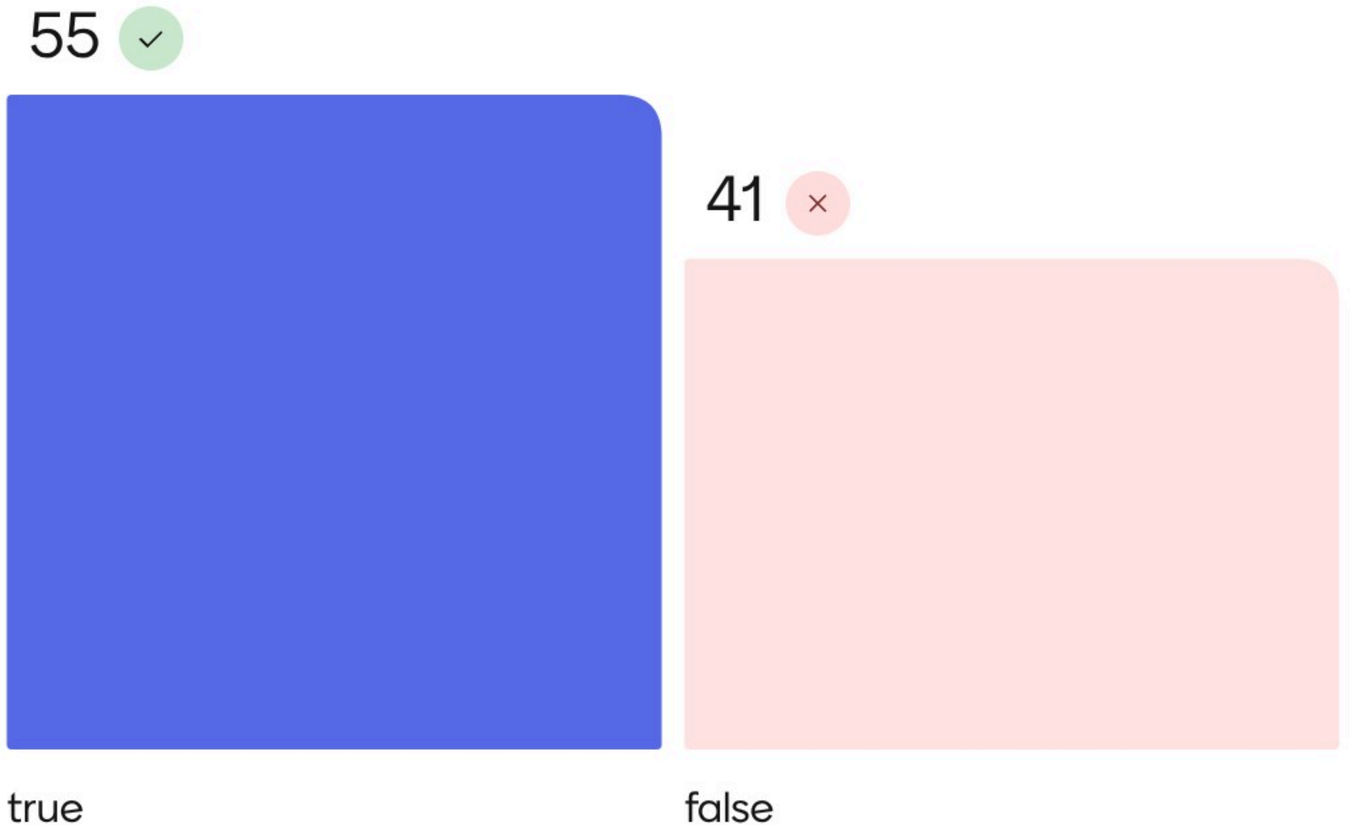
What values do these JavaScript expressions evaluate to?

```
''      == 'zero'

''      == 0

'0'     == 0

false   == 'false'
```

'' == 'zero'

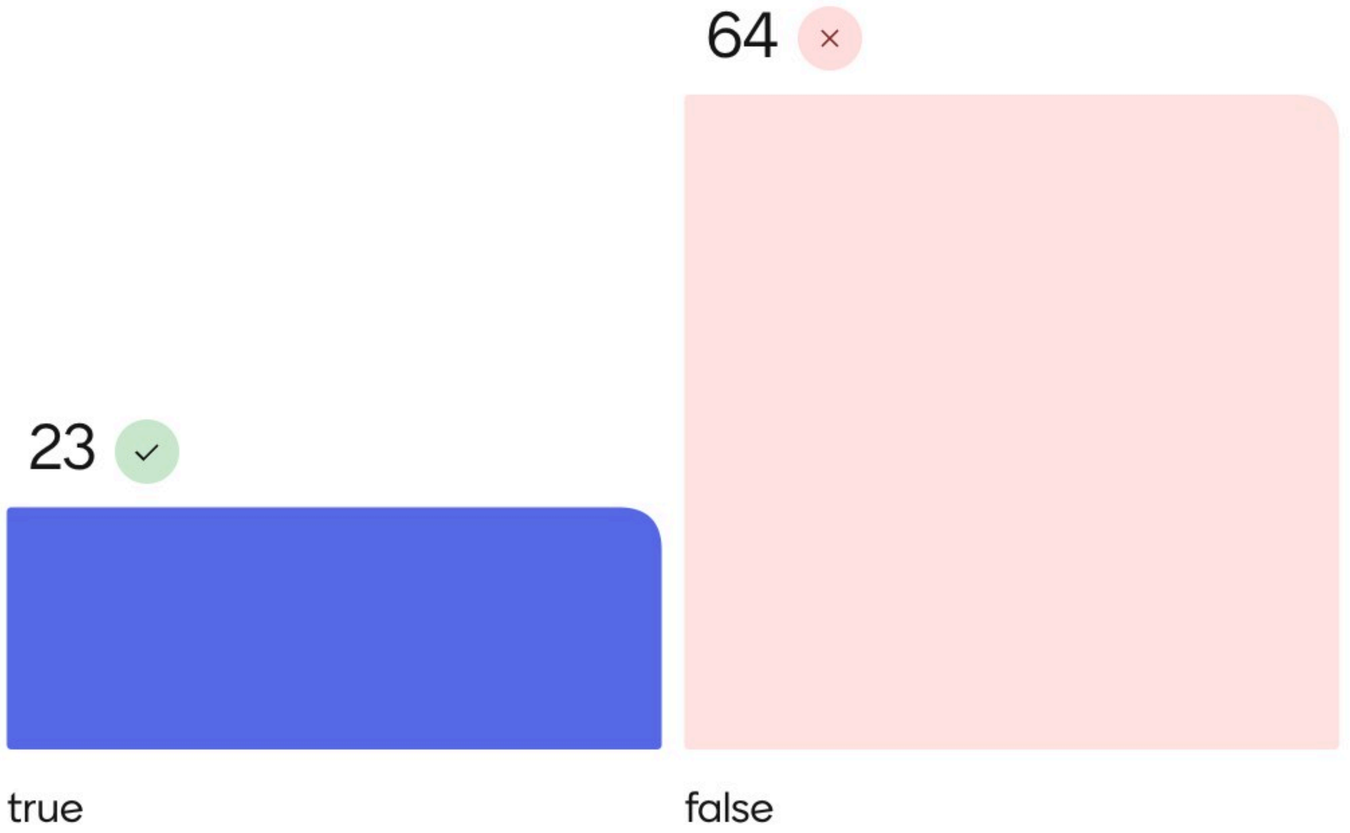


" == 0



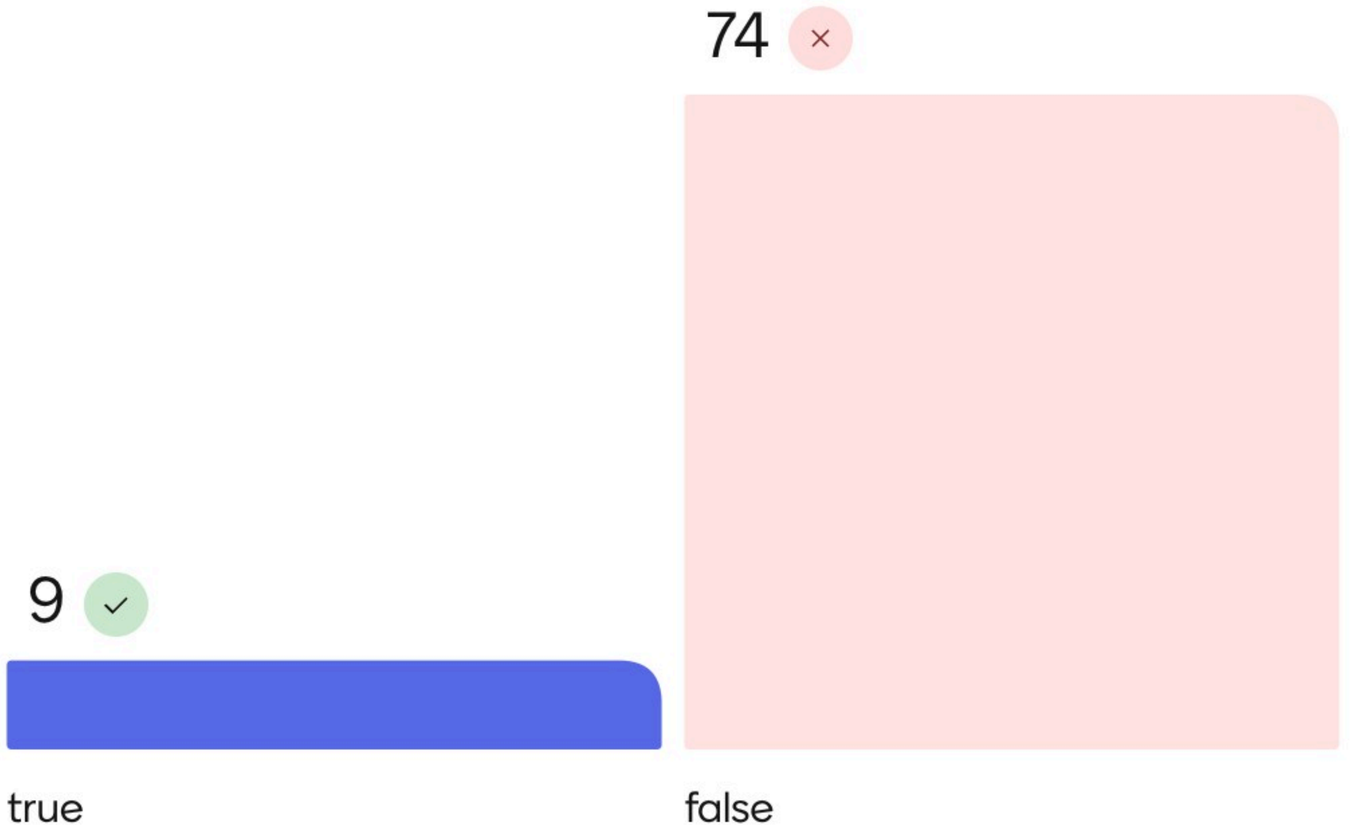
Mentimeter

'0' == 0



15 87

false == 'false'



Quiz

What values do these JavaScript expressions evaluate to?

```
''      == 'zero'   // false
''      == 0        // true
'0'     == 0        // true
false   == 'false'  // false
```

Two strings that are not the same

String and number: String is coerced into a number (here: 0)

Boolean and another type:

- Boolean gets coerced to a number (here: 0)
- String also get coerced to a number (here: NaN)
- The two numbers differ

Why do we need types?

