

Software Engineering

Programming, Paradigms and Languages

There will be slides that explicitly ask you to ask questions.

Nonetheless, you are welcome to ask questions anytime in between as well.

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

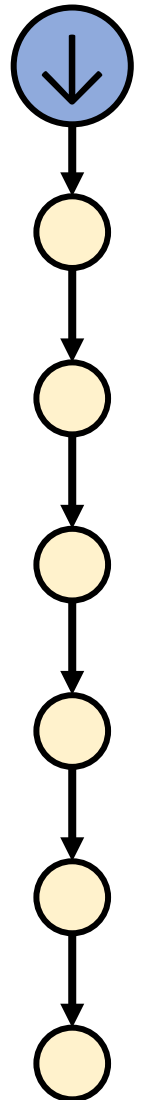
Cover advanced object-oriented principles.

How to model OO systems (UML) and map models to code.

Object-oriented modeling techniques.

Design patterns as means to realize OO concepts (I).

Design patterns as means to realize OO concepts (II).



Last Lecture

- Component diagrams.
- Architectural patterns: Layers, publish-subscribe, microservices, model-view-controller.
- API.

Aims of this Lecture

- Overview of programming paradigms and languages.
- Revisit imperative concepts.
- Introduce "under the hood" principles.
- Revisit object-oriented programming basics.



Paradigms and Languages

Programming Paradigms and Languages

In general, we can differentiate between two major approaches to program construction:

Imperative Programming

The focus lies on the "how", i.e., the developer provides specific step-by-step instructions how to achieve a desired result.

Declarative Programming

The focus lies on the "what", i.e., the developer describes characteristics of the result, and the machine finds a way to achieve this result.



```
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
bubble_sort([],Sorted) :-
    Sorted = [].
bubble_sort([X], Sorted) :-
    Sorted = [X].
bubble_sort(Terms, Sorted) :-
    bubble(Terms, Terms), Sorted = Terms ;
    bubble(Terms, Partials), bubble_sort(Partials, Sorted).

bubble([], Bubbled) :- Bubbled = [].
bubble([X], Bubbled) :- Bubbled = [X].
bubble([X,Y|Terms], [Y|Bubbled]) :-
    Y < X, bubble([X|Terms], Bubbled).
bubble([X,Y|Terms], [X|Bubbled]) :-
    X <= Y, bubble([Y|Terms], Bubbled).
```



SWI Prolog

```
bubblesort :: Ord a => [a] -> [a]
bubblesort list = case sort list of
    sortedList | sortedList == list -> list
                | otherwise -> bubblesort sortedList

Where
    sort (x1:x2:xs) | x1 > x2    = x2 : sort (x1:xs)
                    | otherwise = x1 : sort (x2:xs)
    sort xs                = xs
```



Programming Languages

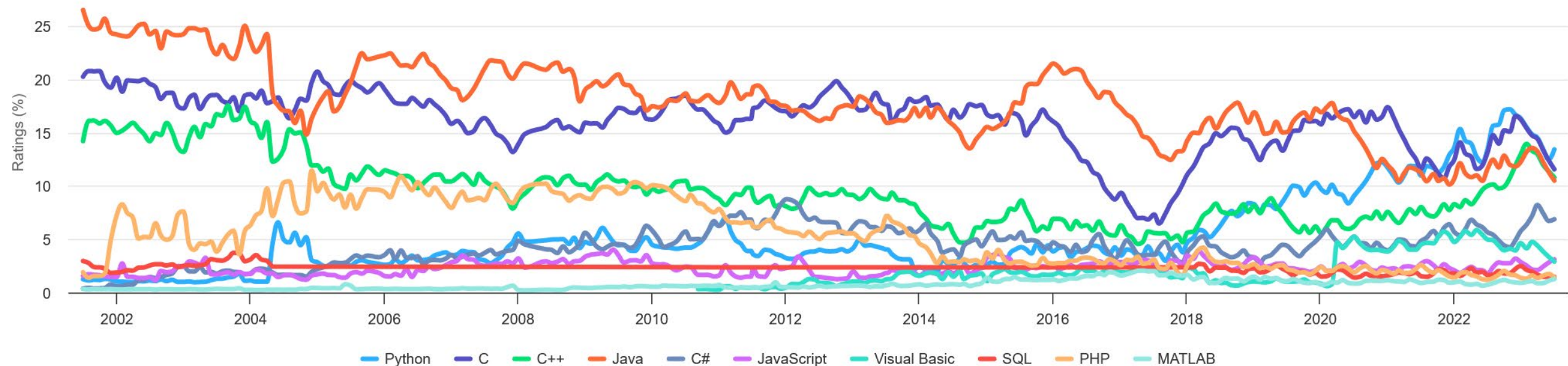
Until today, there are thousands of languages and only a fraction of them are used for more than ten years.









Languages that were in use for more than 25 years include Ada, C, C++, COBOL, Java, Objective C, PL/I, SQL, Visual Basic, ...









