



UNIVERSITY
OF COLOGNE

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

Object-Oriented Software Engineering

Modern and Advanced Programming

Adrian Bajraktari | bajraktari@cs.uni-koeln.de | SoSe 2024 | 29.04.2024

**This chapter has many different small
concepts from different languages.
Focus on the concepts, not on the syntax.**



Collection Handling

Python Lists, Tuples, Sets and Dictionary

List

Lists are the "arrays" of Python. Lists are indexed, ordered, and modifiable collections. They grow dynamically, i.e., adding and removing works without manual resizing.

```
studentList = ["Alice", "Bob", "Charly"]  
alice = studentList[0]
```

Tuples

Tuples are indexed, ordered, immutable collections.

```
studentTuple = ("Alice", "Bob", "Charly")  
alice = studentTuple[0]  
_, bob, _ = studentTuple
```

Set

Sets are unordered, immutable collections that do not allow duplicates..

```
studentSet = {"Alice", "Bob", "Charly"}  
studentSet.add("Dave")
```

Dictionaries

Dictionaries are ordered, mutable collections that do not allow duplicates.

```
studentDict = { "name": "Alice",  
                "matNr": 934969 }  
name = studentDict["name"]
```



Multiple Return Values

Multiple Return Values

Some languages allow procedures to return more than one value.

Multiple Return Values in Python

In Python, returning more than one value creates a tuple of all values. These can be stored in separate variables via tuple unpacking.

```
class Student:
    def __init__(self, firstName, lastName):
        self.firstName = firstName
        self.lastName = lastName

    def getName(self):
        return self.firstName, self.lastName

student1 = Student("Alice", "Wonderland")

first, last = student1.getName()
```



Enumerate

Enumerating through Keys and Values of a Collection

In Python, we can iterate through a collection's keys and values simultaneously by using `enumerate()`.

```
lectures = [ "oose", "swt", "ti" ]  
  
for key, value in enumerate(lectures):  
    print(key, ": ", value)
```



```
grades = { "oose": 1.0, "swt": 1.0, "ti": 3.0 }  
  
for key, value in grades.items():  
    print(key, ": ", value)
```



List Comprehension

Enumerating through Keys and Values of a Collection

In Python, we can iterate through a collection's keys and values simultaneously by using `enumerate()`.

```
grades = { "oose": 1.0, "swt": 1.0, "ti": 3.0, "ra": 5.0}

modifiedRecord = [str(grade) + ": " + str(course)
                  for course, grade in grades.items()
                  if grade <= 4.0]

print(modifiedRecord)
```



List Access and Slicing

Slicing in Python and NumPy

Python provides a convenient way to access ranges of elements in a list using slices. NumPy extends these capabilities to multi-dimensional arrays.

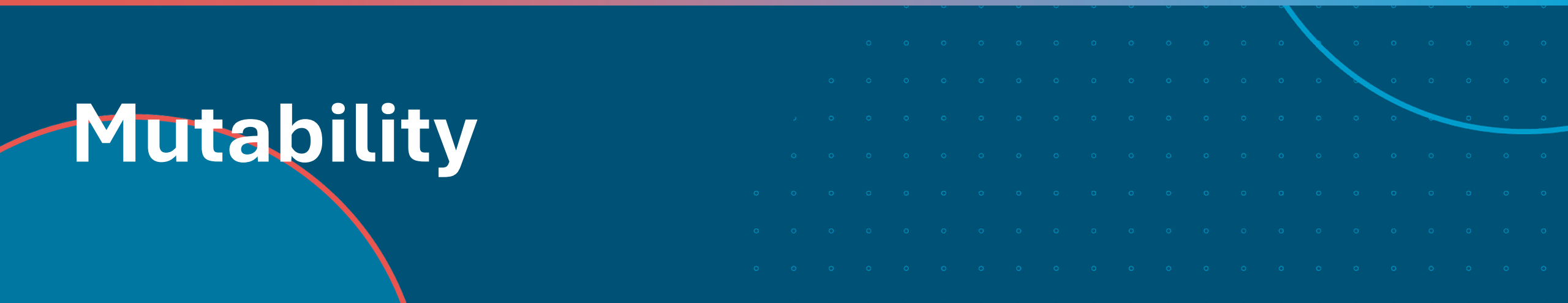
```
lectures = [ "luds", "math1", "math2", "prog1", "prog2", "oose", "swt", "ti", "m1" ]

lectures[2]           # math2
lectures[4:]          # ["prog2", "oose", "swt", "ti", "m1" ]
lectures[:4]          # [ "luds", "math1", "math2", "prog1" ]
lectures[2:5]         # [ "math2", "prog1", "prog2" ]
lectures[1:6:2]       # [ "math1", "prog1", "oose" ]
lectures[-1]          # m1
lectures[0:-4]        # [ "luds", "math1", "math2", "prog1", "prog2" ]
lectures[::-1]        # Reverses the list
```



```
import numpy as np
list = np.array([[ 1, 2, 3, 4 ], [ 5, 6, 7, 8 ], [ 9, 10, 11, 12 ]])
print(list[:, 1:])    # prints all rows, but only columns >= 1
```