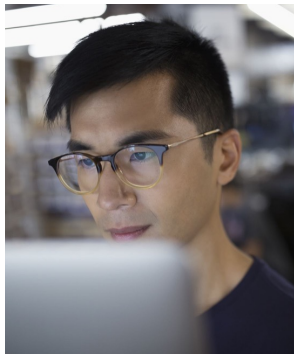
- **Reason 1:** Provide **context** for operations
- **Reason 2: Limit valid operations**
- **Reason 3: Code readability** and **understandability**
- **Reason 4:** Compile-time **optimizations**

*» **Object-oriented programming** is a method of implementation in which programs are organized as **cooperative collections** of **objects**, each of which represents an **instance** of some **class**, and whose classes are all members of a hierarchy of classes united via **inheritance** relationships. «*

Grady Booch, “Object-oriented Analysis and Design with Applications”, 1991.

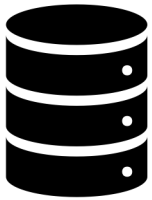
Examples for Objects

Manager



Programmer

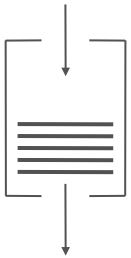
Contract



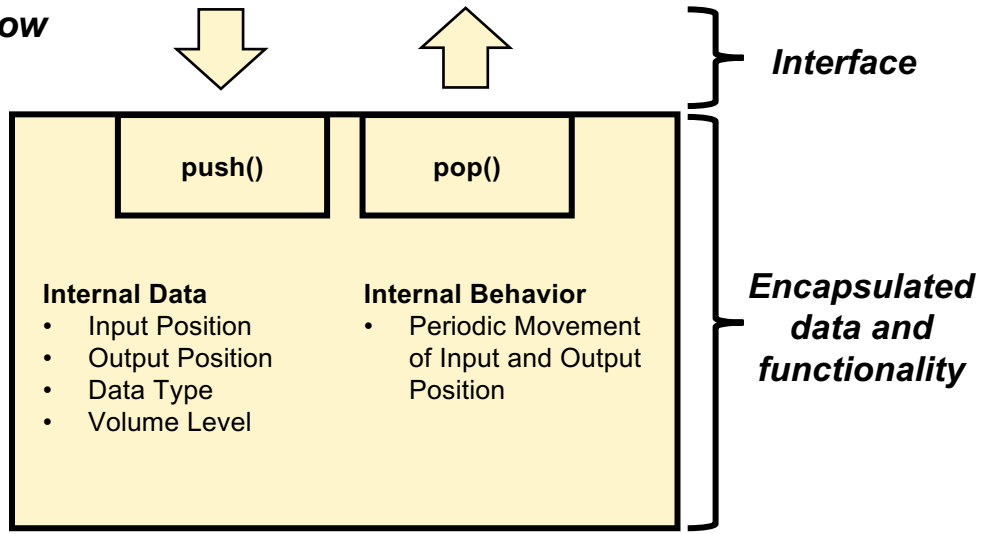
Database

Window

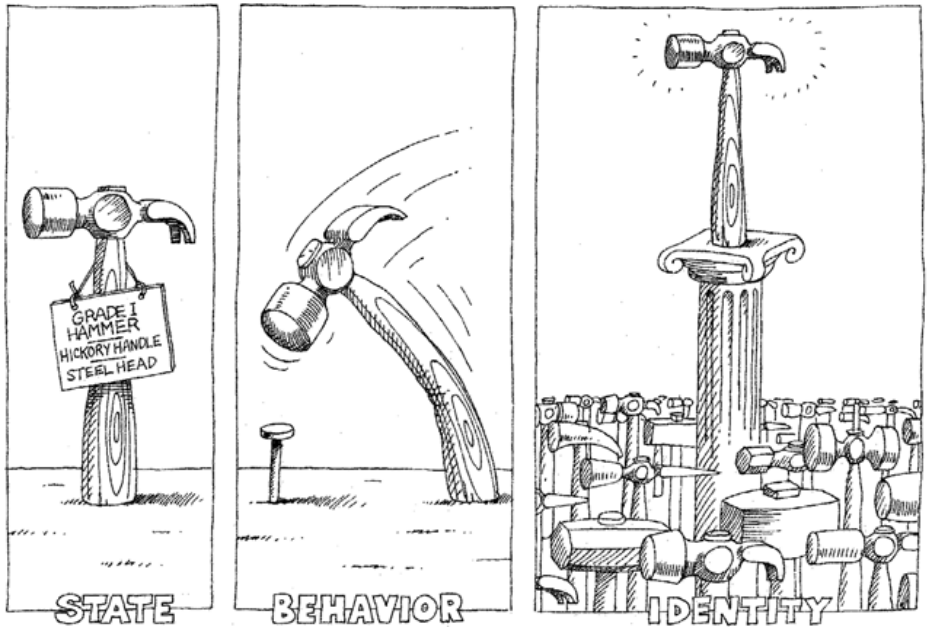
“Coffee”
String



Queue



Objects



*» An **object** has **state**, exhibits some well-defined **behavior**, and has a unique **identity**.«*

- are **self-contained** units
- provide specific **service** or solve a specific task
- have a **state** and manage their **data**
- can be **instantiated** and **destroyed**

Grady Booch, “Object-oriented Analysis and Design with Applications”, 1991.

Classes

- A **class** is the **abstraction** (i.e., the data type) of an object and defines its **structure** (the attributes and methods).
- We **implement** the **class**, not the object.
- Objects are instances of **exactly** one class.
- Classes can have their own (class) methods and (class) attributes. A class attribute only exsists once per class (for all its objects).

- Examples:
  - Professor → Nils Jansen, Yannic Noller, …
  - University → RUB, TU Dortmund, HU Berlin, …
  - Airline → Lufthansa, Singapore Airline, …

Advantages

- Any relevant information (attributes, methods) linked to a concept is contained within a class.
- The object-oriented philosophy is that every object is a **blackbox** with a defined set of methods
  - We do not know or need to know how the data is stored inside the class
  - It is fine to use attributes within the class, but avoid doing so outside

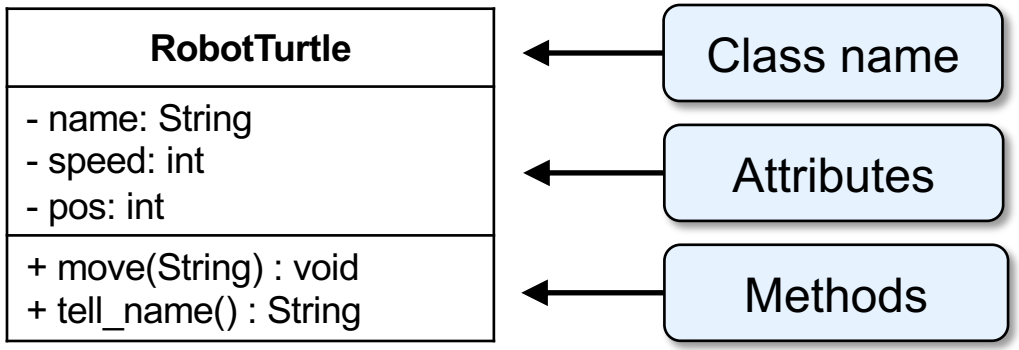
Example

- `RobotTurtle` as a game character
  - name
  - speed
  - position

  - tell name
  - move

UML Class Diagram

- Unified Modeling Language (UML) gives some specifications how to represent the classes visually



Attributes and Methods

- **Attributes**: define the characteristic of the object
  - usually a *noun*
  - variables defined within the object
- **Methods**: define what the object can do
  - usually a *verb*
  - functions that apply to our user-defined data type
- Attributes and methods **define** your object

```
public class RobotTurtle {

    private String name;
    private int speed;
    private int[] pos;

    public RobotTurtle(String name, int speed) {
        this.name = name;
        this.speed = speed;
        this.pos = new int[2];
    }

    public void move(String direction) {
        if (direction.equals("up")) {
            this.pos[1] = this.pos[1] + this.speed;
        …
    }

    public void tell_name() {
        System.out.println("My name is " + this.name);
    }
}
```

Class Definition

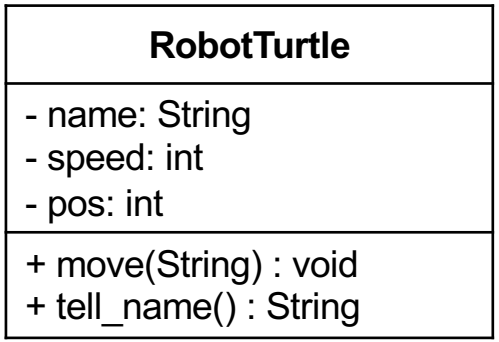
Attribute(s)

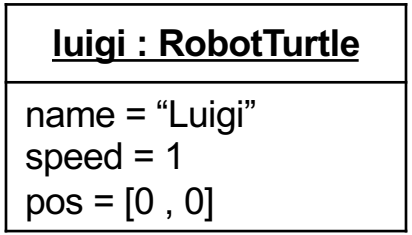
Constructor(s)

