Konzepte objektorientierter Programmierung – Lecture 2 –

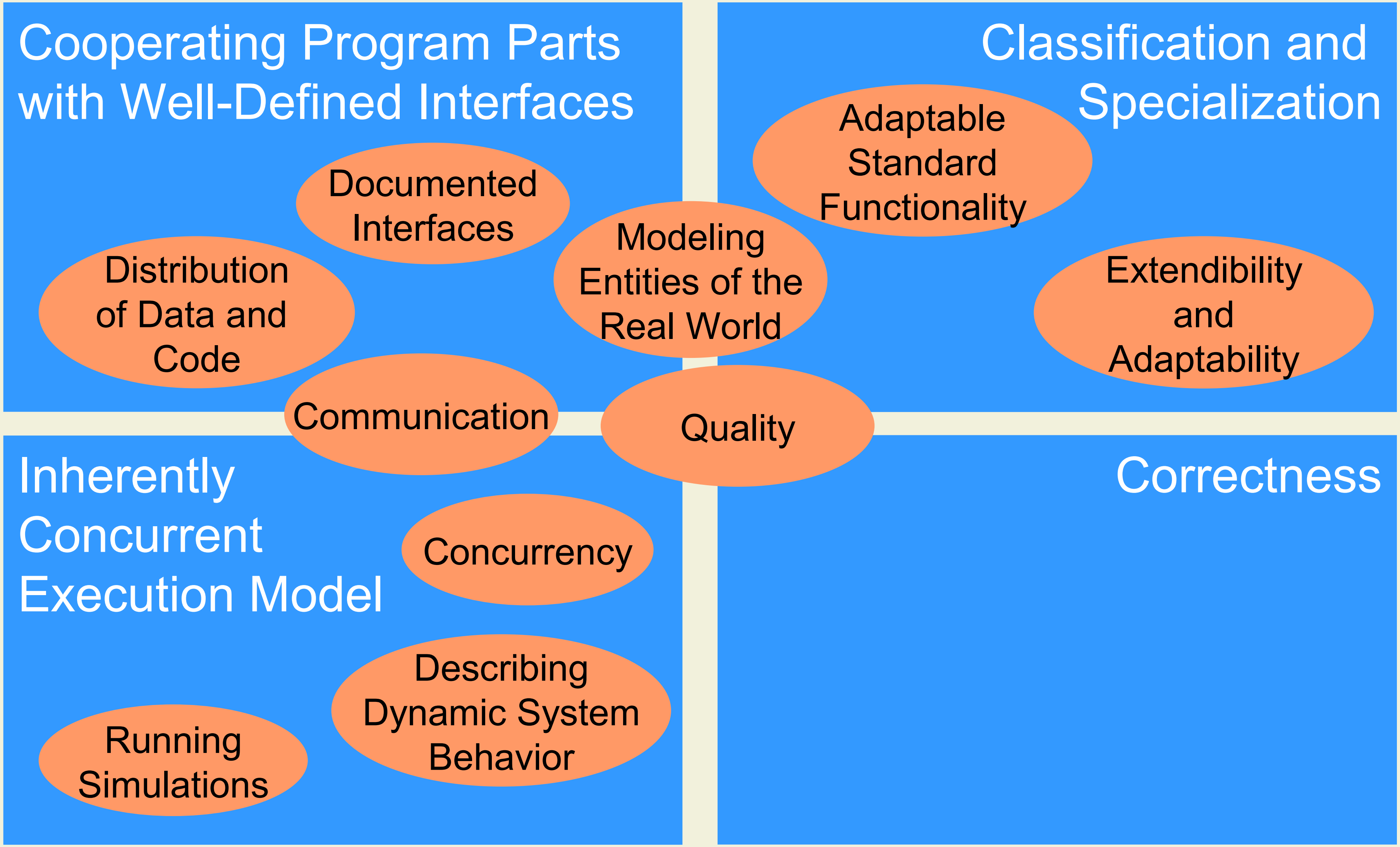
Prof. Dr. Peter Müller

Chair of Programming Methodology

Herbstsemester 2008

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements



Concepts: Summary

Core Concept

Object Model

Interfaces and Encapsulation

Classification and Polymorphism

Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Inherently Concurrent Execution Model

→ Active objects
→ Message passing

Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

Concepts: Summary

Requirement	Core Concept	Language Concept	Language Constructs
Inherently Concurrent Execution Model	Object Model	Classes	Single Inheritance
Cooperating Program Parts with Interfaces	Interfaces and Encapsulation	Inheritance	Multiple Inheritance
Classification and Specialization	Classification and Polymorphism	Subtyping	Inheritance w/o Subtyping
Correctness		Dynamic Binding	Etc.
		Etc.	