Mutability

Mutability

Declaring Constants

Constants are variables explicitly declared as constant, i.e., once assigned, they can not be re-assigned.

- Java: `final`
- C, C++, JavaScript: `const`
- TypeScript: `const` (checked at runtime) and `readonly` (checked at compile time)
- Kotlin, Scala: Differentiate between `var` (mutable) and `val` (immutable).
- C#: `readonly`
- D: transitive mutability via `immutable`
- Python: override `__setattr__` and `__delattr__` operators.

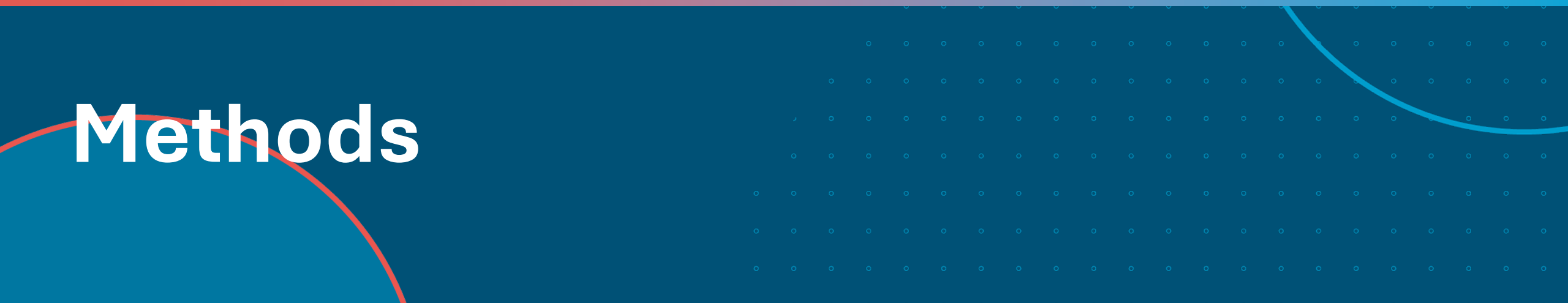
```
val a = 42
```

```
a = 6
```

```
var b = 42
```

```
b = 6
```





Methods

Operator Overloading in Python

Operator Overloading

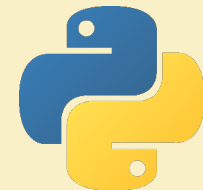
In languages that allow operator overloading, the behavior for some or all operators provided by the language can be replaced with custom behavior.

```
class A:
    def __init__(self, a):
        self.a = a

    def __add__(self, o):
        return self.a + o.a

ob1 = A(1)
ob2 = A(2)
ob3 = A("Object-Oriented")
ob4 = A(" Software Engineering")

print(ob1 + ob2) # Compiled to print(ob1.__add__(ob2))
print(ob3 + ob4) # Compiled to print(ob3.__add__(ob4))
```



