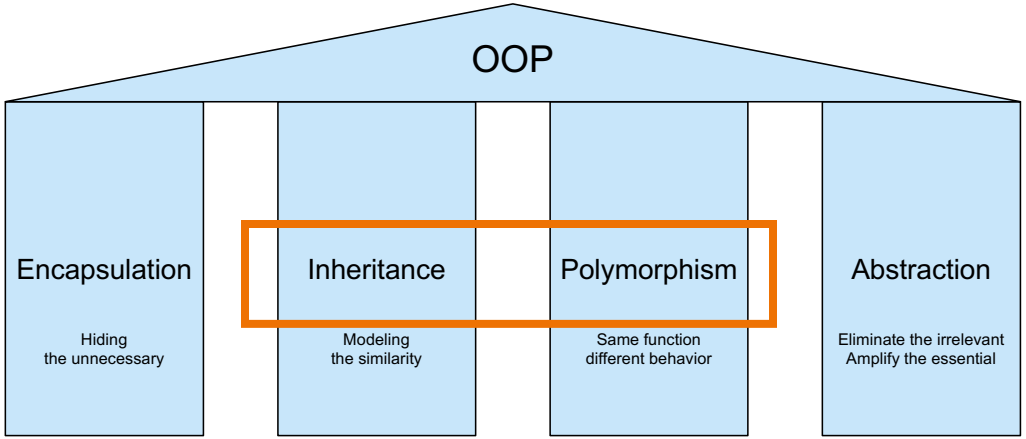


RUHR-UNIVERSITÄT BOCHUM

Vorkurs Informatik

Vorlesung 4: Principles of Object-Oriented Programming II

Prof. Dr. Yannic Noller
Software Quality group

Agenda

- Recap: Encapsulation, Composition
- Four Principles of OOP
  - #2 Inheritance
  - #3 Polymorphism
  - (#4 Abstraction)

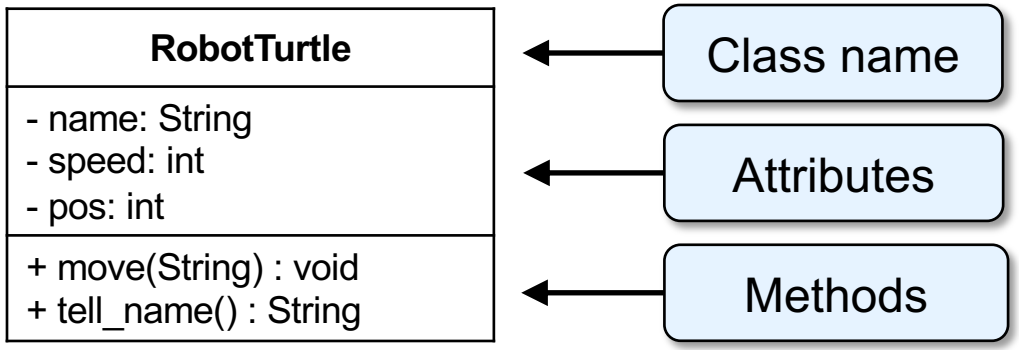
Recap: Classes and Objects

- 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, …

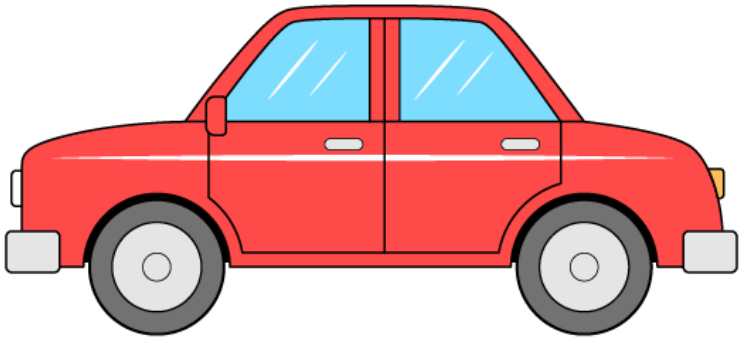
Recap: UML Class Diagram

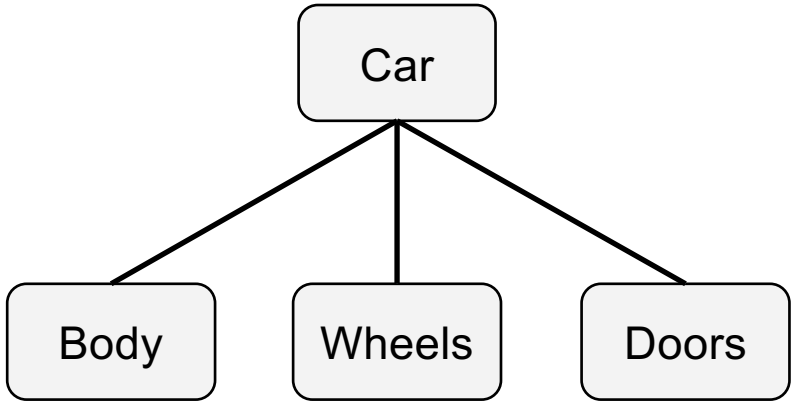
- Unified Modeling Language (UML) gives some specifications how to represent the classes visually



