UNIVERSITY
OF COLOGNE

Software & System Engineering / Software Technology
Abteilung Informatik, Department Mathematik / Informatik

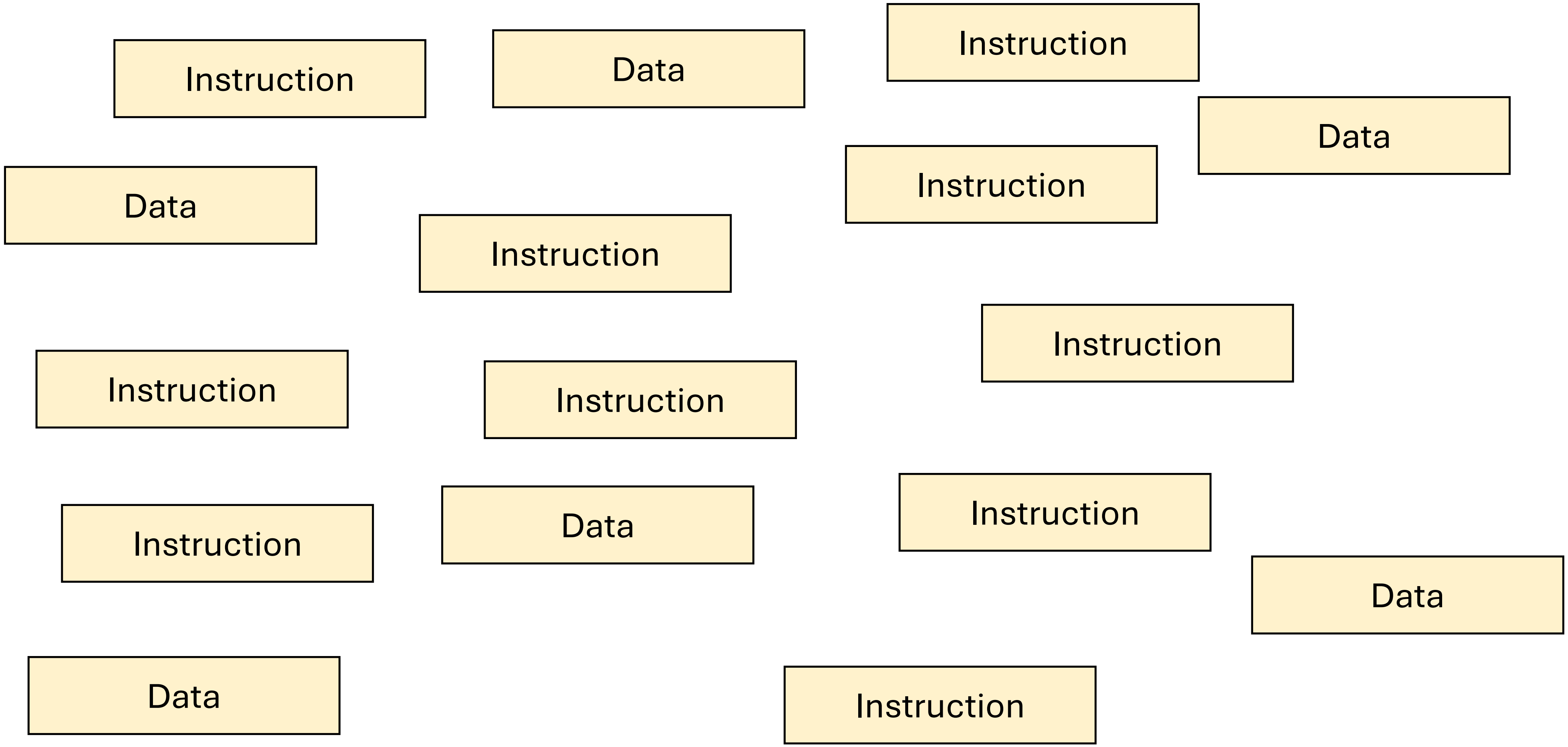
Object-Oriented Software Engineering

Fundamentals of Object-Oriented Programming

Adrian Bajraktari | bajraktari@cs.uni-koeln.de | SoSe 2024 | 13.04.2024

What is Object-Oriented Programming?