Inlining

Procedure Inlining

The code of procedures declared as `inline` replaces every call, i.e., the code is inserted at each location the procedure is called.

```
inline float calculateFinalGrade() {  
    float total = 0;  
    for(int i = 0; i < count; i++) {  
        total += this.grades.get(i).grade * this.grades.get(i).ects;  
    }  
    return total / 180  
}  
  
void printTranscript() {  
    ...//print all courses  
    this.calculateFinalGrade();  
}
```




Explicit and Automatic Getter / Setter

Explicit Getter and Setter

In JavaScript, an object or class can define explicit **get** and **set** methods that bind a method to a property.

Get methods are called when reading from the property and set methods are called when assigning a value to the property.


```
const record = {  
  log: ['oose', 'ti', 'swt'],  
  get latest() {  
    return this.log[this.log.length - 1];  
  }  
};  
  
console.log(record.latest);  
// Expected output: "swt"
```



Automatic Getter and Setter

C# allows to define getter and setter similar to JavaScript. On top, if we define an attribute with non-explicit get and/or set, the compiler will generate getter and setter.

```
public class Student {  
  public string Name { get; set; }  
}
```



Extension Methods / Extension Properties

Extensions

In Kotlin, we can add new methods or attributes to existing types as if we defined them as such. The classes and objects are not really altered. Instead, we define static methods that can be used via member access notation.

For extension properties: Since they are no real members of the object, they can not be initialized directly. Instead, we can only define them in terms of getters and setters.

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {  
    val tmp = this[index1]  
    this[index1] = this[index2]  
    this[index2] = tmp  
}
```

```
val <T> List<T>.lastIndex: Int  
    get() = size - 1
```

Traits

Method extensions are similar to traits, an approach appearing in more and more modern languages, most prominently in Rust.



The null Problem

C.A.R. Hoare, satirically called the inventor of the null reference, introduced the concept of null references in ALGOL W in 1965. He calls it his billion-dollar mistake.

Preventing Dereferencing null

Most modern languages have some mechanic to elegantly prevent accessing null references. These are either language-level concepts (Kotlin, Scala,...) or standard library facilities.



Dealing with null

Nullability

Nullable and Non-Nullable Expressions

Kotlin provides a language feature to indicate whether an expression can be null, and to skip subsequent member accesses.

```
if(bob != null && department != null
    && head != null) {
    print(bob.department.head.name)
}
```



```
print(bob?.department?.head?.name) //null if any part is null
print(bob?.department?.head?.name ?: "no head found")
bob!!.department!!.head!!.name //NPE if violated
```

```
class Student constructor
    (var firstName: String) {...}

var student = Student(null);
```

```
class Student constructor
    (var firstName: String?) {...}

var student = Student(null);
```



Optional

Java's Optional

Java provides a standard library class that wraps around nullable values and provides methods to execute code if the value is not null.

```
if(pName != null) {  
    System.out.println(pName.length());  
}
```



```
Optional<String> opt = Optional.of(pName);  
opt.ifPresent(name -> System.out.println(name.length()));
```





Special Classes and Typing

Enumeration

Enum

An enumeration (enum) is a class that provides all its possible instances out-of-the-box. Thus, enums can not be instantiated.

Enums are used when the state space of a type is very limited, i.e., declaring all possible instances statically is feasible. This is usually used for states, days of the week, months, ...

Like normal classes, enums can

- have a (private) constructor,
- have instance variables and methods,
- have class variables and methods,
- be generic,
- Implement interfaces.

But they can neither inherit from classes (they inherit from `java.lang.Enum`) nor be instantiated from outside (all constructors are implicitly private).