Recap: Composition

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





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

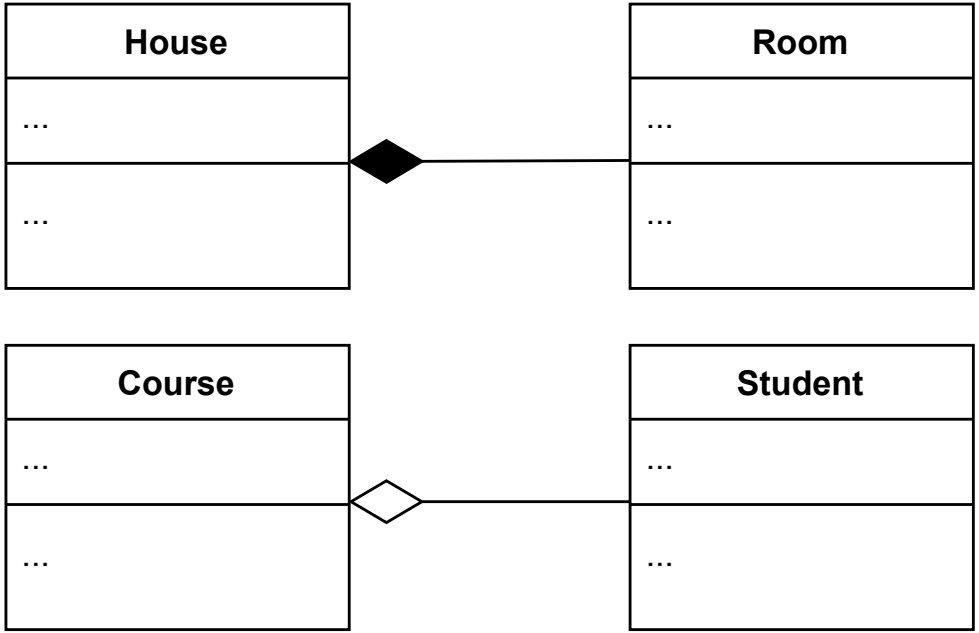
(c) TennisPlayer and Racket

(d) Periodical and Magazine

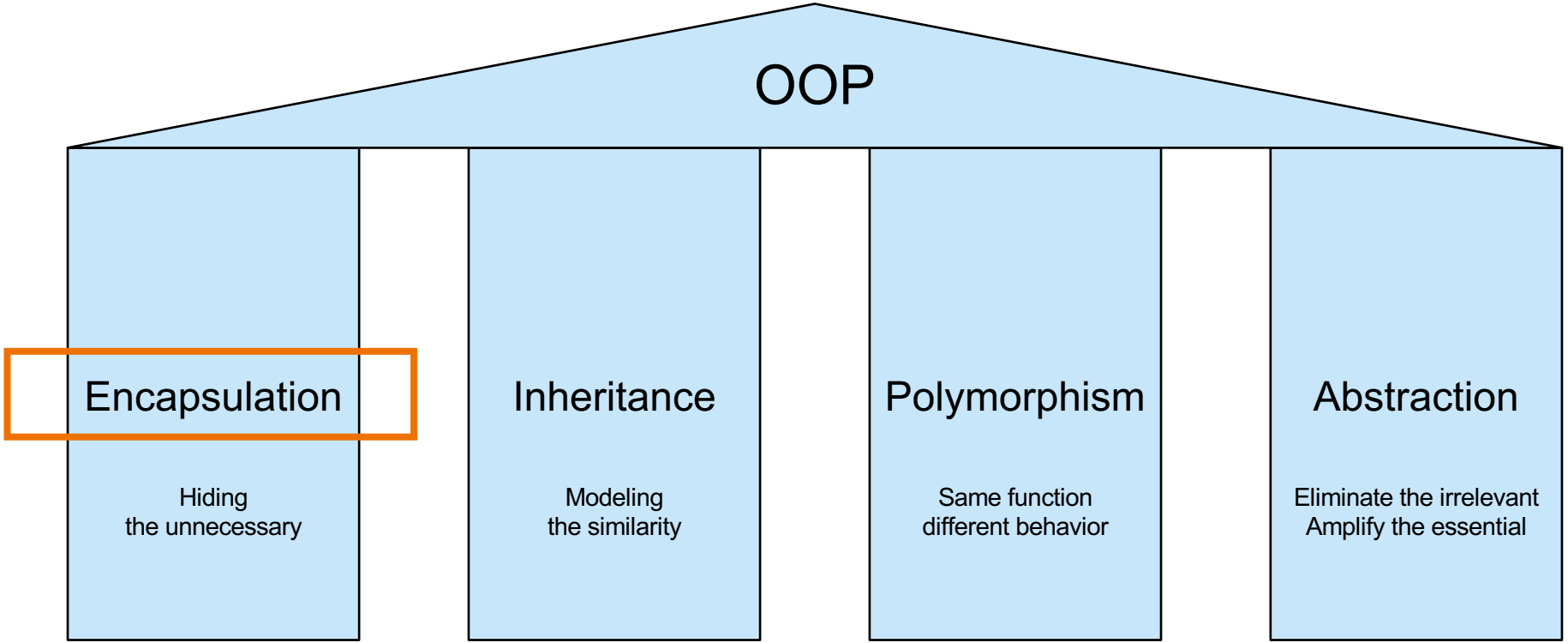
Recap: UML – Relationships between Classes