Programming Paradigms – Unstructured Programming



Instruction

Data

Instruction

Data

Instruction

Data

Instruction

Instruction

Instruction

Instruction

Instruction

Data

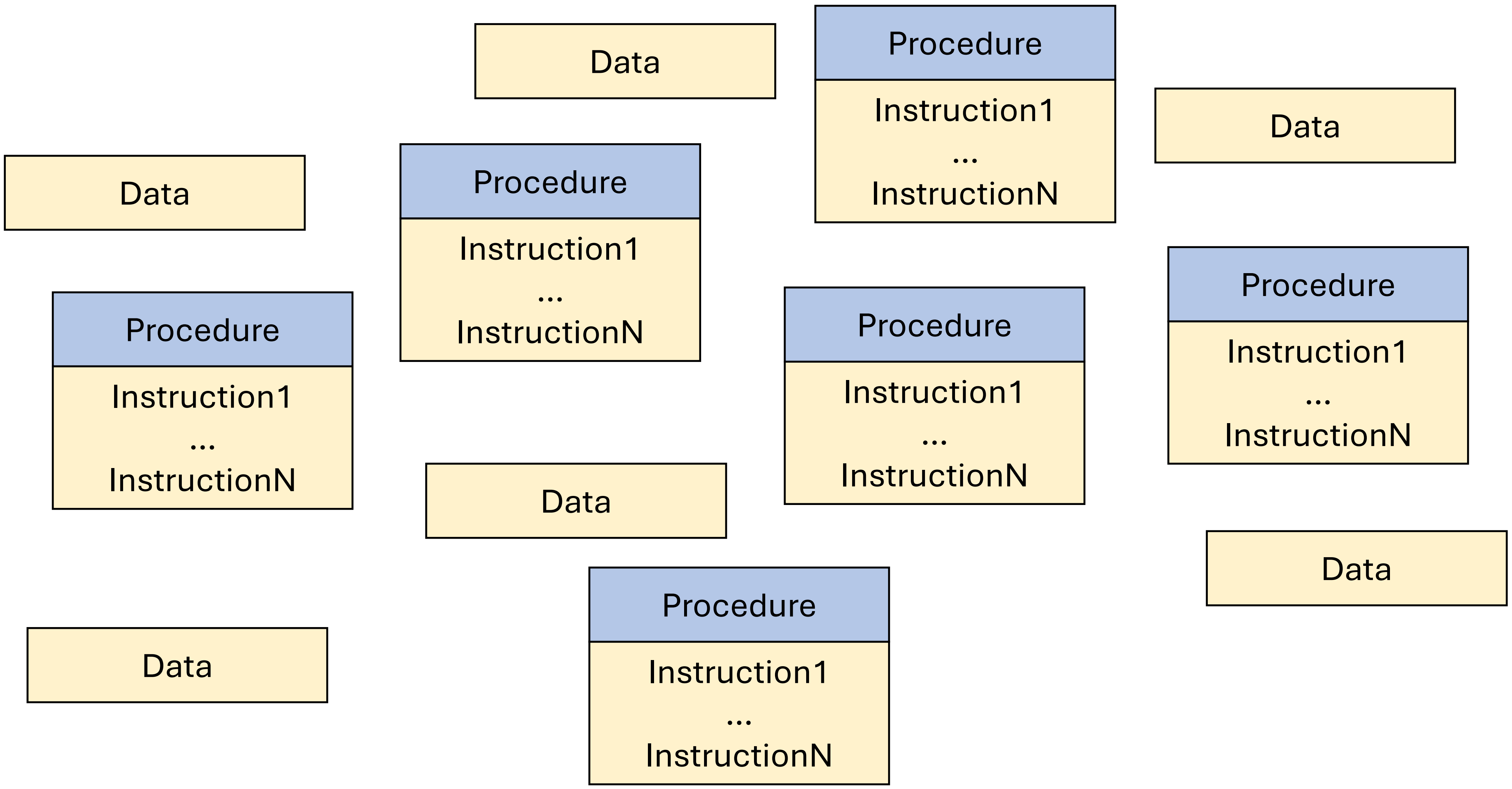
Instruction

Data

Data

Instruction

Programming Paradigms – Procedural Programming



Data

Procedure

Instruction1

…

InstructionN

Data

Data

Procedure

Instruction1

…

InstructionN

Procedure

Instruction1

…

InstructionN

Procedure

Instruction1

…

InstructionN

Procedure

Instruction1

…

InstructionN

Data

Data

Data

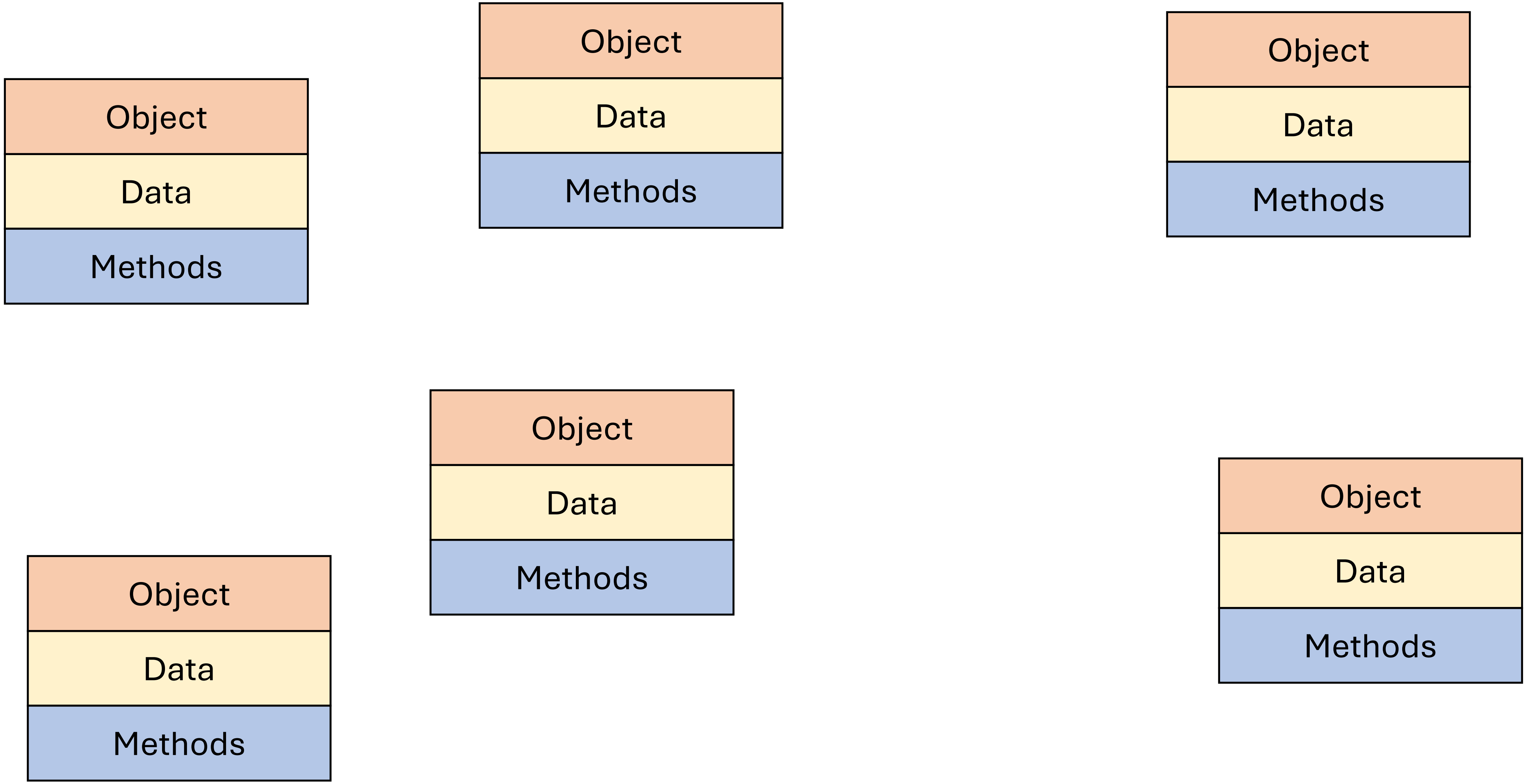
Procedure

Instruction1

…

InstructionN

Programming Paradigms – Object-Oriented Programming



Object
Data
Methods

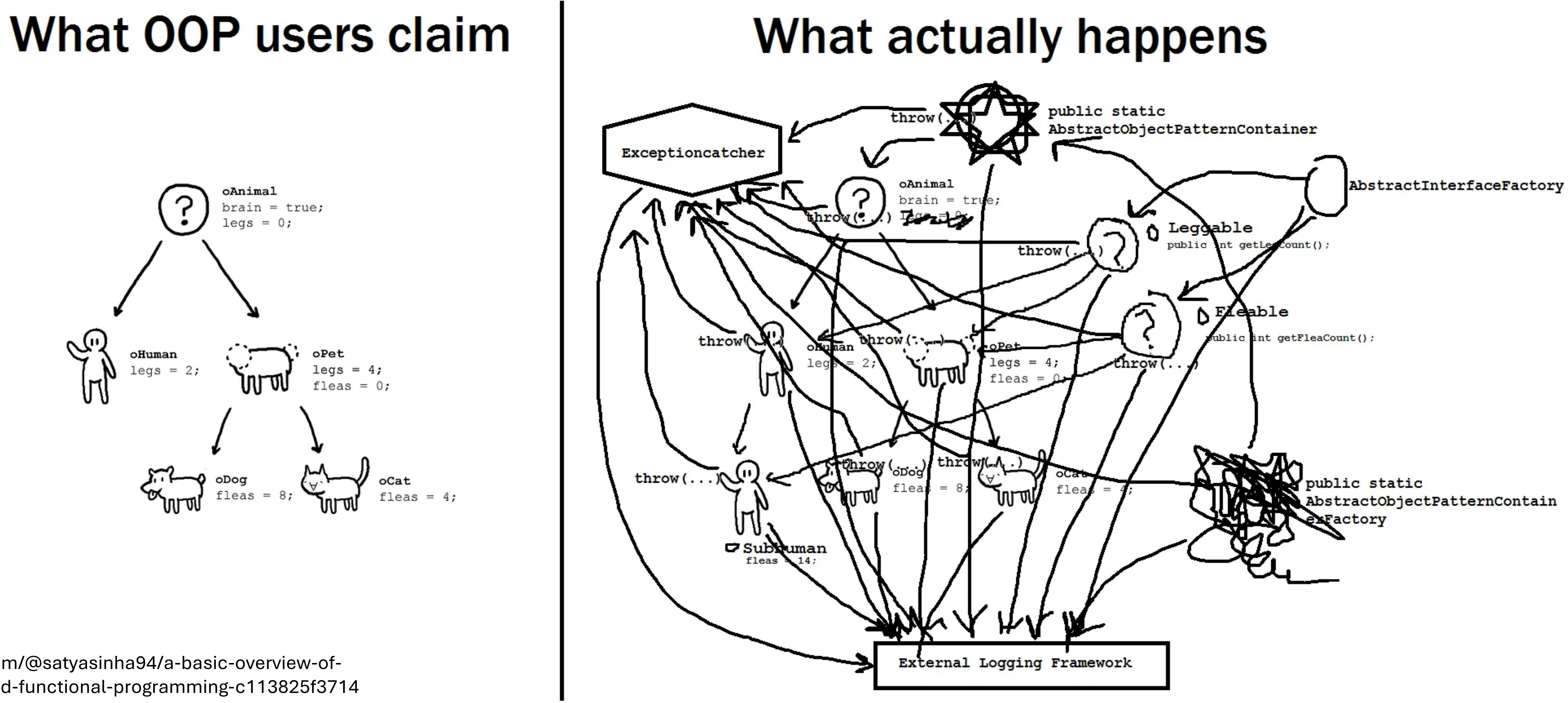
Object
Data
Methods

Object
Data
Methods

Object
Data
Methods

Object
Data
Methods

Object
Data
Methods



https://medium.com/@satyasinha94/a-basic-overview-of-object-oriented-and-functional-programming-c113825f3714

Introduction to **Object-Orientation**

Objects

Object

An object is a dynamically created, encapsulated unit of

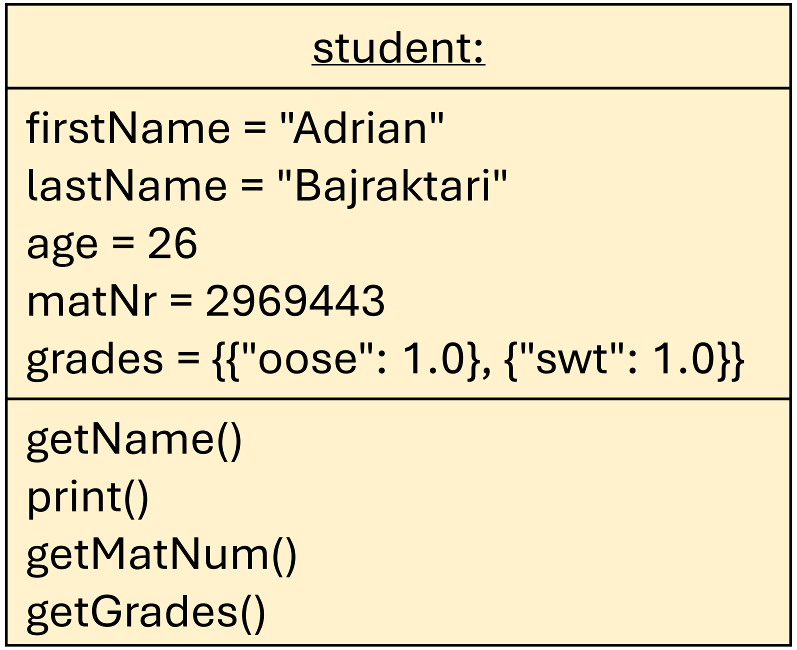
- identity,
- data (values of variables, its state), and
- operations (methods, its behavior).

Encapsulation

- State only modified through interface.
- Other objects only depend on interface.

Interface

- Set of operations provided to a set of clients.
- Different clients may be presented different interfaces.