```
public enum ExamState {  
    NOT_REGISTERED,  
    REGISTERED,  
    ACCEPTED,  
    IN_PROGRESS,  
    IN_CORRECTION,  
    NOT_PASSED,  
    PASSED  
}
```

```
private ExamState examState =  
    ExamState.ACCEPTED;
```

```
public enum ExamState {  
    NOT_REGISTERED("You have not..."),  
    REGISTERED("You are registered ..."),  
    ...  
    private String message;  
  
    ExamState(String message) {  
        this.message = message;  
    }  
    public String getMessage() {...}  
}
```



Nested Classes

Nested Class

Classes can be [nested](#).

Static Inner Class

A static inner class works just like a normal class, only that the reference looks like:
`OuterClass.InnerClass`.

```
public class Map {  
    ...  
    public static class Entry {  
        ...  
    }  
}
```

```
Map.Entry e = new Map.Entry();
```

Non-Static Inner Class

To instantiate non-static inner classes, we first need an instance of the outer class on which the inner class instance is created.

Non-static inner classes have a compiler generated reference to their "outer" object. They can access any members of the outer instance.

```
public class Map {  
    ...  
    public class Entry {  
        ...  
    }  
}
```

```
Map m = new Map();  
Map.Entry e = m.new Entry();
```



Local Class

Local Class

A local class is a class definition inside a method which is only usable within this scope. They have access to any member of the enclosing object, but also to variables of the enclosing method, *if* they are **effectively final**.

```
public void method() {  
    class Local {  
        ...  
    }  
    ...  
}
```

Anonymous Class

Anonymous classes are local classes with no name, intended for a one-time object creation. They need an interface as static type.

```
public interface Person {  
    String getName();  
    int getId();  
    void setLocation(...);  
}
```

```
Person person = new Person() {  
    String name;  
    int id;  
    Location location;  
  
    String getName() {...}  
    int getId() {...}  
    void setLocation(...) {...}  
}
```



Records and Value Types

Record Classes

Record classes model plain data aggregates. They specify in their header a description. Constructor, toString(), hashCode(), equals(), and accessors are usually auto-generated.

Attributes are final, accessors have the same name. The constructor has the same signature as the header.

You can still define them all explicitly, except for instance variables.

Record classes are final.

```
record Student(String firstName,  
               String lastName, int age) { }
```

Value Types

Value types are class-like types whose values are immutable objects with no identity.

In principal, the idea is to treat these objects like primitives, including call-by-value and storing inline.



Sealed Classes

Sealed Classes

Sealed classes do not allow any class to inherit that is not explicitly permitted.
Subclasses must either be sealed as well, final or explicitly non-sealed.

```
public sealed class ScientificEmployee  
    ... permits PhD, Professor, PostDoc { }
```

Non-Sealed Classes

Non-sealed classes revoke the sealing imposed by their parent class.

```
public non-sealed class PostDoc { }
```



Smart Casting / Pattern Matching

Smart Cast

In statically typed languages, a regular issue is to handle an expression in the context of a type that it may or may not have at runtime.

The approach: Type check the expression at runtime and cast it.

```
if(expression instanceof Student) {  
    Student student = (Student) expression;  
    ...student.getName()...  
}
```



```
if(expression instanceof Student student) {  
    ...student.getName()...  
}
```



```
val anInt: Int? = a as? Int //null if violated
```



Manifested vs. Inferred Types (Type Inference)

JavaScript vs. TypeScript

TypeScript is a statically typed version of JavaScript. The code on the right runs fine in JavaScript but produces a compile error in TypeScript.

In contrast to Java which uses **manifest types** (you must declare static types in source code), TypeScript allows you to use manifest types or **type inference** (the compiler derives the type of an object via its structure and values of its properties).

```
let student = {  
  name: "Alice",  
  matNr: 4242  
};  
  
student.name = 123
```



Nominal and Structural Subtyping in TypeScript

TypeScript supports **nominal subtyping** (you declare one class as a subclass of another) and **structural subtyping** (the compiler infers if the object type is a subtype of the required type).

