Introduction to Software Engineering

The Specification Phase

2.1 Object Orientation and UML class diagrams

Prof. Walter F. Tichy

ქუთაისის
საერთაშორისო
უნივერსიტეტი

Definition of “Object” and “Class“

- Let G be the the union of all past, present, and future substantial „things“ and concepts (G is called the universal set)
- For example, G contains:
  - Persons: Prof. Tichy, you, , … (substantial)
  - Air (substantial)
  - My life insurance contract (conceptual)
  - Democracy (conceptual)
  - European Soccer Championship (Event → conceptual)
  - Swimming (activity → conceptual)
  - Swimming (capability → conceptual)
  - Blue (color/property → conceptual)
  - Software (conceptual)
  - etc.

Object

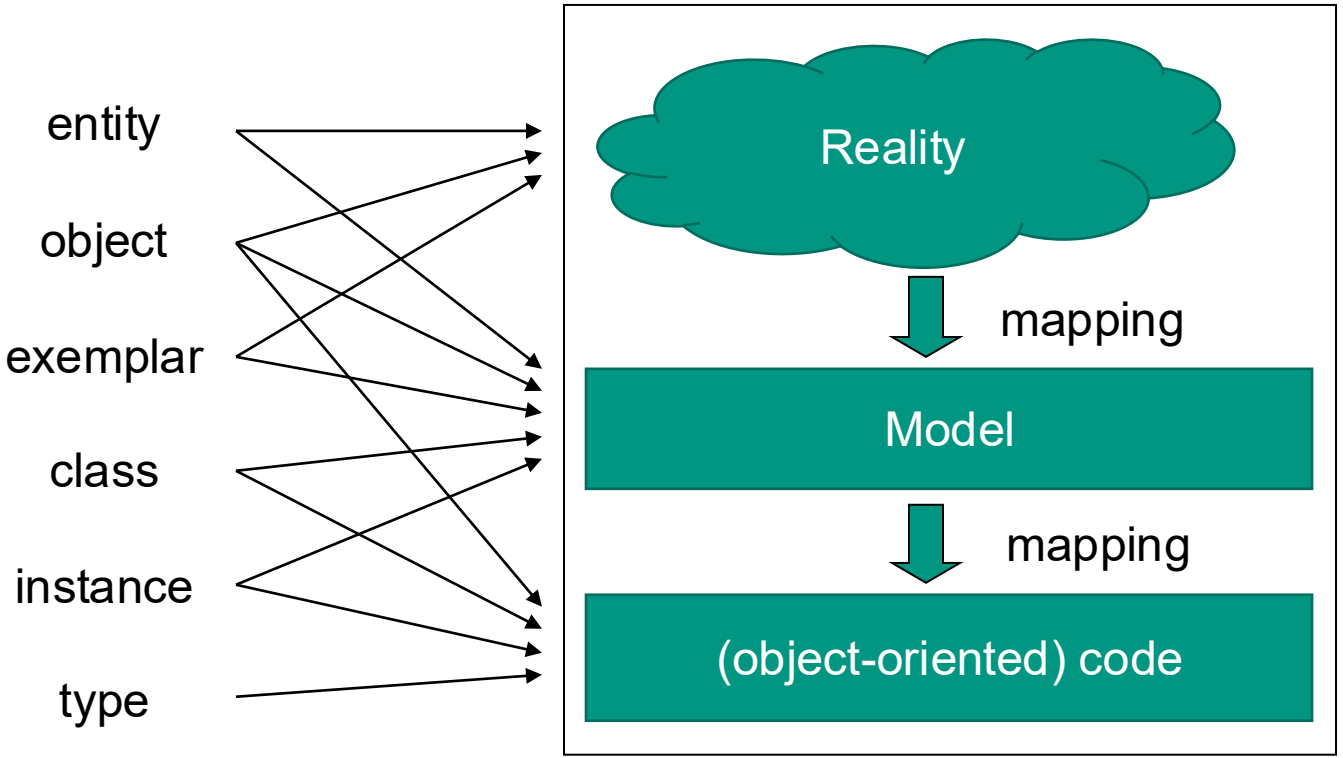
- Def. **Object**: An element out of the set G that is recognizable and clearly distinguishable from others by at least one individuum.
- Def. **Ω**: The set of all objects. ($\Omega \subset G$)

- Exercise: Compare this definition with the following:
  - Everything that can be tagged with a noun or a name.
  - Charles S. Peirce: „By an object, I mean anything that we can think, i.e. anything we can talk about.“

Class and Exemplar

- Def. **Class**: An arbitrary category over the set of all objects Ω.
  - Notes
    - „category“ is derived from „praedicare“; one states what is part of the category and what is not.
    - Usually, a category results in some form of sameness of the objects in the category.
    - A category can also be empty. A class states the basic idea or concept of objects, independent of their existence.
- Def. **Exemplar**: A concrete element of a given class.
  - An exemplar is member of at least one class.
  - „Instance“ is a synonym of exemplar

In which domains are these terms usually used?



Further hints

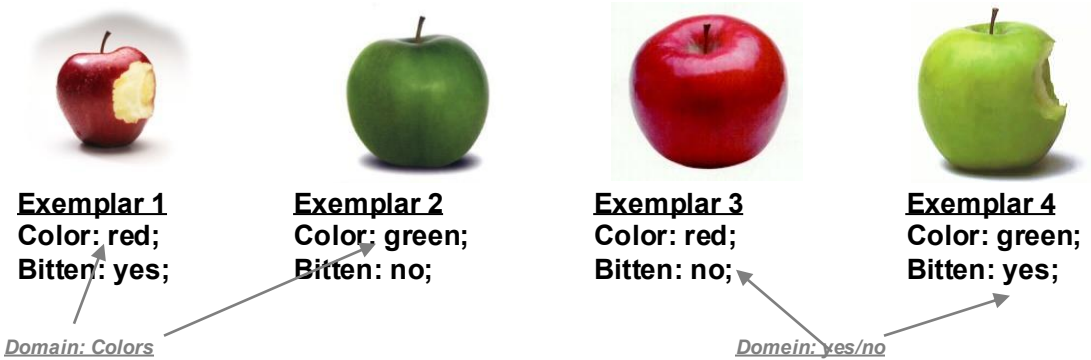
- The term “object“ is often used in the specification phase as a representative for a whole class.
  For instance, we may use an object (“Bob“) instead of “class person“.
- Compare modulo arithmetic, where we use the remainder as representatives for the entire class. (˜1 is the class of integers whose remainder after division by 4 is 1.)

$$\mathbb{Z}_4 = \{\tilde{0}, \tilde{1}, \tilde{2}, \tilde{3}\}$$

- Use the term „class“ if the class has been defined and the term „instance“ when an element of a given class is meant.

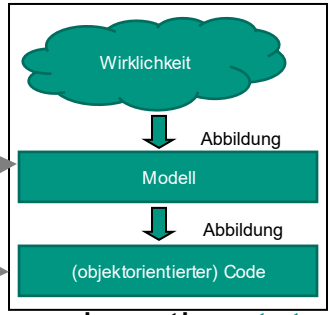
Attributes

- Def. **Attribute**: a property that is defined and present in all exemplars of a class
  - its value can be specified for every exemplar independently of other exemplars and
  - has a clearly defined value and
  - the value is from a defined domain that is the same for all exemplars of that class
  - Notation: `Attributename: Domain [= value];`



Hints for Java programmers

- Differentiate between attribute and instance variable!
  - Often there is a 1:1 mapping of attributes to instance variables



  - But the reverse is not true. Often, we need to store non-attributes such as the state or associations in instance variables. More about this later.
  - Attributes may have additional constraints or assertions which cannot be expressed by the domain of the attribute but requires additional code. (In UML, one can specify this with the Object Constraint Language OCL)

Object identity

- Def. **Object identity:** The existence of an object is independent of its attribute values. Two objects are distinguishable even when they have the same attribute values.

  Hints
  - Object identity is already given by our defintion of attributes.
  - Two (or more) objects can be equal, without being the same. How to define equality then?

Comparing objects

- Def. **Equality of order n**
  - Equality of order 0: the objects are identical (there is only one object)
  - Equality of order 1 : the objects are identical, or the objects are different but have pairwise identical values for all attributes (equality of order 0 or pairwise equality of order 0 in all attributes). Example: apples previous page.
  - Equality of order 2 : equality of order 1 or pairwise equality of order 1 in all attributes. Example: 2 bags of apples of equality order 1.
  - Equality of order 3 : equality of order 2 or pairwise equality of order 2 in all attributes*.
  - etc.

* Of course we need to take into considerations associations and state as well (see later)

What has this to do with classes in programming languages?

- With programming languages, we can only describe classes and objects that contain information and methods.
- For example, in Java,
  `class IntPair {int x,y}`
  describes the set that consists of all objects with *at least* two integer attributes called x and y.
- Obviously, `IntPair` is a subset of the universal set G.
- What about
  `class IntTriple extends IntPair {int z}` ?

- Note the “at least” statement. `IntTriple` is therefore a subset of IntPair. IntPair is consists of all objects containing `int x,y` and potentially more.
- Same idea applies to methods, but there’s more about signatures later.

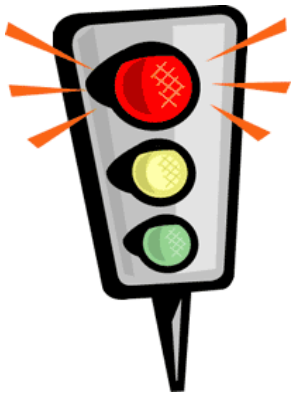
Substitution of objects

It should be possible to use `IntTriple` in the same context as `IntPair` (where only x and y are used). This is called substitution of a subclass object for a superclass object.

```
IntPair pair = new IntPair();
pair.x = 0; pair.y=1;
…
IntTriple triple = new IntTriple();
pair = triple;                          // substitution permitted?
print(pair.x, pair.y);                  // permitted?

//the other way around
triple = pair;                          //permitted?
print(triple.z);                        //permitted?
```

State of objects

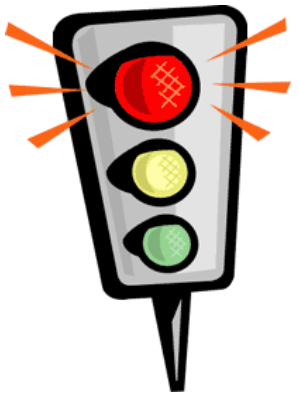
- Often, the state of objects is defined as the concatenation of all attribute values, or simply as the sequence of bits which store the attribute values.
- This definition is problematic: For example, is the value of the attribute „operating hours“ important for the state of a traffic light?



State of objects

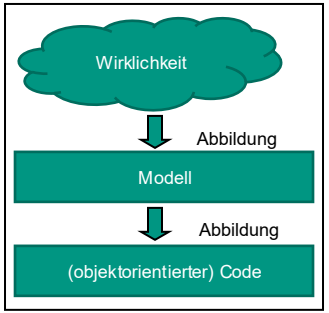
- Def. **State**: As long as an object is in a certain state and in the same calling or use conext, it will always react in the same way. If the state changes, the object will react in at least one context in a different way. (external view)

Exercise: Compare this definition with::
„**state** - A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.“
OMG, UML 2.4 Infrastructure Specification, Ch. 4, Terms and Definitions



Hints for Java programmers

- The state of an object must be saved in instance variables (in the same way as attribute values)



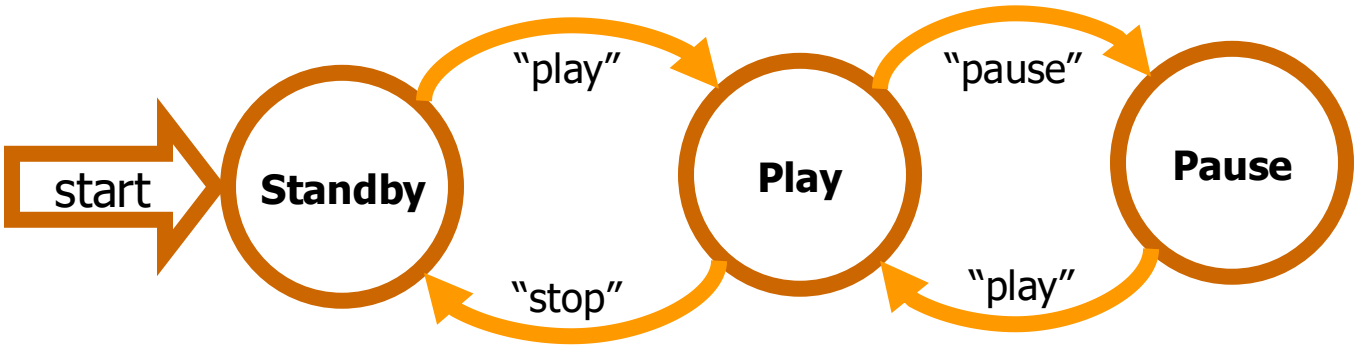
- For this we can use
  - dedicated variables(explicit state)
  - or the state can be computed from the values of other instance variables (implicit state)
    - Example: compute the state of the right to vote from birth date

Encapsulation

- Def. **Encapsulation:** The state is visible from the outside, but is controlled inside the object (controlled change)
  - Example: the state of a traffic light should not be changed arbitrarily or at arbitrary times
  - **Hint:** if the state is implicit, the change of an attribute value may cause a change of state.
  - Example: For an egg timer, if the “remaining time” changes to 0, the egg timer must change into state “ringing”.