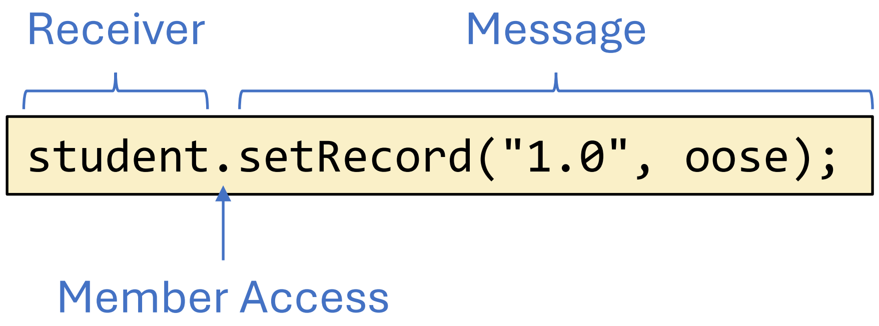


student:

firstName = "Adrian"
lastName = "Bajraktari"
age = 26
matNr = 2969443
grades = {{"oose": 1.0}, {"swt": 1.0}}

getName()
print()
getMatNum()
getGrades()

OOP Terminology

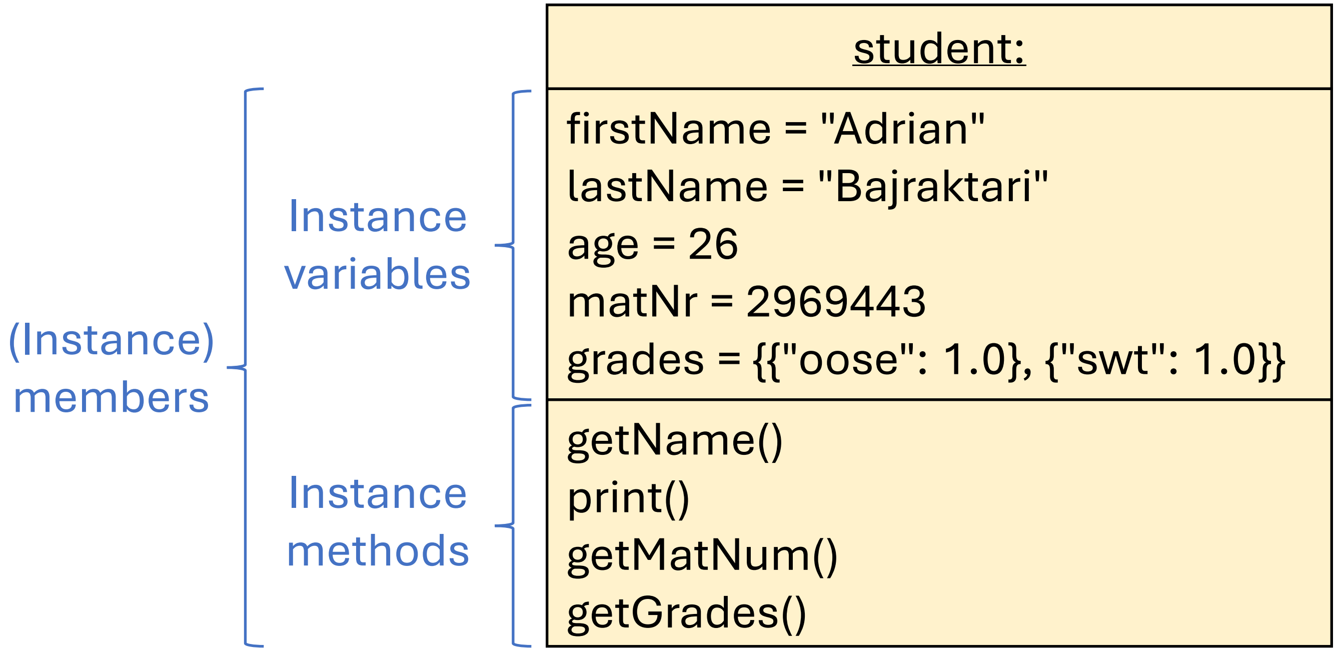


Receiver

Message

```
student.setRecord("1.0", oose);
```

Member Access



(Instance) members

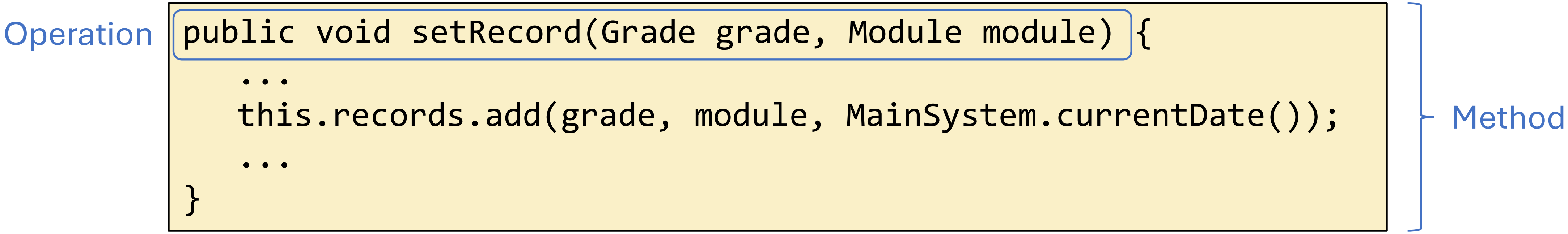
Instance variables

Instance methods

student:

firstName = "Adrian"
lastName = "Bajraktari"
age = 26
matNr = 2969443
grades = {{"oose": 1.0}, {"swt": 1.0}}

getName()
print()
getMatNum()
getGrades()



Operation

```
public void setRecord(Grade grade, Module module) {
    ...
    this.records.add(grade, module, MainSystem.currentDate());
    ...
}
```

Method

Subroutine Terminology

- Procedure: Named block of code that takes arguments as parameters and return (a) value(s). Has or performs side effects.
- Function: A procedure where the output only depends on the input, i.e., it does not have any side effects.
- Method: A procedure that is bound to an object. Methods without side effects do not make sense.
- Operation: Set of methods with same interface. Which method is executed depends on the receiver object.

Examples of Side Effects

```
float calculate(int a, float b) {
   int x = a * a;
   return x + a * b;
}
```

Example of a Function

Result only depends on input, no other actions visible outside the function call.

```
std::string systemInfo = "v0.1";
std::string getSystemInfo() {
   int x = a * a;
   return x + a * b;
}
```

Example of a Procedure

Result only depends on input, no other actions visible outside the function call.

```
class Person { ...
   boolean oldEnough(int limit) {
      return this.age >= limit;
   }
}
```

Example of a Method

Result depends on object state (state read side effect).

```
class Person { ...
   void addGrade(Grade g, Module m) {
      this.grades.add(m, g);
   }
}
```

Example of a Method

No result, but transitive state is modified (state write side effect).

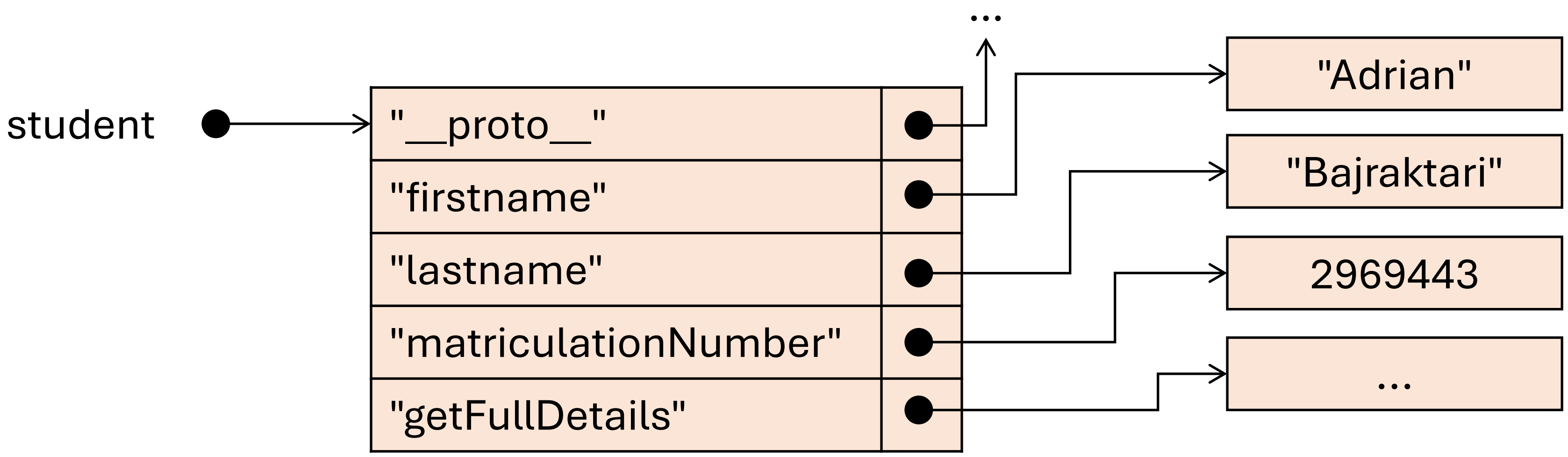
Object-Based Programming

JavaScript Ex-Nihilo Object Creation

```
let student1 = {};                 //empty object
student1.firstname = "Tobi";       //add property to object

let student2 = {                   //object with property
   firstname: "Tobi"
}

let studentAdrian = {
   firstname: "Adrian",
   lastname: "Bajraktari",
   matriculationNumber: 2969443,

   getFullDetails: function() {
      return `${this.firstname} ${this.lastname} has
              matriculation number ${this.matriculationNumber}.`
   }
};
```

Object Implementation

```
let student = {
   firstname: "Adrian",
   lastname: "Bajraktari",
   matriculationNumber: 2969443,
   getFullDetails: function() {
      return `${this.firstname} ${this.lastname} has
               matriculation number ${this.matriculationNumber}.`
   }
};
```



student

"__proto__"

"firstname"

"lastname"

"matriculationNumber"

"getFullDetails"

...

"Adrian"

"Bajraktari"

2969443

...

Self-Reference

Self-Reference

An object can access its own members via a self-reference.

Java, C++, JavaScript:	`this`
Smalltalk, Python:	`self`
Eiffel:	`current`

This self-reference is auto-added by the compiler in most languages (not in Python).

Class-Based Programming

- Explicitly refer to `this` in messages to instance members.
- If no recipient provided, `this` is inserted by the compiler automatically.
- In a message to `o` the compiler inserts `o` for the parameter `this` in the call signature.