```
interface Person {  
  getName(): String;  
}  
  
class Student { ...  
  getName() { ... }  
}  
  
function print(p: Person) {  
  console.log(p.getName());  
}  
  
print(new Student("Alice"));
```



Drawbacks

- Random substitutions possible, but unlikely.
- Relation between types harder to trace.

Benefits

- Type checking at compile time.
- Autocompletion in IDEs.
- No overhead for devs.
- Regular JavaScript code can be checked.

Union and Intersection

Union Type

Union types allow any instance of any component type T_1, \dots, T_n . any instance of any of T_i is allowed.

Since we allow any instance of these types, we can only access members that are common between all types T_i , i.e., we can only call the intersection of all members.

```
function getLength(obj: string | string[]) {  
    return obj.length;  
}
```

TS

Intersection Type

Intersection types restrict the set of possible instances to those which are instance of all component types T_1, \dots, T_n .

Since we know that the instance has all members of all types, we can access all members of every T_i , i.e. the union of all members.

```
function print(person: Student & Employee) {  
    console.log(person.getMatNr()  
        + ", " + person.getName() + ", "  
        + person.getSSID() + ": "  
        + person.getPosition());  
}
```

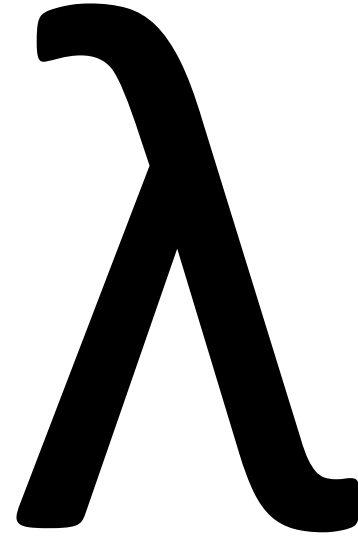
TS

Functional Programming

Functional Programming is a programming paradigm famously known for side-effect-free programs. Example languages: Haskell, Lisp, Scheme, ML, Clojure, Scala (partly), Erlang, F#.

Functional Programming in Java

Java introduced some functional-ish language concepts with [lambda expressions](#) and [streams](#).



Functional Aspects in Java

Functional Interfaces and Lambda Expressions

Functional Interface

A functional interface is an interface that only declares one operation.

```
interface FunctionalInterface {  
    returnType method(ParaType1 para1,...);  
}
```

Lambda Expression

A lambda expression is an expression that realizes a functional interface. It works like a method object. We can store and pass around its definition.

```
FunctionalInterface fi = (para1,...) -> {  
    statement1; ...  
    ... return returnExpression;  
};  
fi.method(arg1, ...);
```

On the Notation of Lambdas

- When exactly one parameter, parentheses can be omitted.
`(para1) -> {...}` → `para1 -> {...}`
- When the statement block consists of one statement, `{ }` can be omitted. The value of the statement becomes the return value, so no explicit return is necessary.
`(para1, ...) -> {statement}` → `(para1, ...) -> statement`
- Instead of calling another method as sole purpose of a lambda, use method references instead:
`... lambda = (para1, ...) -> rec.method(...);` → `... lambda = rec::method;`



Generic Functional Interfaces

Generic Functional Interfaces

To loosen the restrictions on types, we can define generic functional interfaces.

This way, lambda expressions only depend on the number of parameters and the position of type parameters.

```
interface FunctionalInterface<T, U, V> {  
    V method(T para1, U para2);  
}
```

```
FunctionalInterface<Integer, Integer, CourseMember>  
    fi = (courseId, matNr) -> {  
        Student s = Student.findStudent(matNr);  
        return createCourseMember(matNr, courseId);  
    };
```

Predefined Functional Interfaces

The Java Standard Library provides a set of commonly used generic functional interfaces.

- `Predicate<T>`: Takes an argument and returns a `Boolean`.
- `Supplier<T>`: Takes no arguments and returns a value.
- `Consumer<T>`: Takes an argument and returns nothing.
- `Function<T, R>`: Takes a `T`-argument and returns an `R`.

...and many more in `java.base/java.util.function`



Streams

Streams

Streams are a functional-inspired approach to processing collections. Streams provide methods that operate on the data and return a stream object. These often take a lambda expression as argument. Usually, we use stream-returning methods in a method call chain, i.e.,
`Stream.of(...).filter(...).map(...).reduce(...)`