Brief History





- 1949: assembly language
- 1954: first high-level language FORTRAN (Formula Translator) by IBM
- 1959: functional language Lisp
- 1967: first object-oriented lang. Simula
- 1972: logical language Prolog
- 1972: procedural language C
- 1973: declarative domain-specific lang. SQL
- 1985: C++ as object-oriented extension of C
- 1990: functional language Haskell
- 1991: multi-paradigm language Python
- 1995: scripting language JavaScript
- 1996: primarily object-oriented language Java
- 2002: C# by Microsoft
- 2009: Go by Google
- 2014: Swift by Apple
- 2015: Rust by Mozilla Research

Programming Language Popularity (TIOBE Index)



	Programming Language		Ratings	Change
1		Python	13.42%	-0.01%
2		C	11.56%	-1.57%
3		C++	10.80%	+0.79%
4		Java	10.50%	-1.09%
5		C#	6.87%	+1.21%
6		JavaScript	3.11%	+1.34%
7		Visual Basic	2.90%	-2.07%
8		SQL	1.48%	-0.16%

9		PHP	1.41%	+0.21%
10		MATLAB	1.26%	+0.53%
11		Fortran	1.25%	+0.49%
12		Scratch	1.07%	+0.35%
13		Go	1.07%	-0.07%
14		Assembly language	1.01%	-0.64%
15		Delphi/Object Pascal	0.98%	-0.08%
16		Ruby	0.91%	-0.08%

17		Rust	0.89%	+0.47%
18		Swift	0.88%	-0.39%
19		R	0.87%	+0.11%
20		COBOL	0.86%	+0.33%

Choice of Programming Languages

Desired Properties [Ludewig and Lichter]	Criteria in Practice
<ul style="list-style-type: none">• Modular implementation• Separation of interfaces and implementations• Type system: statically/dynamically typed languages• Readable syntax (FORTRAN vs. ALGOL60)• Automatic pointer management (C vs. Java)• Exception handling	<ul style="list-style-type: none">• Language required by the company or customer?• Existing infrastructure?• Domain-specific languages available?• Language known/liked by developers?• Available libraries?• Available tool support?• Language popularity?• What may change in the future?

Choosing the right language involves many aspects and decisions, but it is usually best to go with a language that is neither dead nor brand new and covers many use cases in the development.

Programming Language Efficiency

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Questions?