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



Recap: Four Principles of OOP



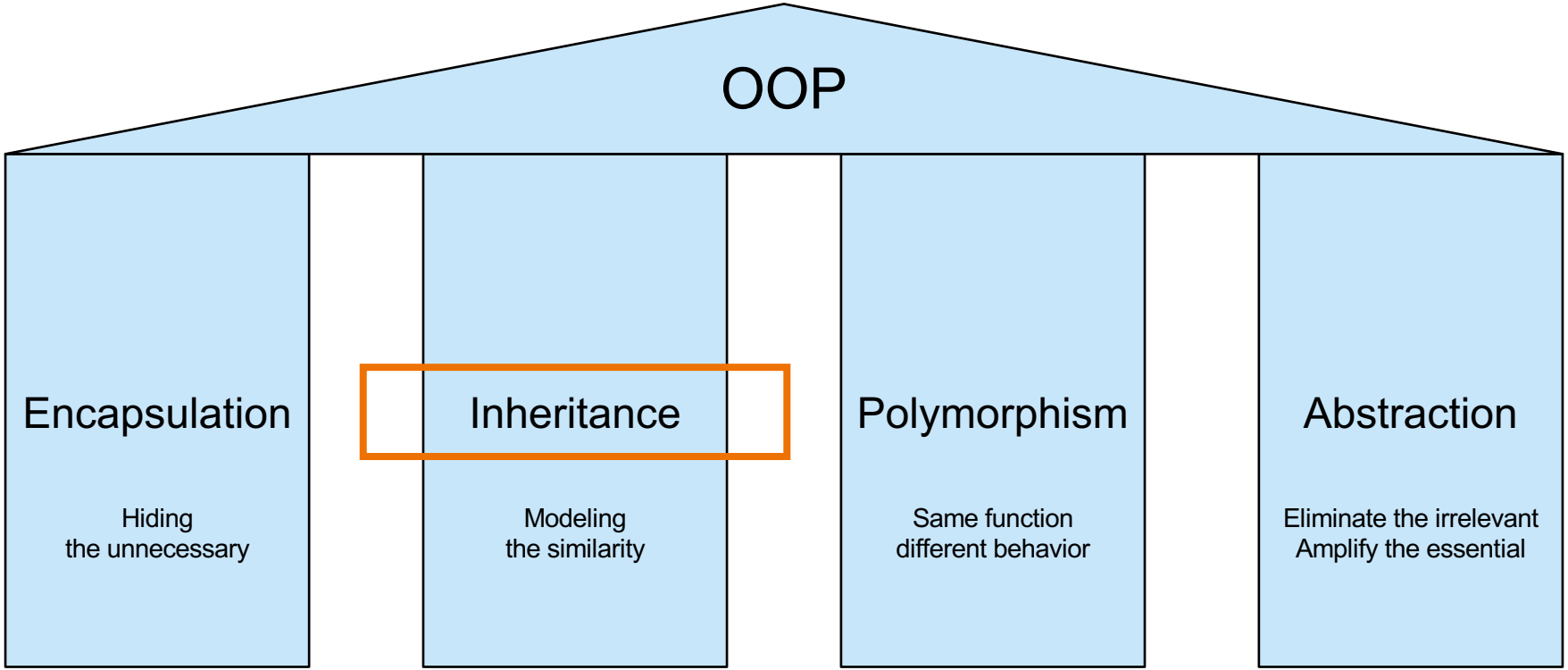
Recap #1 Encapsulation

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.