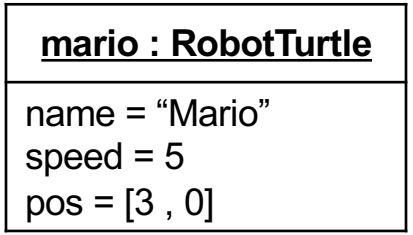
Method(s)

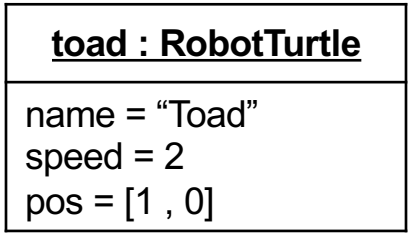
Object Instantiation

- Object instantiation refers to the **creation** of an object
  - an instantiated object is named and created in **memory**









The Constructor

- A **constructor** is a special (optional) **method**, which is called for the **initialisation** of an object.
- Constructors
  - have the same **name** as the class
  - do **not** have any **return value** (also not void!)
  - can have an arbitrary number of **parameters**, and can be **overloaded**.

```
public class Person {
   private int age;

   Person() {...}

   Person(int age) { this.age = age; }
}
```

The Constructor

- A **constructor** is a special (optional) **method**, which is called for the **initialisation** of an object.
- Constructors
  - have the same **name** as the class
  - do **not** have any **return value** (also not void!)
  - can have an arbitrary number of **parameters**, and can be **overloaded**.

```
public class Person {
   private int age;

   Person() {...}

   Person(int age) { this.age = age; }
}
```

```
Person donald = new Person();

Person dagobert = new Person(60);
```

```
Person() {
   this(1);
} ...
```

Constructor Chaining

The Constructor

- A **constructor** is a special (optional) **method**, which is called for the **initialisation** of an object.
- Constructors
  - have the same **name** as the class
  - do **not** have any **return value** (also not void!)
  - can have an arbitrary number of **parameters**, and can be **overloaded**.

Will be generated by **default** if no other constructor is defined.

```
public class Person {
   private int age;

   Person() {...}

   Person(int age) { this.age = age; }
}
```

```
Person donald = new Person();

Person dagobert = new Person(60);
```

```
   Person() {
      this(1);
   } ...
```

Constructor Chaining

```
RobotTurtle luigi = new RobotTurtle("Luigi", 1);

luigi.tell_name();

System.out.println(Arrays.toString(luigi.getPos()));

luigi.move("up");
System.out.println(Arrays.toString(luigi.getPos()));

luigi.move("right");
System.out.println(Arrays.toString(luigi.getPos()));

luigi.move("down");
System.out.println(Arrays.toString(luigi.getPos()));

luigi.move("left");
System.out.println(Arrays.toString(luigi.getPos()));
```

Instantiate Object

↳ Coding Example

Usage of object's methods

Output:

```
My name is Luigi
[0, 0]
[0, 1]
[1, 1]
[1, 0]
[0, 0]
```

Quiz

How do we destroy an object?

- **Java Virtual Machine** and its **Garbage Collector** will handle it.

```
protected void finalize() {
    ... // clean up
}
```

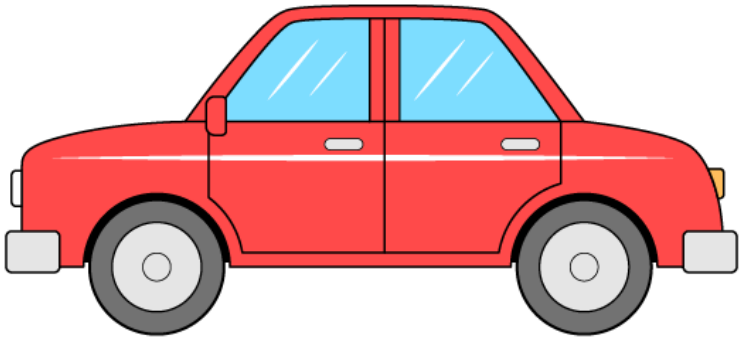
deprecated since Java 9

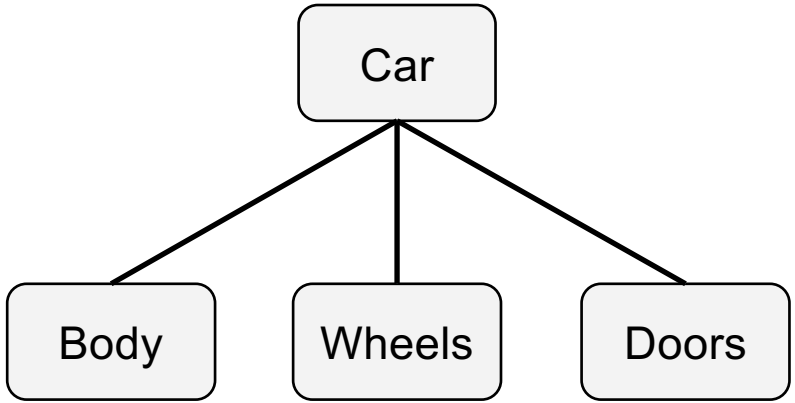
- housekeeping tasks, releasing of locks, closing of database/network connections, …

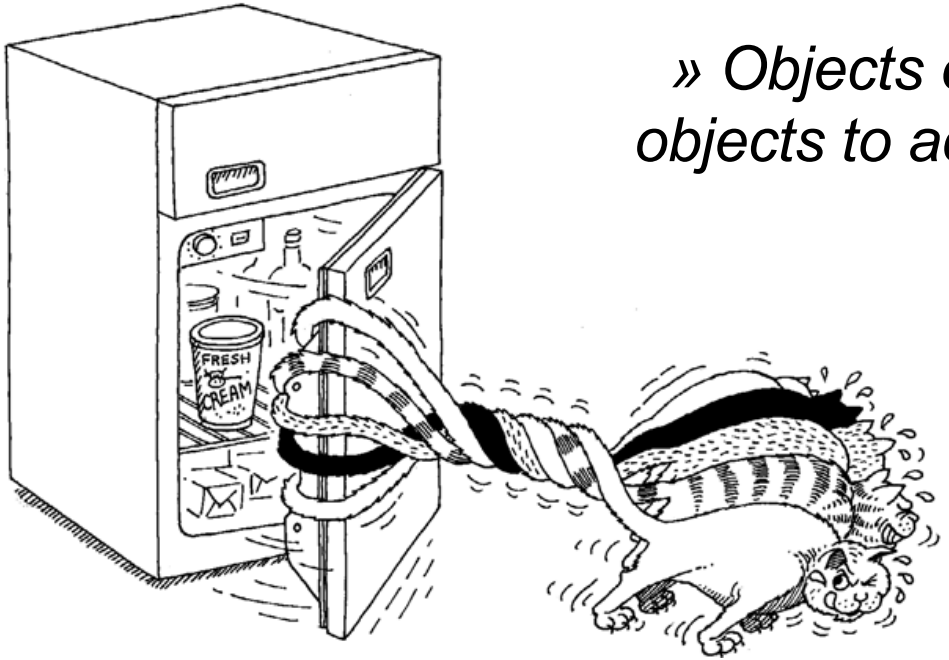
Relationship between classes: **Composition**

Composition

- Composition: an object can be **composed** of other objects.







*» Objects **collaborate** with other objects to achieve some behavior.«*

Grady Booch, “Object-oriented Analysis and Design with Applications”, 1991.

Composition

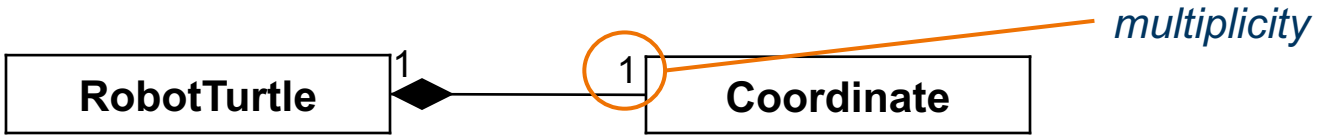
- Composition: an object can be **composed** of other objects.

```
public class RobotTurtle {

    private String name;
    private int speed;
    private int[] pos;

    public RobotTurtle(String name, int speed) {
        this.name = name;
        this.speed = speed;
        this.pos = new int[2];
        this.pos = new Coordinate(0, 0);
    }
…
}
```

↳ Coding Example

UML Class Diagram

- UML allows to specify the **relationship** between different classes

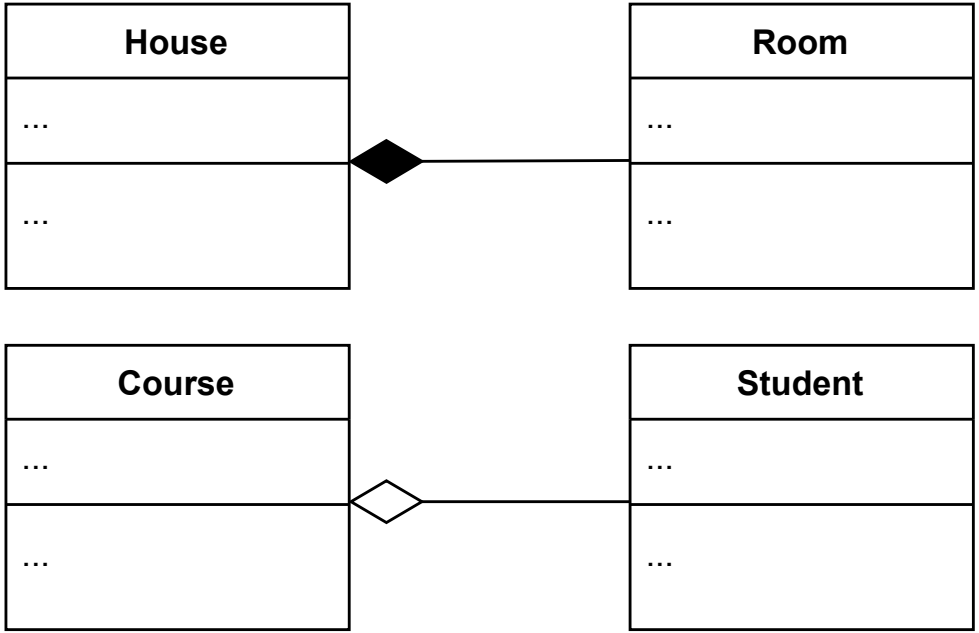


- The **black diamond** shows a **composition** relationship, i.e., an instance of RobotTurtle contains an instance of Coordinate
- **Multiplicity**: the “1” on both sides show that 1 instance of RobotTurtle is associated with exactly 1 instance of Coordinate

UML: Relationships between Classes

Composition
implies a relationship where the child cannot exist independent of the parent

Aggregation
implies a relationship where the child can exist independently of the parent



Quiz

In the following class names, which two would most likely be in a has-a relationship?