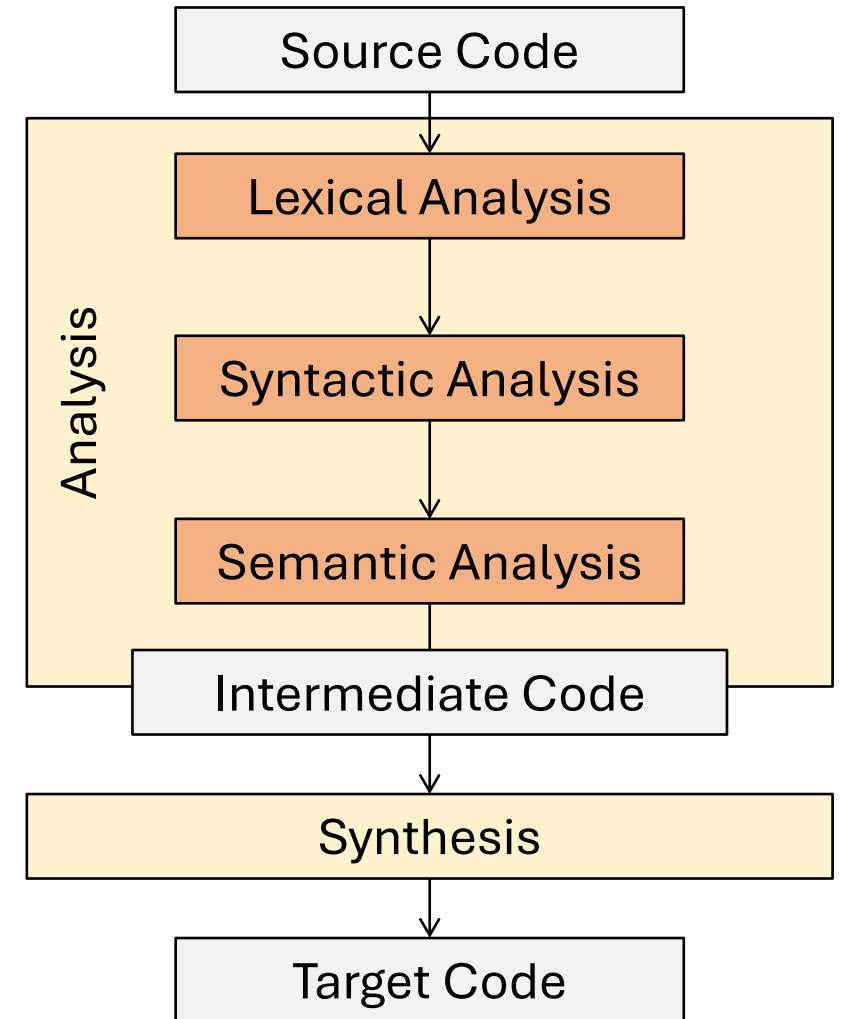


Executable Programs

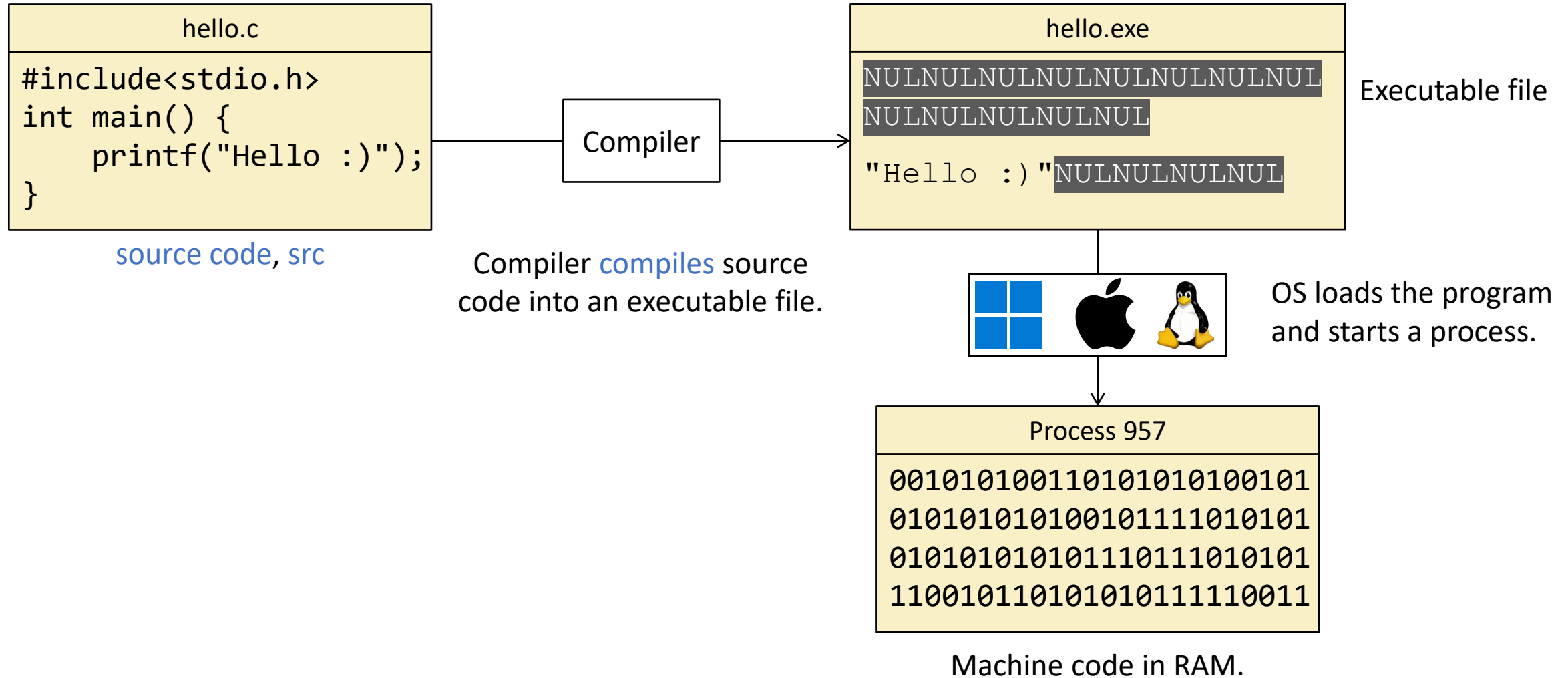
From Source Code to Executable Programs

Programming Languages are translated to machine code via compilers (and linkers).