Change of state

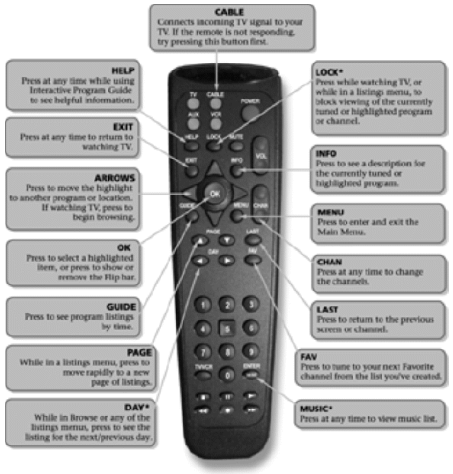
- Example encapsulation principle: You shouldn´t fiddle with the interior of a recorder to set it into the state „play“



Sending a message to an object

- „Sending a message“ means that a certain object (the recipient of the message) should perform a state change, or perform an action in the current state.

- Therefore: Sending a message = method call on a certain object.
- Example: The play button sends the message „Change into the `play´ state“ to the device.

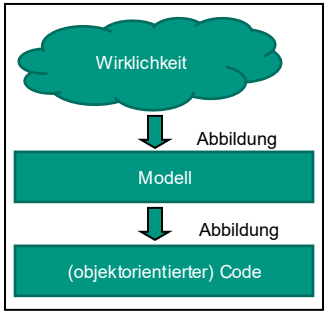


Methods and state changes

- Methods may change the state of an object.
  - The available methods define the permissible state changes that can be requestes of an object (external view)
  - Problem: When may I send which message to an object?
    - Answer: This is defined with a state diagram or state chart (see later)

Hints for Java programmers

- The state diagram specifies a finite automaton.
- Is this automaton complete?
  - What happens if I send a message X in state Y, and this was not anticipated?
  - Choices
    - Return an error
    - Throw an exception
    - Generate Java.lang.Error-Instance
    - „garbage in – garbage out“
    - Ignore
    - crash



Methode signatures

- Def. A **Method signature** consists of
  - *Method name*
  - *Return type*
  - *Parameter list*
    - The parameters carry the data of the message
    - The receiving object (*o* in method call *o.m()*) can be viewed as the “zero’th“ Parameter of the call, or as address of the message.

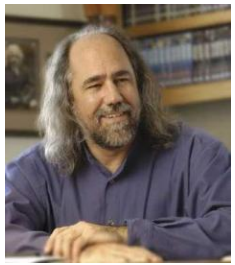
- Notation:
  `Methodname (Parameterlist) : Returntype;`
- Parameterlist:
  `Parametername : Type [, Parametername : Type]*`

- Paramterlist and Returntype may be empty.

UML Class Diagrams

What is UML?

- UML: „Unified Modeling Language“
- UML is the union of three object-oriented modeling formalisms:
  - Booch (Grady Booch)
  - OOSE (Ivar Jacobson)
  - OMT (James Rumbaugh)
- UML is standardized



Grady Booch

Ivar Jacobson

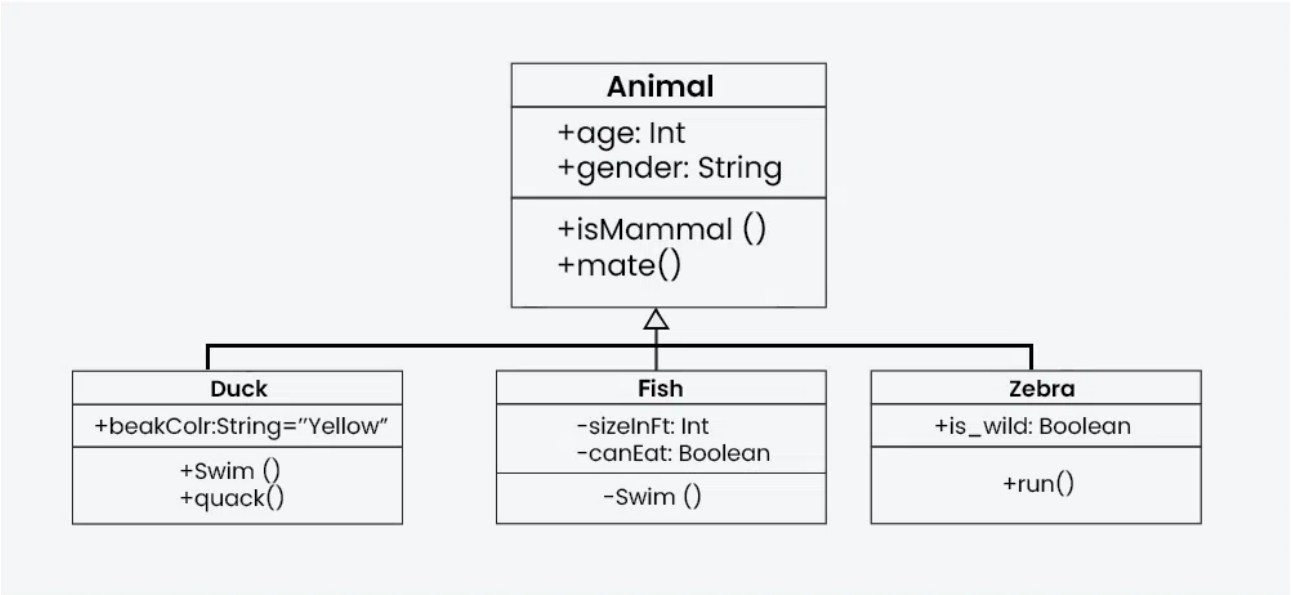
James Rumbaugh

What is a UML class diagram?

- Class diagrams are a type of UML diagram used to **visually represent the structure and relationships of classes** within a system.
- They provide a high-level overview.
- They are used for constructing, documenting, understanding, and communicating the structure of object-oriented software systems
- They are used as blueprints (or plans) for building new systems.
- They are a fundamental tool in object-oriented design.

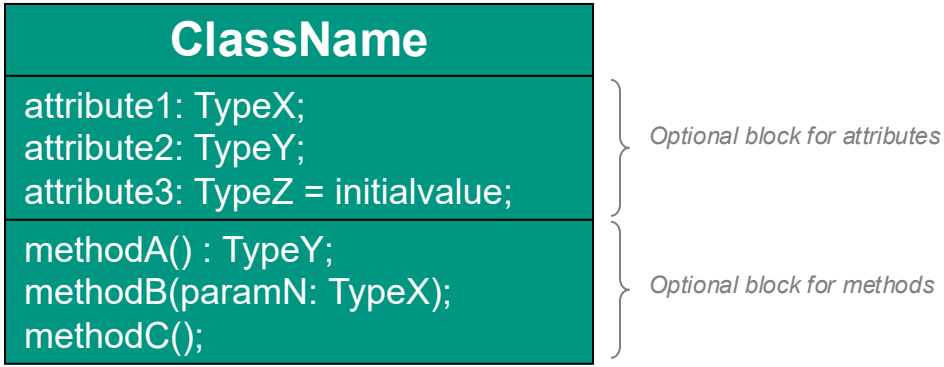
What is a UML class diagram?

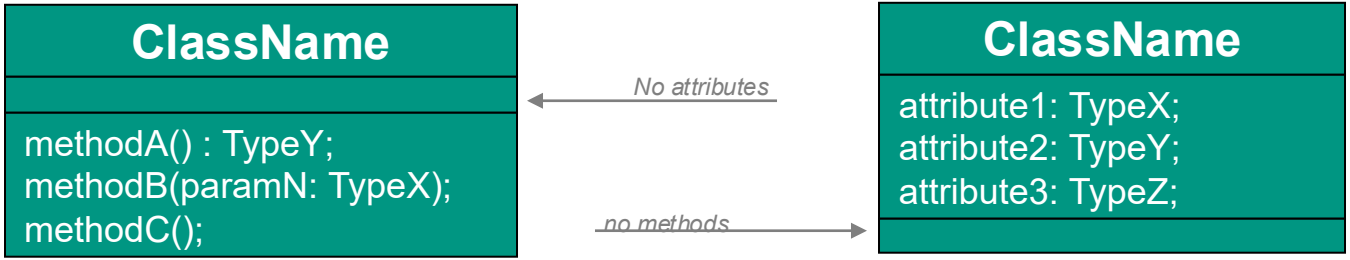
- Classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods.
- Lines connecting classes illustrate relationships such as one-to-one or one-to-many.



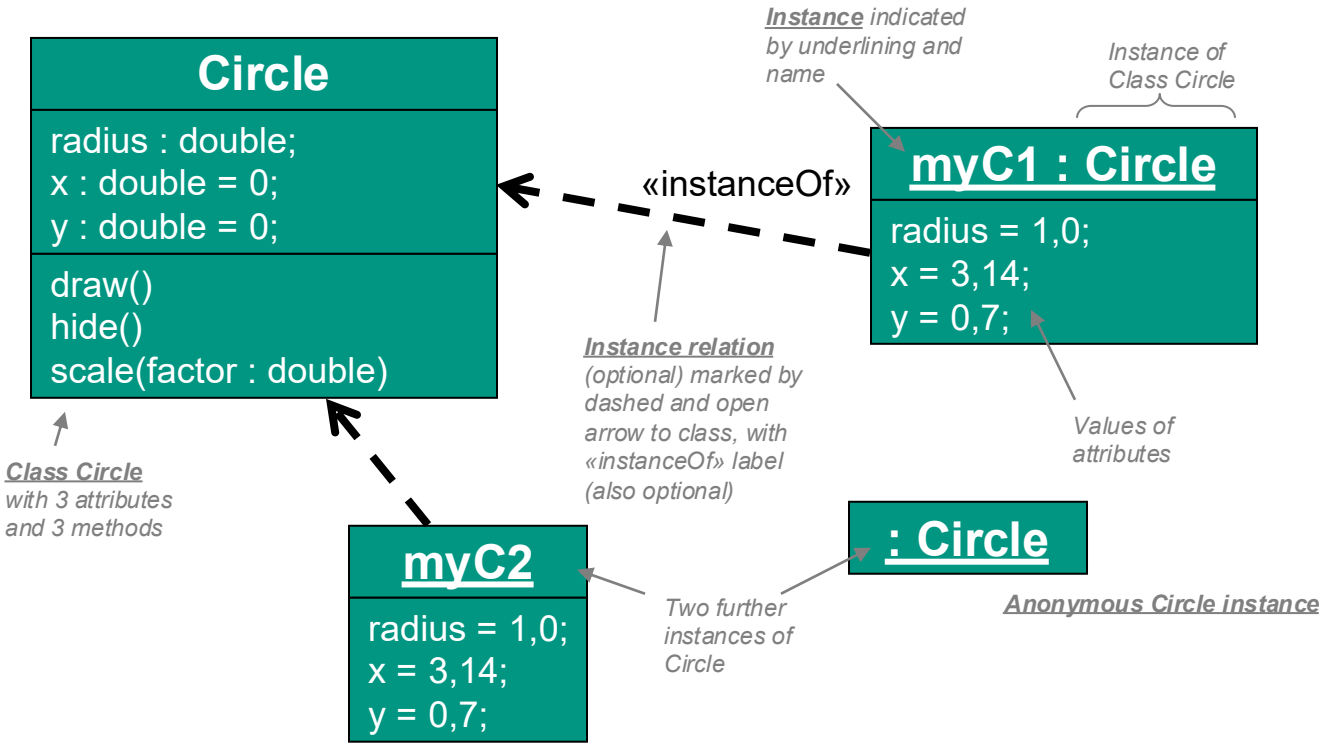
Class Diagram example

Notation of a class in UML



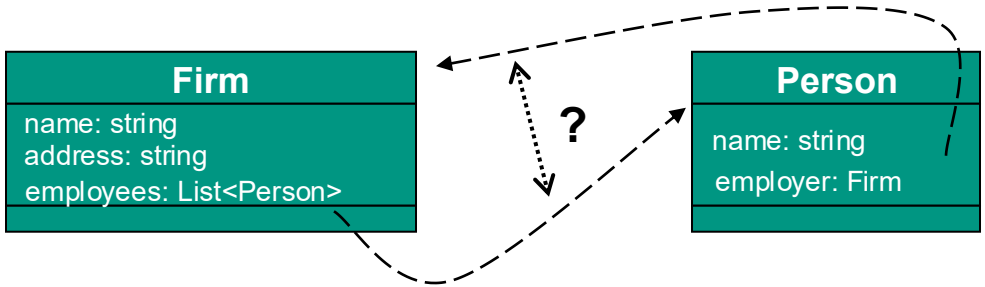


Object/Instance diagramm

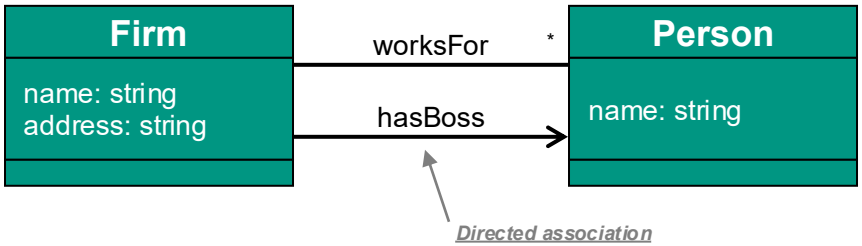


Associations: Defining Relations between Objects

- Motivating example:
  - If we want to express that a person works for a certain firm, we could add an attribute “Firm“ in class “Person”.
  - If we also want to know which persons work at a given firm, we would need an attribute for a list of persons in class “Firm” (`List<Person>`).
- Problem: How do you express that these two attributes belong together? If you change one, you may have to change the other.