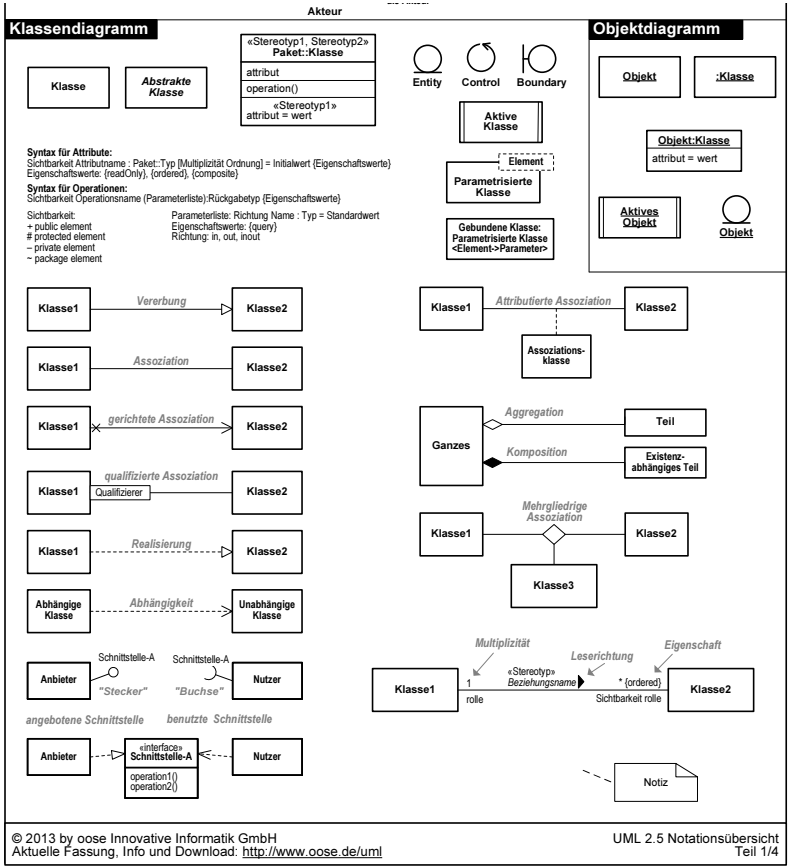
(a) Vehicle and Car

(b) Aircraft and Drone

(c) TennisPlayer and Racket

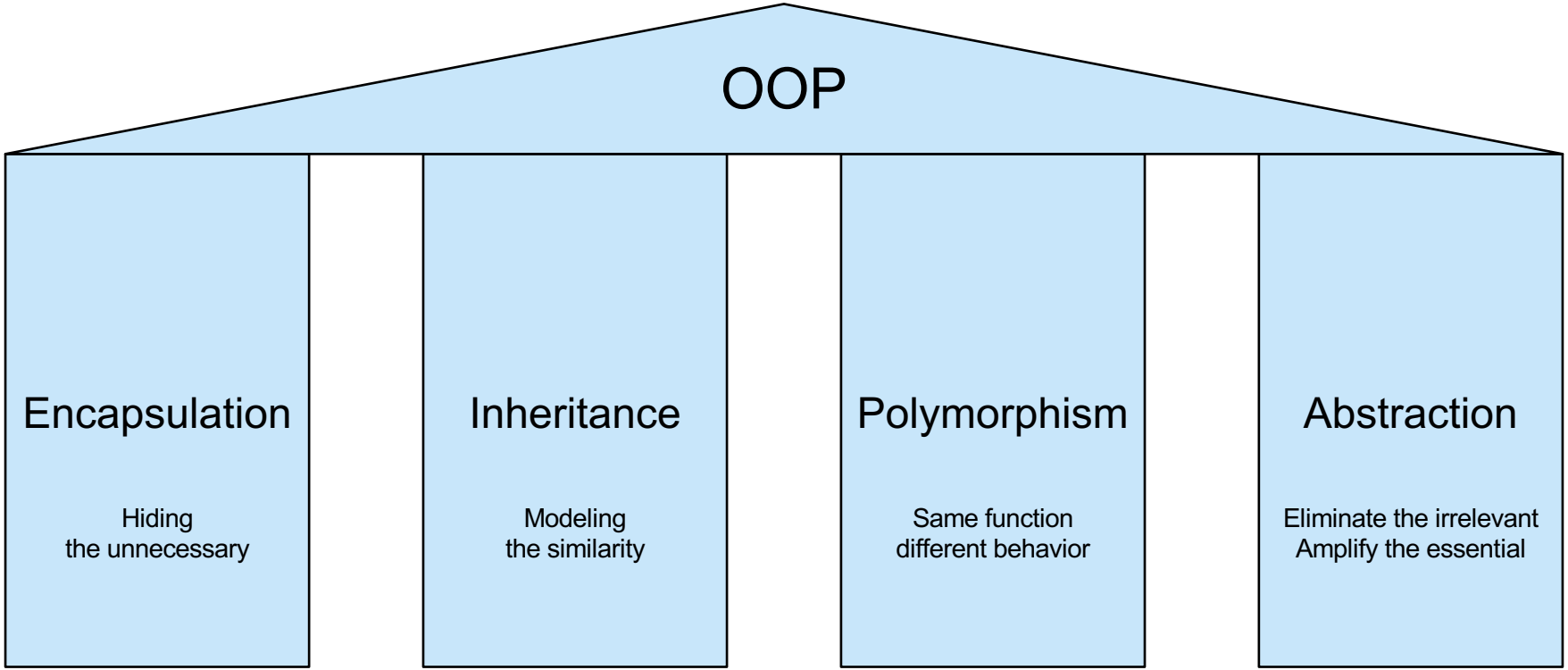
(d) Periodical and Magazine

UML Notation Overview



Four Principles of OOP

Four Principles of OOP



#1 Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.

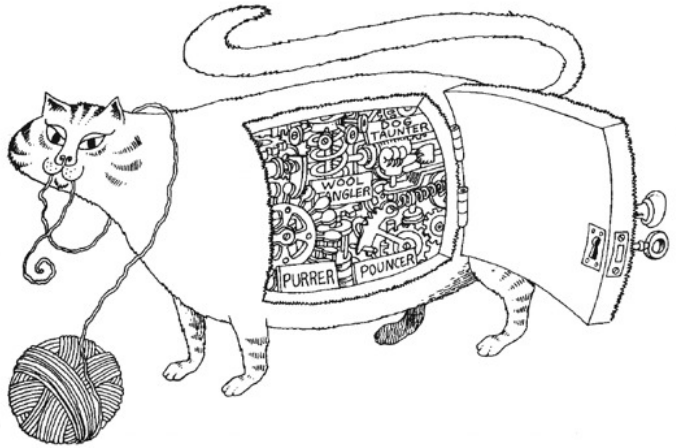
Without encapsulation

With encapsulation

Encapsulation in Java

- 1. Limit access to internal data by providing the methods that operate on the data
  - e.g., getter/setter methods, and other methods for the behavior
- 2. Restrict the direct access to the internal data of an object
  - Use the modifiers properly

*» **Encapsulation** hides the details of the implementation of an object.«*



Grady Booch, “Object-oriented Analysis and Design with Applications”, 1991.

Access Modifiers in Java

- **Modifiers** change the properties of classes, attributes and methods.
- **Accessibility** of attributes and methods:

Attribute/Method visible in …	`public`	`protected`	`(default)`	`private`
own **class**	yes	yes	yes	yes
same **package**	yes	yes	yes	no
subclass	yes	yes	no	no
any other class	yes	no	no	no

- By default, classes are only visible within their own package (“implicit friendly”). The modifier `public` means that a class is also visible in visible in other packages.

Information Hiding & Hyrums Law

- **information hiding** — A software development technique in which each module's interfaces **reveal as little as possible** about the **module's inner workings**, and other modules are **prevented** from using information about the module that is not in the module's interface specification. *IEEE Std 610.12 (1990)*

- **Hyrum's Law** (an observation from software engineering practice):

  "With a sufficient number of users of an API,
  it does not matter what you promise in the contract:
  all observable behaviors of your system
  will be depended on by somebody."