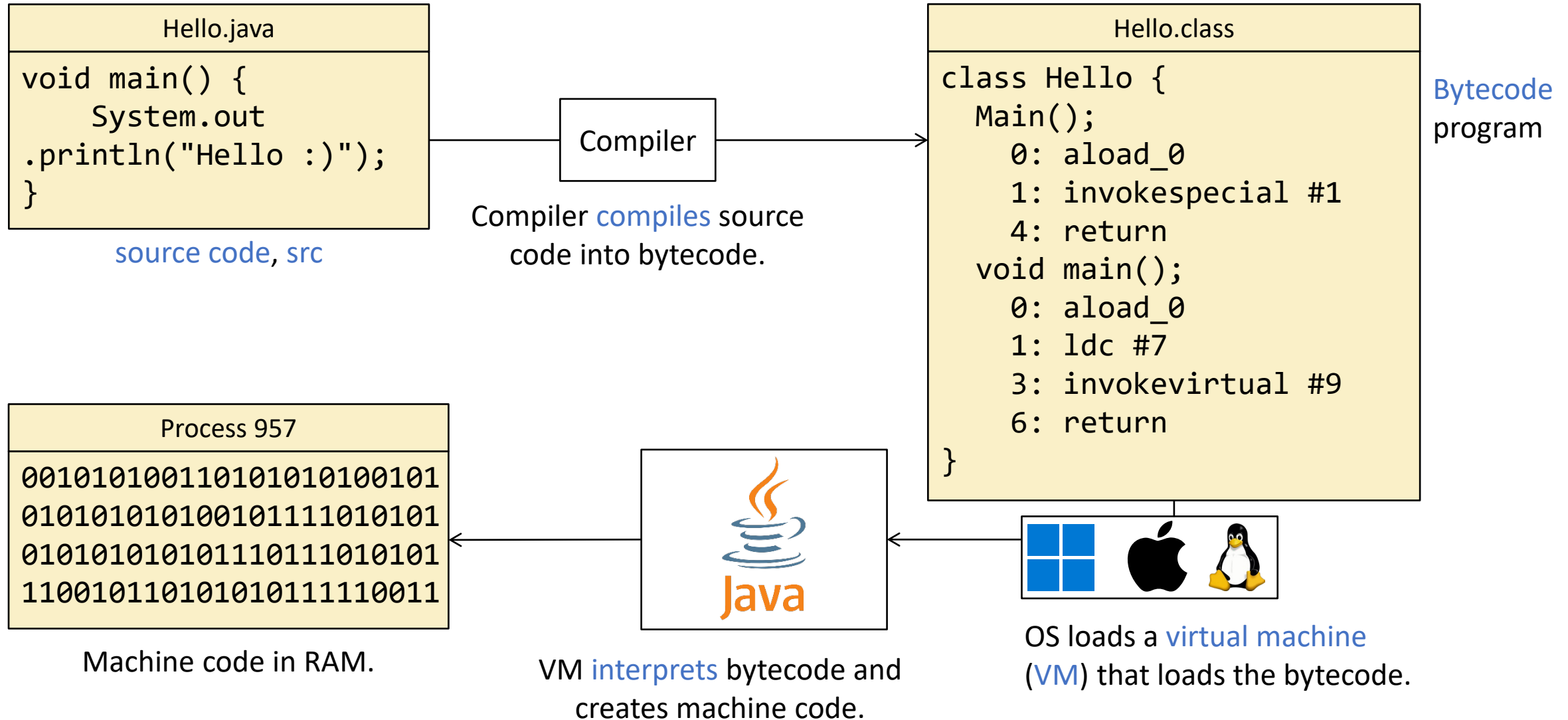
- Lexical analysis (Scanner): Identify **Tokens**.
- Syntactical analysis: Create **abstract syntax tree (AST)**.
- Semantic analysis: Refine AST.
- Synthesis: Create target code using AST and mapping rules.



Compilation



Compilation of Partly-Compiled Languages



Questions?

<https://www.giga.de/spiele/star-wars-battlefront/tipps/star-wars-battlefront-imperator-guide-so-raeumt-ihr-mit-palpatine-auf/>



Imperative Programming

Crash Course for students
of the Java Programming
Course

Imperative Programming: Overview

The term imperative Programming usually covers multiple paradigms:

- "Pure" imperative programming / unstructured programming.
- Structured Programming.
- Procedural Programming.
- Modular Programming.
- Object-Oriented Programming.

We will go through these paradigms and see how they map to Java. We assume you are familiar with the concepts these paradigms provide, even if you were not familiar with the naming. This is why we do not go into details here.

"Pure" Imperative Programming / Unstructured Programming

Unstructured Programming involves the fundamental concepts of programming: variables and operations.

- Declaration of a variable:

```
[modifiers] <type> <variable identifier> [= initial value]; float salary = 2750.0f;
```

- Types define how to interpret a bitstring in memory.

Primitive types in Java: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`.

- Operators are expressions comprising multiple part-expressions to yield a value:

```
<operator><expression> //unary <expression><operator><expression> //binary
```

- Operations in Java: `+`, `-`, `*`, `/`, `%`, `=`, `~`, `^`, `|`, `&`, `||`, `&&`, `>>`, `>>>`, `<<`, `++`, `--`, `!`, `<`, `>`, `==`, `<=`, `>=`, `!=`, `○=`

Structured Programming

Pioneered by David Parnas and Edsger Dijkstra, see "goto considered harmful".

Structured programming introduces control flow structures, i.e., branching and looping.

- Conditional branching:

```
if(condition) {  
    //code  
}  
else if(condition2) {  
    //code  
}  
//...  
else {  
    //code  
}
```

```
switch(var) {  
    case 0: { ...; break; }  
    case 1: { ...; break; }  
    case 2: { ...; break; }  
    case 3: { ...; break; }  
    case 4: { ...; break; }  
    case 5: case 6: case 7: case 8: {  
        ...; break;  
    }  
    default: { ...; }  
}
```

Structured Programming

Structured programming introduces control flow structures, i.e., branching and looping.

- Loops:

```
while(condition) {  
    //code  
}
```

```
for(init; condition; update) {  
    //code  
}
```

```
do {  
    //code  
} while (condition)
```

```
for(<type> element: collection) {  
    //code  
}
```

- Interrupting loops:

- Terminate enclosing loop:

```
break label;
```

- Jump to next iteration of enclosing loop:

```
continue label;
```

Procedural Programming

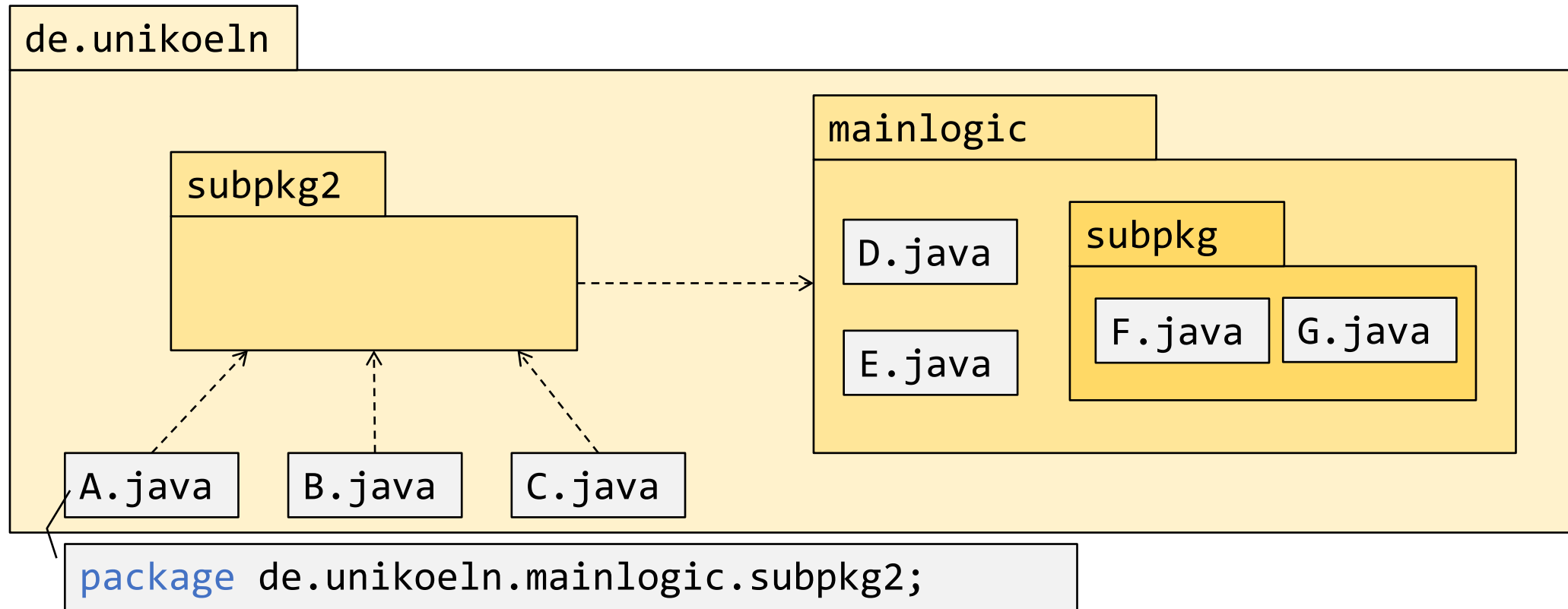
Procedural programming takes structural programming to another level: We define named blocks of code, **procedures**, to which we can jump by calling them.

```
returnType procedureIdentifier(type1 para1, type2 para2) {  
    //code  
}
```

- Calling a procedure: `myProcedure(1, "hey", true)`
- Procedures with return type `void` do not return any value. They are only sensible if they perform any kind of side effect: state change, printing to console, writing to file.
- Procedures with any other return type compute a result: `return <value>;`
 - `return` is the last executed instruction in a procedure. Any instruction after that in the control flow is **unreachable**.

Modular Programming

Modular Programming concerns with the decomposition of code into modules. In Java, there are three levels of modules: classes, packages and modules (which should be named components).



Questions?