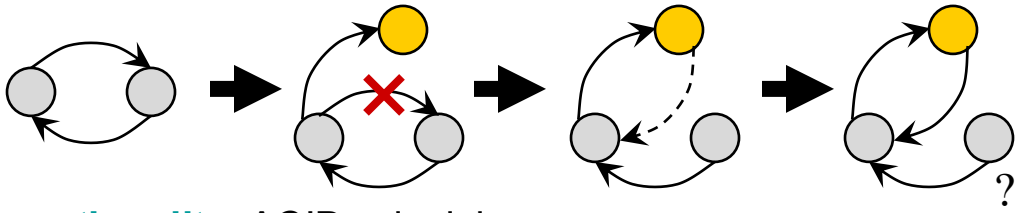


Answer: Use an association



Note: Class diagrams are multigraphs, which means that more than one Association may exist between a pair of nodes.

Added Value of Associations



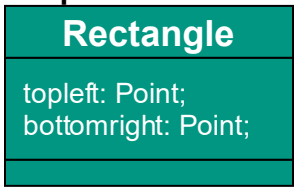
Transactionality, ACID-principle

- **A**tomicity – Indivisibility of changes, “all or nothing“
- **C**onsistency – Changes results in a consistent state (target state or, in case of abortion, in starting state)
- **I**solation – Changes are not affected by concurrently running, other changes
- **(D**urability – After completion, changes are permanent and visible for all other threads—not guaranteed for main memory)

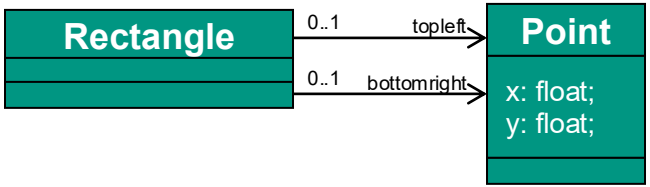
When to use Associations and when to use Attributes

- Example:*

a)



b)



- Do I need transactionality? → Association
- Do I need several of the same attributes? → Association
- Navigation from attribute to object desired? → Association
- Otherwise: your choice
  - Preferred style: attributes only for primitive types (int, float, string, bool, …)

* Two points are enough to locate rectangles in axis-parallel orientation

The Connection between Associations and Relations

- Assume there are two sets X and Y and x_i is an element of X, and y_j and element of Y. Then the cross product is the set of all pairs (x_i , y_j) for all i and j.
- A binary relation is a subset of the cross product of two sets (the sets need not be different)
- Ternery, quaternary, n-ary relations are subsets of the cross products of 3, 4, n sets.
- Relations can be represented as n-dimensional arrays, pairs (only for binary relations), tuples, or graphs
- A „link“ is a single elment of a cross product, a tuple, or an edge in a graph. (in n-ary relations, an edge has more than two ends, in case n>2)
- For more information, look up “binary relations” in Wikipedia.

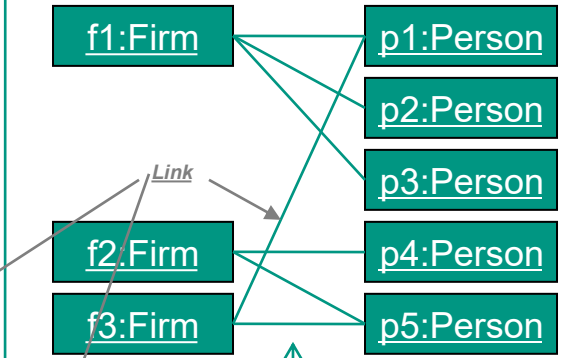
Example of a binary relation, in three representations

Let Firms = {f1, f2, f3} und Persons = {p1, p2, p3, p4, p5} (sets)
A possible relation of the two sets could be:

Person

p5		✓	✓
p4		✓	
p3	✓		
p2	✓		
p1	✓		✓
employes	f1	f2	f3

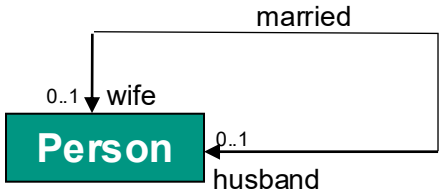
Firm



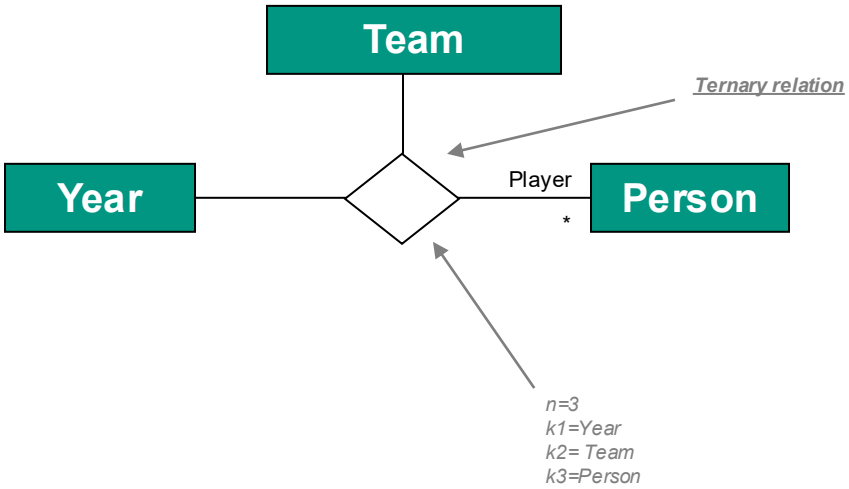
The relation in the array above can also be specified as a tuple set:
{(f1,p1), (f1,p2), (f1,p3), (f2,p4), (f2,p5), (f3,p1), (f3,p5)}, or as a graph.
A single tuple or an edge is called a link.

Then what is an association?

- Def. An **association** in UML defines the properties of a relation between sets.
  - The sets are given as classes.
  - Multiplicities specify, in how many tuples an object of a given class may appear. (for instance: 1:1 means one-to-one, 1:n one-to-many, m:n many-to-many)
  - Duplicates of tuples or duplicate of links are allowed, so relations are actually not only sets, but can be multisets or multigraphs.
  - The classes in associations need not be different (relation with self is possible).

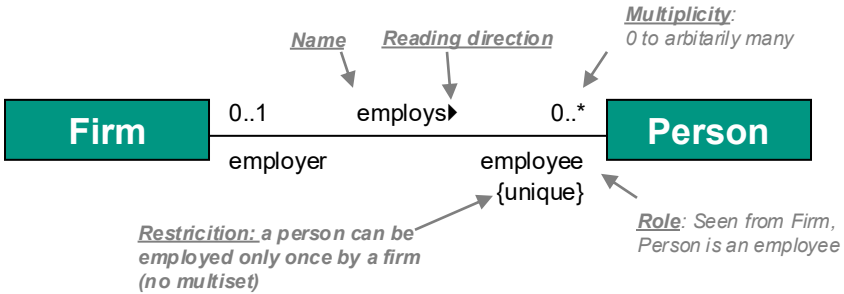


Example of a ternary association (Hypergraph)



Associations in UML

- A relation is specified by an association in UML

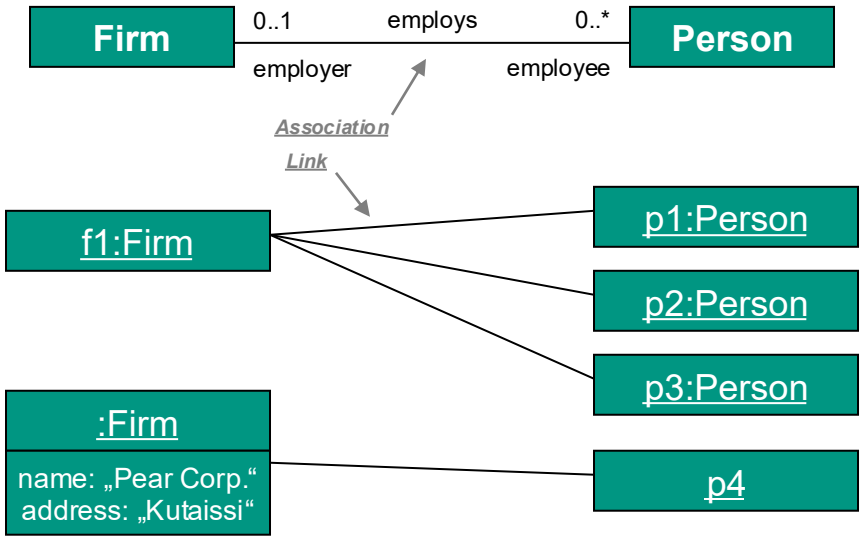


- Name, reading direction and roles are only tags to help with interpretation. They have no exactly defined semantics.

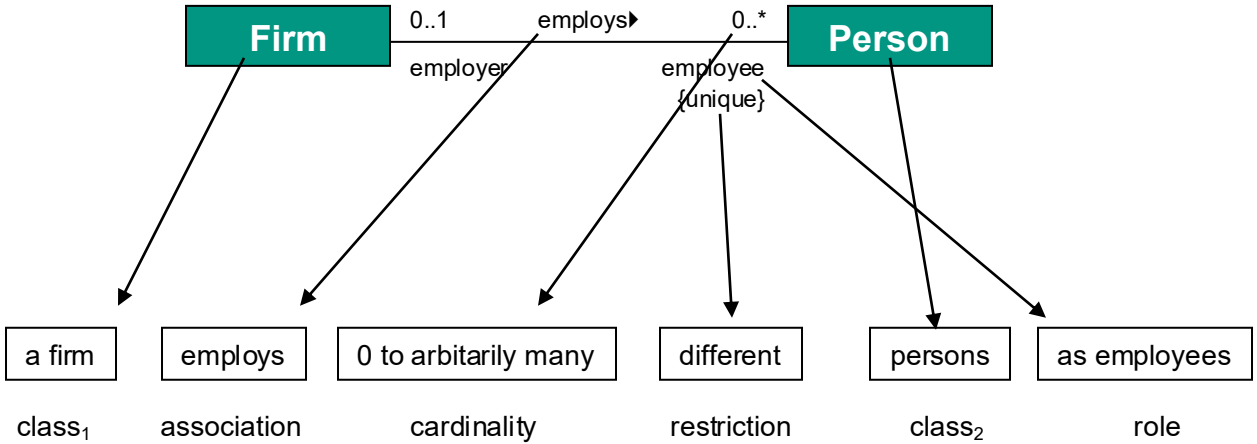
Association vs. Link

An association connects classes and specifies **potential relations** between examplars

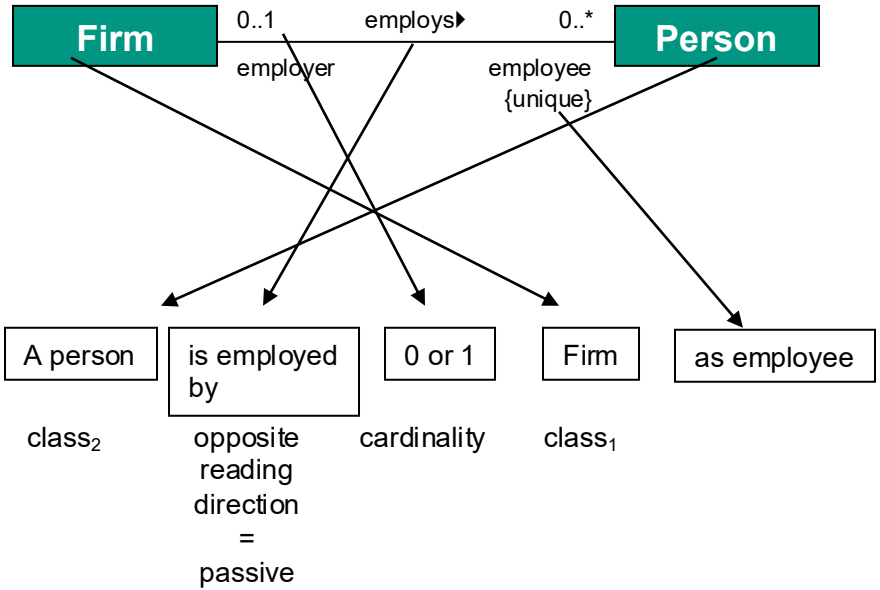
A link connects examplars and expresses an actual relation between exemplars.



How to read associations



How to read associatins



Cardinalities at the ends of associations

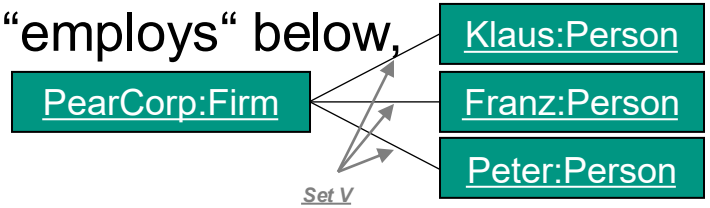
- Def. **Cardinality**: the number of elements of a set (→ integer ≥ 0)
- Def. **Multiplicity**: an interval of specified cardinalities
  - “0..1“ means “0 or 1“
  - “0..*“ = “*“ means “arbitrarily many, including 0“
  - “1“ means “always exactly 1“
  - “1..*“ means “arbitrarily many, but always at least 1“
  - “17“ means “always exactly 17“
  - Careful: missing cardinality means “always exactly 1“.