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

```
studentAlice.setName("alice");
```

Java

Original Code

↓

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

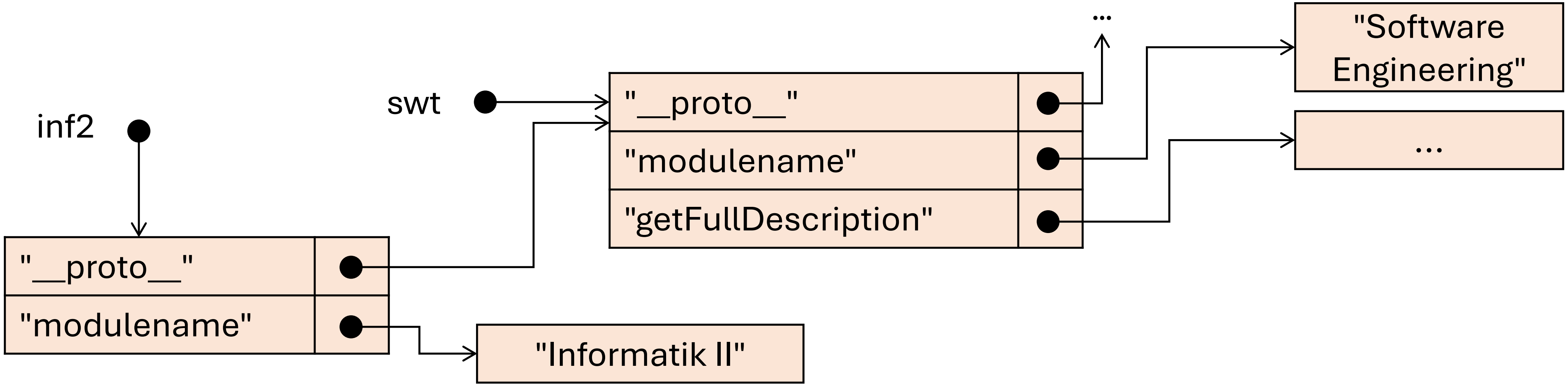
```
studentAlice.setName(studentAlice, "alice");
```

Code with explicit `this`

Object-Based Inheritance: Delegation

```
let swt = {
   modulename: "Software Engineering",
   getFullDescription: function() {
      return `Module name:
         ${this.modulename}, ...`;
   },
   ...
};
```

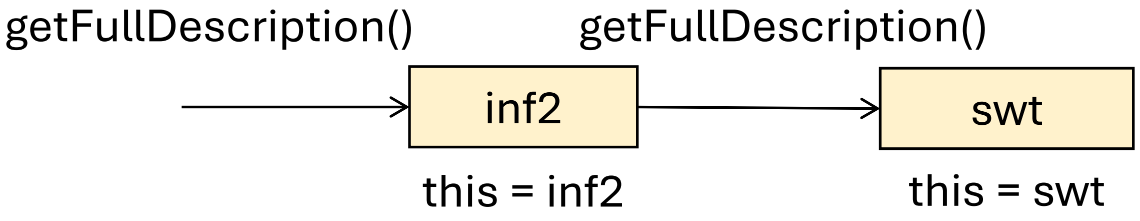
```
let inf2 = {
   __proto__: swt,
   modulename: "Informatik II"
};

console.log(inf2.getFullDescription());
//Prints: "Module name: Informatik II"
console.log(swt.getFullDescription());
//Prints: "Module name: Software Engineering"
```

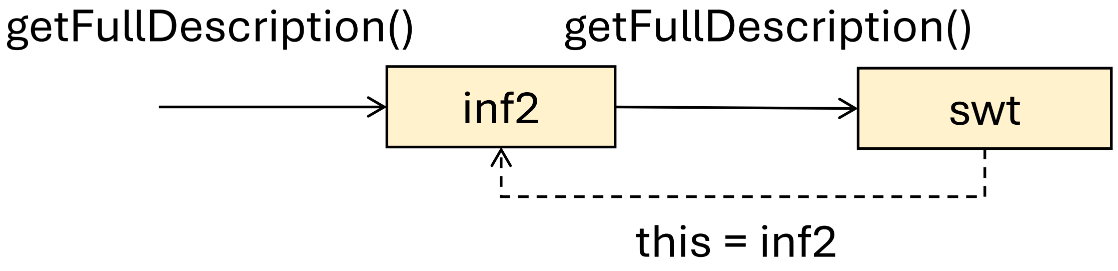


Forwarding vs. Delegation

```
let swt = {
   modulename: "Software Engineering",
   getFullDescription: function() {
      return `Module name:
         ${this.modulename}, ...`;
   },
   ...
};
let inf2 = {
   modulename: "Informatik II",
   getFullDescription: function() {
      return swt.getFullDescription()
   }
};
```

```
let swt = {
   modulename: "Software Engineering",
   getFullDescription: function() {
      return `Module name:
         ${this.modulename}, ...`;
   },
   ...
};
let inf2 = {
   __proto__: swt,
   modulename: "Informatik II"
};
```

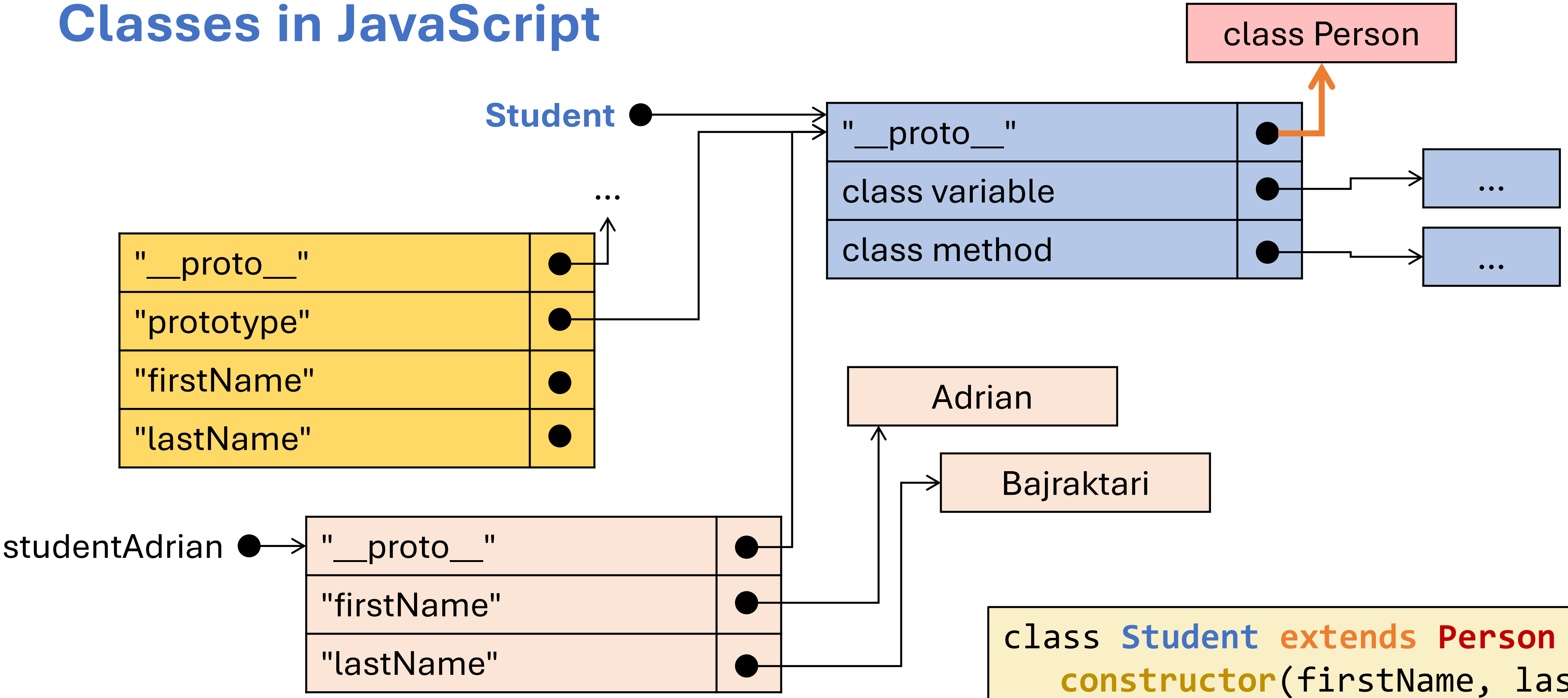




Class-based vs. Object-based Inheritance (Delegation)

Class Inheritance	Delegation
• Class-level: Affects all instances of the class. • Static: fixed on compilation. • Inherited members are part of the object (variables) or its classes' vTable (methods).	• Object-level: Defined for each object individually. • Dynamic: Can be changed at runtime. • "Inherited" members are part of the parent

Classes in JavaScript



```
class Student extends Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
}
```