Memory Management

Process Memory Layout

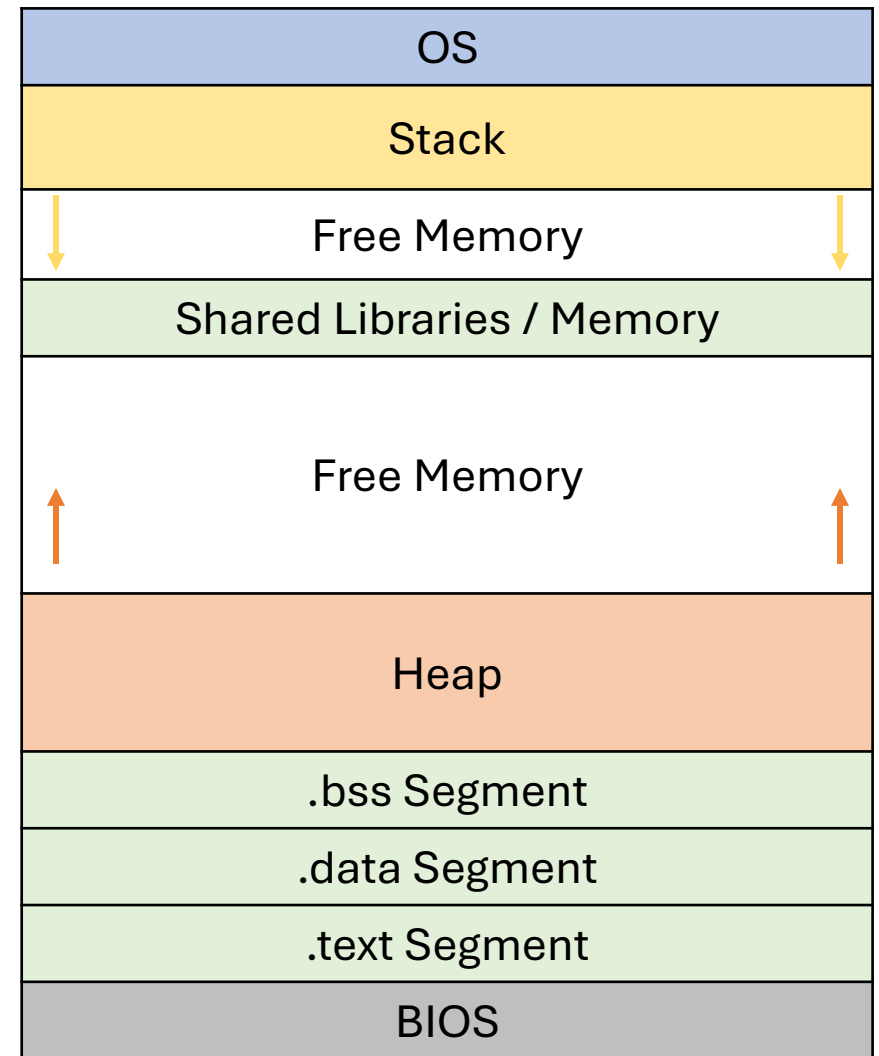
Stack

The stack memory is used for storing values of local variables and expressions.

Heap

The heap memory is used to store objects (more on that later).

The other segments of a processes' memory are not relevant for us and will therefore be referred to as **static memory** in the upcoming lectures.



Argument Passing

We call procedures via their identifier and pass **arguments** (actual values) for their parameters. `myProcedure(a, "hey", true)`

Throughout different programming languages, there are different argument passing mechanisms.

- **Call by value**: the arguments are evaluated before being passed to the procedure. The passing happens via copying the value. By this, it is not possible to change, e.g., a variable passed as argument outside the procedure. This is what Java uses.
- **Call by reference**: The parameter becomes an alias for the argument. Thus, it is possible to change the original value from within the procedure.
- Others, not as common.

Call by Value Passing

The value of arguments is copied to the parameters (which act as local variables of the procedure). Changing the value of the parameters does not affect the original variables.

```
int pow(int x) {  
    int result = x * x;  
    x += 5;  
    return result * x;  
}  
  
int x = 5;  
  
println(pow(x)); //250  
println(x);      //5
```

0xA7	5 (x)	5 (x)	5 (x)	5 (x)	5 (x)
0xA6		return address	return address	return address	return address
0xA5		base pointer	base pointer	base pointer	base pointer
0xA4			25 (result = x * x)	25 (result = x * x)	25 (result = x * x)
rdi		5 (x)	5 (x)	10 (x)	10 (x)
rax					250 (result * x)

Stack evolution over time →

Iteration vs. Recursion

Iteration and Recursion are two principals of repeating the same set of instructions. Iteration uses loops; recursion is the self call of a procedure.

```
while(condition) {  
    //code  
}
```

```
returnType name(type1 para1) {  
    //code  
    name(x);  
}
```

Iteration is faster and is stable in memory usage.

Recursion can be dangerous in terms of memory usage but allows for better parallelization. Plus, recursion is the only way to loop in strictly functional programming languages.

Iteration vs. Recursion: The Stack Memory

Iteration overrides the same cells on the stack its body is using in each iteration.

```
int f = factorial(10); //<- Ret F0
```

```
void factorial(int n) {  
    int result = n;  
    for(int i = 1; i < n; i++)  
        result *= i;  
    return result;  
}
```

		It. 1	It. 2	It. 3	It. 4
0xA2	Ret F0				
0xA1	result	= n	= n*1	= n*1*2	= n*1*2*3
0xA0	i	= 1	= 2	= 3	= 4

Iteration vs. Recursion: The Stack Memory

Recursion, being based on function calls, always creates a new stack frame, thus possibly causing a stack overflow error.

```
int f = factorial(10); //<- Ret F0
```

```
int factorial(int n) {  
    int result = n;  
    if(n==0) return 1;  
    return result * factorial(n - 1); //<- Ret F1  
}
```

```
"int factorial(int n - 1) {  
    int result = n - 1;  
    if(n==0) return 1;  
    return result * factorial(n - 2); //<- Ret F2  
}"
```

```
"int factorial(int n - 2) {  
    int result = n - 2;  
    if(n==0) return 1;  
    return result * factorial(n - 3); //<- Ret F3  
}"
```

...

0xA7	Ret F0	Stackframe "n"
0xA6	result (n)	
0xA5	Ret F1	Stackframe "n-1"
0xA4	result (n - 1)	
0xA3	Ret F2	Stackframe "n-2"
0xA2	result (n - 2)	
0xA1	Ret F3	
0xA0	...	

Questions?



