

Software Engineering

Advanced Programming

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

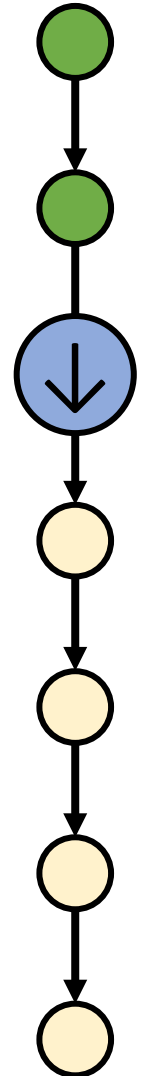
Cover advanced object-oriented principles.

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).

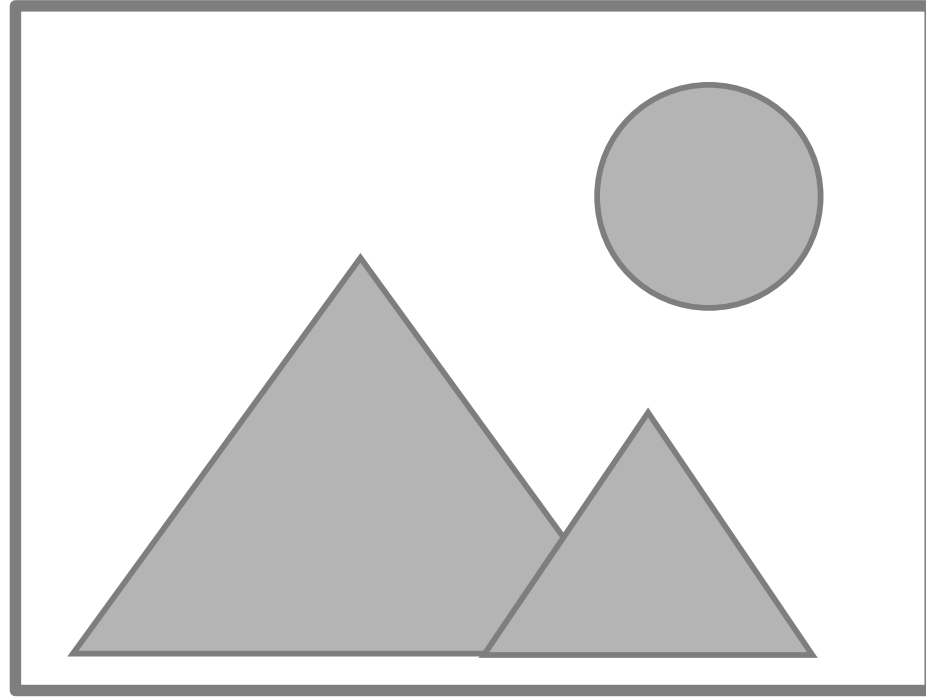


Last Lecture

- Basics of OOP
- What is object identity?
- What are types (for)?
- What is inheritance? (and what is it not?)
- What are interfaces (for)?
- What is dynamic binding and how does it work?
- Exception handling.
- Introduced threading.

Aims of this Lecture

- Introduce generic programming.
- Introduce special class variants.
- Introduce "functional" Java.
- Object-based programming with JavaScript.



Generic Types and Generic Programming

Motivation

In statically typed languages, how can we write code that works on different types? E.g., how can we implement data structures that store arbitrary types of data?

```
class IntList {  
    IntNode head;  
    ...  
}  
  
class IntNode {  
    int data;  
    IntNode next;  
    ...  
}
```

First Idea: Clone it!

Copy code and replace all occurrences for each data type.