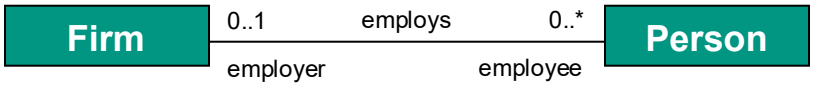
Interpreting Multiplicity

- Suppose we are given an n-ary relation and an end E with multiplicity p. We want to define what multiplicity p at end E means.
  - Let K be the class at end E of the association
  - For the other n-1 ends choose arbitrary but fixed instances of the corresponding classes (more than n-1 possible)
  - Let V be the set of links which go from the n-1 chosen instances to instances of K.
  - Let M be the set of instances of class K included in the above links.
  - Then the multiplicity p restricts the cardinality of M.

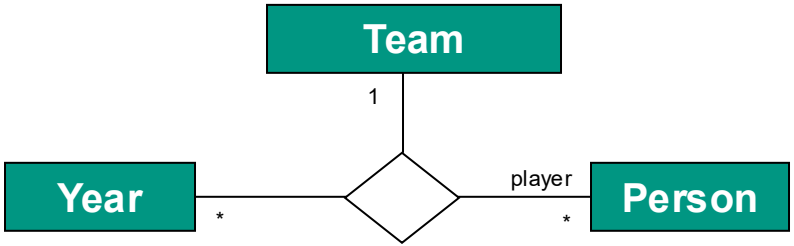
Example

- Given the binary (n=2) association “employs“ below, with end E=“employee“ and p=0..*
  - Then K=Person
  - For the other end, choose instance PearCorp : Firm
  - Find all links from PearCorp to an instance of class Person
  - Let V={(PearCorp, Klaus), (PearCorp, Franz),(PearCorp, Peter)}
  - Then M ={Klaus, Franz, Peter}
  - The multiplicity p=(0..*) limits the cardinality of M





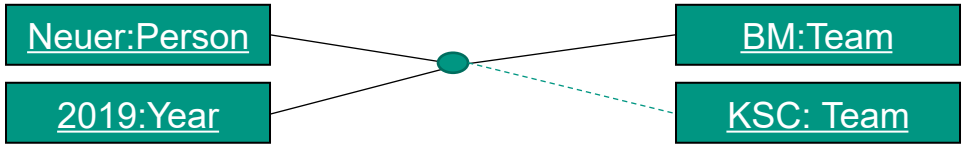
Where do we need this?
n-ary Associations!



How to interpret p=1 at end Team?

N-ary multiplicity

- K = Team
- Choose instances of Person and Year and look at tuples that include these instances plus instances of Team.



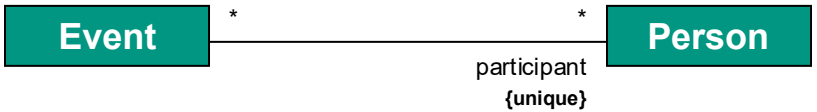
Is another team allowed for Neuer and 2019?
And for 2020?

N-ary Associations and multiplicity

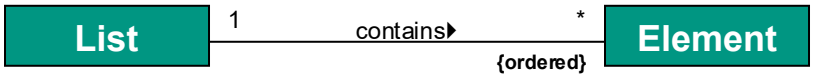
- In the above ternary association we have E/K = Team, p = 1
  - Choose 2002:Year, MiroslavKlose:Person, MarioBasler:Person and HarryKoch:Person
  - Let V be the set of ternary relations that go from instances of Year and Person to instances of class Team.
  - Then M is the set of instances of Team occuring in V
  - p restricts the cardinality of M
- This means that the Persons MiroslavKlose, MarioBasler und HarryKoch play in exactly one team each (not necessarily the same team) in 2002.

Example for Restrictions

- A person can participate in arbitary many events, arbitary may persons can participate in any event, but in each event only once:



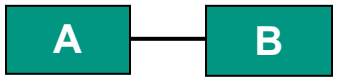
- The elements of a list have an order:



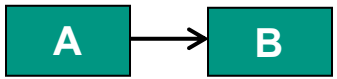
Navigation of Relations

- Exemplars may only send messages to other exemplars if they „know“ them (have links to them). A link makes objects known to each other. In other words, links establish message channels.

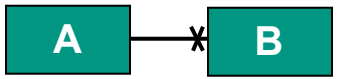
- Navigation specifies in which **direction** the messages can traverse a link



Not specified. Traversal in both direction potentially possible, but not guaranteed.



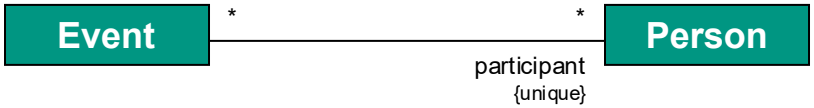
Traversable from instances of A to instances of B.



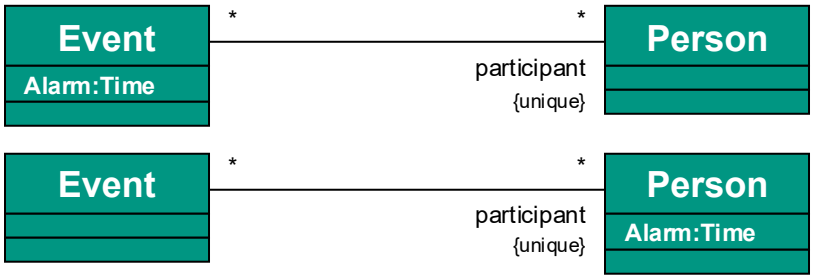
Not traversable from instances of A to instances of B.

How to store information about relations?

- Sometimes one would like to store information about a relation.



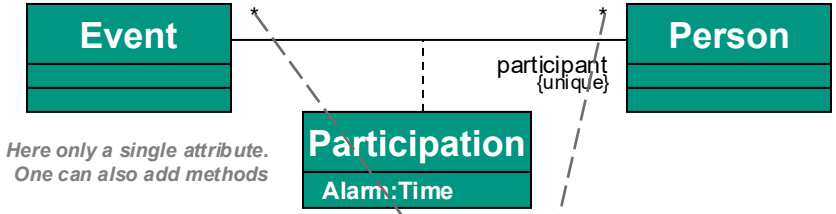
- A participant wants to enter an alarm for his/her events:



Wrong: Alarm applies to all persons

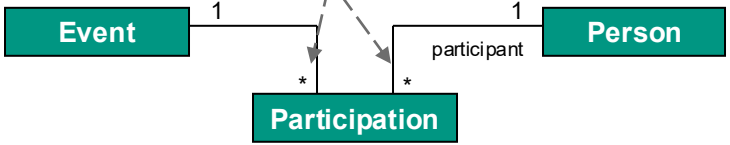
Wrong: Alarm applies to all events of a person

Answer: Association classes put attributes in relations

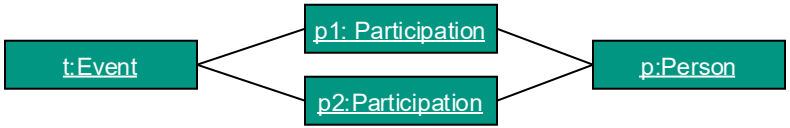


Note: class diagrams are multi-hyper-graphs with attributed nodes and edges!

- Convention: assoc. class name=name of association
- Can be simulated with normal classes:



- But:



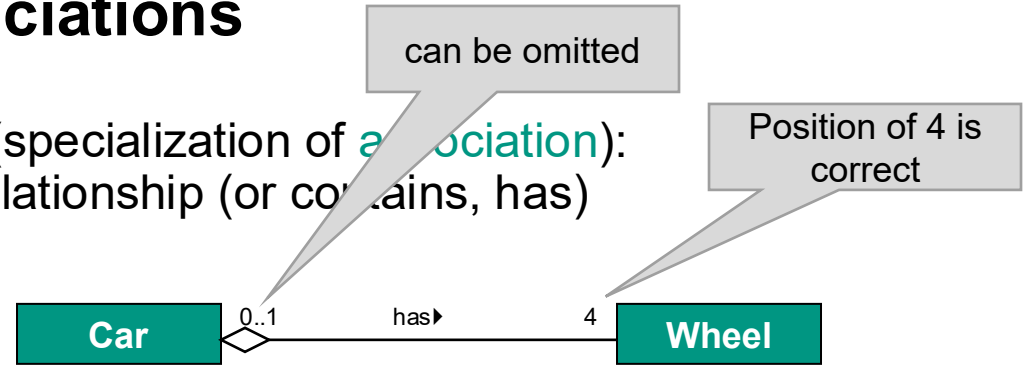
This is legal under the above model, but does not enforce uniqueness

Association classes

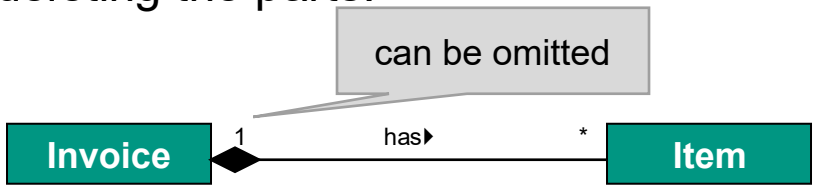
- Hint:
  - The existence of an instance of an association class depends on the existence of its relation!
  - If the relation is deleted, then the instance of the association class with all its data also disapears.
  - Also: If one end of a relation is deleted, then the whole relation is deleted, and the instance of association class with it!
- Conclusion: Don´t use association classes like normal classes!

Special associations

- Aggregation (specialization of association):
  Part-whole relationship (or contains, has)

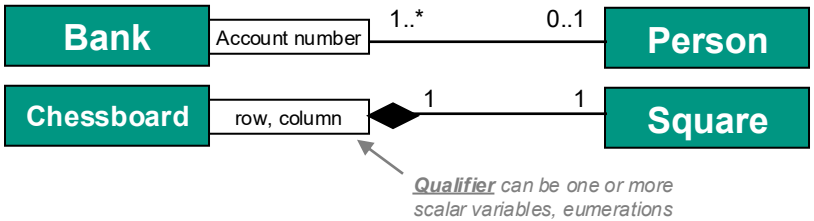


- Composition (specialization of aggregation):
  More constraint: Parts cannot exist without the whole. This is important when creating and deleting the parts.



Qualified associations

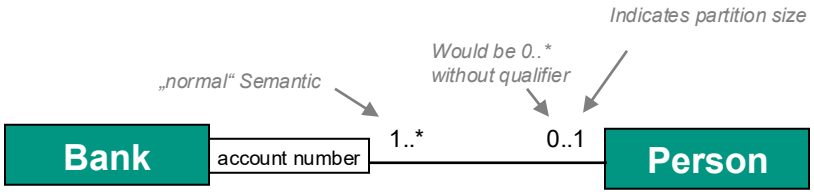
- Def. **Qualifier**: An attribute (or combination of attributes) that partition the set of associated exemplars
- Def. **Qualified assoziation**: An association where the set of associated elements is partitioned by a qualifer.
- Examples



Multiplicity in qualified associations

- Qualified associations always code 1:n or m:n relations, (why?)
  - The (frequent!) Partitionsize „1“ requires a unique qualifier

- Interpretation of multiplicities:



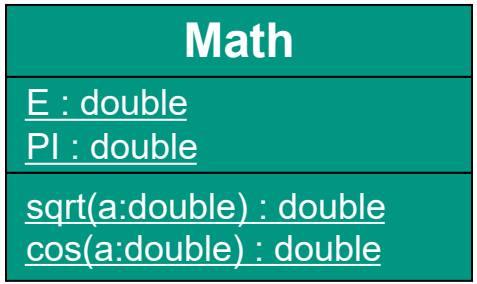
Multiplicity and qualification – Example 1

- At bank X there is no or maximally one person assigned to account number Y.
- Nobody or maximally 1 Person owns an account with number Y at Bank X.
- A person must have at least one account number with at least one bank.
- A person may have any number of account numbers at a bank, an account number at any number of banks, or any number of account numbers at any number of banks.

Multiplicity and qualification – Example 2

- At every bank, every account number Y is assigned to at least one person.
- Several persons can share the same account number at a bank.
- A person need not have an account number at any bank.
- A person can have arbitrarily many accounts at arbitrarily many banks.

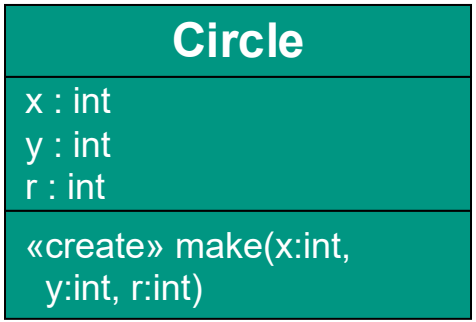
Class attributes and class methods



- The class as set of exemplars is itself viewed as an object. This object can have its own attributes and methods (independent of the attributes and methods of the exemplars)
- These attributes and methods are called class attributes and class methods.
- They are marked by underlining them in the class box. (in Java, they are marked with the keyword static.)
- Class attributes and class methods exist independent of the existence of exemplars of the class.

- The class attributes and methods are available at runtime in any exemplar of the corresponding class and can be used exactly like normal attributes and methods.