Agenda for Today

2. Core and Basic Language Concepts

2.1 Core Concepts

2.2 Basic Language Concepts

Objectives

- Repetition on abstract level
- Relationship between core and language concepts

[This lecture is largely based on Section 2.1 of Arnd Poetzsch-Heffter's course *Fortgeschrittene Aspekte objektorientierter Programmierung, 2002/2003*]

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

2. Core and Basic Language Concepts

2.1 Core Concepts

- Object Model
- Interfaces and Encapsulation
- Classification and Polymorphism

2.2 Basic Language Concepts

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

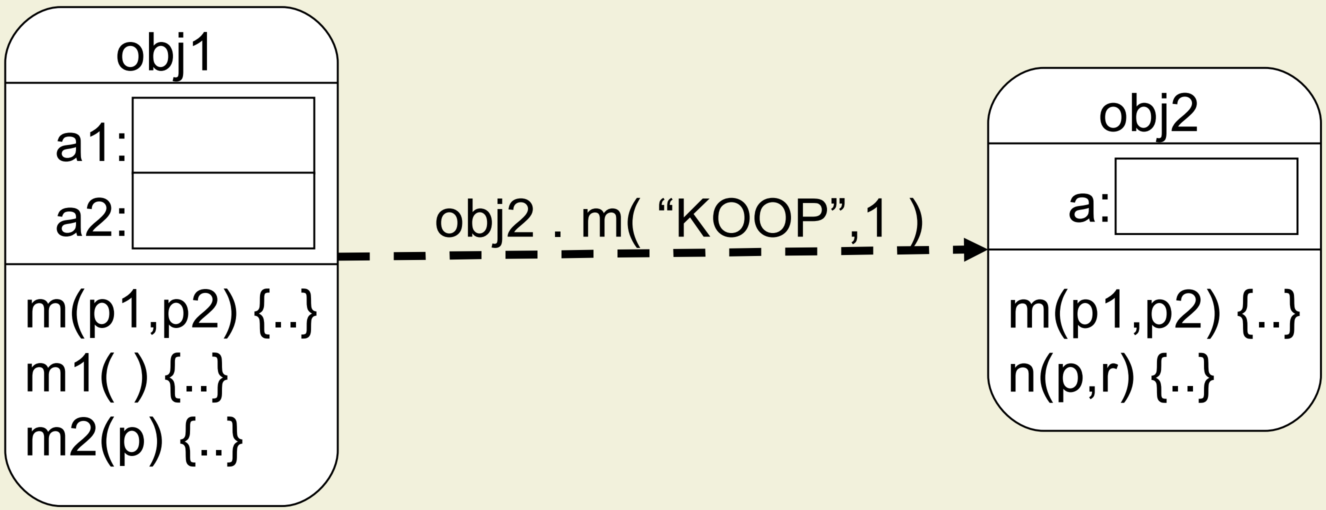
Object Model: The Philosophy

“The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat. It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.“

[Object-oriented Programming in the BETA Programming Language]

The Object Model

- A software system is a set of cooperating objects
- Objects have state and processing ability
- Objects exchange messages