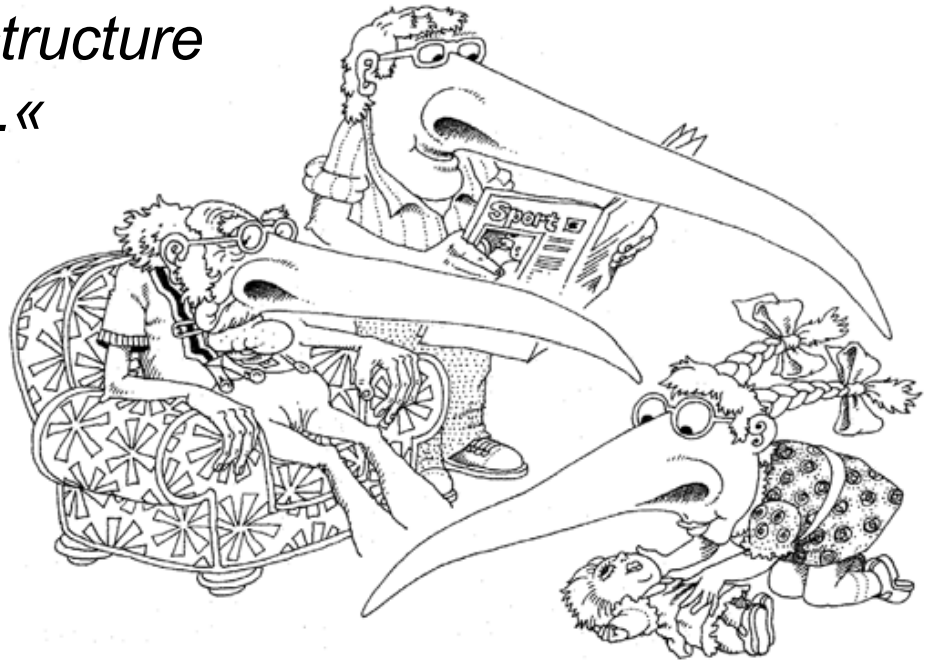
Without encapsulation

With encapsulation

Four Principles of OOP



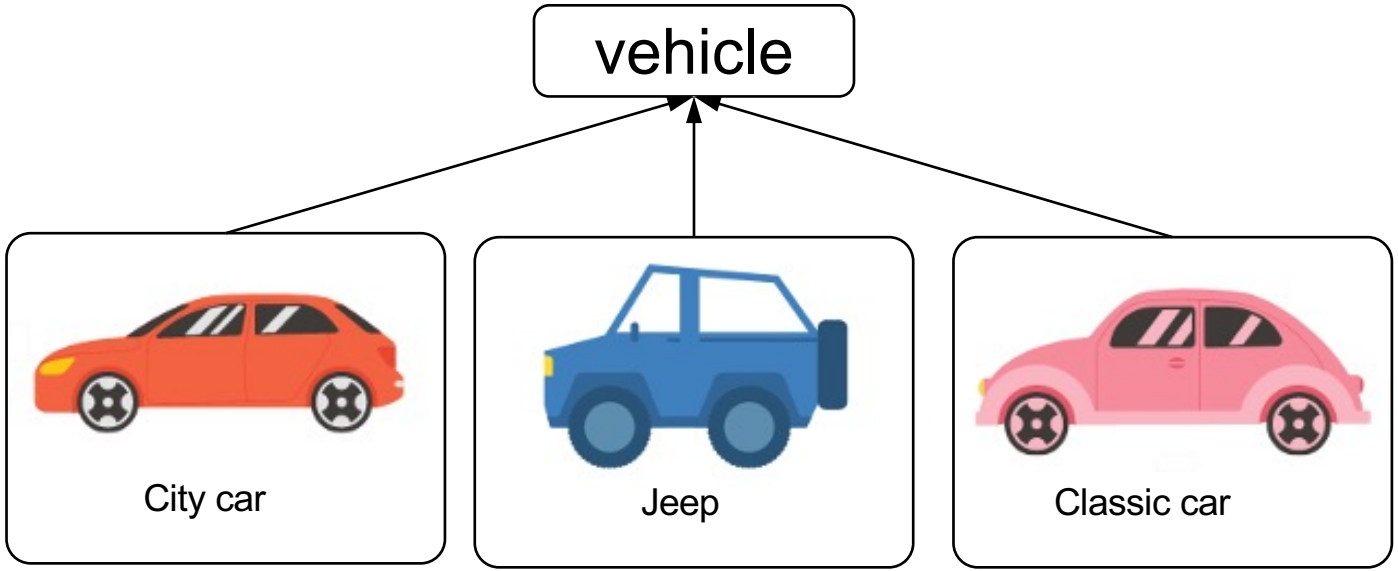
*»A subclass may **inherit** the structure and behavior of its superclass.«*



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

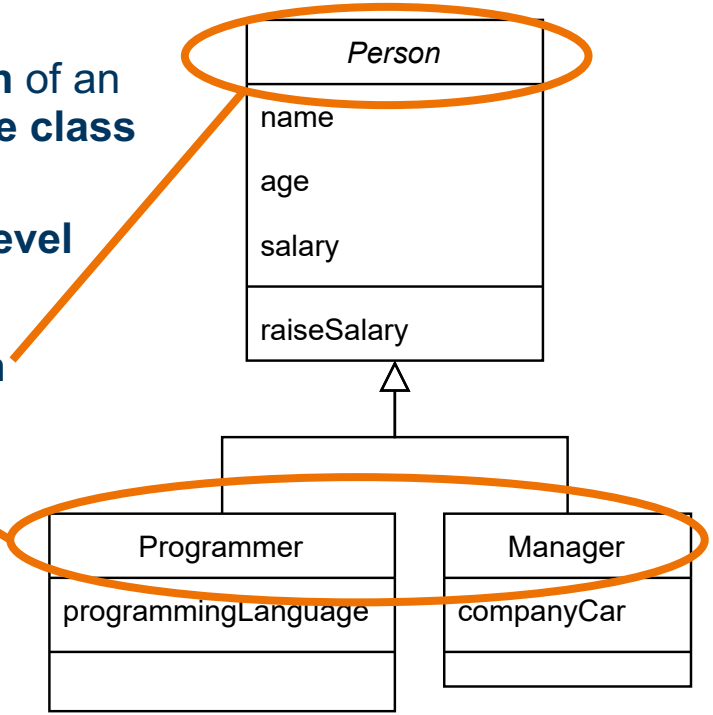
#2 Inheritance

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**
- **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:

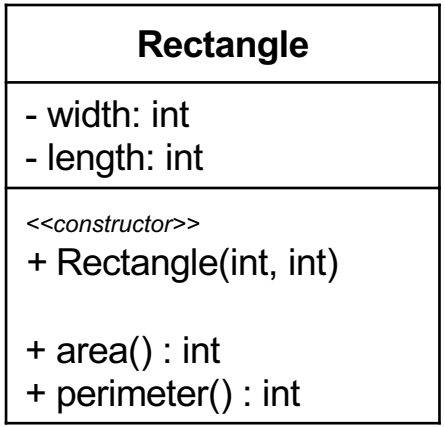
```
public class Programmer extends Person {
    ...
}
```