Modifier `final` – Changeability

- The modifier **`final`** restricts the **changeability** of the affected program elements:
  - **no subclasses** may be derived from **`final`**-classes
  - **`final`**-methods may **not be overriden**
  - **`final`**-attributes **may not be changed**, i.e. they are **constants**
  - **parameters** of methods and **local variables** declared as **`final`** may not be changed after initialization.

```
void increaseSalary(final double amount){
    amount = 2 * amount; // not allowed!
    ...
}
```

Modifier `static` – Lifetime

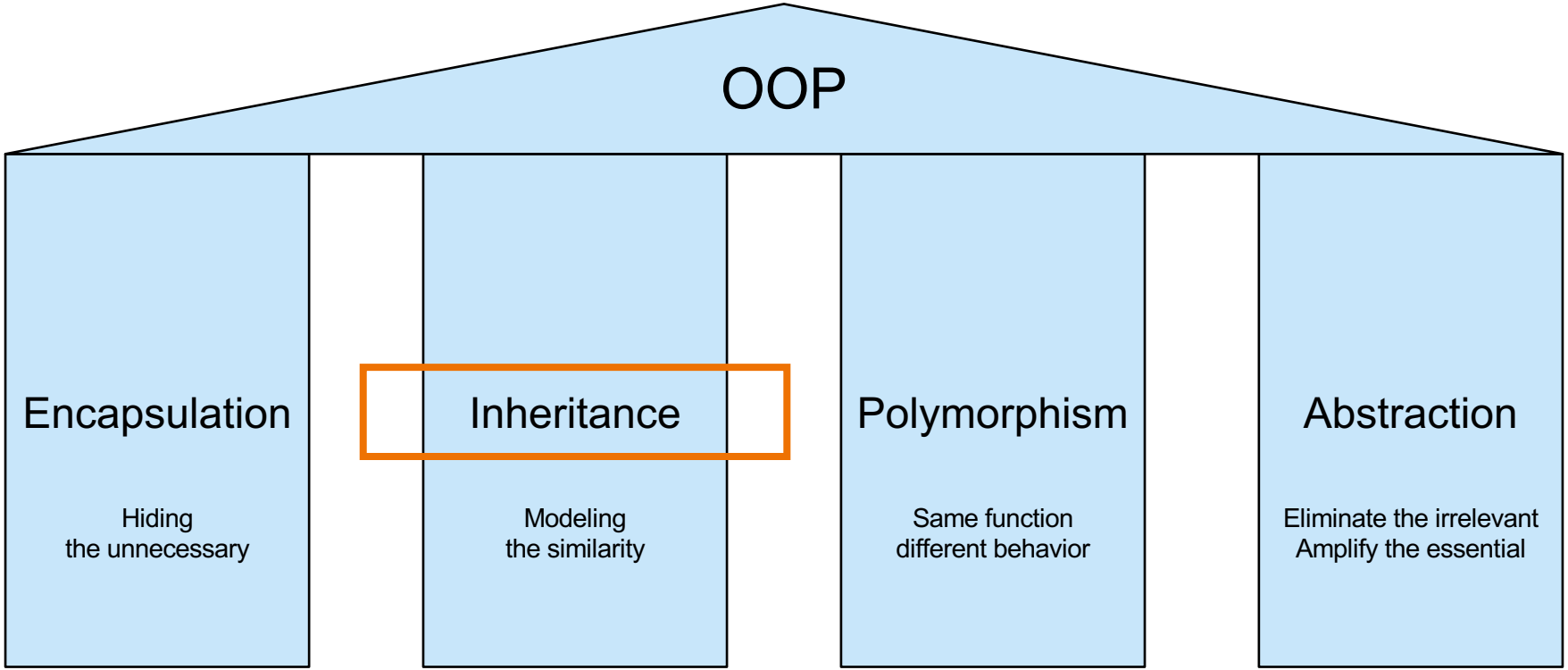
- The modifier **`static`** influences the **lifetime** and is used to define static attributes and methods.
- In contrast to “normal” attributes and methods, static attributes and methods are not bound to specific objects.
- static attributes (class attributes) are only created once per class: all objects of the class “share” the attribute.

```
public class void Person {
    public static int counter = 0;
    Person() { counter++; // counts instances}
    ...
}
```

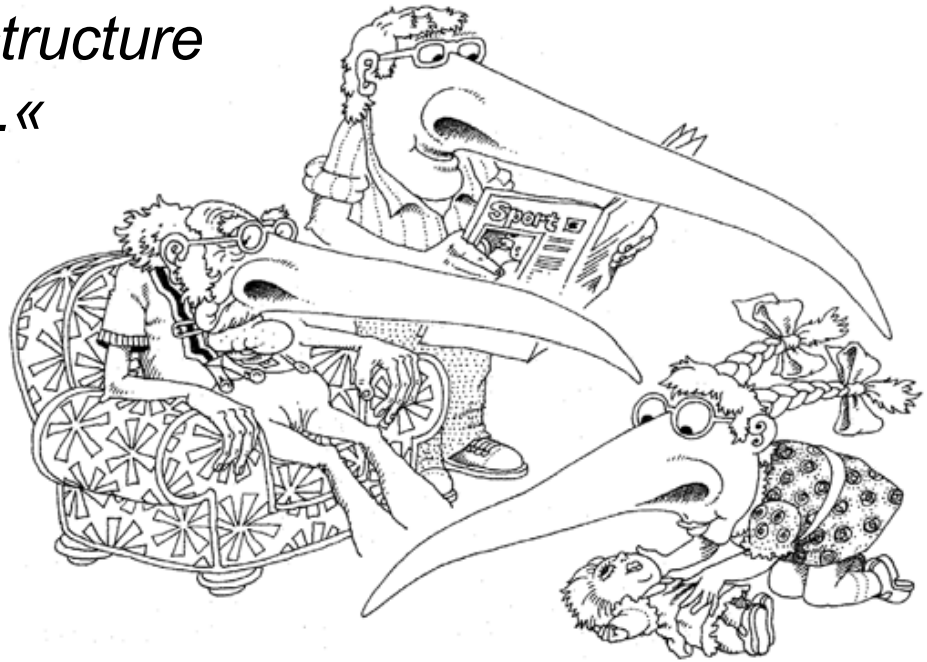
- External access to static attributes: **`ClassName.attribute`**

```
int numberofPersonObjects = Person.counter;
```

Four Principles of OOP



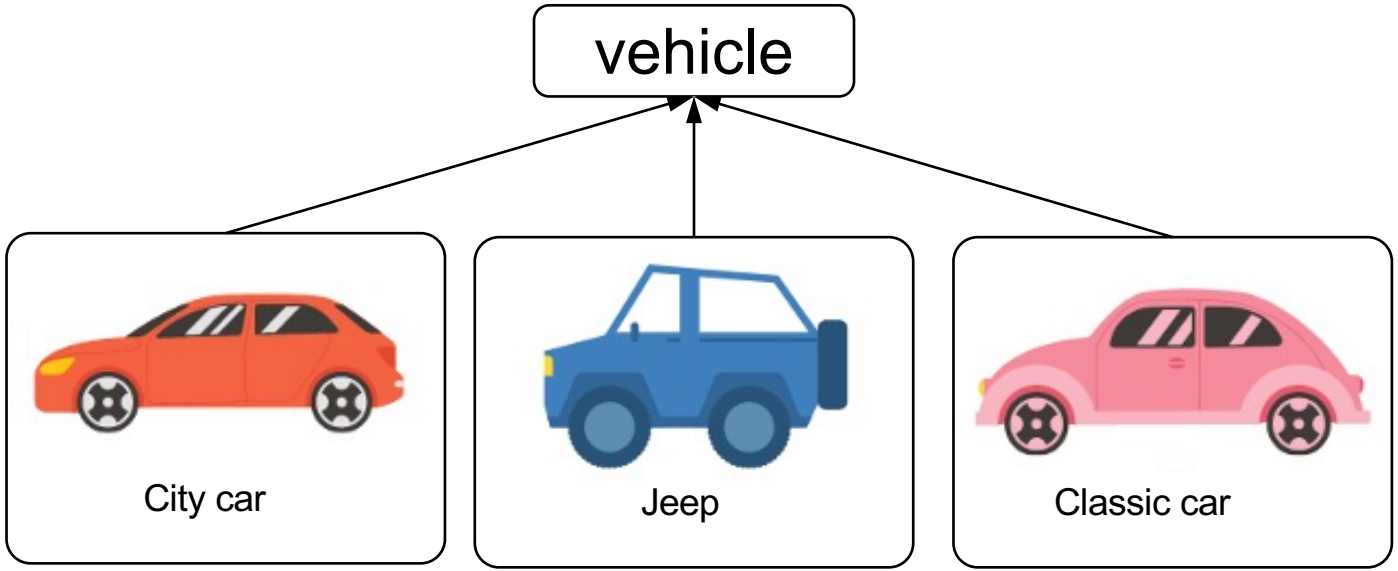
*»A subclass may **inherit** the structure and behavior of its superclass.«*



Grady Booch, “Object-oriented Analysis and Design with Applications”, 1991.

#2 Inheritance

Inheritance is a mechanism where you can derive a class from another class for a hierarchy of classes that share a set of attributes and methods.

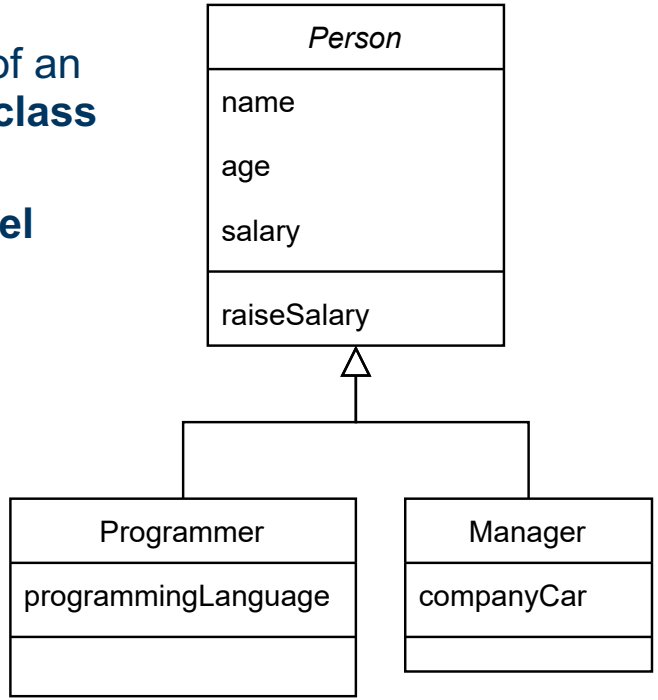


Inheritance

- If we need a **new class**, which represents an **extension** of an **existing class**, we can **reuse** the existing class as **base class** and **inherit** its structure.
- Inheritance supports the **reusability** on a **conceptual level**