```
Stream.of(element1, ..., elementN); //Creating a stream of single elements
Stream.of(array);                    //Creating a stream from arrays
collection.stream();                //Creating a stream from a collection
```

• <code>toArray()</code>	
• <code>reduce(...):</code>	Reduce stream to one element.
• <code>collect(...):</code>	Create collection of given type of stream elements.
• <code>min(...)/max(...):</code>	Find minimum/maximum of elements.
• <code>map(...)/mapToXXX(...):</code>	Apply provided mapping function to each element of the stream.
• <code>limit(...):</code>	Cap size of stream.
• <code>forEach(...):</code>	Perform an action for each element of the stream.
• <code>flatMap(...)/flatMapToXXX(...):</code>	Like map, but replaces original.
• <code>findAny()/findFirst():</code>	Find elements.
• <code>filter(...):</code>	Returns stream consisting of elements that match criteria.
• <code>distinct():</code>	Returns stream with no duplications, according to <code>Object::equals</code> .
• <code>count():</code>	Returns the number of elements in the stream.
• <code>anyMatch(...)/allMatch(...):</code>	Returns whether any/all elements match the criteria.



Coding Time!



Reflection

Reflection

Reflection

Reflective Languages represent language concepts within the language itself.
Biggest example: Lisp.

Reflective *object-oriented* languages represent all language concepts as objects.
Biggest examples: Smalltalk, Self.

Java is **partly reflective**.

Reflection in Java

Classes are objects. They are instances of `Class`.

Each object knows its class: `getClass()`.

Methods and attributes can be represented as objects.
`Class` has corresponding methods for that.

Java Tutorial for Reflection

<https://docs.oracle.com/javase/tutorial/reflect/index.html>

Drawbacks

- Bad Performance (Optimizations do not apply to reflective code).
- Security risk.
- No static checking.

Benefits

- Extensibility: Dynamic instantiation of unknown classes.
- Good for Debugger (need access to private members)
- Good for testing (might need access to private members)

Examples

Find class of an object

```
Object o;  
  