- 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

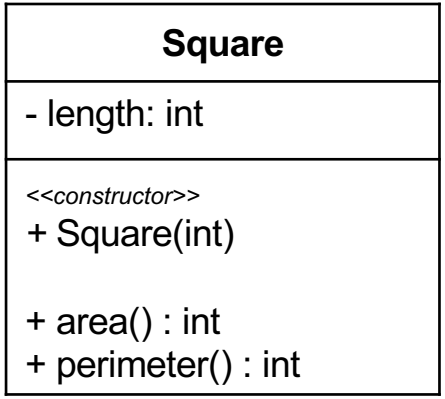
Example: Rectangle

```
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

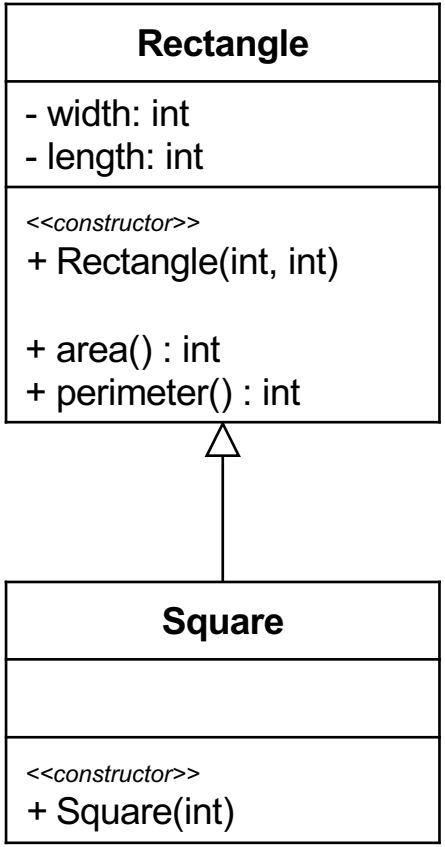
```
public class Square {

    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;
    }
}
```



Example: Square

```
public class Square extends Rectangle {

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

}
```

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



`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.

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();
```

Quiz

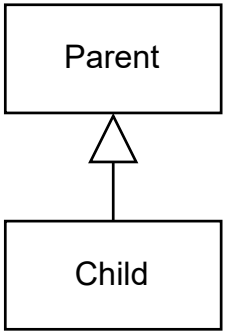
What will be the output?

```
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()"
```

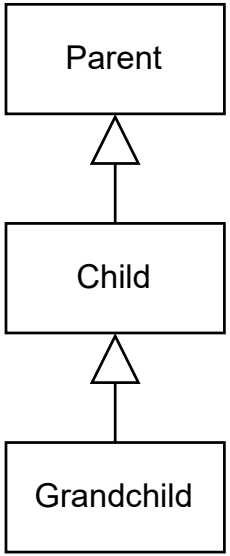
- If `super` is not called, there will be an **implicit** call of the base class constructor without parameters!