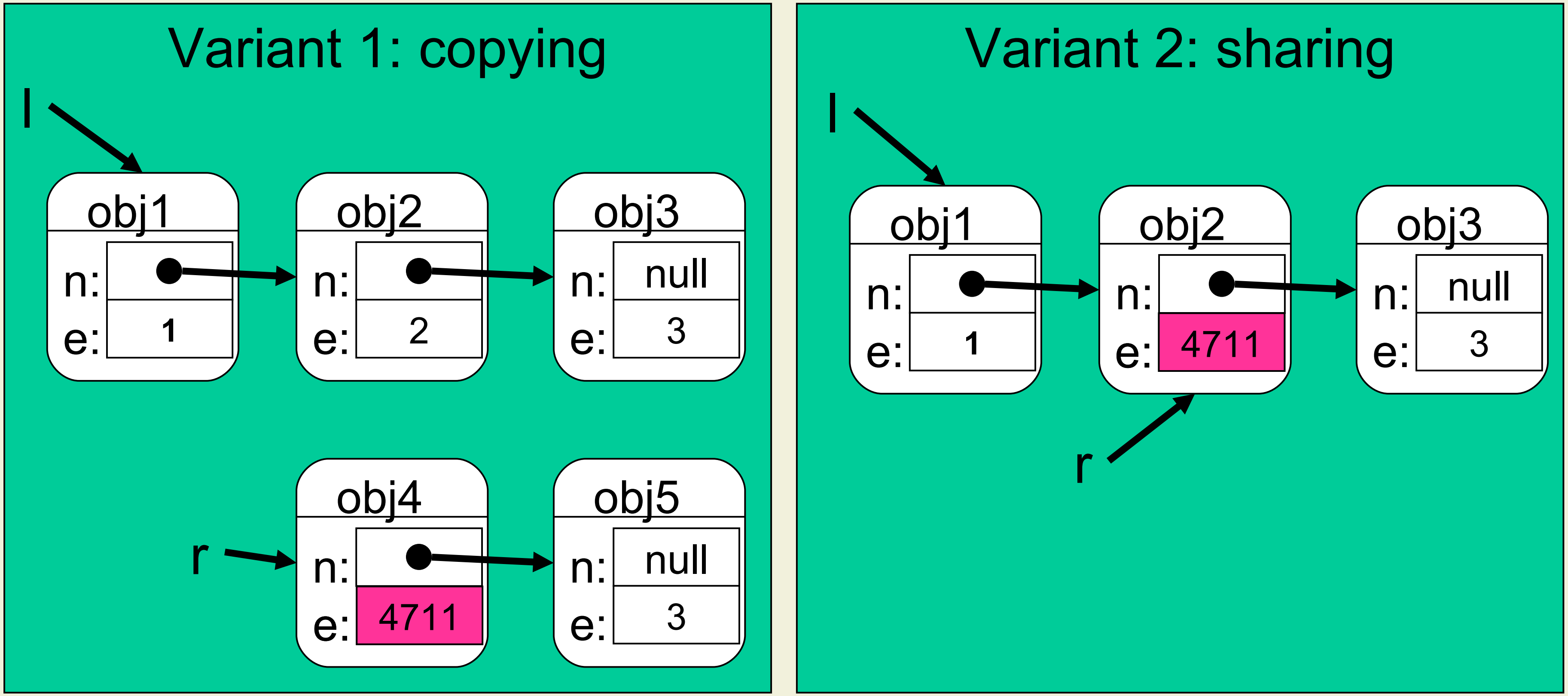
Characteristics of Objects

- Objects have
  - State
  - Identity
  - Lifecycle
  - Location
  - Behavior

- Compared to imperative programming,
  - Objects lead to a **different program structure**
  - Objects lead to a **different execution model**

Object Identity: Example

- Consider
  r = l.rest( ); r.set(4711); **int** i = l.next.get();



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

2. Core and Basic Language Concepts

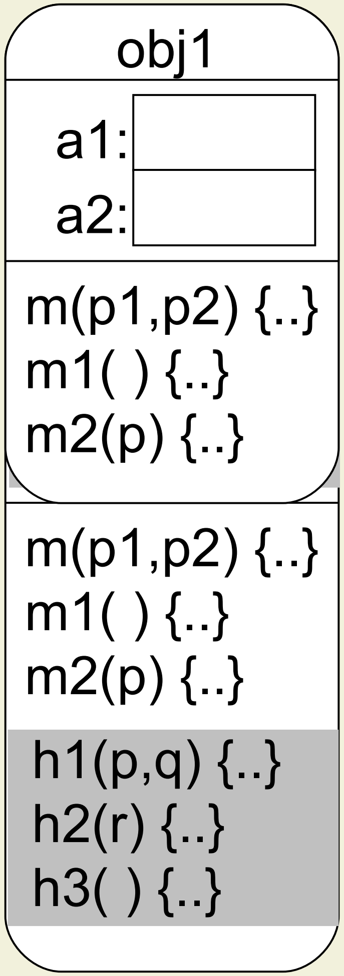
2.1 Core Concepts

- Object Model
- Interfaces and Encapsulation
- Classification and Polymorphism

2.2 Basic Language Concepts

Interfaces and Encapsulation

- Objects have **well-defined interfaces**
  - Publicly accessible attributes
  - Publicly accessible methods
- **Implementation is hidden** behind interface
  - Encapsulation
  - Information hiding
- Interfaces are the basis for **describing behavior**