- Easy, but absolute no-go. Bloats code up. Every time you find a bug in one variant, you also need to check all other variants as well.
- "Rule of three" in Refactoring book. Having two variants of something is okay, but the third time you need to generalize it.

```
class IntList {  
    IntNode head;  
    ...  
}  
class IntNode {  
    int data;  
    IntNode next;  
    ...  
}
```

```
class FloatList {  
    FloatNode head;  
    ...  
}  
class FloatNode {  
    float data;  
    FloatNode next;  
    ...  
}
```

```
class PersonList {  
    PersonNode head;  
    ...  
}  
class PersonNode {  
    Person data;  
    PersonNode next;  
    ...  
}
```

Second Idea: Subtyping

Make use of subtyping! Find a common supertype of all data types in question. This is also known as manual genericity and has been the way to do it in Java until Java 5.

Problem: Many casts and runtime type checks. Most of the time, the only logical common supertype of "everything" is Object, which does not really provide much to work with.

```
class List {  
    Node head;  
    ...  
}  
  
class Node {  
    Object data;  
    Node next;  
    ...  
}
```

```
List list = new List();  
list.add("str");  
String s = (String) list.get(0); //cast necessary  
...  
Integer i = (Integer) list.get(0); //cast + runtime error
```


Genericity

Language level genericity allows to specify the type of certain elements on the client side.

Generic classes have one or many **type parameters**. Variables, parameters, returns and local variables of the class may have this type parameter as static type or have generic types that use this type parameter themselves.

```
public class Node<T> {  
    T data;  
    Node<T> next;  
    setData(T data);  
    ...  
}
```

```
Node<Person> n = new Node<Person>();
```

The value of the type parameter is determined at instantiation time.

Generic and Parameterized Types

A **generic type** is a template for unlimited many **parameterized types**.

Parameterized Types are created through **type instantiation**.

```
Node<Person> p = new Node<Person>();  
Node<Lecture> l = new Node<Lecture>();  
Node<Department> d = new Node<Department>();
```

Type arguments may be arbitrarily "stacked" parameterized types as well:

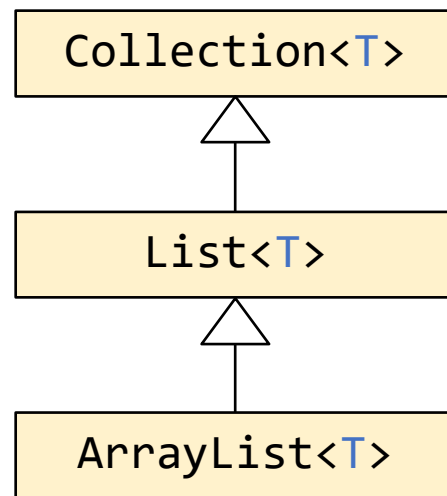
```
Node<List<ThreeTuple<Student, Examiner, Examiner>>>  
    listOfStudentExaminerMappings = new Node<>();
```

Questions?

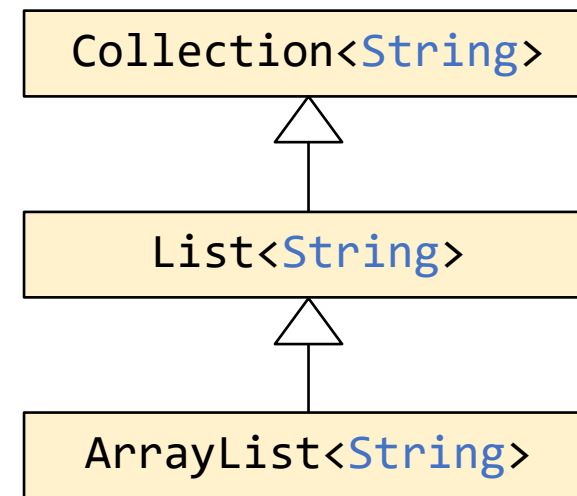
Generic Types and Subtyping

If a generic type $\text{Sub}\langle X_1, \dots, X_n \rangle$ is subtype of a generic type $\text{Super}\langle Y_1, \dots, Y_n \rangle$, then the parameterized type $\text{Sub}\langle P_1, \dots, P_n \rangle$ is subtype of $\text{Super}\langle P_1, \dots, P_n \rangle$. Mind that the type arguments are exactly the same!