JS

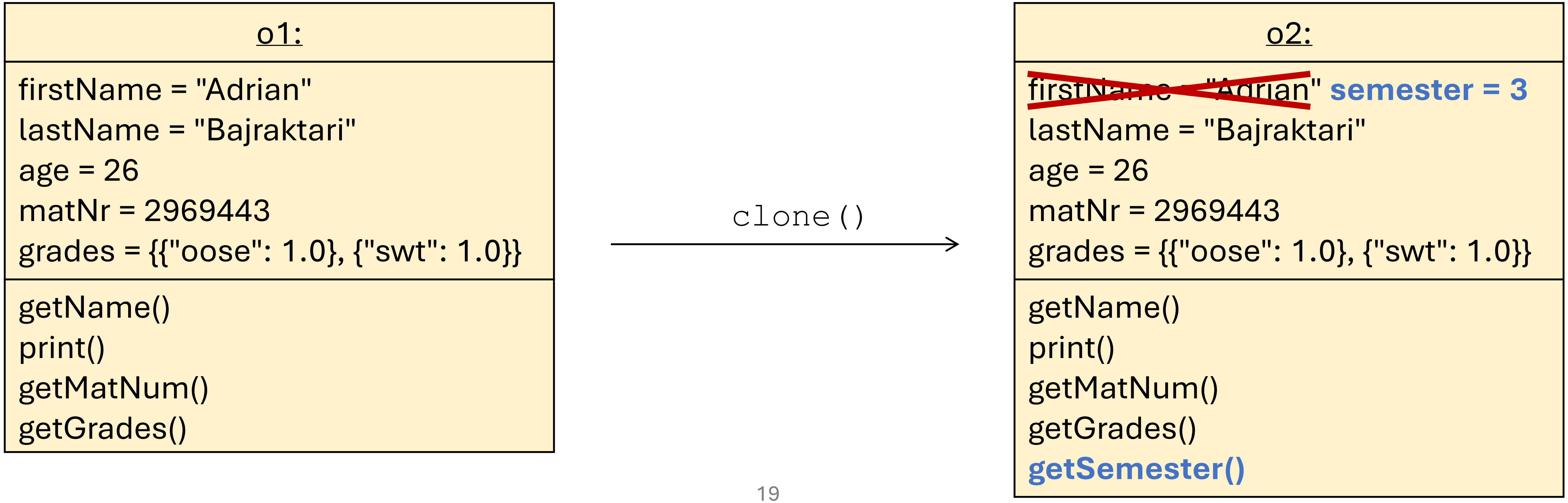
Prototype-Based Programming

Prototype-Based Programming

Objects are created by cloning existing objects and modifying the copies' properties.

Example: Self

```
o1 = {...};
o2 = o1.clone();
delete o2.firstname;
o2.semester = 3;
o2.getSemester = function() {…}
```



o1:

firstName = "Adrian"
lastName = "Bajraktari"
age = 26
matNr = 2969443
grades = {{"oose": 1.0}, {"swt": 1.0}}

getName()
print()
getMatNum()
getGrades()

`clone()`

o2:

~~firstName = "Adrian"~~ **semester = 3**
lastName = "Bajraktari"
age = 26
matNr = 2969443
grades = {{"oose": 1.0}, {"swt": 1.0}}

getName()
print()
getMatNum()
getGrades()
getSemester()

Coding Time!

Class-Based Programming

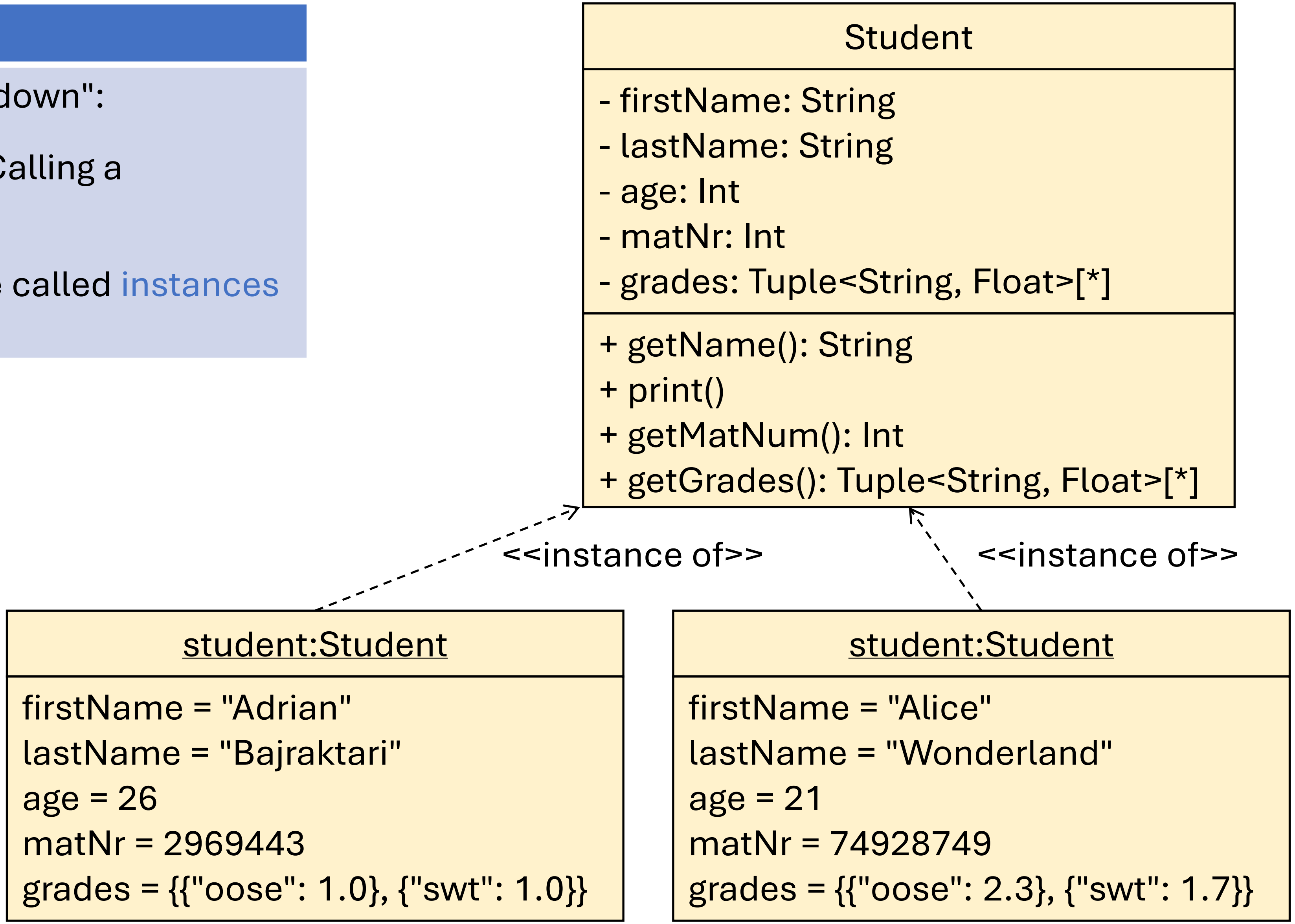
Instantiation – Class-based Object Creation

Class-Based Programming

Specification through "writing it down":

Creation through instantiation: Calling a constructor.

Objects created from a class are called instances of the class.



Student

- firstName: String
- lastName: String
- age: Int
- matNr: Int
- grades: Tuple<String, Float>[*]

+ getName(): String
+ print()
+ getMatNum(): Int
+ getGrades(): Tuple<String, Float>[*]

<<instance of>>

<<instance of>>

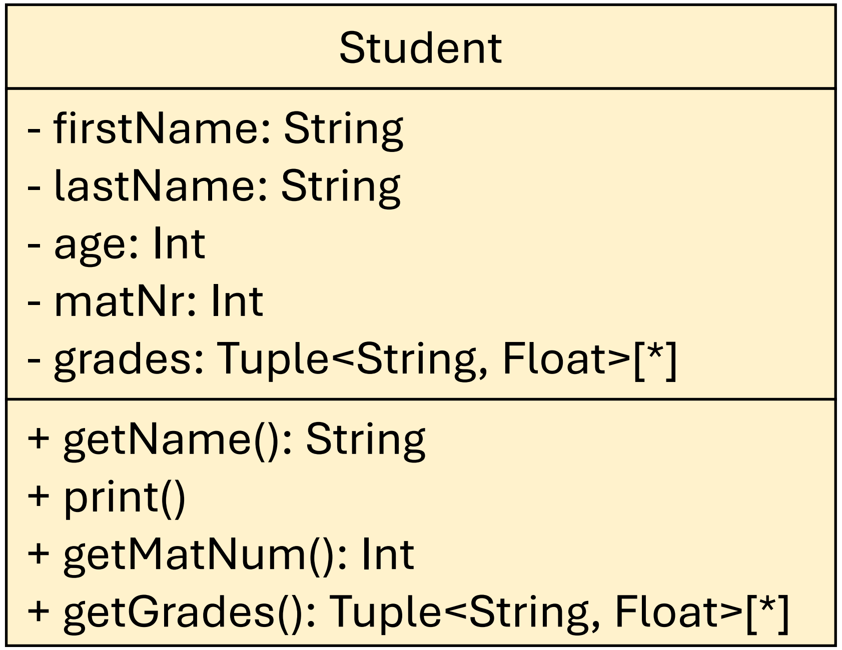
student:Student

firstName = "Adrian"
lastName = "Bajraktari"
age = 26
matNr = 2969443
grades = {{"oose": 1.0}, {"swt": 1.0}}

student:Student

firstName = "Alice"
lastName = "Wonderland"
age = 21
matNr = 74928749
grades = {{"oose": 2.3}, {"swt": 1.7}}

Classes: Example



Student

- firstName: String
- lastName: String
- age: Int
- matNr: Int
- grades: Tuple<String, Float>[*]

+ getName(): String
+ print()
+ getMatNum(): Int
+ getGrades(): Tuple<String, Float>[*]

```
public class Student {
   private String firstName;
   private String lastName;
   private int age;
   private int matNr;

   Student(String firstName, String lastName,
            int age, int matNr) {
      this.age = age;
      this.firstName = firstName;
      this.lastName = lastName;
      this.matNr = matNr;
   }

   public void print() {
      System.out.println(this.firstName + " "
         + this.lastName + ", " + this.age +
         ", " + this.matNr);
   }
   ...
}
```

Why Classes?