Four Types of Inheritance

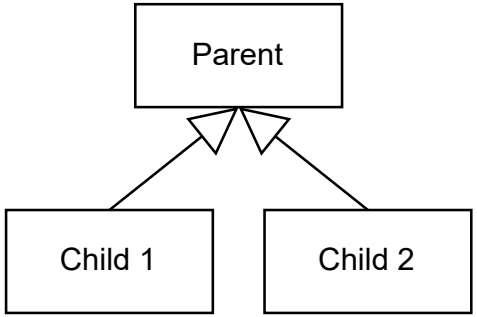
Single Inheritance



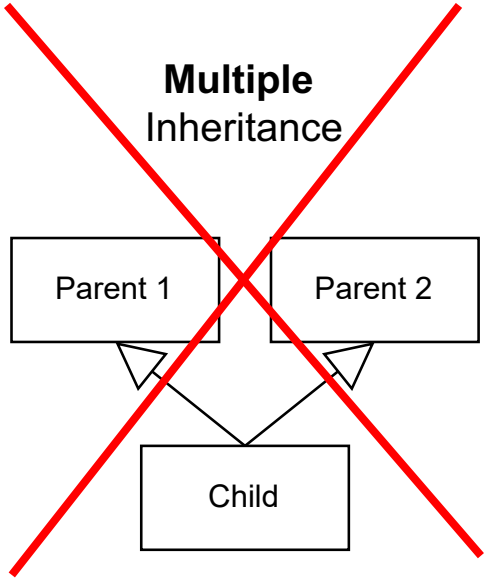
Multi-level Inheritance



Hierarchical Inheritance

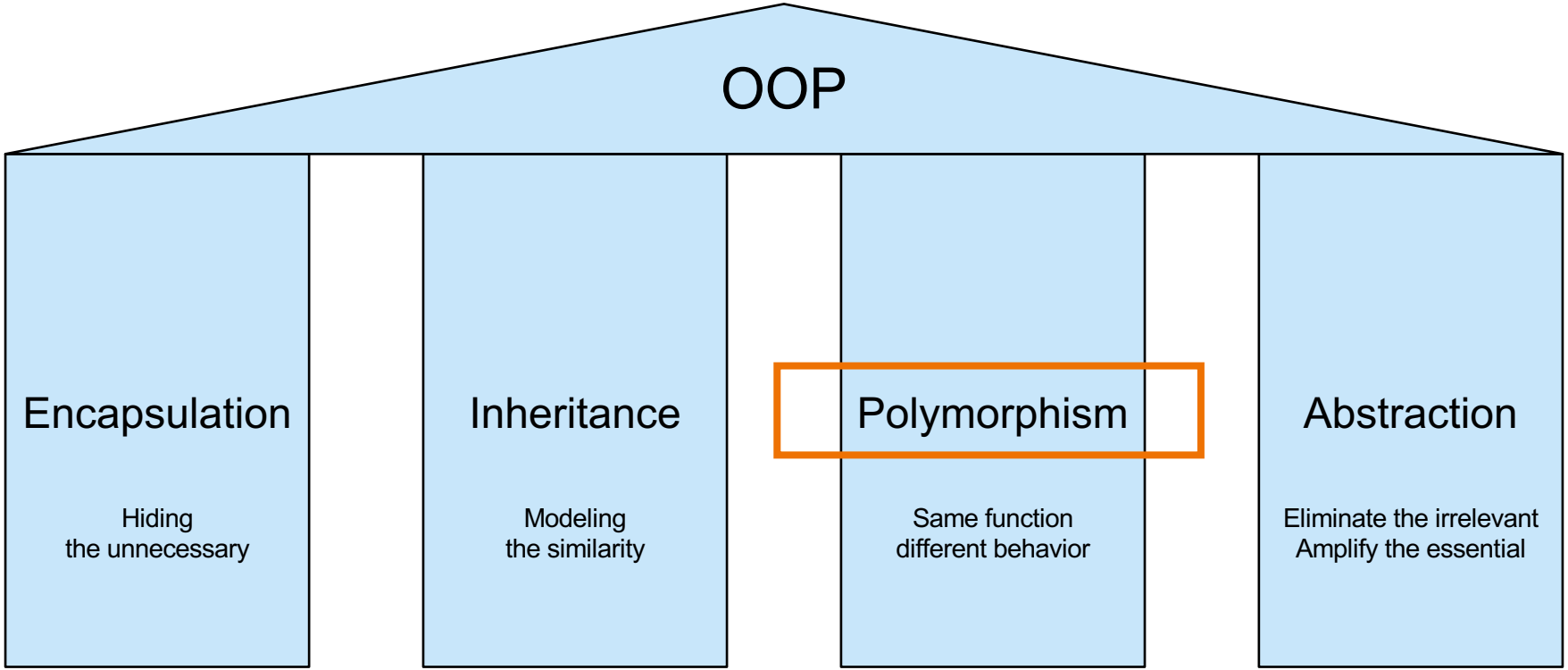


Multiple Inheritance



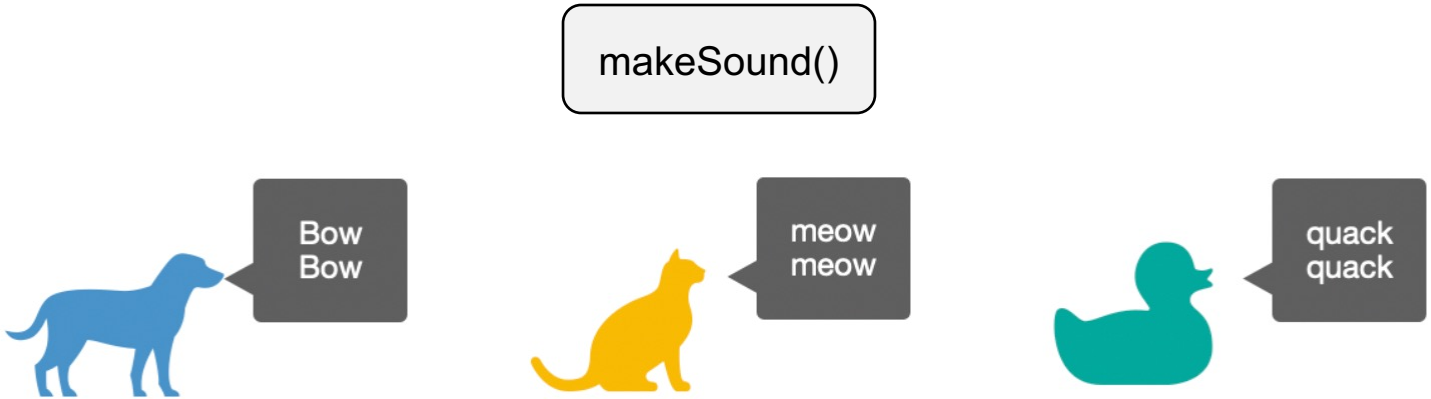
Java only supports the **single inheritance**, i.e., a class can only have one base class.