2. Core and Basic Language Concepts

2.1 Core Concepts

- Object Model
- Interfaces and Encapsulation
- Classification and Polymorphism

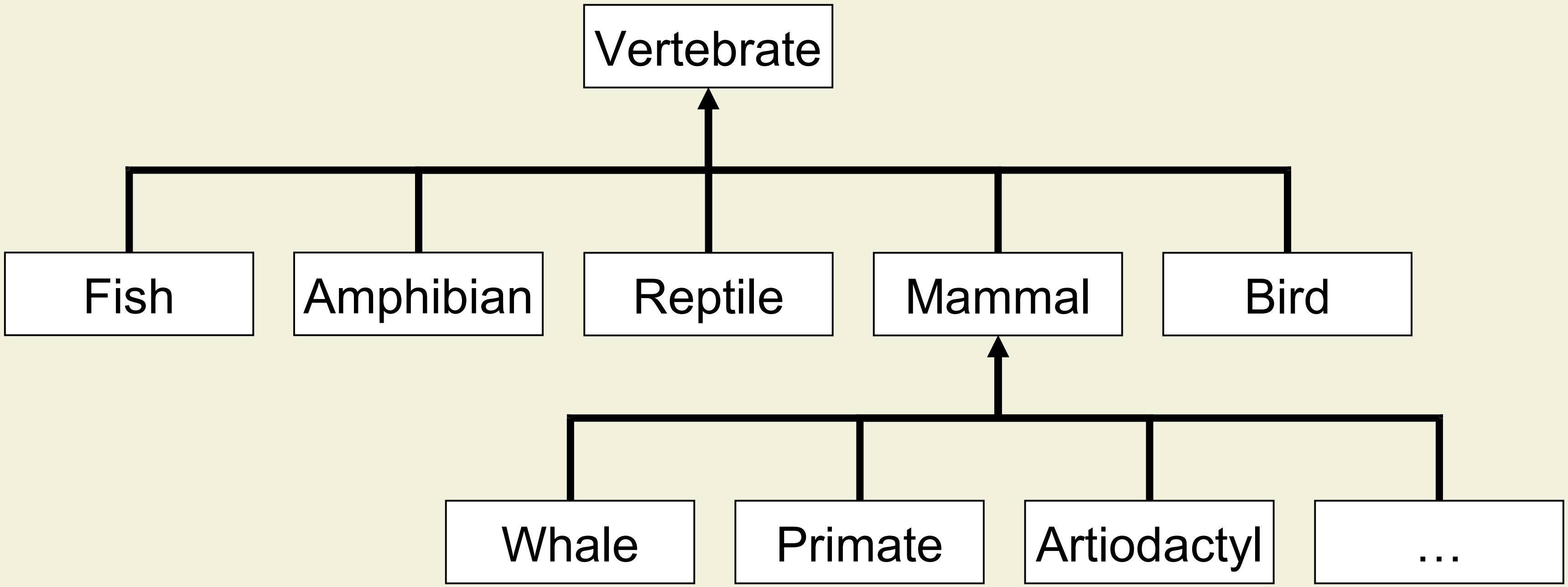
2.2 Basic Language Concepts

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

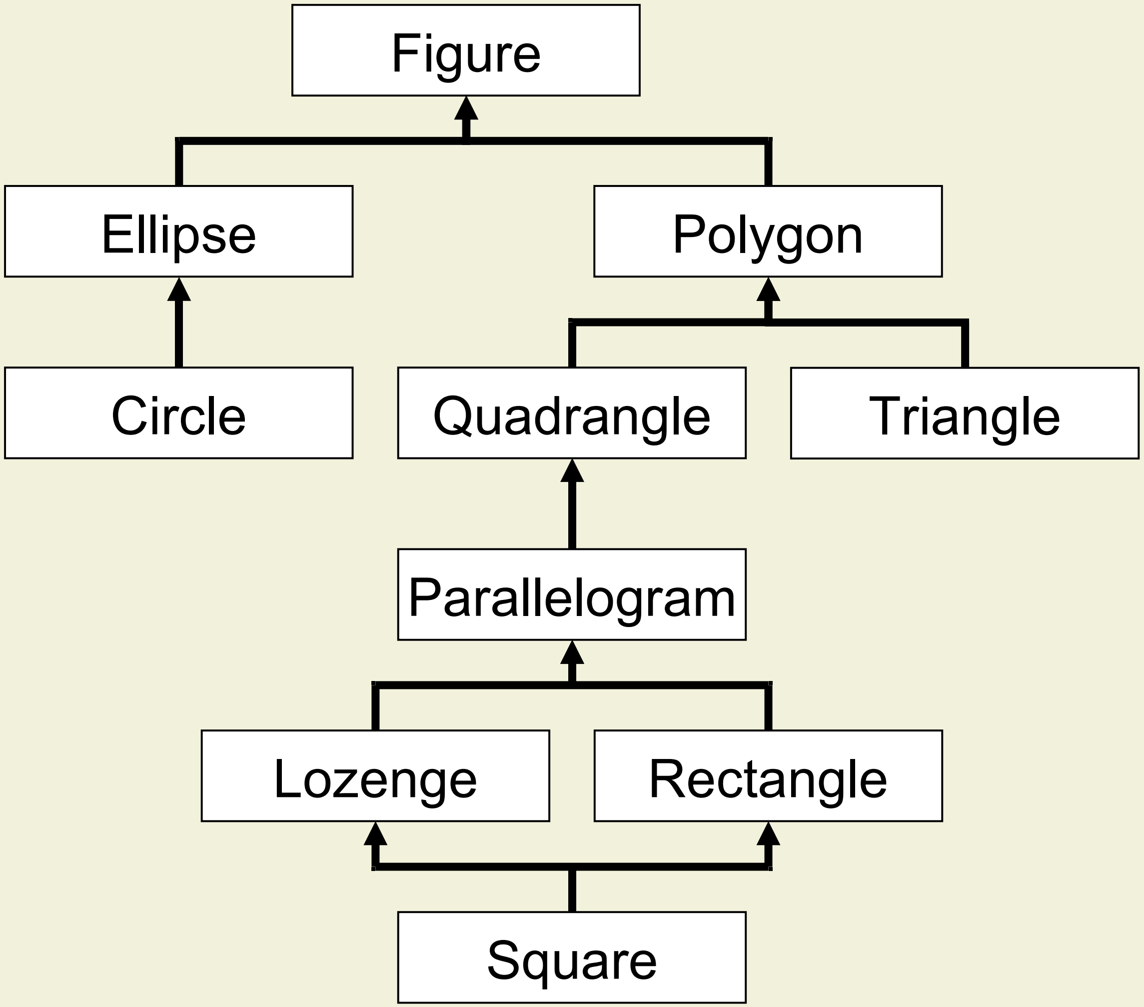
Classification

- Definition
  Classifying is a general technique to hierarchically structure knowledge about concepts, items, and their properties.
  The result is called classification.