Characteristics of Dynamic, Object-Based Programming	Problems arising
• Unlimited flexibility. • Few code due to dynamic. • Less complexity.	• Unlimited flexibility allows for arbitrary complex, intangible object changes. • Error-prone (security, logical, bugs) that are only discovered at runtime. • Non-trivial programs are hard to comprehend.

Class-Based Programming

Classes prescribe a common structure among their objects, allowing optimizations and easier comprehensible code.

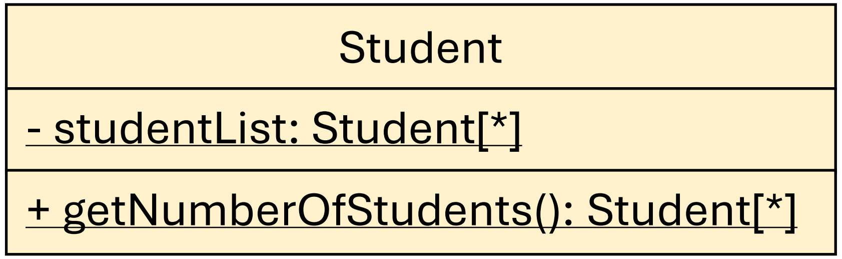
Object-Based Programming with Classes

Many dynamic modern OO languages use a mix form of both paradigms.

Class Methods and Variables

Self-Reference

Class methods work independent of instances. They are called via the class.
Java, C++, JavaScript: `static`
Python: variable defined in class / `@classmethod`



Student

- studentList: Student[*]

+ getNumberOfStudents(): Student[*]

```
public class Student {
    private static Student[] studentList
            = new Student[10];
   ...

    public static int getNumberOfStudents() {
      return studentList.length;
   }
   ...
}
```

```
int numberVar = Student.studentlist.length;
int numberMet = Student.getNumberOfStudents();
```

Constructors and Initialization

Object Creation via Instantiation

Constructor

The constructor is a special class method that returns a reference to a newly created instance of the class.
Constructors encapsulate logic to initialize a new instance.

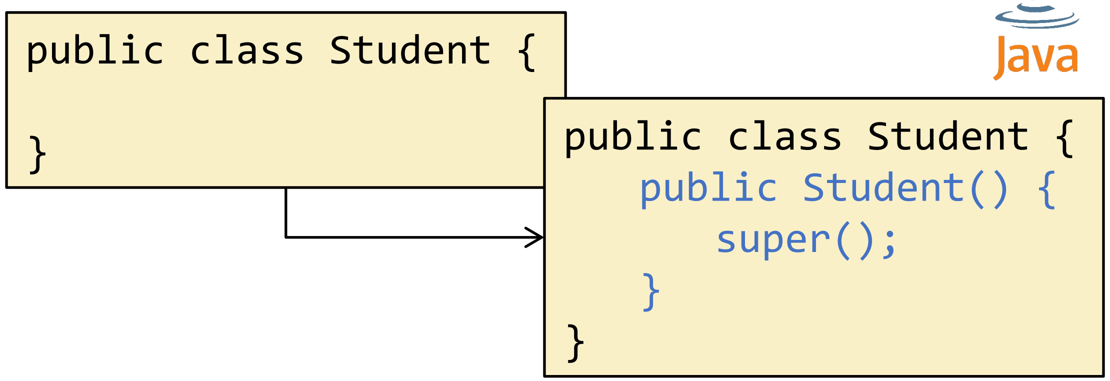
```
public Student(...) {
   //init instance variables
}
```

Overloading Constructors

A class can have more than one constructors in most languages (not in Python and JavaScript).

Implicit Default Constructor

If there is no explicit constructor, the compiler adds a public, parameterless default constructor.



```
public class Student {

}
```

```
public class Student {
   public Student() {
      super();
   }
}
```

About Constructors

Calling other Constructors

As first statement in a constructor, one can

- call a specific constructor of the super class with `super(…)`, or
- call a specific constructor of the same class with `this(…)`.

If none of both is explicitly written, the compiler adds a `super()` call (default constructor of superclass, only if applicable!)

Cost of Object Creation

Creating objects is expensive. To reduce object creation overhead, one could

- Clone existing objects of the same type.
- Use object pools (e.g., multiton or flyweight pattern).
- In case of strings: use StringBuffer or StringBuilder instead of concatenating strings.

Copy Constructors

Copy Constructor

A copy constructor is a constructor that takes an instance of the class as argument and creates a new instance with the same values as the parameter.

Support for Copy Constructors

While only a pattern in other languages, copy constructors are an essential language element in C++. The compiler provides automatically generated copy constructors if none is explicitly given.

```
public Student(Student s) {
    this.name = s.name;
    ...
}
```

Object Identity

Object Identity (OID)

An object's identity is a way for the system to refer to an object.

Equality

```
student1 = new Student();
student2 = new Student();

student1 == student2 //false
```

```
student1 = new Student();
student2 = student1;

student1 == student2 //true
```

value-equality

```
student1 = new Student();
student2 = new Student();

student1.equals(student2)
```

```
boolean equals(Object o) {
   if(o instanceof Student student) {
      return this.firstName == student.firstName
      && this.lastName == student.lastName;
   }
   return false;
}
```

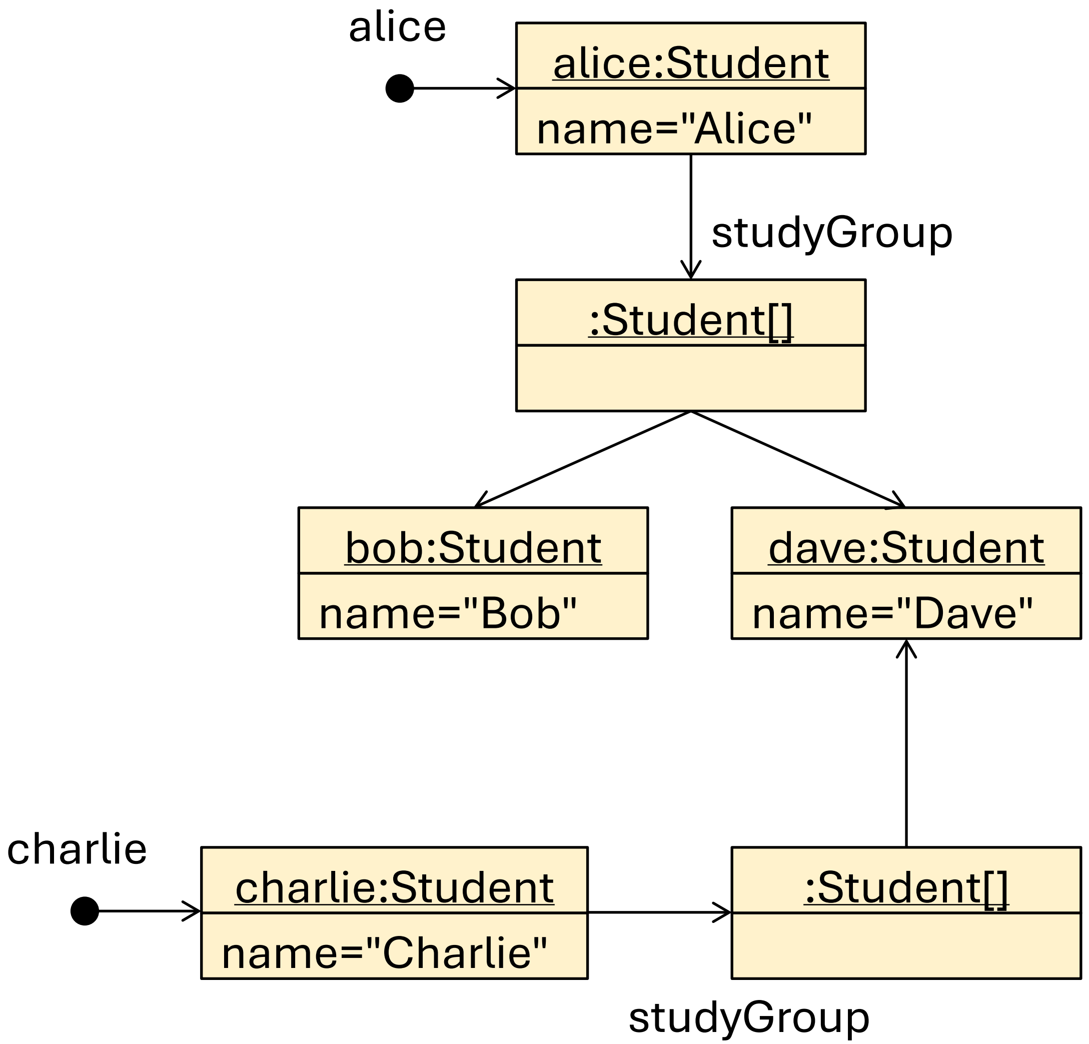
Sharing / Aliasing

Benefit: Sharing.

Danger: Aliasing.