Generic type hierarchy



Parameterized type hierarchy



$T = \text{String}$

$T = \text{String}$

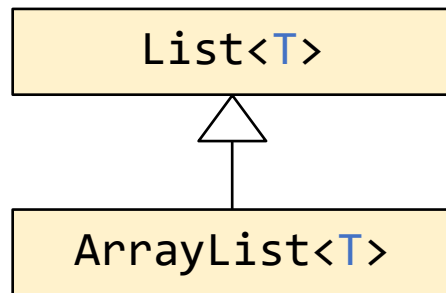
$T = \text{String}$

Generic Types and Subtyping

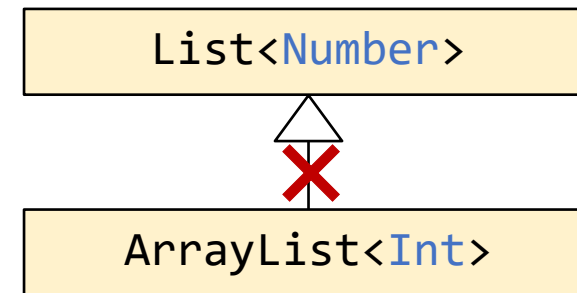
The parameterized type $\text{Sub}\langle P_1, \dots, P_n \rangle$ is never a subtype of the parameterized type $\text{Super}\langle Q_1, \dots, Q_n \rangle$.

- Though generic Sub is subtype of generic Super.
- Even if P_i is subtype of Q_i .
- Even if $\text{Sub} == \text{Super}$.

Generic type hierarchy



Parameterized type hierarchy



Why?

```
List<String> ls = new ArrayList<String>();           //okay
List<Object> lo = ls;                               //not allowed in Java
```

Line 1: okay! ArrayList is a subtype of List.

In line 2, we must ask: "Is a list of strings also a list of objects?"

- One might say "of course" and yes, it is, but
- we cannot treat it like a list of objects:

```
lo.add(new Object());           //insert an Object instance into lo
String s = ls.get(0);           //type error! Cannot assign Object to String
```


So, is there no subtyping possible for type arguments? It is, and it gets complicated.

Bounded Genericity

A type variable `T` can be substituted by anything. Thus, we do not know anything about its interface. Often, we need some information about the interface to work with the data.

```
public class Node<T> {  
    T data;  
    public MyClass(T v) {  
        data = v;  
    }  
    public void setData(T v) {  
        data = v;  
    }  
    public T getData() {  
        return data;  
    }  
}
```

```
public void print() {  
    println(this.getData().intValue());  
}
```



The compiler needs to know that the object returned by `getData()` has an `intValue()` method. Otherwise, it must forbid the message. We thus need a way to restrict the type variable to a set of types of which the compiler knows the interface.

Bounded Genericity

A type variable `T` can be substituted by anything. Thus, we do not know anything about its interface. Often, we need some information about the interface to work with the data.

```
public class Node<T
    extends Number> {
    T data;
    public MyClass(T v) {
        data = v;
    }
    public void setData(T v) {
        data = v;
    }
    public T getData() {
        return data;
    }
}
```

```
public void print() {
    println(this.getData().intValue());
}
```

The compiler now knows that whatever is returned by `getData()` will be an instance of a subtype of `Number`.

Bounded Genericity: Multiple Bounds

A type variable may have more than one upper bound.

```
public class Example<T extends Person & Cloneable & Printable> {  
    ...  
}
```

Syntax: Bounds are separated by &. If one of the bounds is a class and not an interface, it must be placed first.

Semantic: The type argument must be subtype of all provided type bounds.

Generic Methods

Methods can be made generic as well.

```
public class Other {  
    public void print(Node<Number> n) {  
        println(n.getData().intValue());  
    }  
}
```

The above would work, but only allows Number nodes.