Four Principles of OOP

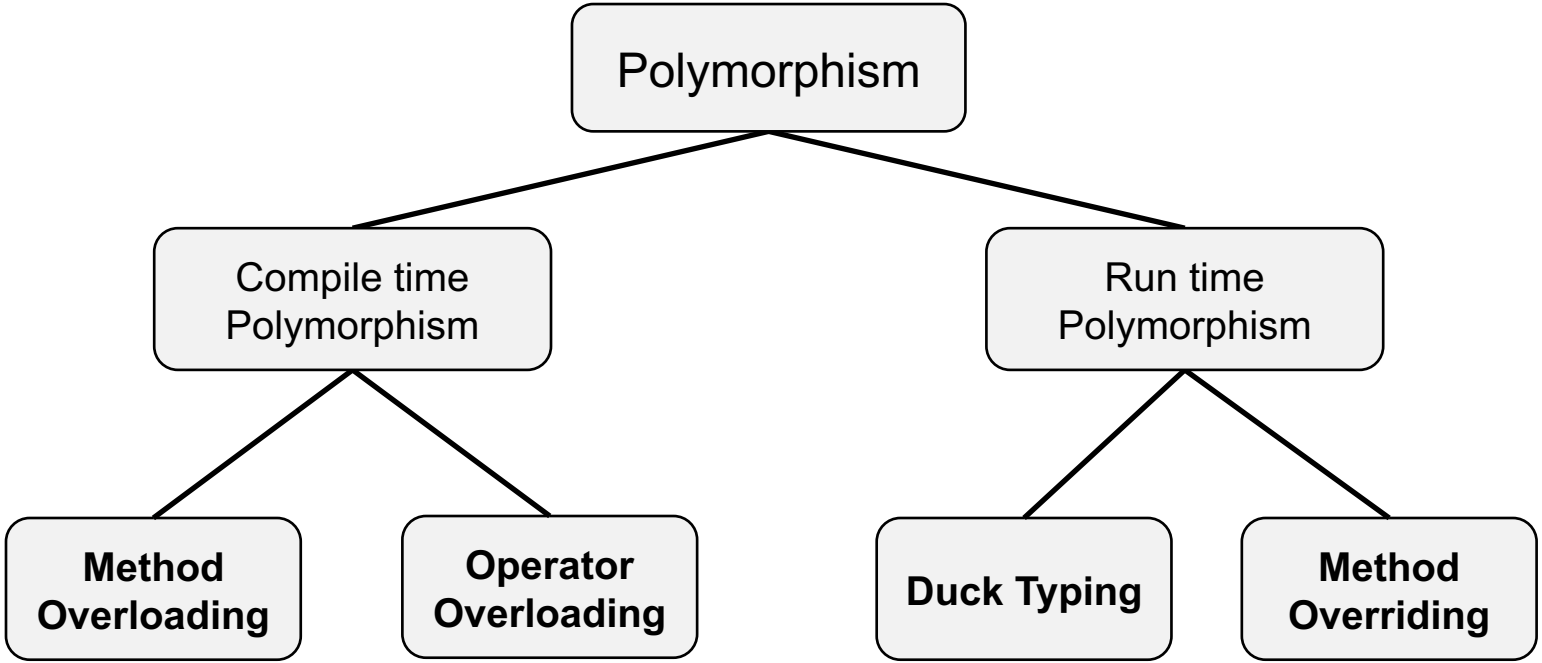


#3 Polymorphism

- **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.")
```

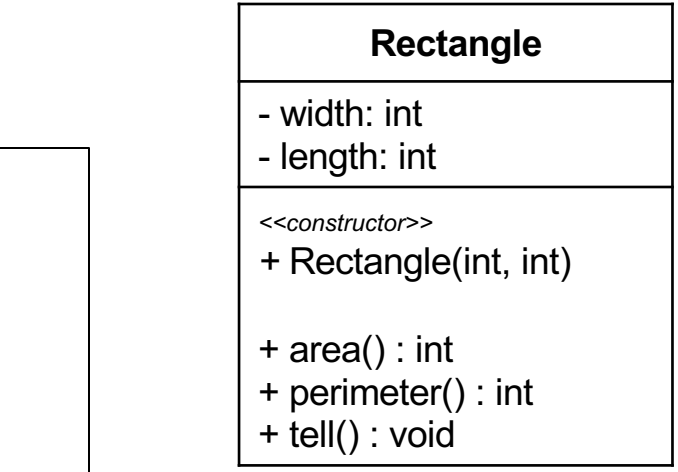
```
obj_usa = USA()
obj_fra = France()

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

Output

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

Override Methods

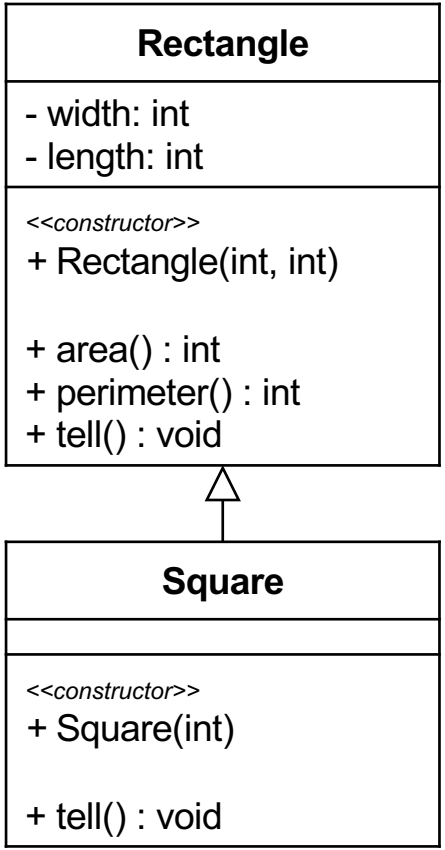
```
public class Rectangle {

    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."); }

}
```



New method shared by superclass and subclass.