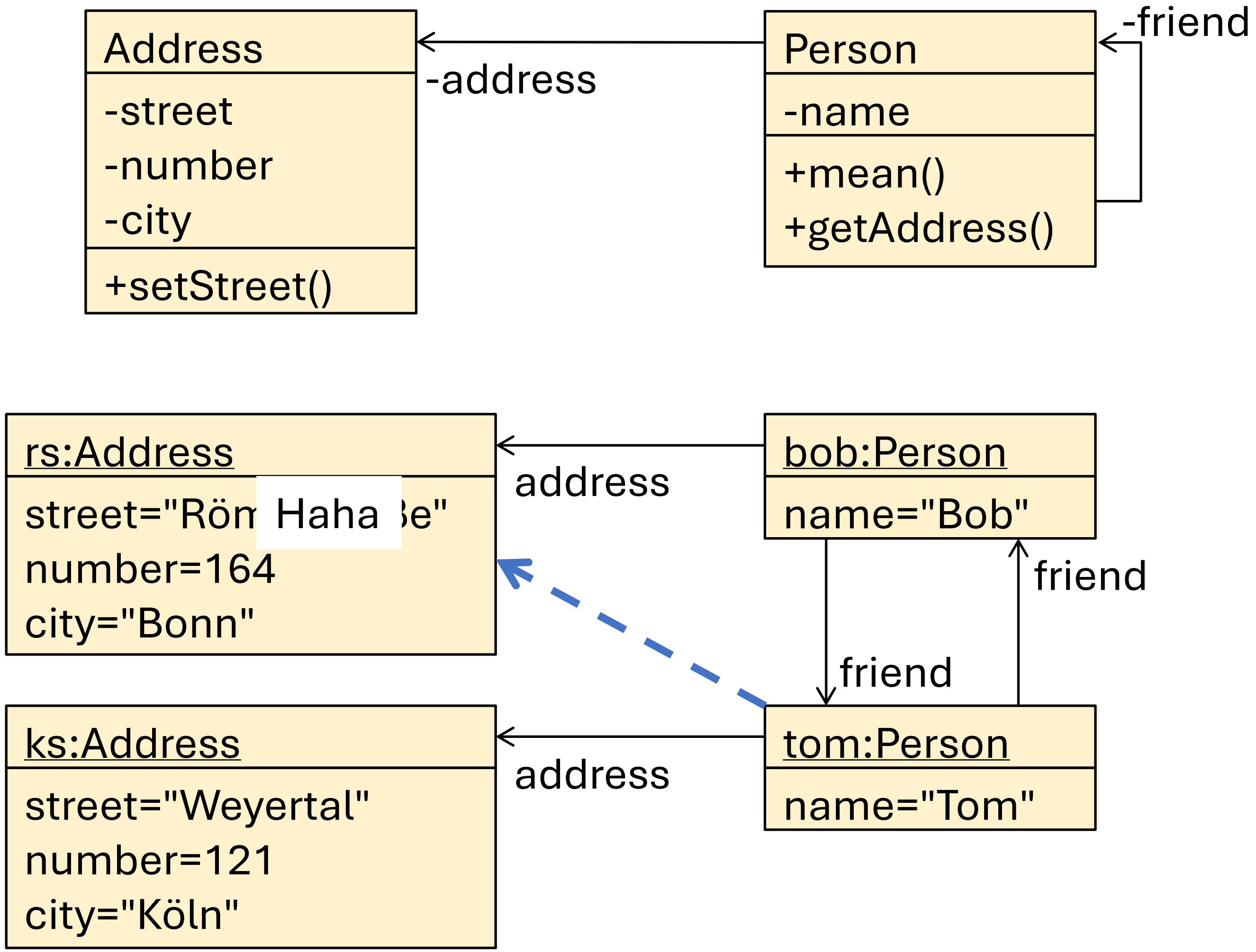
```
public class Student {
   ...
   Student[] studyGroup;
   ...
}
```

```
Student alice = new Student()
```



Example: Person / Address

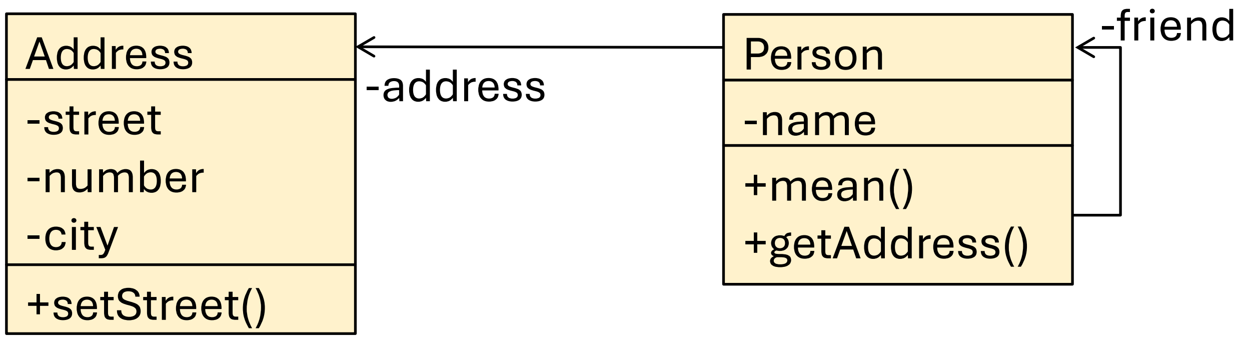
```java
public class Person {
   //private attributes
   private String name;
   private Address address;
   private Person friend;
   ...
   public Address getAddress() {
      return this.address;
   }
   public void mean() {
      Address friendsAddr
         = friend.getAddress();
      friendsAddr.setStreet("Haha");
   }
}
```



Example: Person / Address with Cloning

Person::getAddress():

```
public Address getAddress() {
   Address clone = this.address.clone();
   return clone;
}
```



Address
-street
-number
-city
+setStreet()

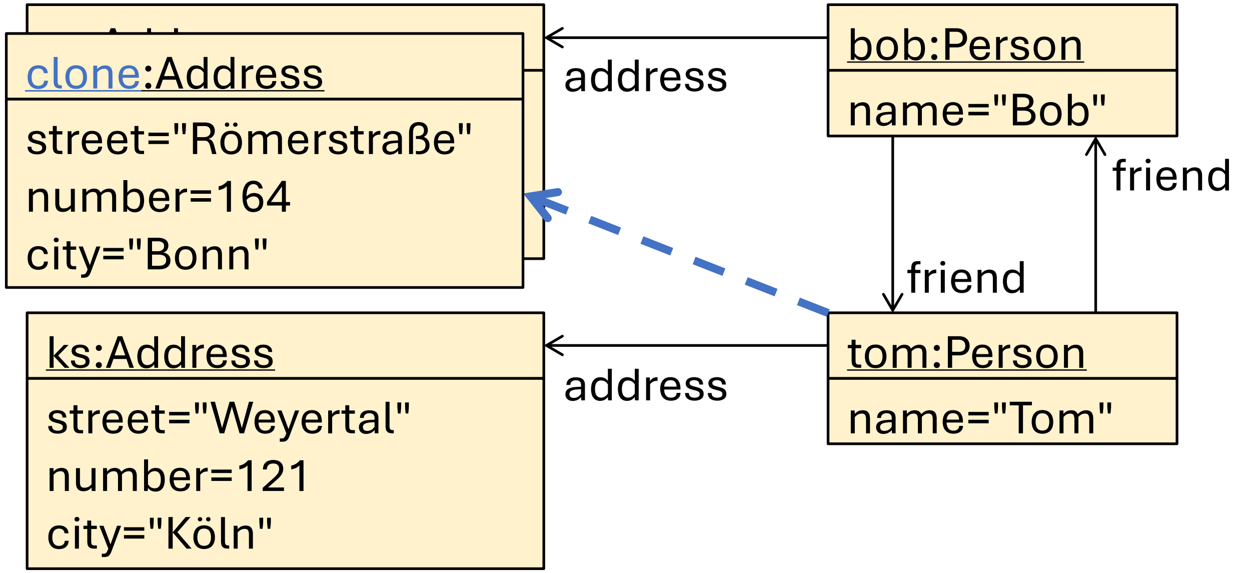
-address

Person
-name
+mean()
+getAddress()

-friend

Address::clone():

```
public Address clone() {
   Address clone = new Address();
   clone.street = this.street;
   ...
   return clone;
}
```



clone:Address
street="Römerstraße"
number=164
city="Bonn"

address

bob:Person
name="Bob"

friend

friend

ks:Address
street="Weyertal"
number=121
city="Köln"

address

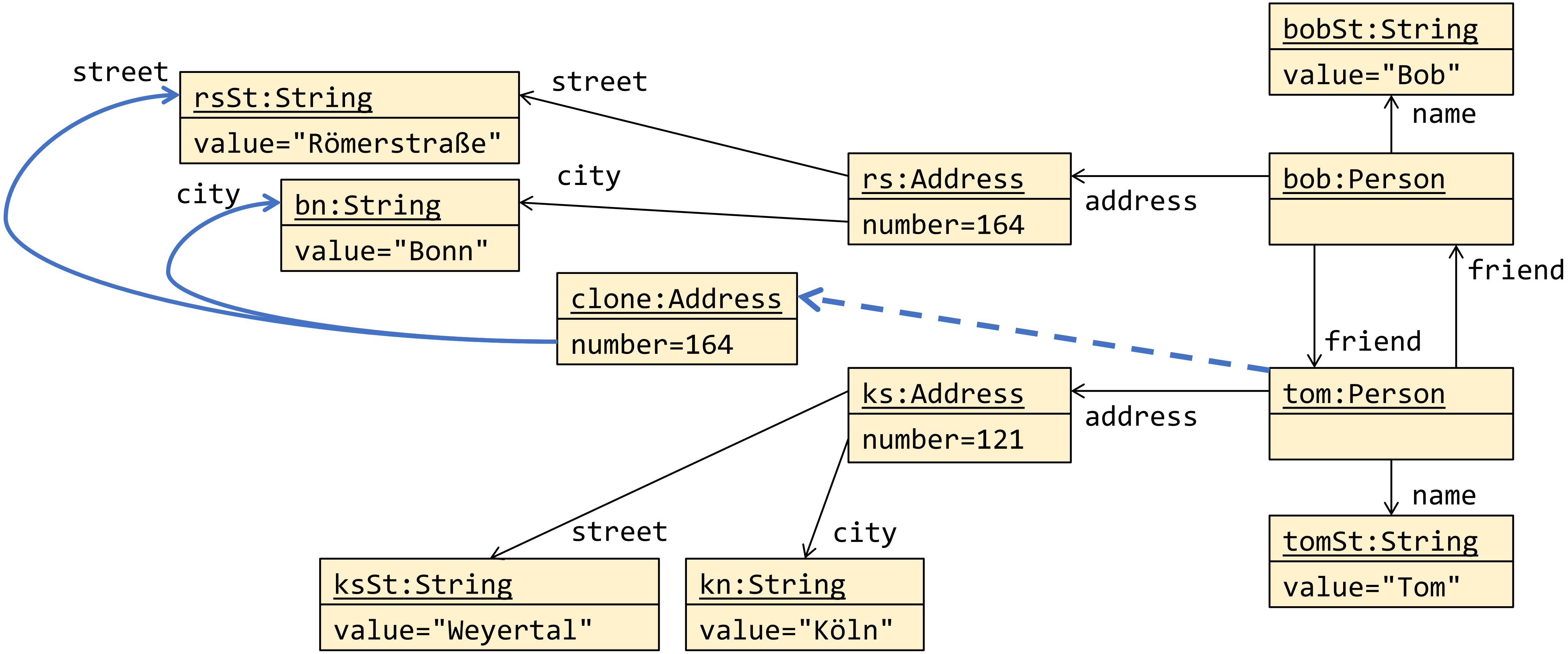
tom:Person
name="Tom"