Generic Methods

Methods can be made generic as well.

```
public class Other <E extends Number> {  
    public void print(Node<E> n) {  
        println(n.getData().intValue());  
    }  
}
```

This now also allows subtypes of `Number`. But if only the `print` method uses the type parameter `E`, it would be bloating to declare the whole class as generic.

Generic Methods

Methods can be made generic as well.

```
public class Other {  
    public <E extends Number> void print(Node<E> n) {  
        println(n.getData().intValue());  
    }  
}
```

This now also allows subtypes of Number and keeps the genericity in the print method. We can use the method as follows:

```
<Integer> print(new Node<Integer>(42));
```

Questions?

Wildcards

If we use a type parameter only once, we can directly get rid of it and use wildcards:

```
public class Other {  
    public void print(Node<? extends Number> n) {  
        println(n.getData().intValue());  
    }  
}
```

```
print(new Node<Integer>(42));
```

This hides the genericity of print from clients.
The same applies everywhere where we want to allow subtyping type arguments (see slides about why subtyping does not work).

Wildcards

The wildcard `?` is an unnamed type, not a type variable.

The following two lines are equivalent in their functionality:

```
public <A extends Number, B extends Integer> void print(Node<A> n, List<B> l);
```

```
public void print(Node<? extends Number> n, List<? extends Integer> l);
```

This way, we reduce one-time type variables.

Wildcards are allowed as types of local variables, attributes, parameters and return types (though this should be avoided).

Wildcards

A wildcard can have

- an upper bound `<? extends Number>`
- a lower bound `<? super Number>`.

Because of reasons, for parameters using wildcards:

- Input parameter: `<? extends UpperBound>`
- Output parameter: `<? super LowerBound>`

Example (generic list copying):

```
<T> void copy(List<? extends T> source, List<? super T> destination) {  
    ...  
}
```


Input and Output Parameter

Generics with an **upper bound** are input parameters, i.e., we can only call methods on them that do not have the type parameter as parameter types, but as return types (we can "read from them", thus "input"). We also call them "producers".

Generics with a **lower bound** are output parameters, i.e., we can only call methods on them that do not have the type parameter as return type, but only as parameter type (we can "write into them", thus "output"). We also call these "consumers".

```
<T> void copy(List<? extends T> source, List<? super T> destination) {  
    ...  
}
```

When and when not to use Wildcards

Wildcards are the preferred way to use bounded genericity if there are no interdependencies between parameterized types, e.g., that they share the same type variable:

```
public <A extends Number, B extends Integer> void print(Node<A> n1, Node<A> n2, List<B> l);
```

In the above code, `B` is superfluous, as it is only used one time. We can replace it with a wildcard.

`A`, however, is used in two parameters. This means, that, whatever we plug into `A`, must be the same for `n1` and `n2`. This relationship can not be expressed via wildcards. Thus, `A` can not be removed.

Restrictions on Generic Types (1/2)

The information whether a type is generic or parameterized exists only until the compiler checked the program. At runtime, parameterized types become "normal": A `List<Integer>` becomes a `List` and is thus not differentiable from a former `List<Float>`.

Thus, it is not possible to use the type parameters for:

- Object creation: no `"new T ()"`,
- Type checking: no `"instanceof T"`,
- Declaration of class variables: no `"static T var;"`,

Restrictions on Generic Types (2/2)

- Create arrays of parameterized types: no `"new List<Number>[10]"`,
- Cannot create generic exceptions: no `"class Except<T>"`,
- Overload generic methods in special cases.

Also, type arguments must always be reference types. Thus, for primitive types, we must use their wrapper classes.

Questions?



Special Classes

```
import java.util.ArrayList;
```

```
public class Student {
```

```
    private String name;
```

```
    private int age;
```

```
    private String address;
```

```
    private int phoneNo;
```

```
    public Student(String name, int age, int
```

```
        phoneNo, String address) {
```

```
        this.name = name;
```



Enumeration

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, ...

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

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

Enumeration Construction

As normal classes, enums can

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

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

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

Nested Classes

Classes can have nested classes inside of them.