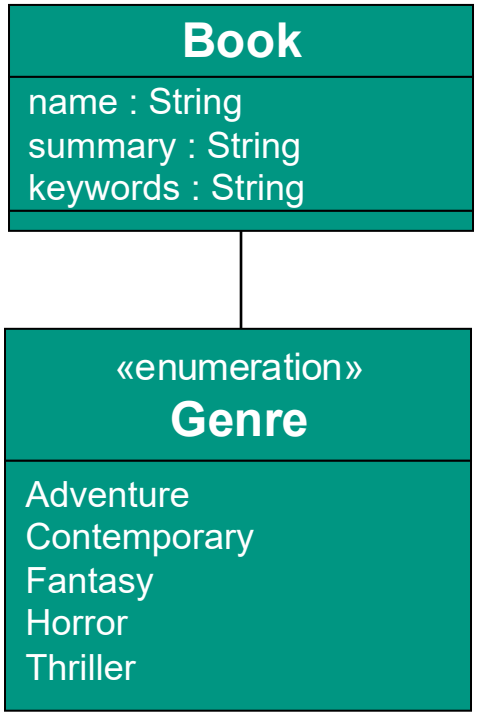
Constructors



- Constructors are marked with the stereotype «create»
  They can have arbitrary names.
  They create exemplars.
- The return type is implicitly the class and is not indicated.
- In Java, the name of a constructor must be the same as the class. Java constructors also have no declared return types.
- Java example: `public Circle(int x, int y, int r) {…}`

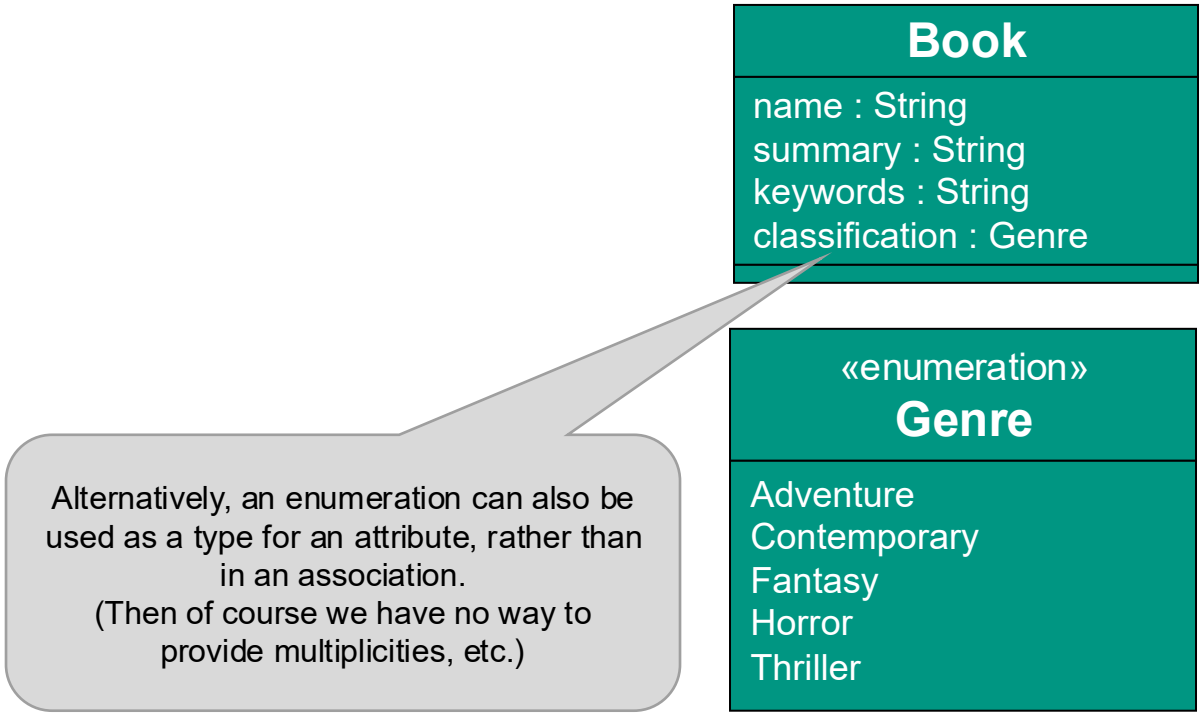
Enumerations

- Enumerations in UML are defined by a special form of class
- The attributes define the possible elements of the enumeration.
- Enumerations are declared with the stereotype «enumeration»
- An enumeration is linked to an object with an association.



- (In UML, stereotypes classify model elements and thus provide extra information)

Enumerations

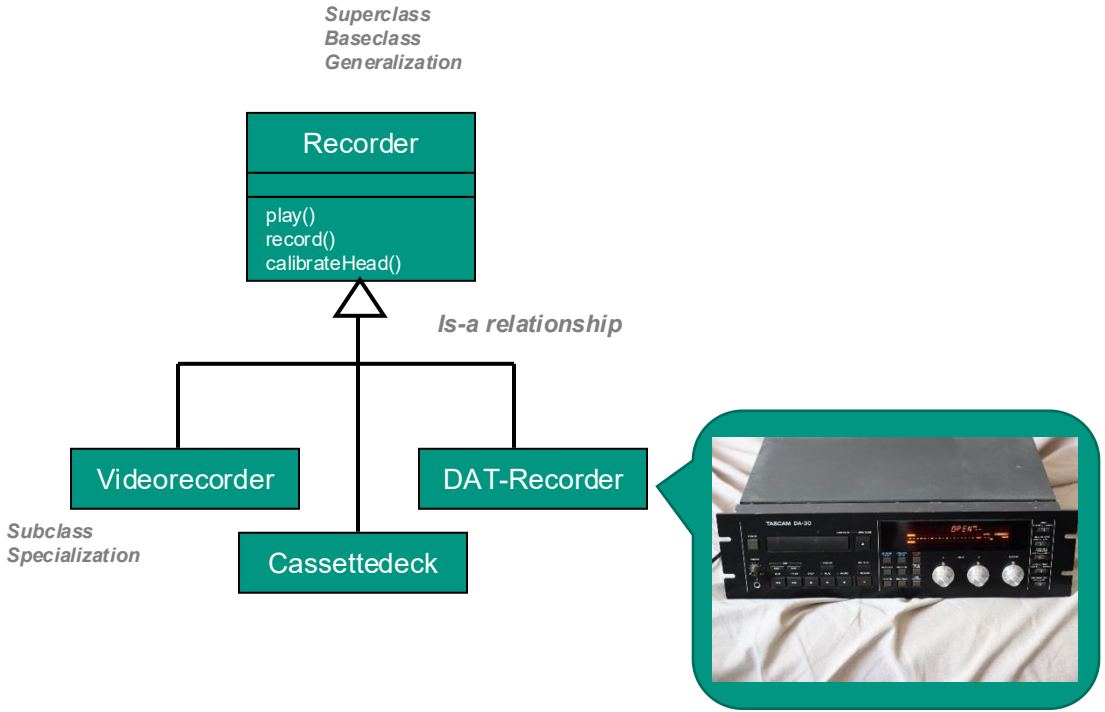


A short quiz

- True or false…?
- In UML an association can have more than two endings.
- The following model is adequate:

Inheritance

Inheritance—concepts and synonyms



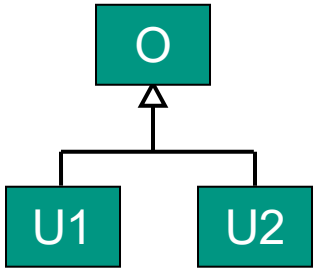
Def. **Inheritance**: Let O and U be classes, and Ω_O und Ω_U the sets of exemplars that make up these classes. Then U is a subclass or specialization of O (and O superclass or generalization of U) iff $\Omega_U \subseteq \Omega_O$

We also say that U „inherits“ from O.

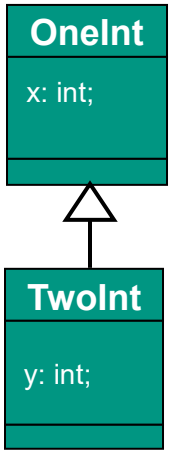
Since exemplars of U are also a exemplars of O, we call the relationship between O and U the is-a relation.

If O has several subclasses, then they should be disjoint (have no common exemplars)

Example of an exception: Platypus is a mammal, but it also lays eggs (i.e., it is also a bird)



Is the subset relation really correct?

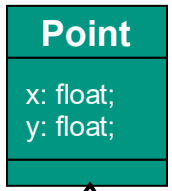


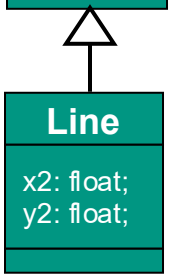
Question: isn´t the set Ω_{TwoInt} larger than Ω_{OneInt}?
TwoInt appears to be an extension of One*Int*!
OneInt has 2^{32} Elements, but *TwoInt* has 2^{64}

Answer*: OneInt* is the set of all objects that
have **at least** an integer variable named *x*.
The set described by *OneInt* includes for example
Objects with two variables, where the second one
Is of type float or called differently.
In fact we have $\Omega_{TwoInt} \subset \Omega_{OneInt,}$ because of the “at least” statement

Is-a relation

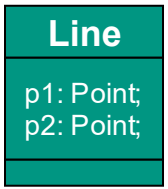
- Suppose need to model a line. We could use the class Point for this. Is inheritance here a good idea?



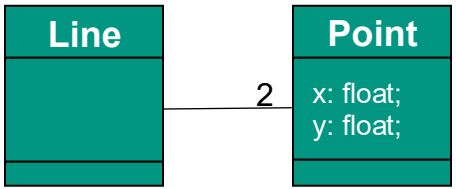


A line is not a special case of a point.
(it is actually infinitely many points)
This model is lazy.

Better:



or:



Finding superclasses

- Check whether different classes have common attributes, states, associations, or methods.
- These can then be „pulled up“ into a common super class.
- Inheritance offers advantages when doing so:
  - Avoids redundancy (shared attributes and methods need to be repeated; instead, they are „inherited“ from the subclasses.)
  - There is a well-founded typing theory for inheritance

Liskov substitution principle

In a program with U being a subclass of O, every exemplar of O can be replaced with an exemplar of subclass U and the program will continue to work correctly.

- All properties* of the superclass must be available in the subclass.
- The subclass has the same or weaker preconditions as the superclass
- The subclass has the same or stronger postconditions as the superclass.

* Attributes, associations, assertions, states, methods

Example for the substitution principle

```
class Subclass extends Superclass … ;

Superclass o = new Superclass();
Subclass u = new Subclass();

o.m(); ✓
u.m(); ✓

o = u; --Substitution

o.m(); ✓!
```

Implications of the substitution principle (I): Adding and deleting properties

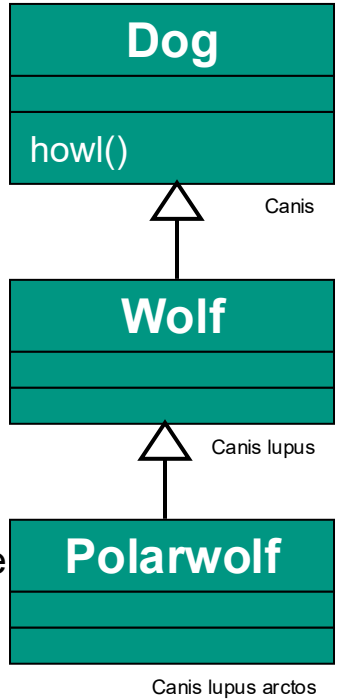
- Inherited properties* are available in a subclass as defined in the superclass.
- The subclass may define additional properties, which make it more specialized.
- The subclass may not delete properties of the superclass.
  - If this were necessary, we wouldn´t have an inheritance relationship.
  - Substitution of subclass for superclass would not be possible. (why not?)

* attributes, methods, states, associations,

Implications of the substitution principle(II) Transitivity

- The inheritance relation is **transitive**.
  - A Dog can howl()
  - A Wolf is a dog.
  - Wolf inherits from Dog
    - ⇨ A Wolf can also howl().
  - A Polarwolf is a Wolf.
  - Polarwolf inherits all methods from Wolf, howl() included
    ⇨ Polarwolf can also howl().

⇒ We can send the message howl() to an instance of Polarwolf

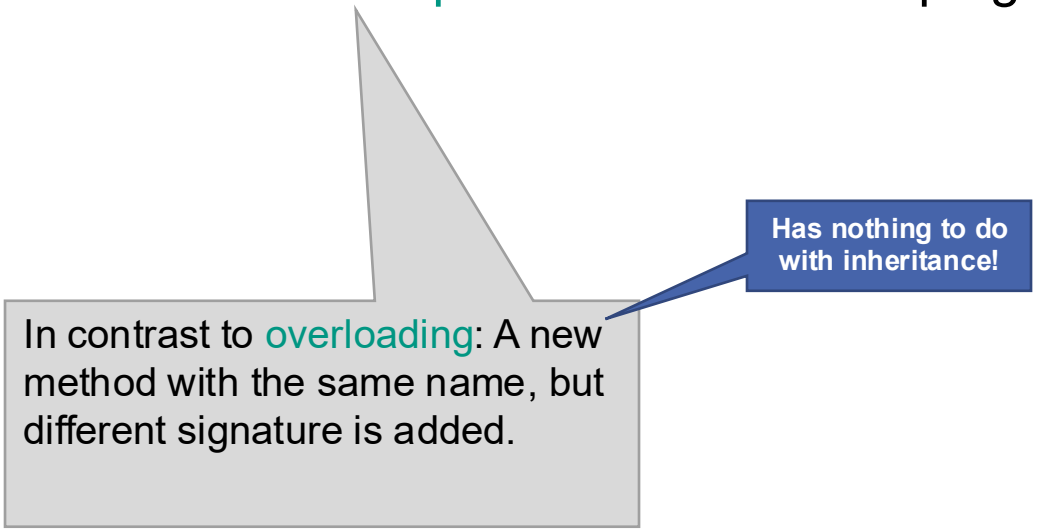


Differentiate!