Classification of Vertebrates



Classification of Figures



Arrows represent the ***"is-a" relation***

Goal: Apply classification to software artifacts

Characteristics of Classifications

- We can classify objects or fields
- Classifications can be **trees** or **DAGs**
- Classifications of objects form **"is-a" relation**
- Classes can be **abstract** or **concrete**

- Substitution principle
  Objects of subtypes can be used wherever objects of supertypes are expected

Classification in Software Technology

- Syntactic classification
  - Subtype objects have **wider interfaces** than supertype objects
  - Subtype objects can understand at least the messages that supertype objects can understand
- Semantic classification
  - Subtype objects provide **at least the behavior** of supertype objects

Interface Narrowing: Example

```
class Super {
  void foo( ) { … }
  void bar( ) { … }
}

class Sub <: Super {
  void foo( ) { … }
  // no bar( )
}
```

```
void m( Super s ) { s.bar( ); }
```

- Sub narrows Super’s interface
- If m is called with a Sub object as parameter, execution fails

Polymorphism

- Definition of *Polymorphism*:
  The quality of being able to assume different forms
  [Merriam-Webster Dictionary]

- In the context of programming:
  A program part is polymorphic if it can be used for objects of several types

Subtype Polymorphism

- Subtype polymorphism is a direct consequence of the substitution principle
  - Program parts working with supertype objects work as well with subtype objects
  - Example: printAll can print objects of class Person, Student, Professor, etc.
- Other forms of polymorphism (not core concepts)
  - Parametric polymorphism (generic types)
  - Ad-hoc polymorphism (method overloading)

Parametric Polymorphism: Example

```
class List<G> {
  G[ ] elems;
  void append( G p ) { … }
}
```

```
List<String> myList;
myList = new List<String>( );
myList.append( “String” );
```

```
myList.append( myList );
```

- Parametric polymorphism uses **type parameters**
- One implementation can be used for different types
- Type mismatches can be detected at compile time

Ad-hoc Polymorphism: Example

```
class Any {
  void foo( Polar p ) { … }
  void foo( Coord c ) { … }
}
```

```
x.foo( new Coord( 5, 10 ) );
```

- Ad-hoc polymorphism allows several methods with the **same name but different arguments**
- Also called **overloading**
- No semantic concept: can be modeled by **renaming** easily

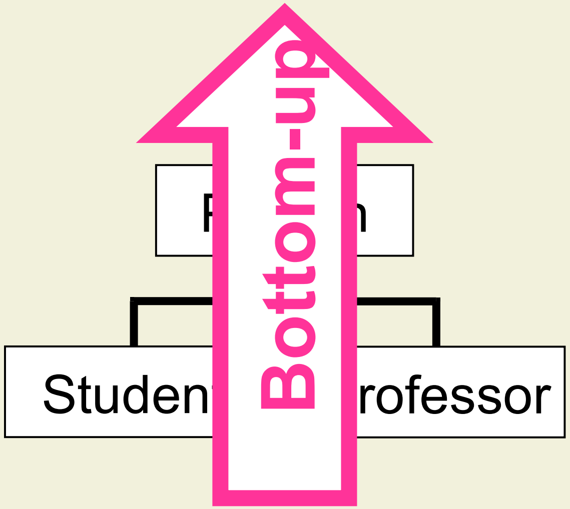
Abstraction

- Definition of *Abstraction*:
  A general concept formed by extracting common features from specific examples

  [WordNet, Princeton University]

Abstracting

- Start from different objects or types with common properties
- Develop a more abstract type, extracting the common properties
- Corresponds to making the interface smaller
- Program parts, that only rely on the common properties work for all objects of the more abstract type



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Example: Abstraction

```
class Student {
  String    name;
  int       reg_num;
  …

  void print( ) {
    System.out.println( name );
    System.out.println( reg_num );
  }
}
```

```
class Professor {
  String    name;
  String    room;
  …

  void print( ) {
    System.out.println( name );
    System.out.println( room );
  }
}
```

Example: Interface Types in Java

```
interface Person {
  void print( );
}
```

```
class Student implements Person {
  …
}
```

```
class Professor implements Person {
  …
}
```

Example: Abstraction (cont'd)

- Abstraction
  - Type Person with method print
  - Algorithm based on Person

```
Person[ ] p = new Person[4];
p[0] = new Student (…);
p[1] = new Professor (…);
…
for ( int i=0; i < 4; i++ )
    p[ i ].print( );
```