Nested classes can have any visibility. Outer classes must always be public or protected.

There are two kinds of inner classes:

- Static, which work just as regular classes, and
- non-static, which's instances need an outer-class instance to be created on.

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

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

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

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

Non-Static Inner Classes

Non-static inner classes have a (compiler generated) reference to their "outer" object.

Through that, they can access the outer instances members (even private ones).

```
public class Map {  
    ...  
    private int count = 5;  
  
    public class Entry {  
        ...  
        public int method() {  
            return count;  
        }  
    }  
}
```

Local Class

Local classes can be declared in methods and are intended as one-time use classes.

They have access to the enclosing objects members (as non-static inner classes) but also to the local variables of the surrounding method, if they are **effectively final**.

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

Anonymous Classes

Anonymous classes are local classes with no name. They are intended for a one-time object creation.

The anonymous class is build upon an already existing type (class/interface) as static type. The idea is, that the anonymous class extends this type, e.g., by overriding.

This is especially useful for one-time objects of interface 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(...) {...}  
}
```

When to use what

- Local class: if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).
- Anonymous class: if you need to declare fields or additional methods, extending and existing type (class or interface).
- Static inner class: if your requirements are similar to those of a local class, but you want to make the type more widely available, and you don't require access to local variables or method parameters.
- Non-static inner class: if you require access to an enclosing instance's non-public fields and methods.

Questions?

λ

Functional Aspects in Java

Functional Programming in Java

Java provides facilities to develop in a functional way.

- Lambda expressions (closures).
- Streams.

Functional Interfaces and Lambda Expressions

Functional interfaces are the basis for "method objects". Functional interfaces only consist of one method.

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

A lambda expression is an expression of the following format that realizes a functional interface:

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

The relation to a functional interface is made via the static type.

Lambda Expressions

A lambda expression...

- is an object of a method.
- can thus be passed around.
- can thus be passed as arguments and returned as results.

A lambda expression can be executed by calling the functional interface's method on the lambda object:

```
fi.method(arg1, ...);
```

Lambda Expressions: Specialties

- When using only one parameter, parentheses can be omitted.

```
(para1) -> {...}
```



```
para1 -> {...}
```

- When the statement block consists of only 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, ...) -> receiver.method(...);
```



```
... lambda = receiver::method;
```

Generic Functional Interfaces

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

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

```
FunctionalInterface<String, Integer, Person> fi = (...) -> {...};
```

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

Generic 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 return 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 are a functional-inspired approach to processing collections.

Streams provide methods that, in some way,

- operate on the data and return a stream object. These often take a lambda expression as argument.
- map a stream to a single result or a collection of results.

A stream can be created as follows:

```
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
```

Stream Operations

Usually, we use stream-returning methods in a method call chain, i.e., `Stream.of(...).filter(...).map(...).reduce(...)`

• <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.

Questions?

> typeof NaN	> true==1	> []+[]
< "number"	< true	< ""
> 999999999999999999	> true===1	> []+{}
< 1000000000000000000	< false	< "[object Object]"
> 0.5+0.1==0.6	> (!+[]+[]+![]).length	> {}+[]
< true	< 9	< 0
> 0.1+0.2==0.3	> 9+"1"	> true+true+true===3
< false	< "91"	< true
> Math.max()	> 91-"1"	> true-true
< -Infinity	< 90	< 0
> Math.min()	> []==0	
< Infinity	< true	

JS

JavaScript

General introduction and
object-based program-
ming

Why JavaScript?

JavaScript (JS) is the language of dynamic websites.

JS was originally developed by Netscape in 1995 and is standardized as ECMAScript. JS developed frontend development but becomes more and more relevant in backend development as well, e.g., via Node.js.

In this lecture, we will use JavaScript for two purposes:

1. As an example of an object-based language that realizes many concepts differently.
2. For the "SE for Web Applications" lectures.

Today and in the next lectures, we focus on everything for the first part. In the WebApp lectures, we will continue with web-related aspects.