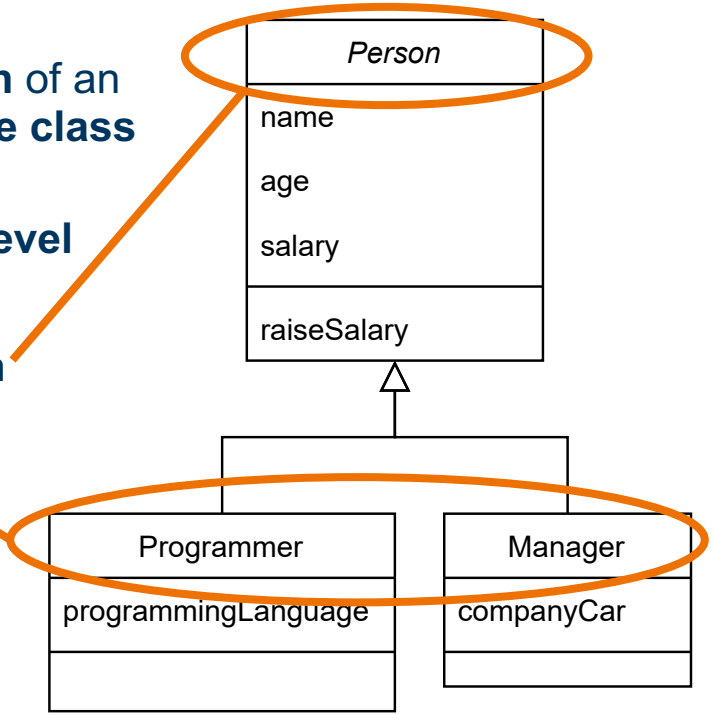
Inheritance

- If we need a **new class**, which represents an **extension** of an **existing class**, we can **reuse** the existing class as **base class** and **inherit** its structure.
- Inheritance supports the **reusability** on a **conceptual level**



Inheritance

- If we need a **new class**, which represents an **extension** of an **existing class**, we can **reuse** the existing class as **base class** and **inherit** its structure.
- Inheritance supports the **reusability** on a **conceptual level**
- **Terminology**:
  - base class, parent class, super class, **generalization**
  - subclass, children, child class, **specialization**
- **Child class** (subclass)
  - **Inherits** all attributes and methods of parent class
  - **Add** more attributes/methods
  - **Override** methods



Inheritance in Java

- The keyword **`extends`** serves for the inheritence:

```
public class Programmer extends Person {
    ...
}
```

- The new class inherits all attributes and methods (unless prevented by the **modifiers**).

Attribute/Method visible in …	`public`	`protected`	`(default)`	`private`
own **class**	yes	yes	yes	yes
same **package**	yes	yes	yes	no
subclass	yes	yes	no	no
any other class	yes	no	no	no

`super` keyword

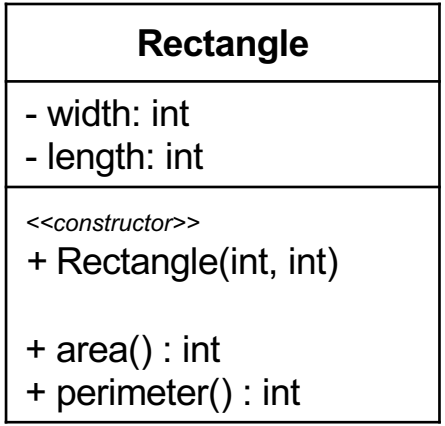
- Within a constructor, we can access the base class via **`super`**:

```
super(length, length);
// parameterized constructor of the base class
```

- Needs to be the first statement in the constructor.

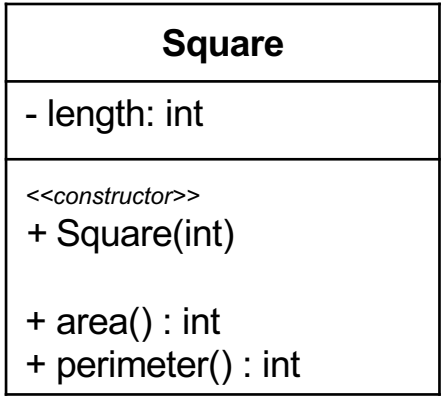
Example: Rectangle

```java
public class Rectangle {

    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int area() { return this.length * this.width; }

    public int perimeter() {
        return 2 * this.length + 2 * this.width;
    }
}
```



Example: Square

```
public class Square {

    private int length;

    public Square(int length) {
        this.length = length;
    }

    public int area() { return this.length * this.length; }

    public int perimeter() {
        return 4 * this.length;
    }
}
```

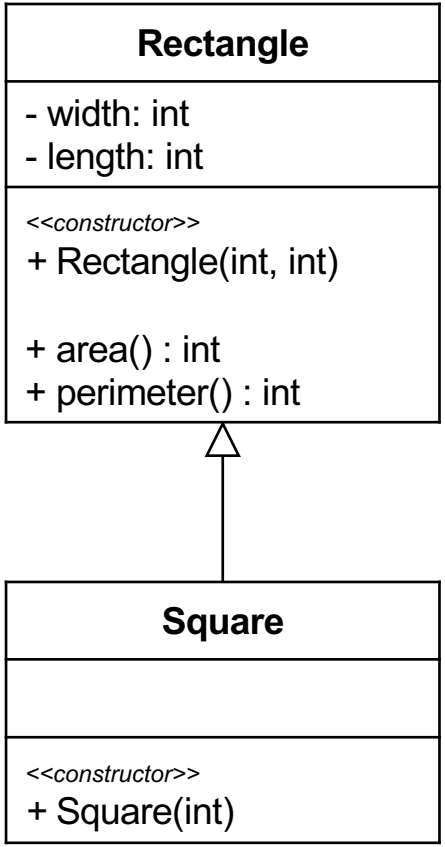


Is there a better, more convenient way?

Example: Square

```
public class Square extends Rectangle {

    public Square(int length) {
        super (length, length);
    }

}
```

Advantage: make the code **simple** and **extensible**

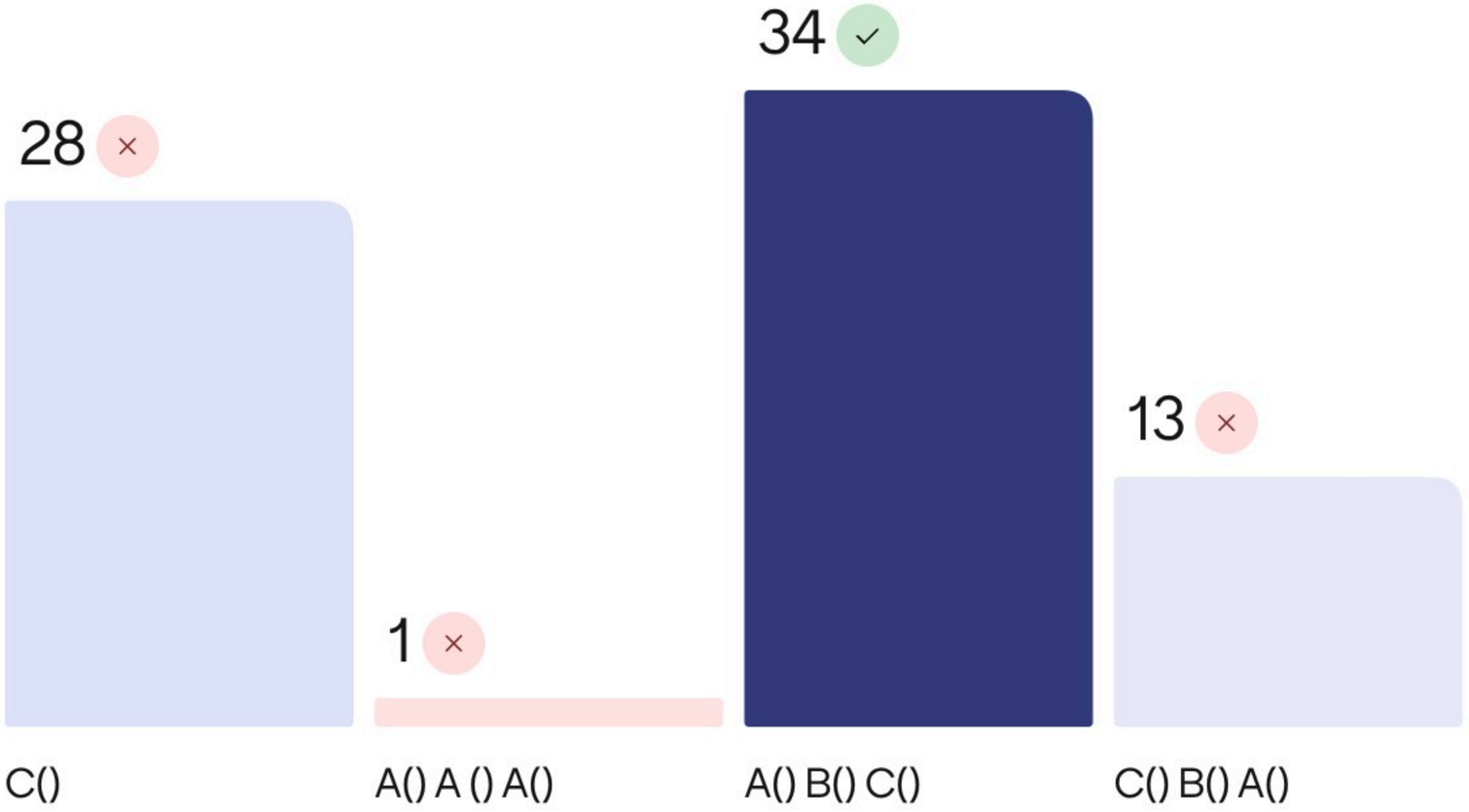


Quiz

What will be the output?

```
class A                 { A() { System.out.print("A() "); } }
class B extends A       { B() { System.out.print("B() "); } }
class C extends B       { C() { System.out.print("C() "); } }
...
C c = new C();
```

What will be the output?




```
class A                { A() { System.out.print("A() "); } }
class B extends A      { B() { System.out.print("B() "); } }
class C extends B      { C() { System.out.print("C() "); } }
...
C c = new C();
```

Quiz

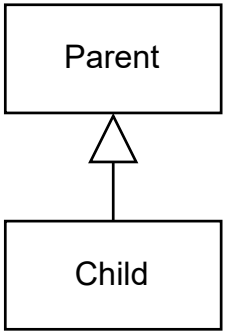
What will be the output?

```
class A                 { A() { System.out.print("A() "); } }
class B extends A       { B() { System.out.print("B() "); } }
class C extends B       { C() { System.out.print("C() "); } }
...
C c = new C(); // Output "A() B() C()"
```

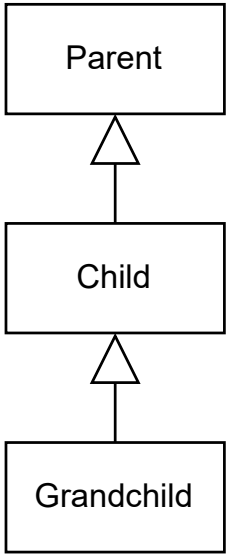
- If **`super`** is not called, there will be an **implicit** call of the base class constructor without parameters!