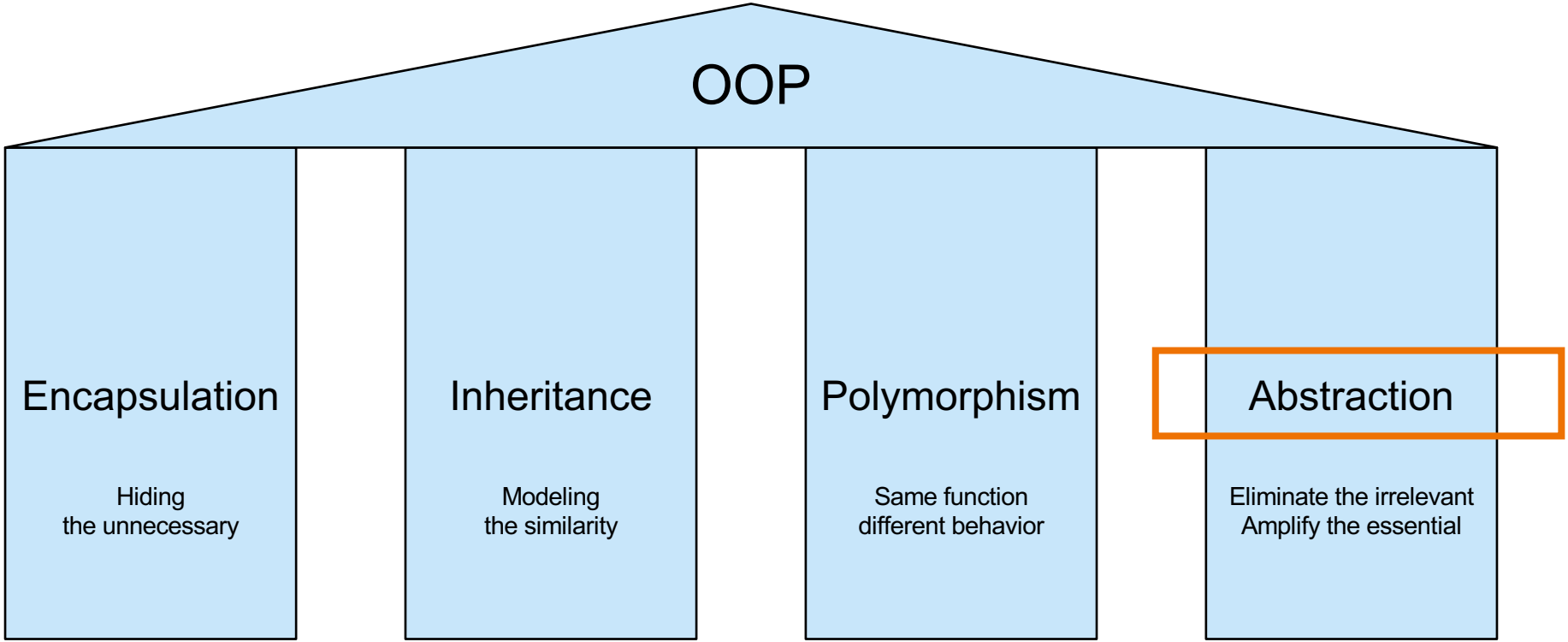
Override Methods

```
public class Square extends Rectangle {

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

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

}
```



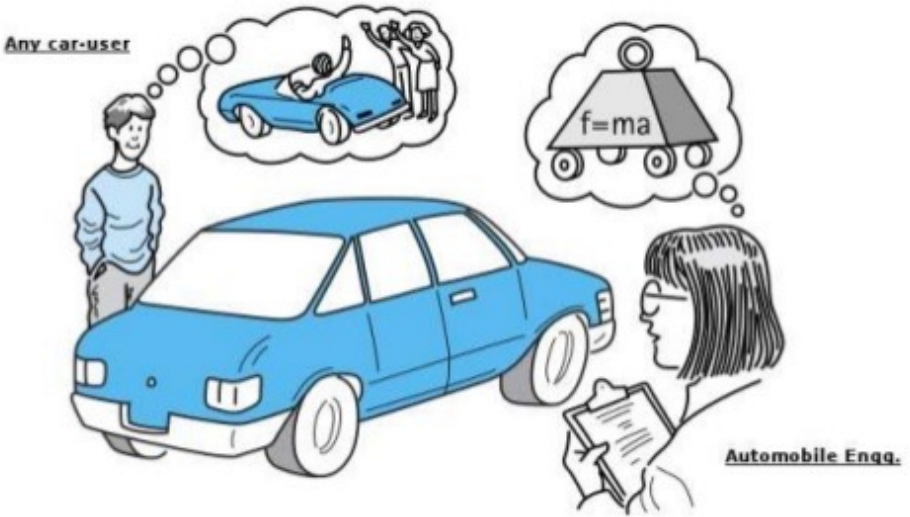
- we can change the functionality of the tell() method in the superclass by redefining it in the subclass
- this is termed **method overriding**
- the implementation of a method will be determined dynamically

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