JavaScript Basics / Differences to Java

- Everything is an object, also "primitive types".
- Objects in JavaScript are associative arrays ("dictionaries").
- `undefined` for no set value, `null` for empty value.
- No static typing. Types only exist at runtime.
- Variables are declared via `var` (discouraged), `let` or `const`.
- There are specific type conversion rules that are applied automatically. Every non-empty (`0`, `"`, `null`) is considered `true`. Be very careful! Some examples:
- Attention! There is no difference between float and ints in JS! So `42 === 42.0` will be `true`!

```
let x = 10 + 5      //15
let x = 10 + "5"    //"105"
let x = 10 * "5"    //50
let x = [10] + 5    //"105"
let x = true + 5    //6
let x = +"10" + 5   //15
Boolean("false")    //true
```

JavaScript Basics / Differences to Java

Operators

- Arithmetic operators: `+`, `-`, `*`, `/`, `++`, `--`, `%`
- Comparison operators: `>`, `>=`, `<`, `<=`, `==`, `!=`, `===`, `!==`
 - `==` and `!=` check for equality including type conversion.
 - `===` and `!==` check for true equality, not applying any type conversion.

Arrays in JS...

- are dynamic, i.e., they can be extended/shrunked via
 - `push()`/`pop()`, modifying the end of the array.
 - `unshift()`/`shift()`, modifying the front of the array.
- Sorted via `sort()` or printed via `join(sep)`.

JavaScript Basics / Differences to Java

Template Literals

- ``` (backquote) as delimiter.
- Can span multiple lines.
- Can embed expressions via `${...}` to replace complex concatenation.

```
function toString() {  
    return `${this.firstname} ${this.name}  
           takes the following courses:  
           ${this.courses.join(',')}`;  
}
```

Loops & Conditionals

- `if` and `switch`.
- `while`, `do-while` and `for` loop.
- `For-in` and `for-of` loop.

Attention:
uses `===`

```
for(index in this.courses) {  
    //operate using the element index.  
}
```

```
for(course of this.courses) {  
    //operate using the element value.  
}
```

JavaScript Basics / Differences to Java

No overloading. Functions are matched by name only.

- Missing arguments are passed as undefined.
- Additional arguments are ignored.
- Optional parameters via default values.

Functions can be nested.

There can be closures (functions passed as arguments or returned)

- Syntactic sugar: `=>` notation.

```
function getGrades(semester = 'all') {  
    if(semester === 'all') {...}  
    else ...  
}
```