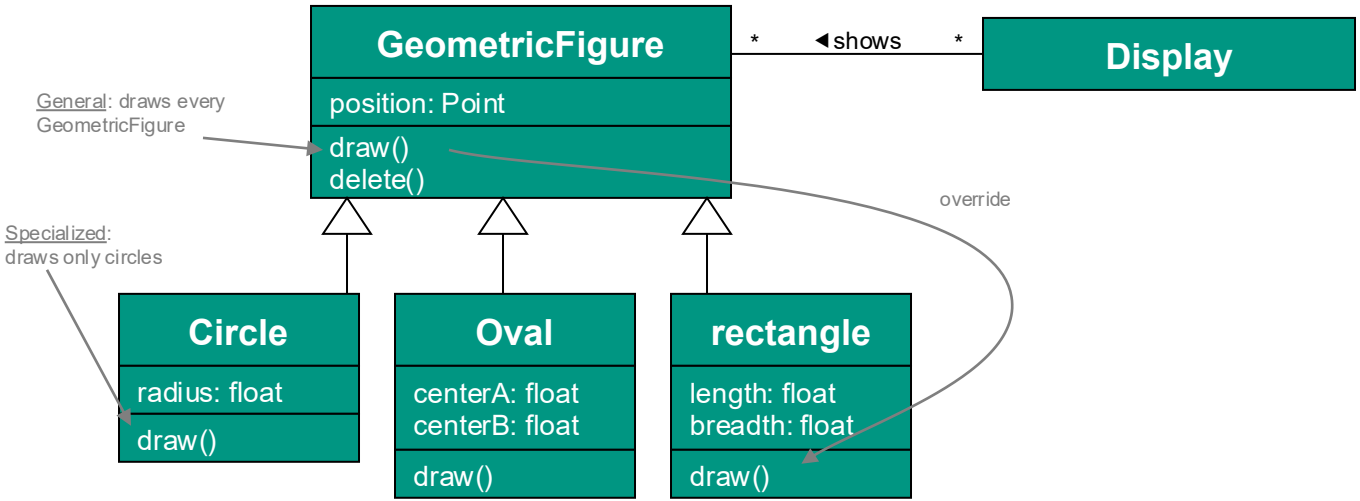
- Def. Signature inheritance: A method defined (and perhaps implemented) in a superclass transfers only the signature of the method to subclasses.
- Def. Implementation inheritance: A method defined and implemented in a superclass transfers both the method´s signature and its implementation to subclasses.
- Implementation inheritance is not possible without signature inheritance.
- Example: Java and C# provide both signature and implementation inheritance. (how?)

Implications of the substitution principle(III): Adaptation of methods

- Methods can be replaced to meet the requirements of the specialization
- Def. Override: an inherited method can be reimplemented while keeping its signature.

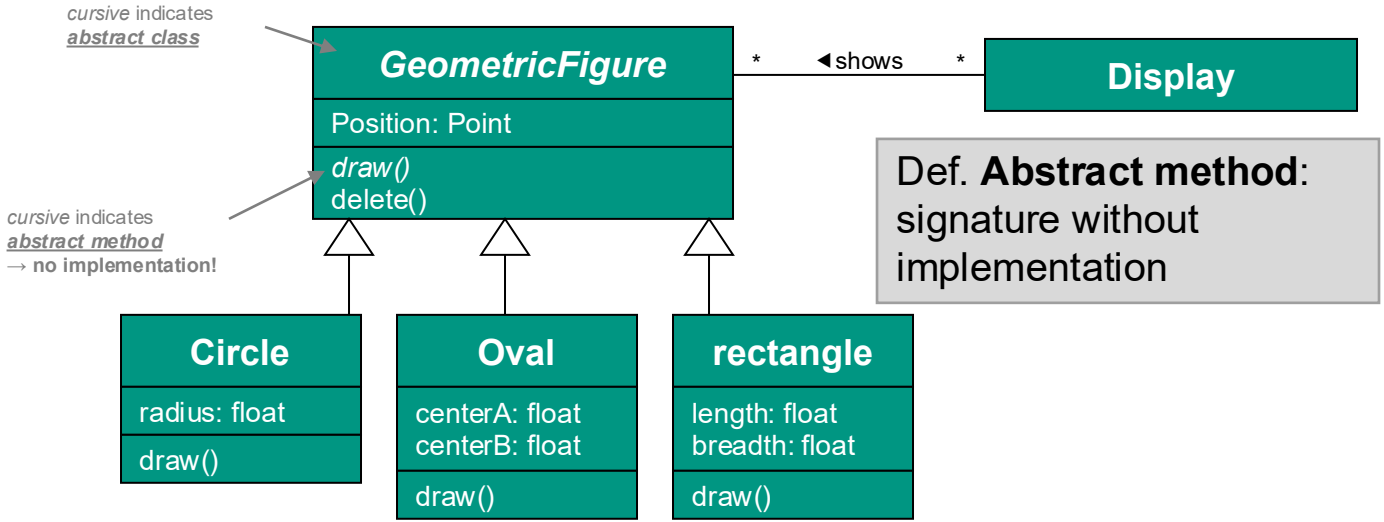


Overriding – Example



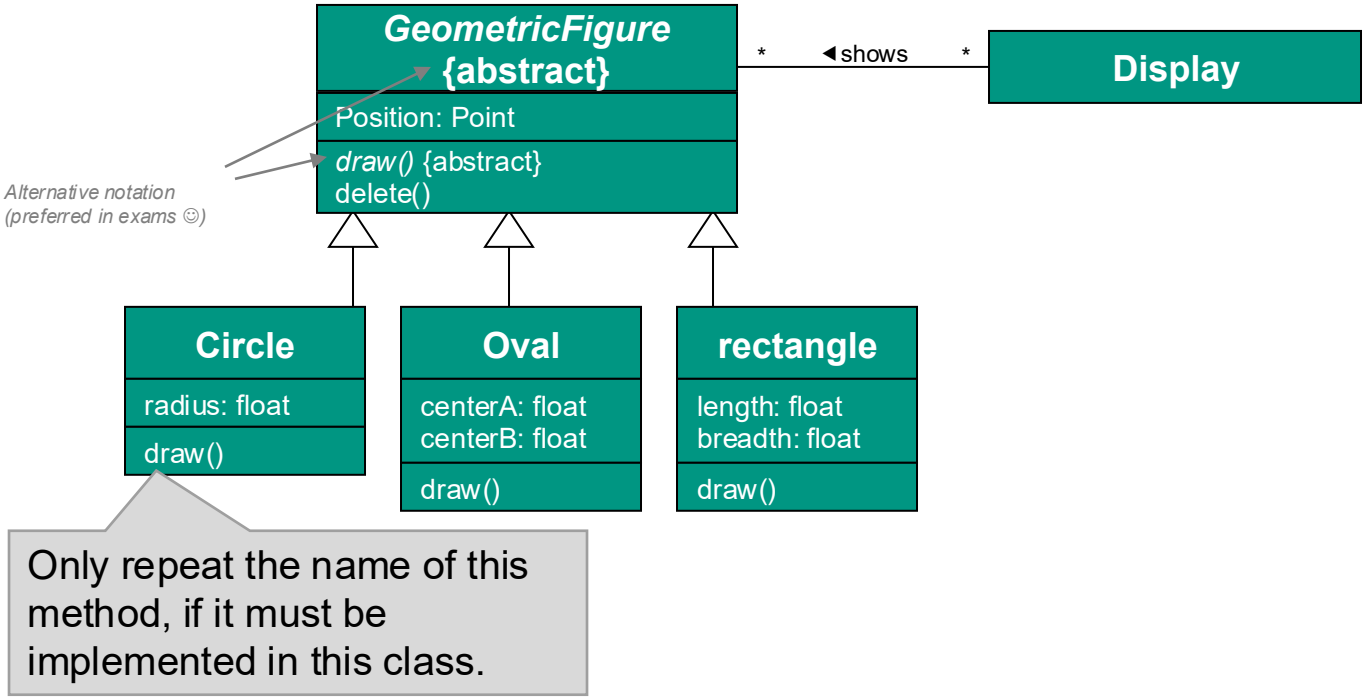
- Insight:
  - Each of the three specializations must implement its own `draw()` method
  - Is it even possible to implement a general `draw()` in `GeometricFigure`?

Solution: Abstract methods



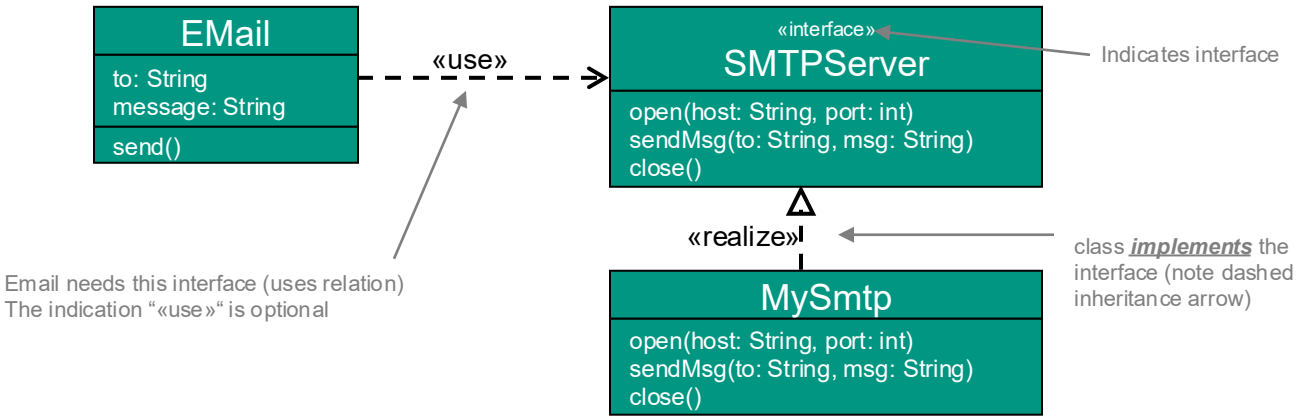
- If there is at least one abstract method in a class, then the class itself is abstract.
- There are no exemplars of abstract classes. Why not?
- The first concrete (non-abstract) class in an inheritance hierarchy “inherits” the obligation to provide implementations for the abstract methods of abstract superclasses. Why?

Abstract methods, alternative notation



Interfaces

- Def. interface: a named set of abstract methods, which need to be implemented by classes that offer that interface.
  - Interfaces cannot be directly instantiated
  - A class may implement several interfaces (multiple inheritance of interfaces).



Use of interfaces

- Interfaces transfer the obligation to implement their methods.
  - Interfaces provide the guarantee that certain methods are present.
  - Exemplars of a class that implements a certain interface can be used as if they were an exemplar of the interface (is-a semantics).
- General idea: an interface indicates how to use an object, but not what it is.

Interface – example

- You receive a remote control.

- You can see that it can send the messages
  - play()
  - stop()
  - ff()
  - rwd()
  - skip()

  to some device.

- Although you do not know what the device is, you can use it, because you know the interface.

Inheritance and interfaces

- Interfaces cannot inherit!
  - If an interface A is extended by an interface B, then the set of abstract methods of A is a subset of the abstract methods of B. (short: $A \subseteq B$).
  - When a class C implements an interface A, then the set of abstract methods of A is a subset of the methods of C, where C may provide additional implementations.
- Classes can implement several interfaces. Since this is simply a union operation of sets, multiple inheritance of interfaces is not problematic.
  - if two inherited interfaces provide the same method, then we simply take the union of the elements in the interfaces, meaning that for two inherited methods only one remains.
- Exercise: Which problems can occur with multiple inheritance of classes?

Implications of the substitution principle (IV): Signature changes

- The substitution principle only requires that it is possible to use an exemplar of a subclass as if it were an exemplar of the superclass. It does not say that the interface must stay exactly the same.

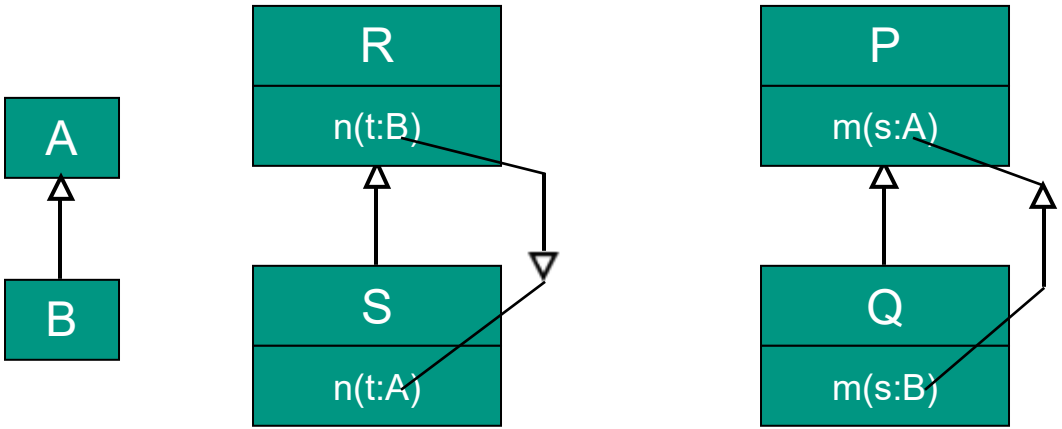
- Which signature changes are possible, without violating the substitution principle?

- Short answer: it is possible to make the types of the parameters and the return type more or less general.