- Applications
  - I/O interfaces
  - Window systems
  - …

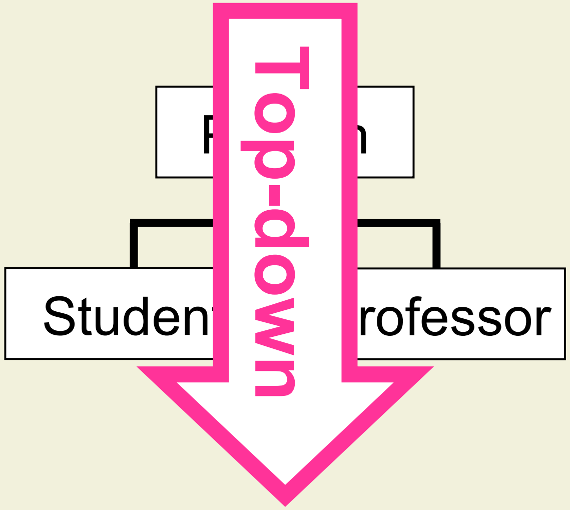
Specialization

- Definition of *Specialization*:
  Adding specific properties to an object or refining a concept by adding further characteristics.

- Example: Professional specialization

Specializing

- Start from general objects or types
- Extend these objects and their implementations (add properties)
- Requirement: Behavior of specialized objects is compliant to behavior of more general objects
- Program parts that work for the more general objects work as well for specialized objects
- Implementation inheritance, reuse



Example: Specialization

- Specialization
  - Develop implementation for type Person
  - Specialize it

```
class Person {
  String    name;
  …
    void print( ) {
    System.out.println( name );
  }
}
```

Example: Specialization (cont'd)

- Inheritance of
  - Attributes
  - Methods
- Methods can be overridden in subclasses

```
class Student extends Person {
  int        reg_num;
  …
  void print( ) {
    super.print( );
    System.out.println( reg_num );
  }
}
```

```
class Professor extends Person {
  String    room;
  …
  void print( ) {
    super.print( );
    System.out.println( room );
  }
}
```


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

2. Core and Basic Language Concepts

2.1 Core Concepts

2.2 Basic Language Concepts

- Description of Objects
- Inheritance
- Dynamic Method Binding
- Contracts

Description of Objects

- Class Concept
- Prototype Concepts

Class Concept

- Programs declare **classes instead of individual objects**
- A class is the **description of the common properties** of the class' objects
- During program execution, objects of declared classes are created (**instances**)
- Classes cannot be modified at runtime
- Class declarations correspond to record declarations in imperative languages

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Prototype Concept

- Programs **describe individual objects directly**
- New objects are created by **cloning** existing objects and **modifying** their properties at runtime
  - Cloning an object means to create a new object with the same properties (but different identity)
  - Modifying means adding attributes, or adding and replacing methods
- Use in the language **Self**

2. Core and Basic Language Concepts

2.1 Core Concepts

2.2 Basic Language Concepts

- Description of Objects
- Inheritance
- Dynamic Method Binding
- Contracts

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inheritance

- **Language concept** to support **reuse** of the properties and the code of another class
- Provides support for **specialization**
  - Adding attributes, methods, etc.
  - Overriding methods
  - Using overridden methods
- Objectives
  - **Avoid** error-prone **duplication of code**
  - **Reduce size** of programs
  - **Specialize** interfaces that cannot be copied or modified

Example: Inheritance

```
class Person {
  String name;
  …
  void print( ) {
    System.out.println( name );
  }

  String lastName( )    { … }
  Person( String n )    { name = n; }
}
```