```
student => student.getGrades();  
(student, module) =>  
    student.getExam(module);  
(student, module) => {  
    module.register(student);  
    student.courses.add(module);  
}
```

JavaScript Object Notation (JSON)

JavaScript objects are in JSON format.

You can serialize JS objects into JSON strings for data exchange via `JSON.stringify(object)`.

You can deserialize objects from JSON strings via `JSON.parse(string)`.

Questions?

Ex-Nihilo Object Creation (from Latin "from nothing")

Objects in (classical) JavaScript are just "written down".

You can either create an empty object and add properties to them, or you provide properties directly at creation time.

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

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

Properties

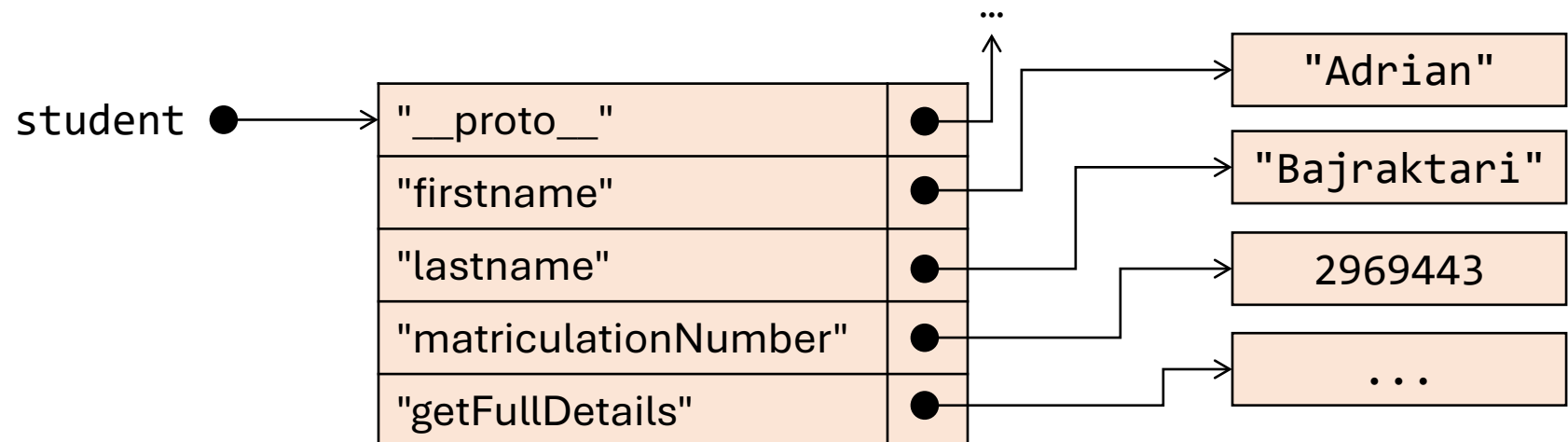
Properties of objects in JavaScript can either be attributes or methods. We can add (by just setting a value) or remove (via `delete`) them dynamically.

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

Accessing properties can be done via `object.property` or `object["property"]`.

Object Implementation

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



Messages in Object-Based Languages

In object-based languages, messages are always dynamically bound method invocations. (Static binding is not possible for dynamically created objects.)

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

Receiver

Message

Object-Based Inheritance: Delegation

Every object has a "super object", referenced by the property `__proto__`. When an object receives a message that it does not understand (i.e., it has no such property), it delegates the message to its super object.

Delegation leaves the receiver of the message as the original receiver. This is different from simply forwarding the message.

Delegation is useful to dynamically "plug" objects together to reuse already implemented functionality in different contexts. This flexibility, however, comes with risks of runtime errors.

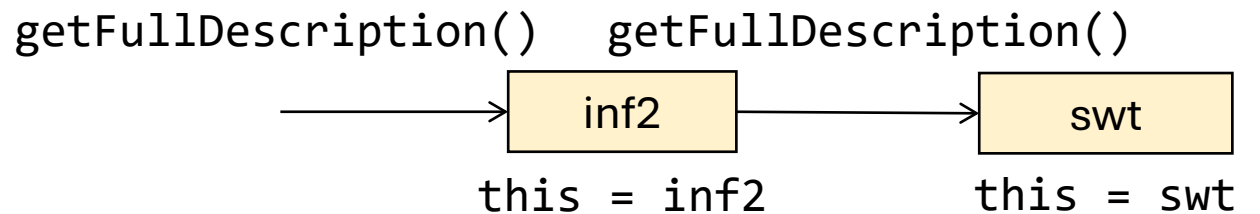
Forwarding

Forwarding is the manual call of another object's method to handle a message.