Implications of the substitution principle (IV): Parameter variance

- Def. **Variance**: Changing the types of parameters and return types of an overriding method.
- Def. **Covariance:** Specializing the parameter or return types of an overriding method
- Def. **Contravariance**: Generalizing the parameter or return types of an overriding method
- Def. **Invariance**: no modification of types

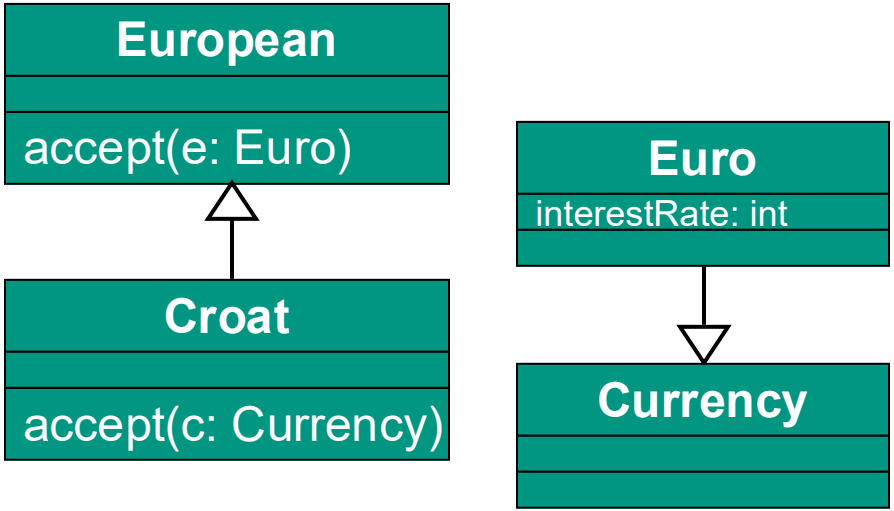
Example of co/contravariance



contravariance of parameter t. (classes and parameters specialize in opposite directions.)

covariance of parameter s. (classes and parameters specialize in the same direction.)

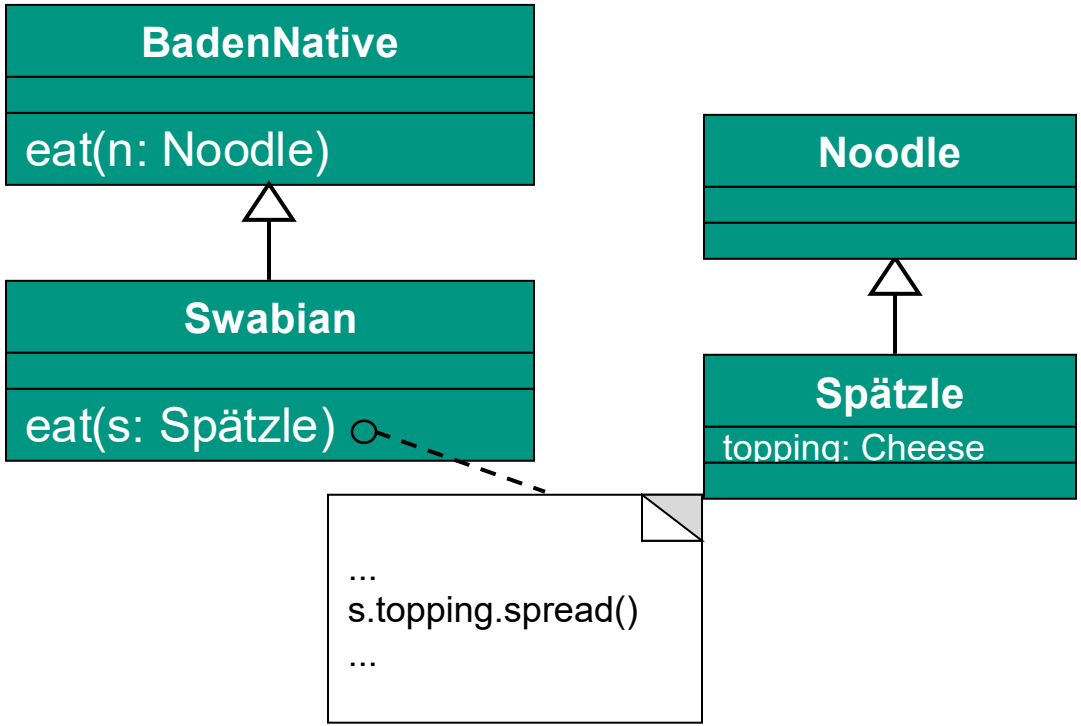
Contravariance and substitution principle – Example



Contravariance and substitution principle – Example

```
European e;
Croat k;

Currency cur;
Euro eu;

e.accept(eu); ✓  e.accept(cur); ??

e = k;  // substitution!

e.accept(eu); ?? e.accept(cur); ??
```

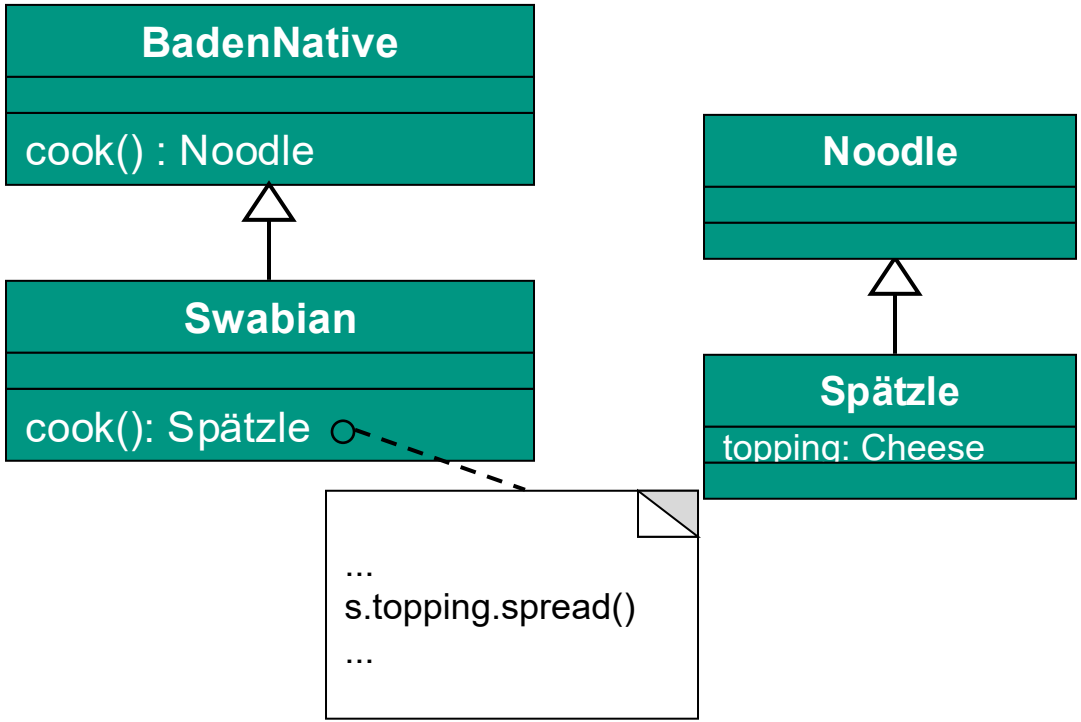
Covariance und substitution principle – Example



Covariance und substitution principle – Example

```
BadenNative b;
Swabian s;

Noodle n;
Spätzle sp;

b.eat(n); ✓  b.eat(sp); ✓

b = s;   // substitution !

b.eat(n); ??  b.eat(sp);  ??
```

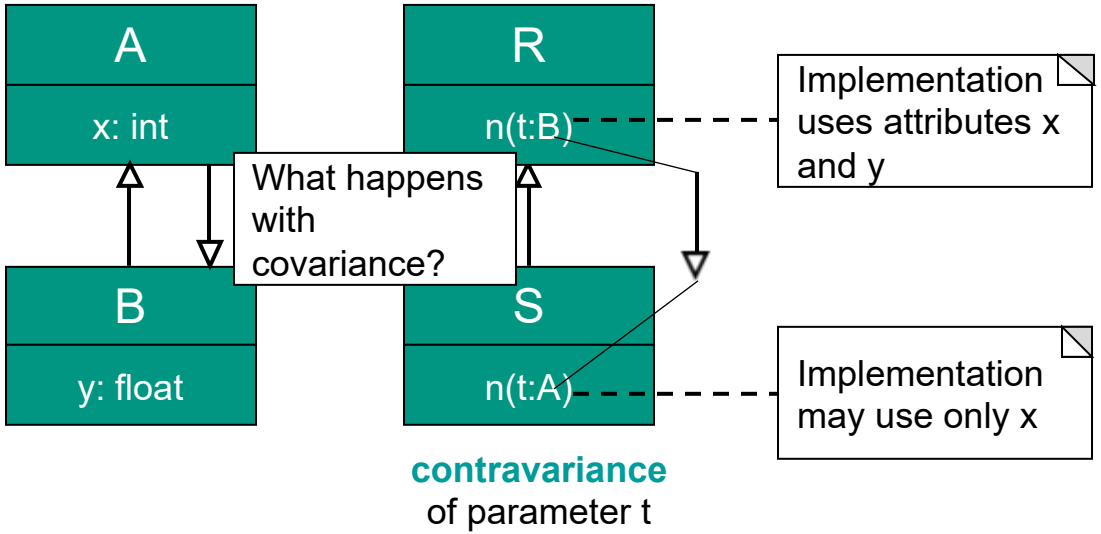
Covariance und substitution principle
Example with return type



Covariance und substitution principle
Example with return type

```
BadenNative b;
Swabian s;

Noodle n;
Spätzle sp;

n = b.cook(); ✓ sp = b.cook(); ??

b = s;   // substitution !

n = b.cook(); ??
```

Co/contravariance again




```
R r; S s;
r.n(new A);   r.n(new B);

r = s;   // Substitution !!!
r.n(new A);   r.n(new B);
```

Co/contravariance again

- Exercise 1: Examine the example before for contravariance of the parameters. Which ones of the calls work/don´t work? Why?

- Exercise 2: Examine again the example before, this time for return values. Let class R have a method `m():A` and S a Method `m():B`. Which ones of the expressions `r.m().x` und `r.m().y` are legal before and after the substituion of `r=s`, given covariance? What happens for contravariance? What is caught by the compiler, and what at runtime?

Quick quiz…

- What is allowed for input parameters?
  - variance (general)
  - covariance
  - contravariance
  - invariance

Implications of the substitution principle(IV): Permitted variance

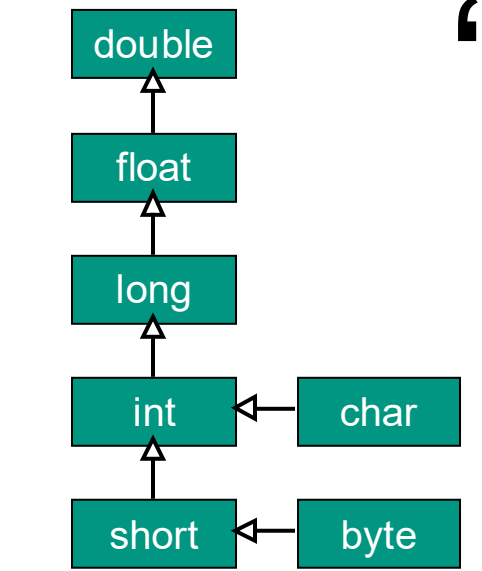
- To allow the substitution principle, the following modifications of parameter and return types are permitted in the overriding methods

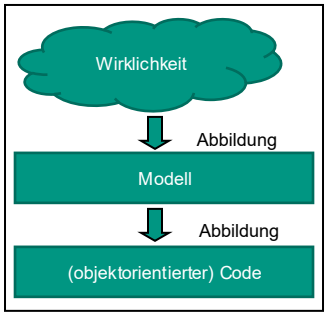
Input parameters	Contravariance
Output parameters, return types, exceptions thrown	Covariance
Parameters, which are simulateneously input and putput parameters	Invariance

- Note: In Java or C# not all these variations are permitted!

Hint for Java-Programmers

Java primitive types have the following inheritance tree:





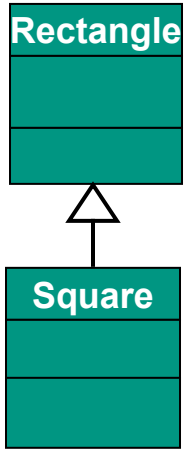
see: The Java Language Specification 2.0, §5.1.2, Widening Primitive Conversion

Implications of the substituion principle: general

- A subclass has the same or weaker preconditions as the superclass.
  - For methods this means that when substituting a subclass for a superclass, the preconditions of the subclass methods are satisfied.
- A subclass offers the same or stronger postconditions as the superclass.
  - For methods this means that when substituting the results of a subclass method, the conditions that are expected of the results of the superclass method are satisfied.
- Subclass methods must not expect more or deliver less.

Implications of the substitution principle(V)

Example:



Is this model correct?
Obviously, squares are rectangles

Implications of the substitution principle(V)

```
class Rectangle {
  private double breadth, height;

  public void setBreadth(double b) {breadth=b}

  public void setHeight(double h) {height = h}

  public double getBreadth() { return breadth }

  public double getHeight() { return height }
}
```

Implications of the substitution principle(V)

```
class Square extends Rectangle {

  public setBreadth(double b) {
     super.setHeight(b);
     super.setBreadth(b);
  }

  public setHeight(double h) {
     super.setHeight(h);
     super.setBreadth(h);
  }
}
```

Implications of the substitution principle(V)

```
// somewhere else....
public void compute (Rectangle r) {
     r.setHeight(5);
     r.setBreadth(6);
     assert(r.getHeight()*r.getBreadth()==30);
}

// somewhere else...
Rectangle r = new Square(....); //substitution
compute(r);  // assert fails. Why?
```

Implications of the substitution principle(V)

- Postcondition of `Rectangle.setBreadth(..)`:

```
assert((breadth==b) && (height==old.height));
```

- This condition is for `Square.setBreadth(..)` not satisfied. The postcondition is weaker.
- The precondition for `Square` is stronger: `breadth==height`

- In others words, the subclass methods expect more and deliver less. Which means the substitution principle is not satisfied.