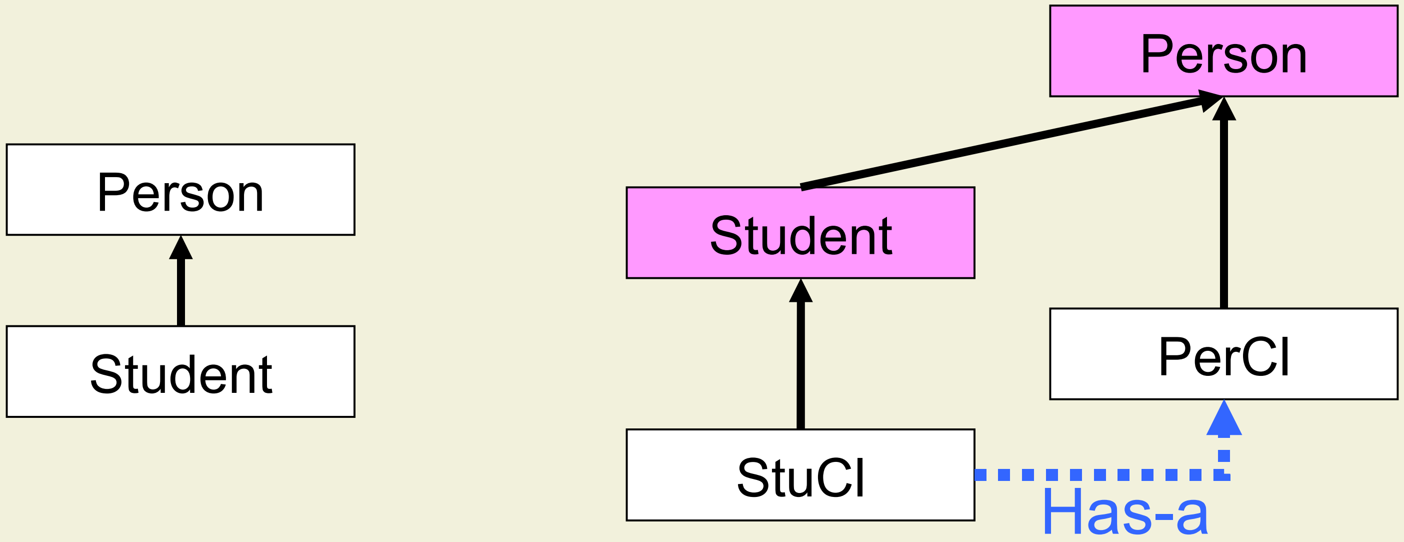
```
class Student extends Person {
  int reg_num;
  …
  void print( ) {
    super.print( );
    System.out.println( reg_num );
  }

  Student( String n, int rn ) {
    super( n );
    reg_num = rn;
  }
}
```

Inheritance versus Subtyping

- **Subtyping** expresses **classification**
- **Inheritance** is a means of **code reuse**
- Inheritance is **usually coupled** with subtyping
  - Terminology: **Subclassing** = Subtyping + Inheritance
- Issues
  - Subtyping without inheritance
  - Inheritance without subtyping

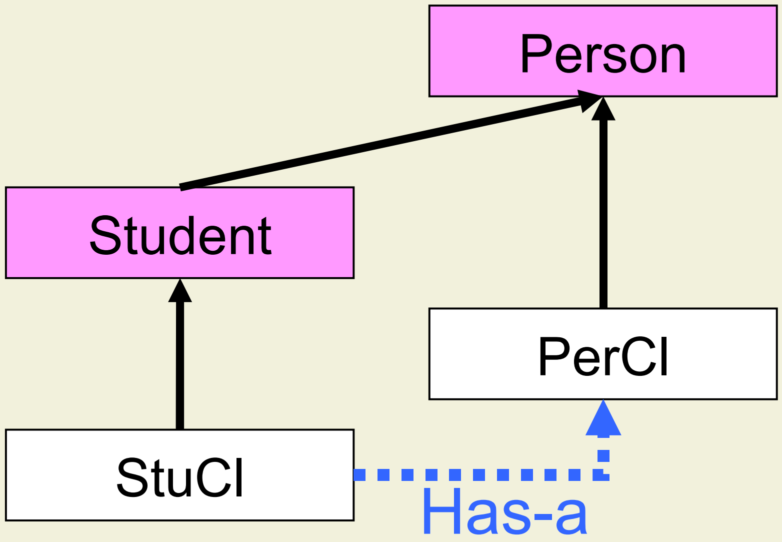
Subtyping without Inheritance

- OO-programming can do without inheritance, but not without subtyping and dynamic method binding
- Therefore, inheritance is **no core concept**
- Technique: **Delegation** instead of inheritance



Subtyping without Inheritance: Example

```
interface Person {
  void     print( );
  String   lastName( );
}
```




```
class PerCl implements Person {
  String name;
  …
  void print( ) {
    System.out.println( name );
  }

  String lastName( )
    { … }

  PerCl( String n ) {
    name = n;
  }
}
```

Using Delegation

```
interface Student extends Person {
  int getRegNum( );
}
```

```
Person[ ] p = new Person[4];
p[0] = new StudCl ("Winter",4711);
p[1] = new PerCl ("Lang");
…
for ( int i=0; i < 4; i++ )
    p[ i ].print( );
```

```
class StuCl implements Student {
  Person perPart;
  int reg_num;
  StuCl( String n, int rn ) { perPart = new PerCl( n );  reg_num = rn; }
  String   lastName( )    { return perPart.lastName( ); }
  int      getRegNum( ) { return reg_num; }
  void     print( )          { perPart.print( ); System.out.println( reg_num ); }
}
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inheritance without Subtyping

- Inheritance can be realized without subtyping
  - Example: Eiffel, Sather (a descendant of Eiffel)
- Using subclassing without establishing the “is-a” relation is problematic

```
class List {
  …
  void appendFront( Object o ) { … }
  void appendBack( Object o ) { … }
}
```