Four Types of Inheritance

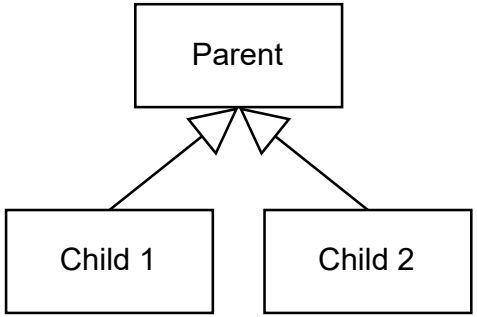
Single Inheritance



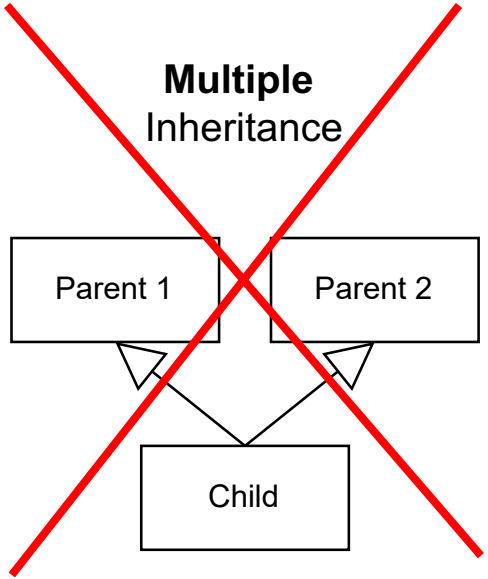
Multi-level Inheritance



Hierarchical Inheritance

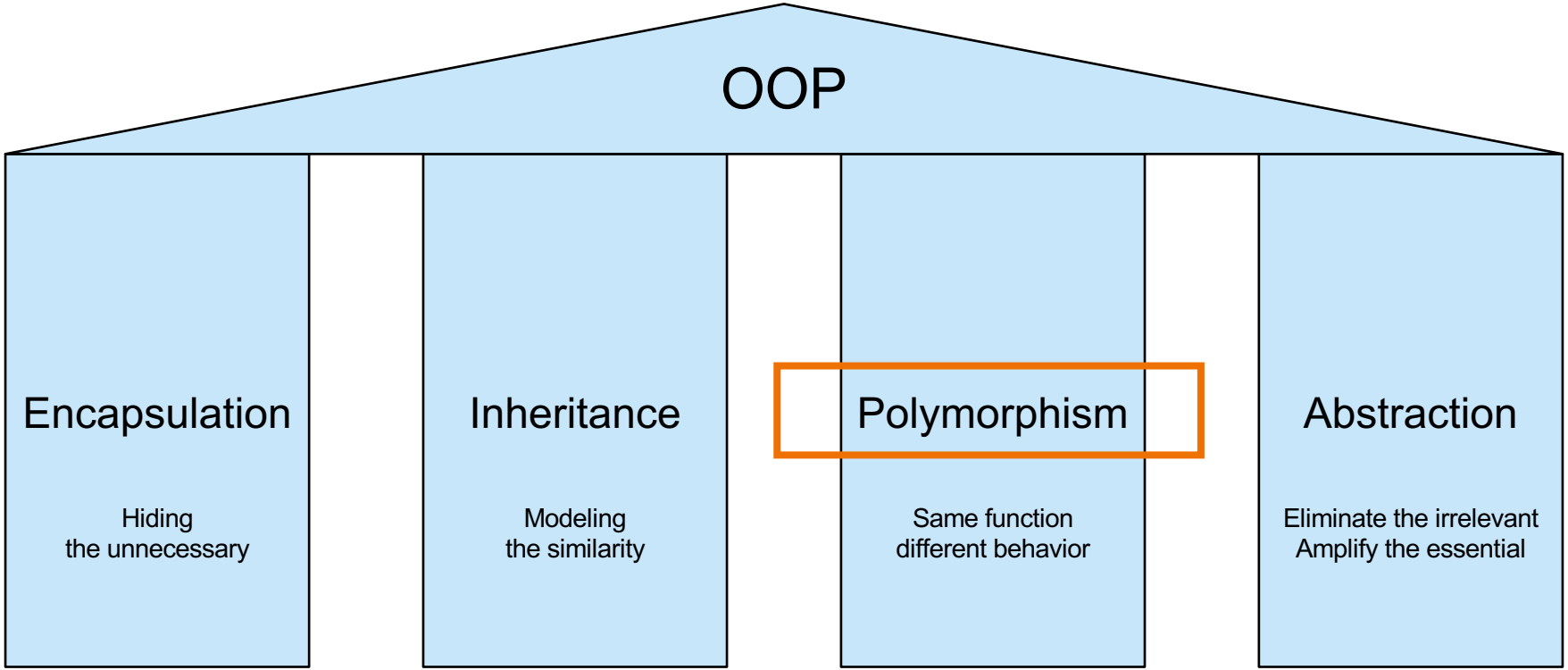


Multiple Inheritance



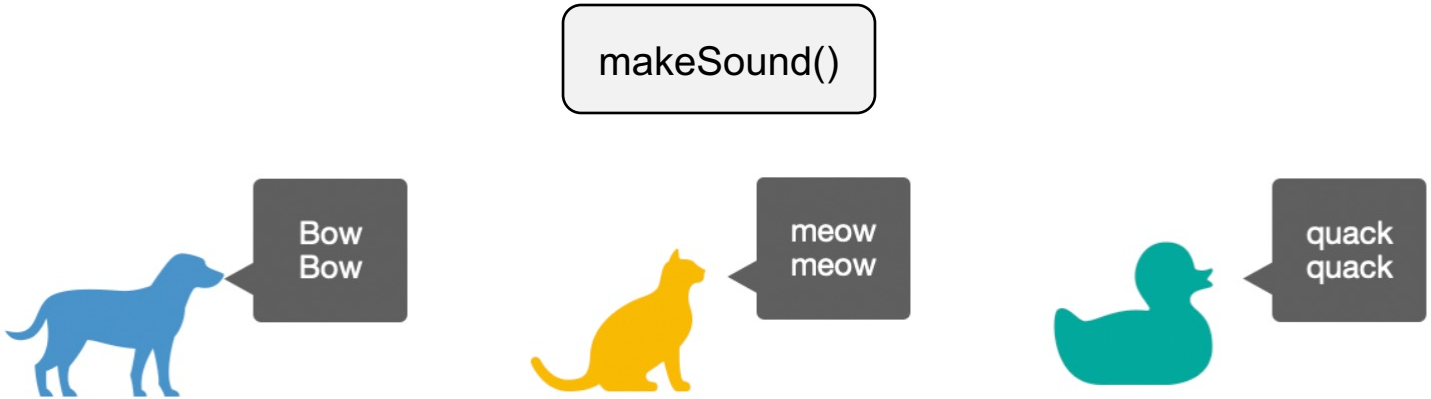
Java only supports the **single inheritance**, i.e., a class can only have one base class.

Four Principles of OOP

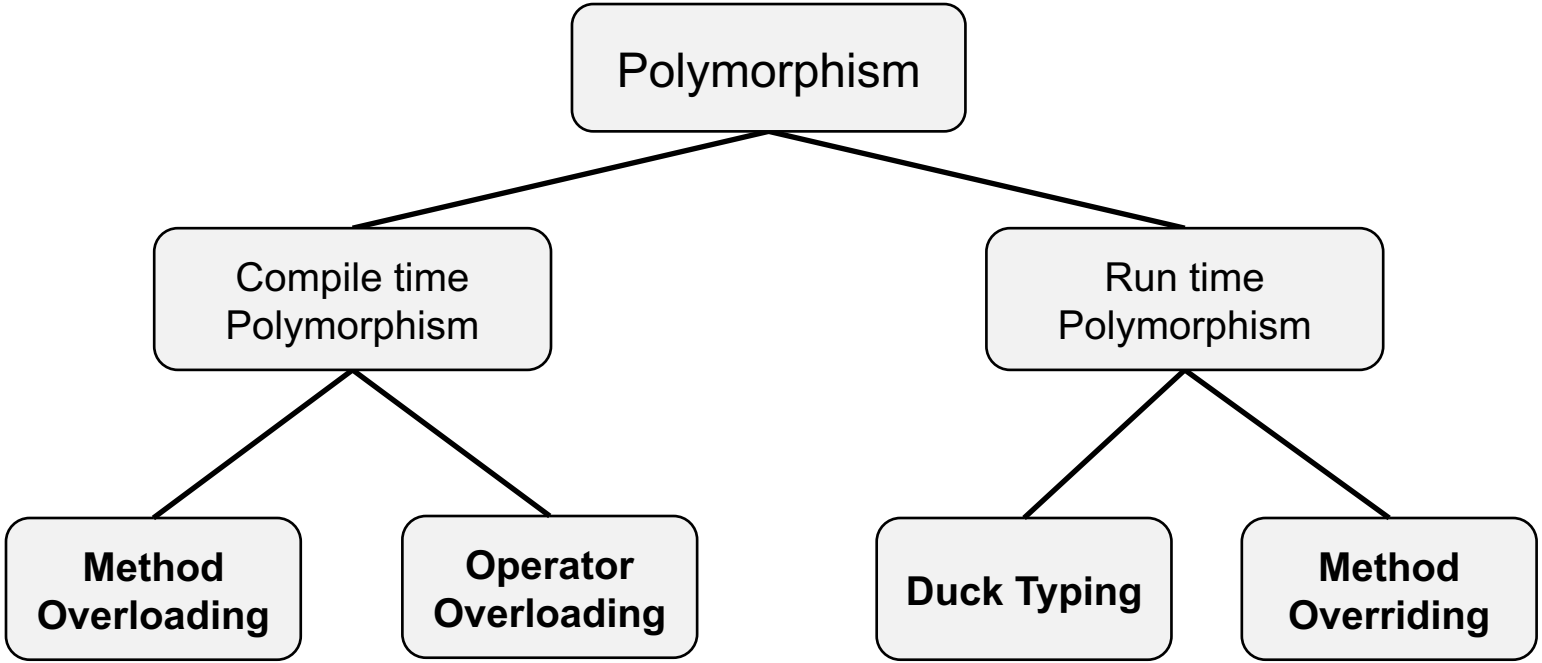


#3 Polymorphism

- **Greek origin**: Polymorphism means existing in many forms.
  - "poly" = many, "morphism" = forms
- In programming, polymorphism means that we can call the same method name, and depending on its parameters or the class, the method will do different things.