Implications of the substitution principle(V)

- Logically, the model is correct, because squares are indeed a subset of rectangles.
- But the behavior of the classes is not consistent.
- We have to check whether pre- and postconditions „fit“ before we substitute one class for another.
- Only if superclass objects can be substituted correctly by subclass objects is it true that a subclass object can be used in the context of a superclass object.

Polymorphism

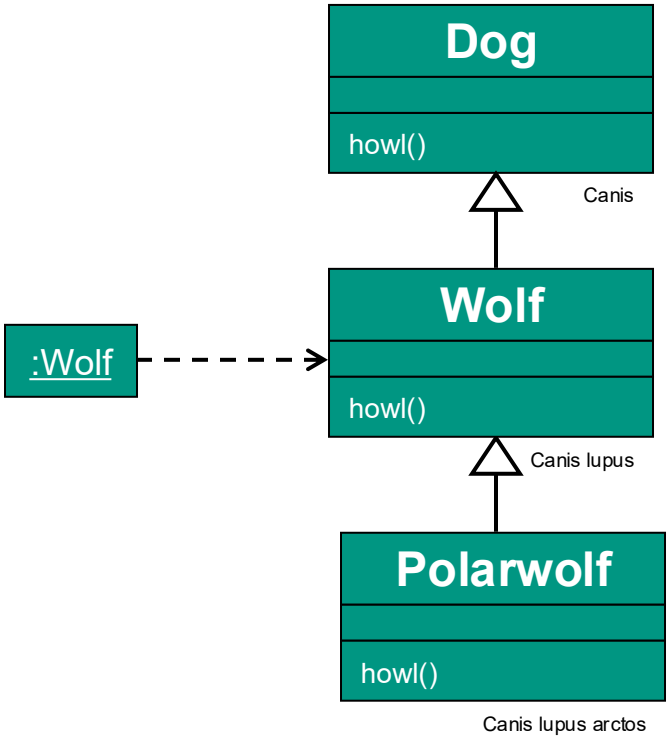
- Polymorphism means „having many forms“
- “Static polymorphism“ (Overloading)
  - There can be several methods with the same name (but their signatures* must be different, so the compiler can determine which one to choose)
- “Dynamic polymorphism“ (using inheritance)
  - At runtime, the method with a given signature is chosen that is the most specialized one in the inheritance hierarchy, starting with the class of the current object.

Has nothing to do with object orientation or inheritance!

* In Java, the parameter list must be different, in Haskell the return type

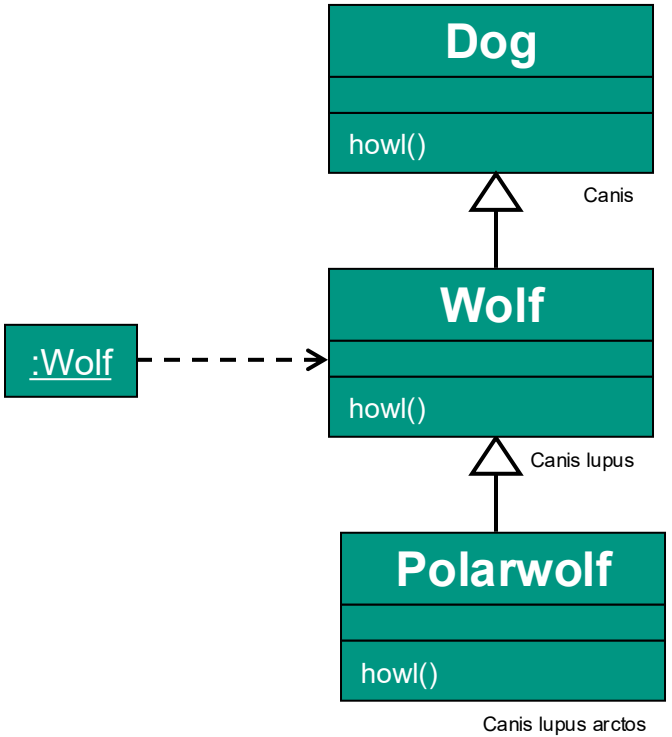
Example:
dynamic polymorphism

```
Dog d;
Wolf w = new Wolf();

d = w;
d.howl();
```

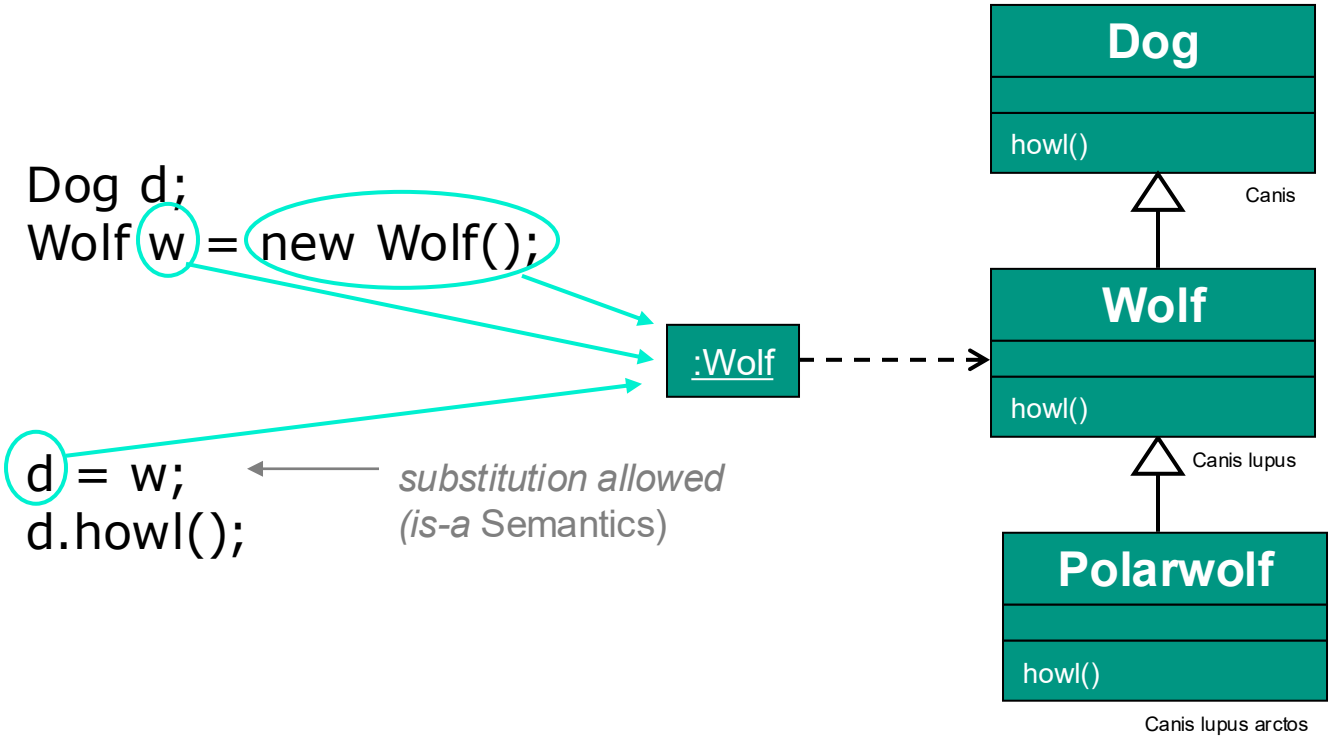


Example:
Dynamic polymorphism

```
Dog d;
Wolf w = new Wolf();

d = w;
d.howl();
```

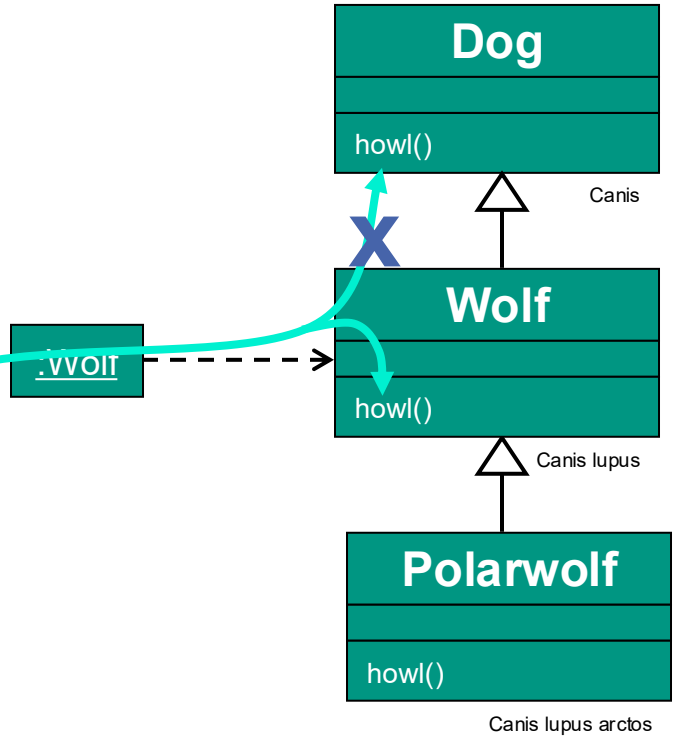


Example:
Dynamic polymorphism

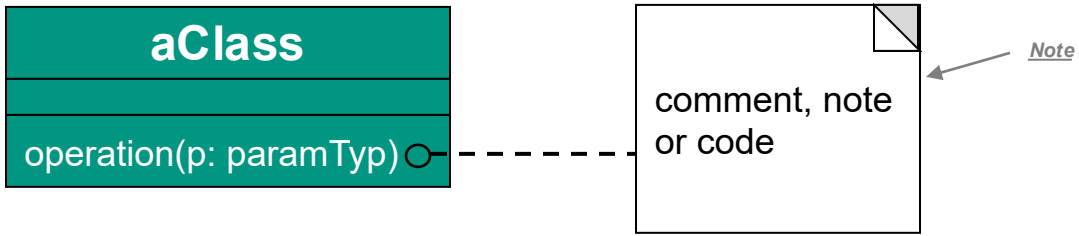


Example:
Dynamic polymorphism

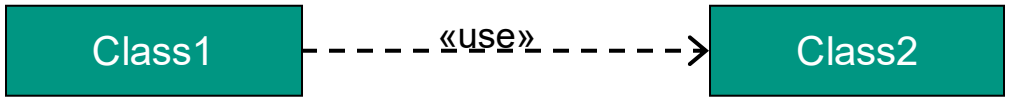
```
Dog d;
Wolf w = new Wolf();


d = w;
d.howl();
```



What else is there in class diagrams?



- Notes/comments/pseudo code added
- Dependency (Class1 depends on Class2, for instance, because Class1 uses Class2 as a parameter, a local variable, or return type)

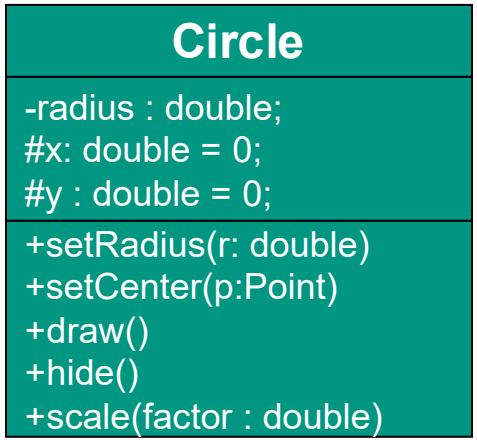


Visibility/access rights

- Attributes and methods of an object may be protected from access by other classes. Access is allowed for:
  - **private** (“**-**“): only (but all!) Exemplars of the same class
  - **protected** (“**#**“): Exemplars of the same class and all derived classes
  - **package** (“**~**“): Exemplars of the same package (subsystem/library)
  - **public** (“**+**“): every Exemplar
  - The visibility symbol (-, #, ~, +) is simply placed before the name of the attribute or method
- Note: “private“ does not mean “only this exemplar“!
- The protection is limited to compile time only.
  - All access rights can be changed at runtime (e.g. with BCEL forJava). They do not protect from malicious attacks. They are only meant for a clean model that avoids excessive information distribution and makes the software easier to modify.

In Java, “protected” includes package visibility

Visibility: Example



No indication of visiblity means “public“

Literature

- Bruegge, A.H. Dutoit, **Object-Oriented Software Engineering: Using UML, Patterns and Java**,
  Read Chap. 2! (not as detailed as this lecture.)

- UML-Spezifikation unter https://www.omg.org/spec/UML/2.5.1/
  - Especially “UML 2.5.1 Superstructure Specification“
  - and “UML 2.5.1 Infrastructure Specification“

Exercises

1. What are class attributes and class methods called in Java? Is there a difference to UML?
2. How do associations and inheritance differ? (Hint: Which UML elements are primarily addressed in associations, and which ones in inheritance?)
3. What may occur in Java interfaces? Is there a difference to UML interfaces?
4. Suppose we had interfaces, but no abstract methods in classes. Would this cause a problem or disadvantage?
5. What type of parameter variance is permitted in Java? (Don´t forget return types.)