The problem: The forwarded message is executed on behalf of the other object, i.e., it uses its own methods and variables instead of the ones of the original receiver.

This will always print "Module name: Software Engineering"

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



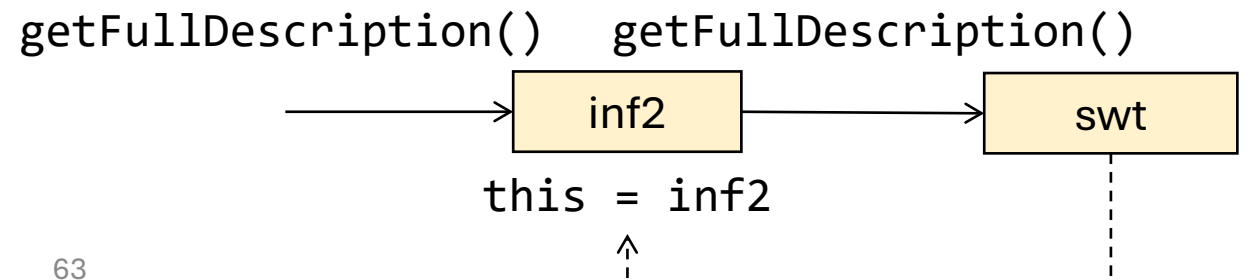
Delegation

Delegation is like forwarding, but the receiver stays the original. That means: accesses to this refer back to the original receiver.

Delegation is ideally automatic. That means, that we do not need to explicitly call another objects method.

In this example, the method will print "Module name: Informatik II" if the receiver is inf2, or "...Software Engineering" if the receiver is swt.

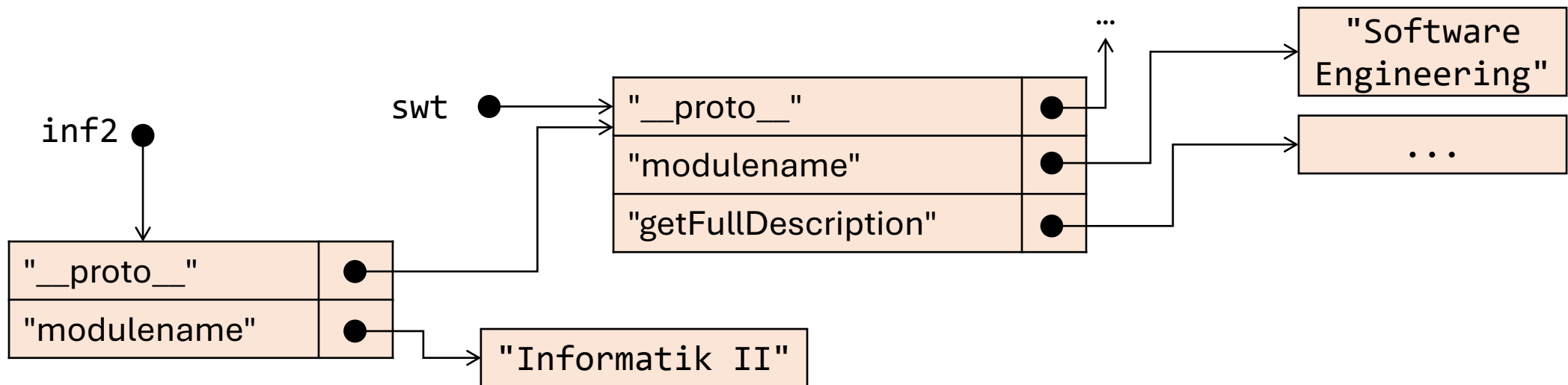
```
let swt = {  
  modulename: "Software Engineering",  
  getFullDescription: function() {  
    return `Module name:  
      ${this.modulename}, ...`;  
  },  
  ...  
};  
let inf2 = {  
  __proto__: swt,  
  modulename: "Informatik II"  
};
```



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"
```



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

Class Inheritance

- 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).

Delegation

- Object-level: defined for each object individually.
- Dynamic: Can be changed at runtime.
- "Inherited" members are part of the parent object.

Classes

Classes in JavaScript are syntactic sugar. Objects instantiated from classes can still be modified (adding/removing properties).

They are realized by a "class object" that is the common `__proto__` of all "instances". All common methods are defined in the class object and via delegation, messages on instances are delegated to the class objects.

```
class Student extends Person{
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}

let student = new Student("Adrian", "Bajraktari");
student.matriculationNumber = 2969443;
delete student.firstname;
```

Class inheritance is in turn realized by setting the `__proto__` of the class object to its super class object.

Questions?

Summary

In today's lecture:

- Generic programming.
- Special classes.
- Lambdas and streams.
- Object-based programming via JavaScript.