Four Types of Polymorphism



Method Overloading

- **Method overloading** is a type of polymorphism in which we can define a number of methods with the **same name** but with a **different number of parameters** as well as parameters can be of d**ifferent types**.

```
public int add(int a, int b) { return a + b; }

public int add(int a, int b, int c) { return a + b + c; }
```

Operator Overloading

```
System.out.println(1 + 2);

System.out.println(”hello ” + “world!”);
```

Java does not support custom operator overloading!

Duck Typing

- **Duck typing** is a concept that says that the “type” of the object is a matter of concern only at runtime and you don’t need to explicitly mention the type of the object before you perform any kind of operation on that object
- Duck typing is a concept related to **dynamic typing**, where the type or the class of an object is less important than the methods it defines.
  - When you use duck typing, you do not check types at all.
  - Instead, you check for the presence of a given method or attribute.
- Duck typing is not directly supported in Java, but there are some dynamic proxies
  - Interfaces, Generics

Duck Typing (e.g., in Python)

```
class USA():
    def capital(self):
        print("Washington, D.C.")

    def language(self):
        print("English.")

class France():
    def capital(self):
        print(”Paris.")

    def language(self):
        print(”French.")
```

```
obj_usa = USA()
obj_fra = France()

for country in (obj_usa, obj_fra):
    country.capital()
    country.language()
```

Output

```
Washington, D.C.
English.
Paris.
French.
```

Override Methods

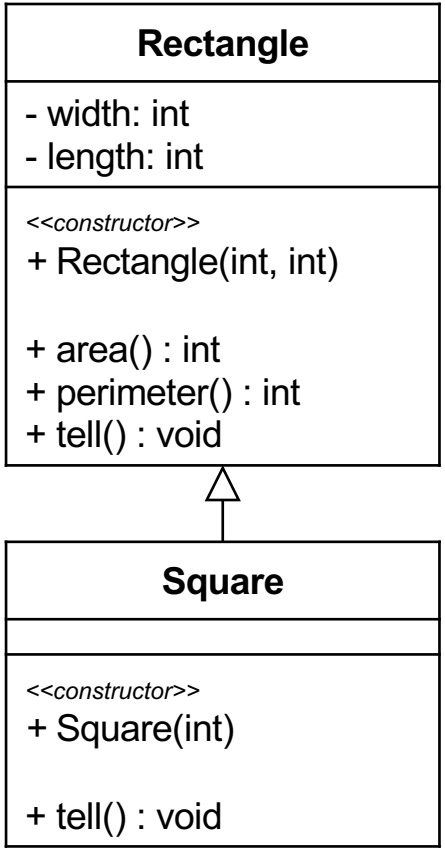
```
public class Rectangle {

    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int area() { return this.length * this.width; }
    public int perimeter() { return 2 * this.length + 2 * this.width;}

    public void tell() { System.out.println("This is a rectangle."); }

}
```

Rectangle
- width: int - length: int
<<constructor>> + Rectangle(int, int) + area() : int + perimeter() : int + tell() : void

New method shared by superclass and subclass.

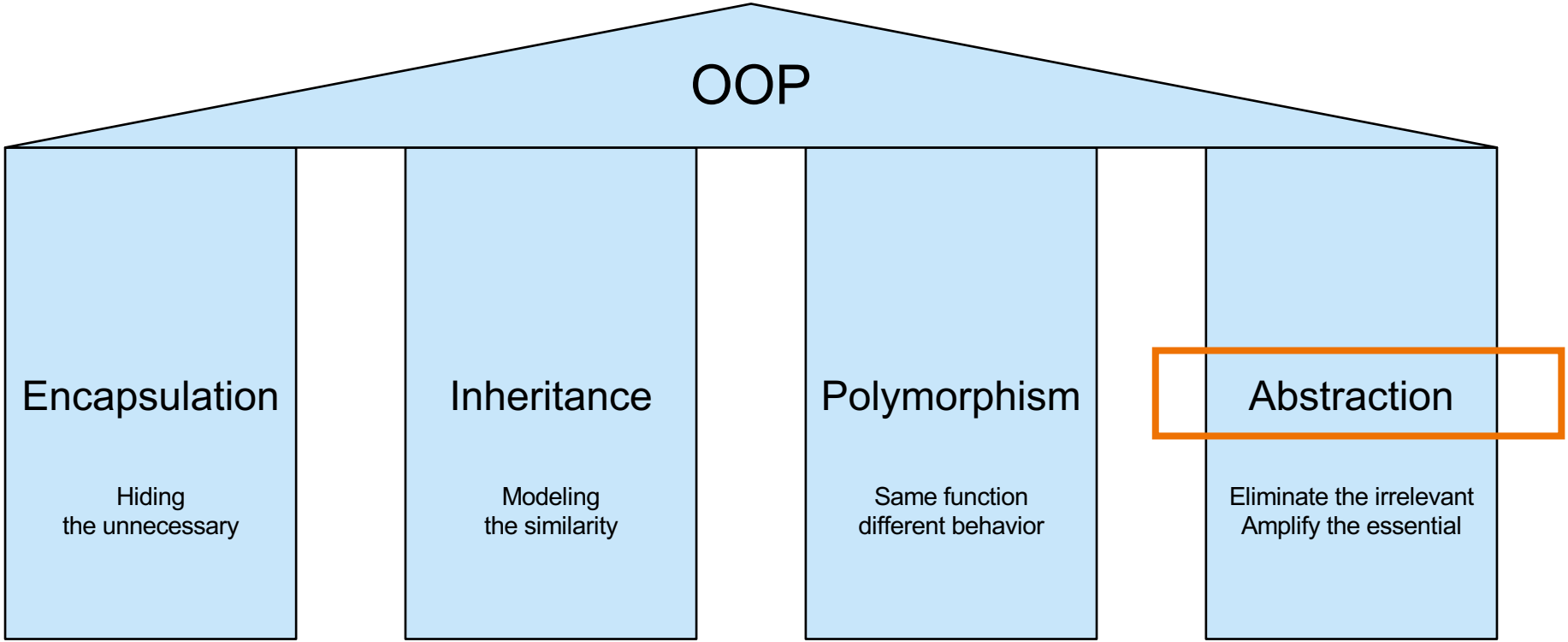
Override Methods

```
public class Square extends Rectangle {

    public Square(int length) {
        super (length, length);
    }

    public void tell() { System.out.println("This is a square."); }

}
```



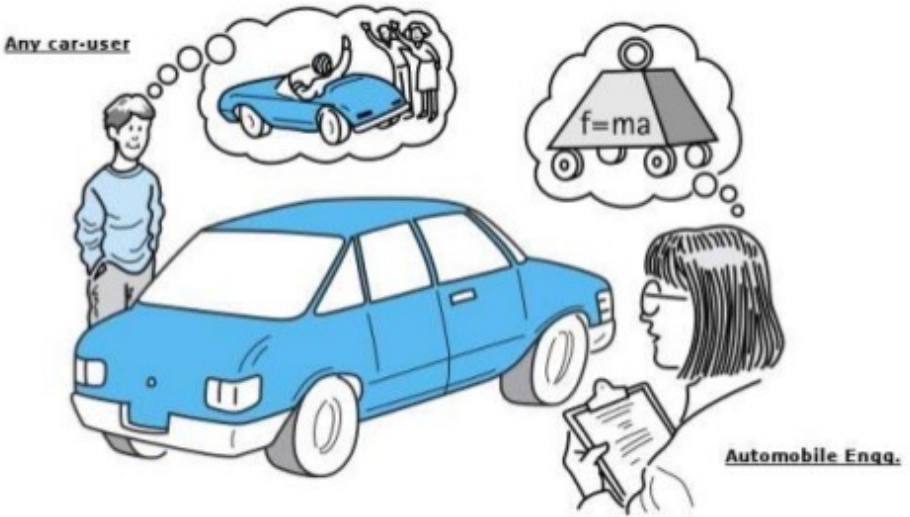
- we can change the functionality of the tell() method in the superclass by redefining it in the subclass
- this is termed **method overriding**
- the implementation of a method will be determined dynamically

Four Principles of OOP



Abstraction

- **Abstraction** includes the essential details relative to the perspective of the viewer.
- Java specifics:
  - Abstract classes
  - Abstract methods
  - Interfaces
  - Generics



More Topics

- **Java**
  - Abstract Classes, Interfaces, Generics, Virtual Machine, Memory Management, Java Bytecode, …
- **Object-Orientation**
  - How to find classes?
  - What notation shall we use for classes? UML.
  - How do I structure my classes? How do I design the instantiation of objects? → Design Principles and Design Patterns.
- **Software Engineering**: Processes and Methods
- Support for **Testing** and **Verification**, Software **Quality**

Summary: Four Principles of OOP in Java

Data Security

Encapsulation

- Getter
- Setter
- Access Modifiers

Inheritance

- Superclass
- Subclass
- Method overriding

Code Reusability, Extensibility

Code Simplicity

Polymorphism

- Method overloading
- ~~Operator overloading~~
- ~~Duck typing~~
- Method overriding

Abstraction

- Abstract class
- Abstract method

Reduce complexity

Summary

- **Programming Languages** (PLs) determine what algorithms and ideas you can express
- **Typesystems** are an essential part of PLs
- **Object-Orientation**
  - Objects: construct classes based on object abstraction
  - **Instantiation** of objects with constructors
  - Four **Principles** of OOP
    - #1 **Encapsulation**: internal data and behavior, information hiding
    - #2 **Inheritance**: code reuse and extension
    - #3 **Polymorphism**: code simplicity, overloading/overriding
    - (#4 Abstraction) ~> reduce complexity