```
class Stack extends List {
  …
  // appendFront used as push
  void appendBack( Object o ) {
    System.out.println (“Should not
      be used!!”);
  }
}
```

2. Core and Basic Language Concepts

2.1 Core Concepts

2.2 Basic Language Concepts

- Description of Objects
- Inheritance
- Dynamic Method Binding
- Contracts

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Method Binding

- Static:
  At compile time, a method declaration is associated with every method invocation

- Dynamic:
  At runtime, the target object for the invocation is determined. The method to be executed is selected based on (the type of) the target object

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dynamic Method Binding

- Is necessary for **specialization**
- Is necessary for working with **abstract classes** and **interfaces**

```
class Super {
  int foo( int i ) { return i; }
}
```

```
class Sub extends Super {
  int foo( int i ) { return i + 10; }
}
```

```
Super s = new Sub( );
int r = s.foo( 5 );
```

2. Core and Basic Language Concepts

2.1 Core Concepts

2.2 Basic Language Concepts

- Description of Objects
- Inheritance
- Dynamic Method Binding
- **Contracts**

Interface Specifications

- Syntax
  - **Names** and **signatures** of methods
  - **Names** and **types** of attributes
  - Etc.
- Semantics
  - **Behavior** of methods
  - **Properties of** object **states**
  - Etc.

Method behavior

- **Preconditions** have to hold in the state before the method body is executed
- **Postconditions** have to hold in the state after the method body has terminated
- **Old-expressions** can be used to refer to values of the prestate in the poststate

```
class BoundedList {
  Object[ ] elems;
  int max; // maximum length
  int free;  // next free slot
  …
  // requires free < max
  // ensures elems[ old(free) ]=e
  void add( Object e )  { … }
}
```

Object states

- **Invariants** have to hold in all states, in which an object can be accessed by other objects
  - in each prestate of a method
  - in each poststate of a method

```
class BoundedList {
  Object[ ] elems;
  int max; // maximum length
  int free;  // next free slot
  /* invariant
     elems != null          &&
     free >= 0              &&
     free <= elems.length   &&
     elems.length==max
     */
  …
  // requires free < max
  // ensures elems[ old(free) ]=e
  void add( Object e )  { … }
}
```

Contracts and Subtyping

```
class Number {

  int n;
  // invariant true

  // requires true
  // ensures n == p
  void set( int  p )
     { n = p; }
  …
}
```

```
class UndoNaturalNumber
              extends Number {
  int undo;
  // invariant n > 0 && undo > 0

  // requires p > 0
  // ensures n == p && undo == old ( n )
  void set( int  p )
     { undo = n; n = p; }
  …
}
```

- Subtypes often have to **adapt** the **contracts** of their supertypes

Rules for Subtyping: Preconditions

```
class Super {
  // requires 0 <= n < 5
  void foo( int n ) {
    char[ ] tmp = new char[ 5 ];
    tmp[ n ] = 'X';
  }
}
```

```
class Sub extends Super {
  // requires 0 <= n < 3
  void foo( int n ) {
    char[ ] tmp = new char[ 3 ];
    tmp[ n ] = 'X';
  }
}
```

```
void crash( Super s ) {
  s.foo( 4 );
}
```

```
x.crash( new Sub( ) );
```

- Subtype objects must **fulfill contracts** of supertypes
- Overriding methods of subtypes can have **weaker preconditions** than corresponding supertype methods

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Rules for Subtyping: Postconditions

```
class Super {
  // ensures result > 0
  int foo( ) {
    return 1;
  }
}
```

```
class Sub extends Super {
  // ensures result >= 0
  int foo( ) {
    return 0;
  }
}
```

```
void crash( Super s ) {
  int i = 5 / s.foo( );
}
```

```
x.crash( new Sub( ) );
```

- Overriding methods of subtypes can have **stronger postconditions** than corresponding supertype methods

Rules for Subtyping: Invariants

```
class Super {
  int n;
  // invariant n > 0
  Super( )       { n = 5; }
  int crash( )   { return 5 / n; }
  }
}
```

```
class Sub extends Super {
 // invariant n >= 0
  Sub( ) {
    n = 0;
  }
}
```

```
new Sub( ).crash( );
```

- Subtypes can have **stronger invariants**

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contracts and Subtyping

```
class Number {

  int n;
  // invariant true

  // requires true
  // ensures n == p
  void set( int  p )
     { n = p; }
  …
}
```

```
class UndoNaturalNumber
            extends Number {
  int undo;
  // invariant n > 0 && undo > 0

  // requires p > 0
  // ensures n == p && undo == old ( n )
  void set( int  p )
     { undo = n; n = p; }
  …
}
```

- Subtypes often have to **adapt** the **contracts** of their supertypes

Rules for Subtyping: Summary

- Subtype objects must **fulfill contracts** of supertypes, but
  - Subtypes can have **stronger invariants**
  - Overriding methods of subtypes can have
    weaker preconditions
    stronger postconditions
    than corresponding supertype methods

- Concept is called **Behavioral Subtyping**
- Consequence of substitution principle