Now is everything ok?

In this case: yeah.

Example: Person / Address

This is how the object diagram from before looks like with all strings not inlined.



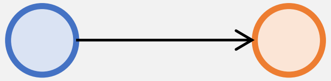
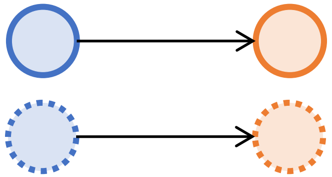
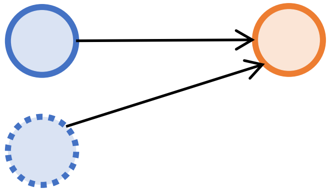
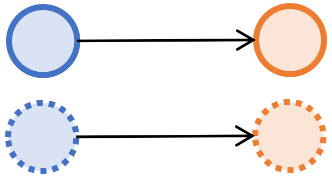
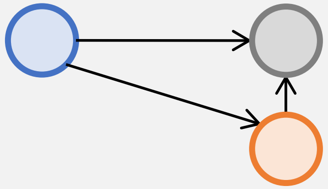
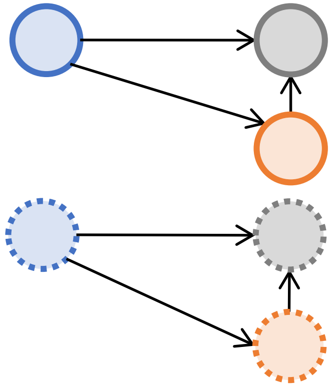
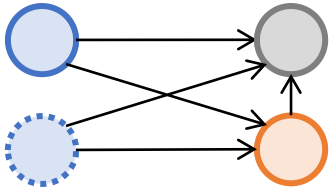
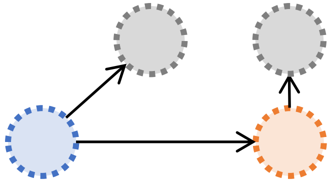
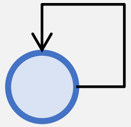
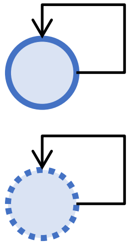
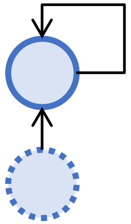
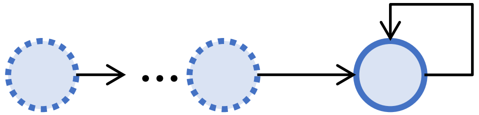
Shallow Cloning and Deep Cloning

```
public Student clone() {
   Student clone = new Student();
   clone.firstname = this.firstname;
   clone.lastname = this.lastname;
   clone.studyGroup = this.studyGroup;

   ...
   return clone;
}
```

```
public Student clone() {
   Student clone = new Student();
   clone.firstname = this.firstname;
   clone.lastname = this.lastname;
   clone.studyGroup =
         this.studyGroup.clone();
   ...
   return clone;
}
```

Cloning: Pitfalls

These cases can be solved with a `Map<Reference of Original, Reference of Clone>` of all copied references: If the original reference is in the map, do not clone again and instead use the associated new reference.

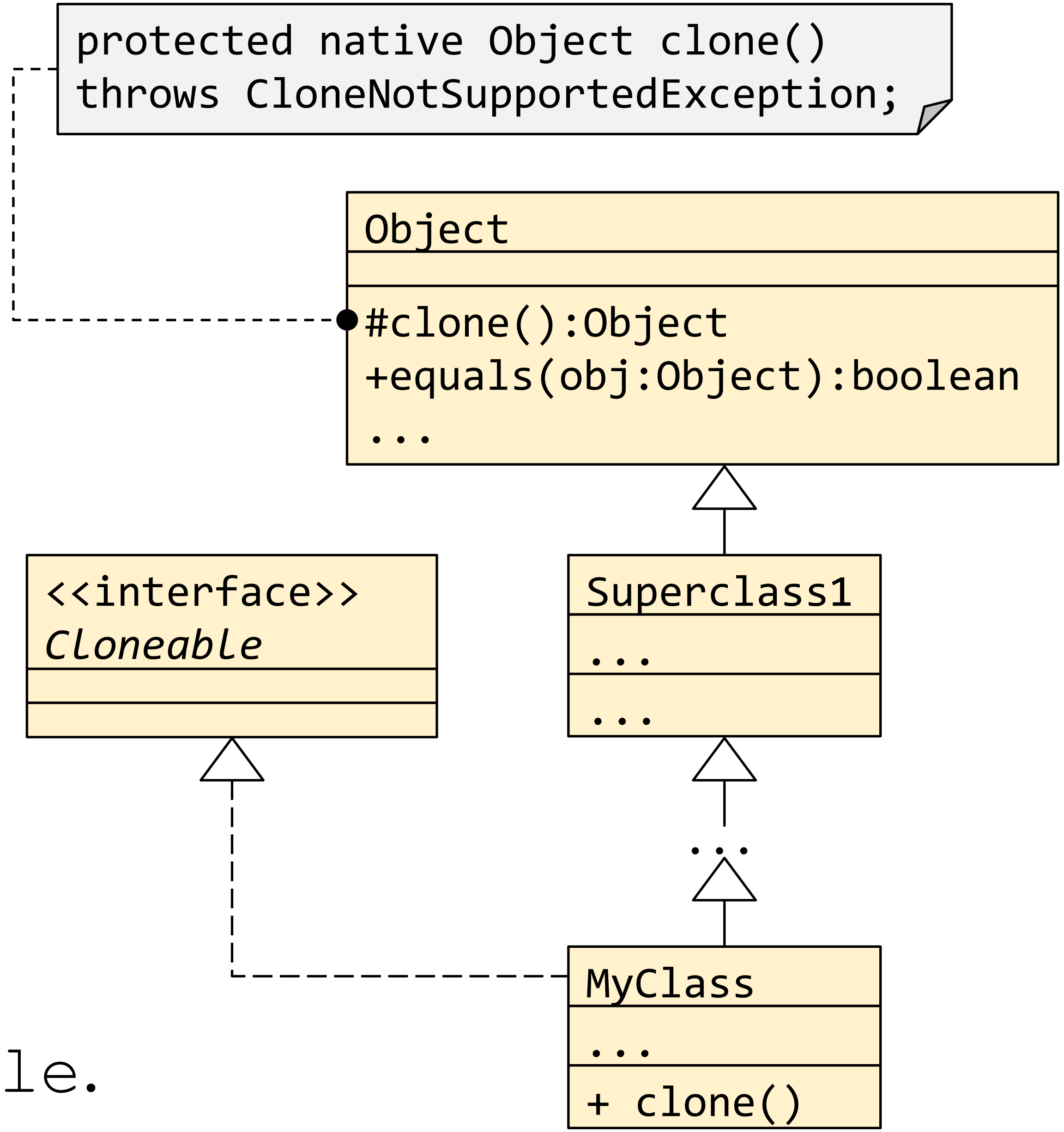
	Expected Result	Shallow Clone	Deep Clone
Simple Case 		 ✗	 ✓
Sharing/Aliasing 		 ✗	 ✗
Cyclic Reference 		 ✗	 ✗

Shallow Cloning in Java

Java provides `Object::clone().`

It is protected and must be overridden as public.

The class must implement `Cloneable`.


protected native Object clone()
throws CloneNotSupportedException;
Object
#clone():Object
+equals(obj:Object):boolean
...
<<interface>>
Cloneable
Superclass1
...
...
...
MyClass
...
+ clone()

Summary

Shallow Cloning	Deep Cloning
• Manual implementation tedious. • Java provides a standard, highly optimized shallow cloning method (→ faster than calling constructors!). • Often times not very useful.	• In Java the only way to prevent other objects to access an object's own transitive state. • Costly: For bigger object structures high space and time costs. • Only for enhanced encapsulation too expensive!

Summary

- Objects = Identity + state + behavior.
- Object-Oriented Programming does not need classes (Object-Based Programming).
- Delegation as object inheritance.
- Classes provide common structure for objects.
- Sharing/aliasing and cloning as issues of object identity.