Objects and Classes

```
import java.util.*; Arrays;
```

```
public class Student {
```

```
    private String name;
```

```
    private int age;
```

```
    private String[] courses;
```

```
    public Student(String name, int age, int  
        roomNumber, String[] courses) {  
        this.name = name;
```

```
StudyProgram
```

```
name: String
```

```
1..*
```

```
Student
```

```
multiple int  
1..many Student()
```

```
1..20
```

```
with int  
Student()
```

```
1..many
```

```
1..many
```

Objects

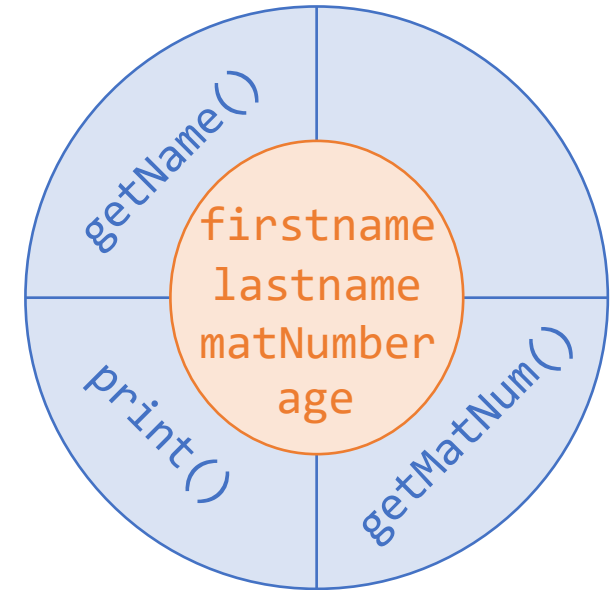
Objects are dynamically created, encapsulated units of **data (variables, its state)** and **operations (methods, its behavior)**.

The **state** of an object is time dependent

- State = Values of variables.
- State in t_i = Values of variables in t_i .

The **behavior** of an object is the set of reactions to **operation** invocations (\rightarrow **methods**).

- Self-state-transition.
- Invocation of operations of other objects.



OOP Terminology

Message: Operation invocation / method call.

```
student.setRecord(Grade.get("1.0"), softwareEngineering);
```

↑
Receiver of
the message

↑
Message

Operation: The parts of a method visible to the outside.

```
public void setRecord(Grade grade, Module module)
```

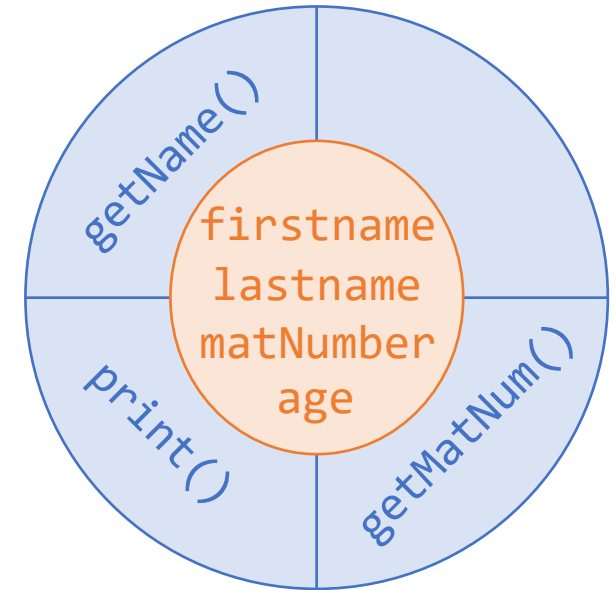
Method: The body of a method.

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

Objects

Encapsulation

- The language ensures that state transitions are only triggered via the provided interfaces.
- If the interface stays the same, changes in the implementation won't affect other objects.



Interface of an object

- Set of callable operations for a set of users.
- Different users may be presented different interfaces.

How are Objects Created?

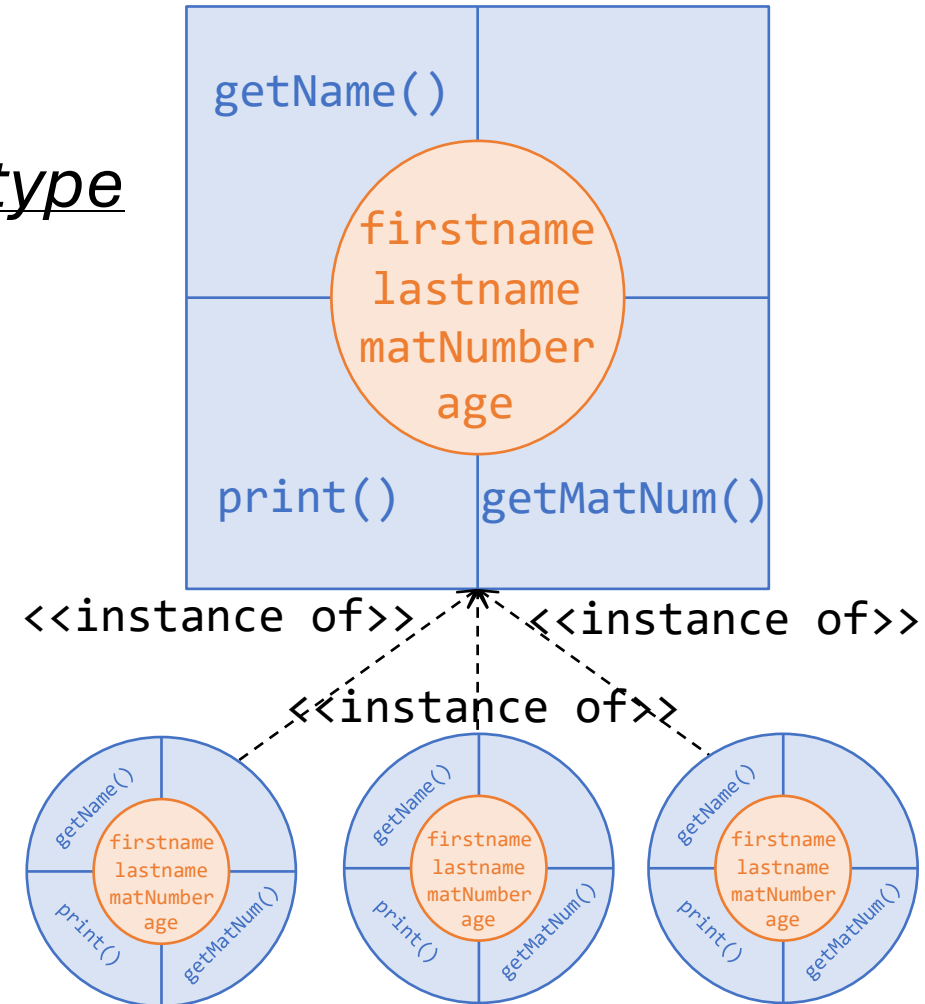
In **class-based** languages:

Specification through "writing it down":

Variables and Methods can be specified per *type* of object (→ in its **class**).

Creation through **instantiation**:

- Objects are created through a special operation, called **constructor**.
- The class is a template for its objects.
- Objects created from a class are called **instances** of the class.



Classes: Example

A class describes a set of "similar" objects

- Structurally equal
- Same interface
 - Same attributes
 - Same operations
 - Same method implementations
 - Different values of variables

```
class Student {
    String firstname;
    String lastname;
    int age;
    int matriculationNumber;

    Student(String firstname, String lastname,
            int age, int matNumber) {
        this.age = age;
        this.firstname = firstname;
        this.lastname = lastname;
        this.matriculationNumber = matNumber;
    }

    void print() {
        System.out.println(this.firstname + " "
            + this.lastname + ", " + this.age +
            ", " + this.matriculationNumber);
    }

    ...
}
```


Classes: Instantiation

```
Student adrian = new Student("Adrian",  
    "Bajraktari", 26, 2969443);
```



firstname	"Adrian"
lastname	"Bajraktari"
age	26
matriculation- Number	2969443

The expression `new Student (...)` creates a new instance of type `Student` and takes as value the reference to the instance.

```
class Student {  
    String firstname;  
    String lastname;  
    int age;  
    int matriculationNumber;  
  
    Student(String firstname, String lastname,  
            int age, int matNumber) {  
        this.age = age;  
        this.firstname = firstname;  
        this.lastname = lastname;  
        this.matriculationNumber = matNumber;  
    }  
  
    void print() {  
        System.out.println(this.firstname + " "  
            + this.lastname + ", " + this.age +  
            ", " + this.matriculationNumber);  
    }  
    ...  
}
```

Messages

```
adrian.setAge(27);
```

firstname	"Adrian"
lastname	"Bajraktari"
age	27
matriculation- Number	2969443

In the object-oriented world, invoking/calling a method is also called a [message](#).

```
class Student {  
    String firstname;  
    String lastname;  
    int age;  
    int matriculationNumber;  
  
    Student(String firstname, String lastname,  
            int age, int matNumber) {  
        this.age = age;  
        this.firstname = firstname;  
        this.lastname = lastname;  
        this.matriculationNumber = matNumber;  
    }  
  
    void print() {  
        System.out.println(this.firstname + " "  
                            + this.lastname + ", " + this.age +  
                            ", " + this.matriculationNumber);  
    }  
    ...  
}
```

Accessing Instance Members

Access to instance members is done via the following scheme:

- Access to instance variables: `objectRef.attribute`
- Access to instance methods: `objectRef.operation(arg1, ..., argn)`

It is generally considered bad coding style to directly access an objects attributes. Instead: Define methods that encapsulate the access:

- **Getter methods** to get the variables value.
- **Setter methods** to set the variables value.

Attention:

```
Student adrian = null;  
int a = adrian.age;  
adrian.print();
```

Access to members via a reference to `null` causes a `NullPointerException` to be thrown!

Class Methods and Variables

Class methods work independent of instances. They are called via a class directly.

- Keyword `static`
- Semantic: Class variables are the same for all instances of a class.

Invocation:

```
class Student {  
    static Student[] studentList  
        = new Student[10];  
    ...  
    static int getNumberOfStudents() {  
        return studentList.length;  
    }  
    ...  
}
```

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

Visibilities: Overview

The following shows which visibility modifier applies to which level. This applies to classes, methods, constructors and attributes.

Visible in:	public	protected	<i>package-private</i>	private
Own class	✓	✓	✓	✓
Own package	✓	✓	✓	✗
Subclasses	✓	✓	✗	✗
Globally	✓	✗	✗	✗

Attention! Visibility is neither access control, nor mutability!

Questions?

Summary

In today's lecture:

- General programming paradigms and languages.
- Imperative programming concepts.
- Memory management.
- Basics of object-oriented programming.

Use this knowledge when developing code, in your job as well as in the exercises.