System.out.println("This object has type " + o.getClass().getName());
```

For `o = "abc"`: `This object has type java.lang.String`

Find whether type of an object is subtype of type of another object:

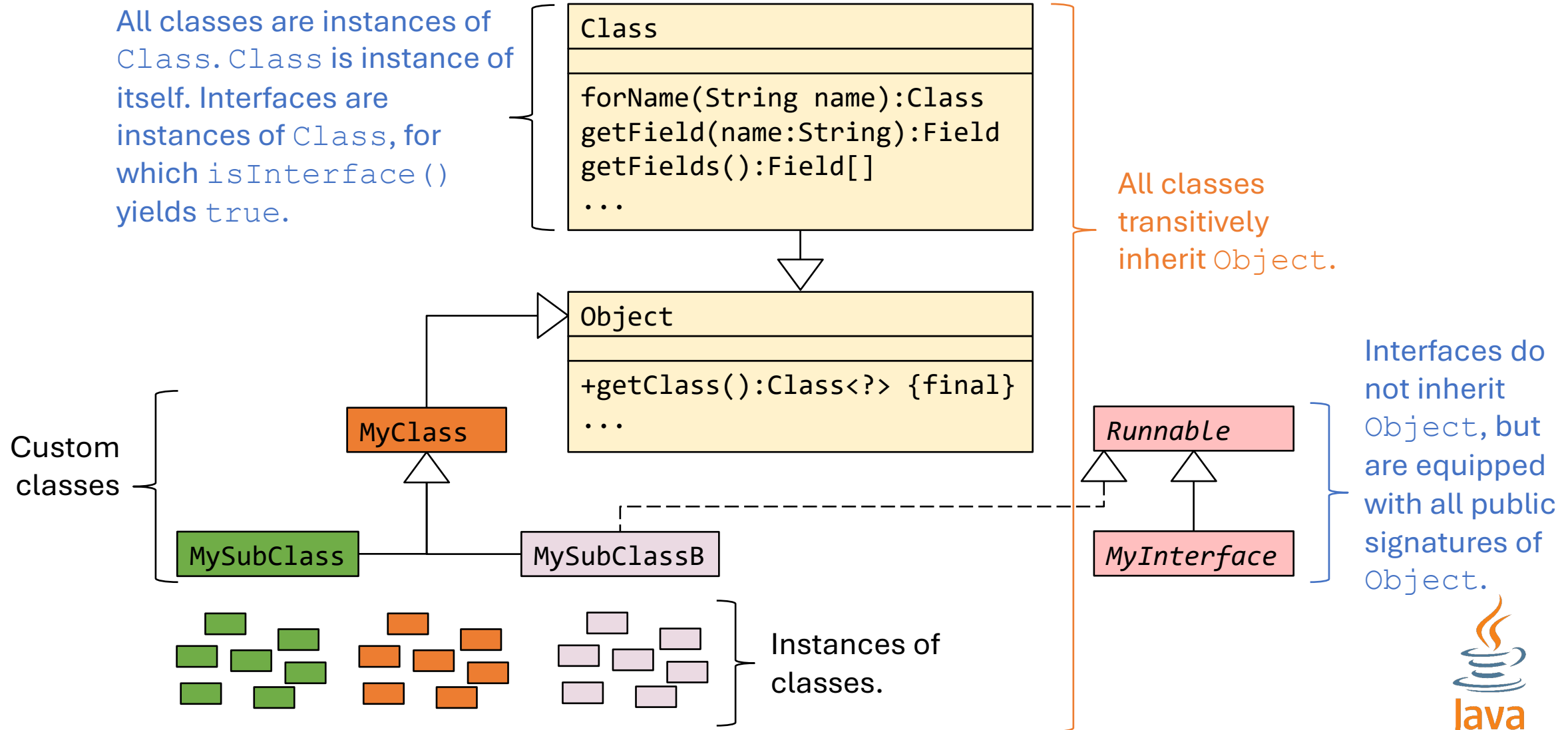
```
Object o1, o2;  
Class c1 = o1.getClass();  
Class c2 = o2.getClass();  
  
if (c1.isAssignableFrom(c2)) System.out.println("Is assignable");  
else System.out.println("Is not assignable");
```

For `o1 = "abc"` and `o2 = "cde"`: `Is assignable`



Reflective Architecture of Java

All classes are instances of `Class`. `Class` is instance of itself. Interfaces are instances of `Class`, for which `isInterface()` yields `true`.



java.lang.Class

Allows access to:

- Classes via their name.
- Members of classes.
- Supertypes of Classes.

Creation of instances via
`newInstance()`.

- ...

Class
<pre>forName(String name):Class getField(name:String):Field getFields():Field[] getMethod(name:String,Class[] paramTypes):Method getMethods():Method[] getInterfaces():Class[] getSuperclass():Class getDeclared...(): ... //Fields, Methods,Constructors, //Classes newInstance() : Object ...</pre>

Allows to circumvent visibility! (attention!)

You can get the `Class` Object of an object via `.getClass()`



Summary (so far)

Summary of the Programming Lectures

The first 5 lectures 02 – 06 were dedicated to programming and programming concepts.

- 02: Basic OOP concepts, distinction in object-based and class-based programming.
- 03: Types, subtyping, inheritance, overloading, overriding and implementation of class-based OOP.
- 04: Basics of memory management, how memory and "objects" are handled in C, garbage collection vs. destructors.
- 05: Modular programming, generics and parametric polymorphism.
- 06: Multiple different programming concepts reappearing throughout popular programming languages.

What is Object-Oriented Programming?

At its core: Objects (identity, state, behavior) as language concept + dynamic binding.

On second level: Classes, subtyping (class-based inheritance or delegation), constructors & destructors.

Not bound to OOP or completely orthogonal: Interfaces, Types, Visibility, Generics, Packages/Modules, Garbage Collection, anything seen today.