Object Oriented System Development
Lecture 5
Contracts, Classes, Objects and Relations

Rogardt Heldal

Use case Register Student

Pre: Student id exists **and** course code exists **and** places exist on the course **and** the student meets all the course pre-requirements

Post: Student is registered on the course

Pre-condition

- Pre:
  - Student id exists **and** course code exists **and** places exist on the course **and** the student meets all the course pre-requirements

This pre-condition is true if all these are true:

- Student id exists
- Course code exists
- …

This pre-condition is false if any of these conditions is false

- Student id exists
- Course code exists
- …

Post-condition

- Post: Student is registered to the course
- This post-condition is true only if a student is registered.
- **That means that a student should always be registered to a course after executing the use case!**

Contract

- Pre and Post-conditions are the contract for the use case.
- Or put in another way:
  - Pre-condition is what should be true for the use case to guarantee the post condition.
  - Post-condition is what should be true after ending any of the scenarios of the use case.
- So, when writing the use case flows:
  - Given the pre-condition, write the flows in such a way that it meet the post-conditions

Contract

- pre-conditions

  implies

  <running a scenario> post-conditions

True implies True (True)

True implies False (False)

False implies True (True)

False implies False (True)

Use case Register Student

Pre: Student id exists **and** course code exists **and** places exist on the course **and** the student meets all the course pre-requirements

Post: Student is register to the course

Use case Register Student

Pre: Student id exists **and** course code exists **and** places exist on the course and student meets all the course pre-requirement

1. User input student id and course code
2. System find student and course
3. System register student to course

Post: Student is register on the course

Use case Register Student

Pre: True

Post::

if Student id exists **and** course code exists **and** places exist on the course and student meets all the course pre-requirement
then Student is registered to the course
else True

Use case Register Student

Pre: True

1. User input student id and course code
2. System find student and course
3. Assume: that student and course existed
4. System register student to course
5. Assume: enough places on the course
6. Assume: student had the require courses

3 alternative
Cases:
Assume:
student don´t
exist
…

Post::
if Student id exists **and** course code exists **and** places exist on the course and student meets all the course pre-requirements
then Student is registered to the course
else True

Use case Register Student

Pre: True

1. User input student id and course code
2. System find student and course
3. Assume that student and course existed
4. System register student to course
5. Assume enough places on the course
6. Assume student had the require courses

3 alternative
Cases:
Assume:
  student don´t
  exist
…

Post::
if Student id exists **and** course code exists **and** places exist on the course and student meets all the course pre-requirements
then Student is register on the course
else **nothing is changed in the system**

Operation

```
pre. y<> 0
div(x,y) : int
    return x/y                    (this is ok, due to pre-condition)
```

Misunderstood pre-condition:

```
pre y<> 0
div(x,y) : int
     if y<> 0                    (y should be different from 0)
     then return x/y
     else …
```

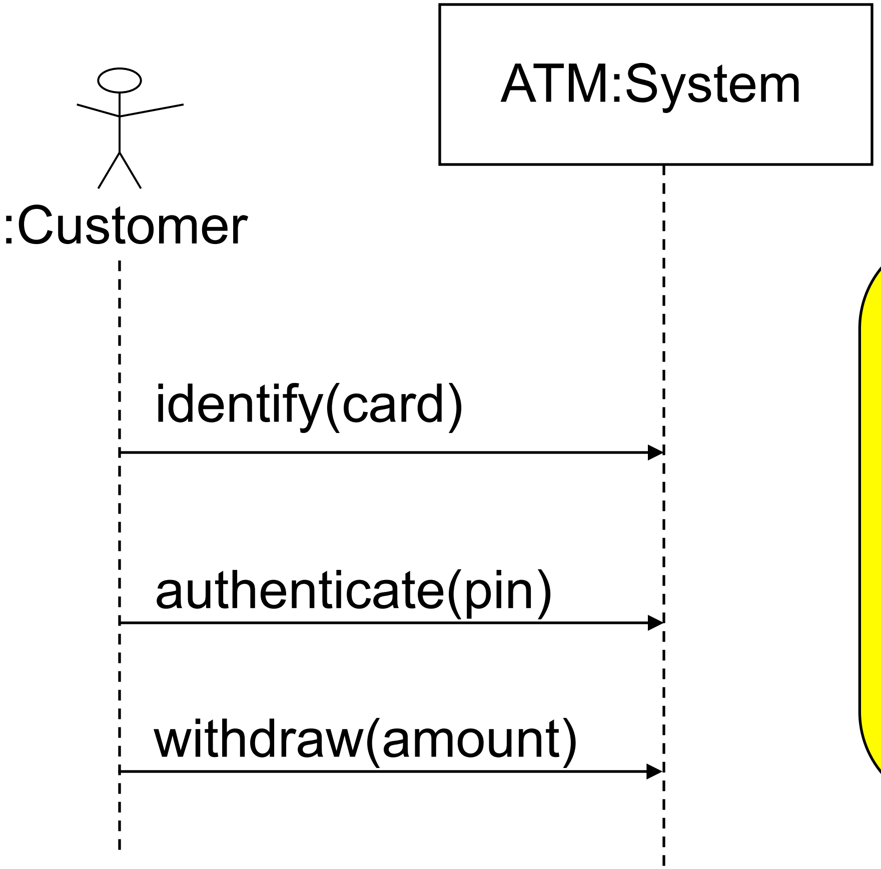
Use case: Withdraw Money

Only main flow:

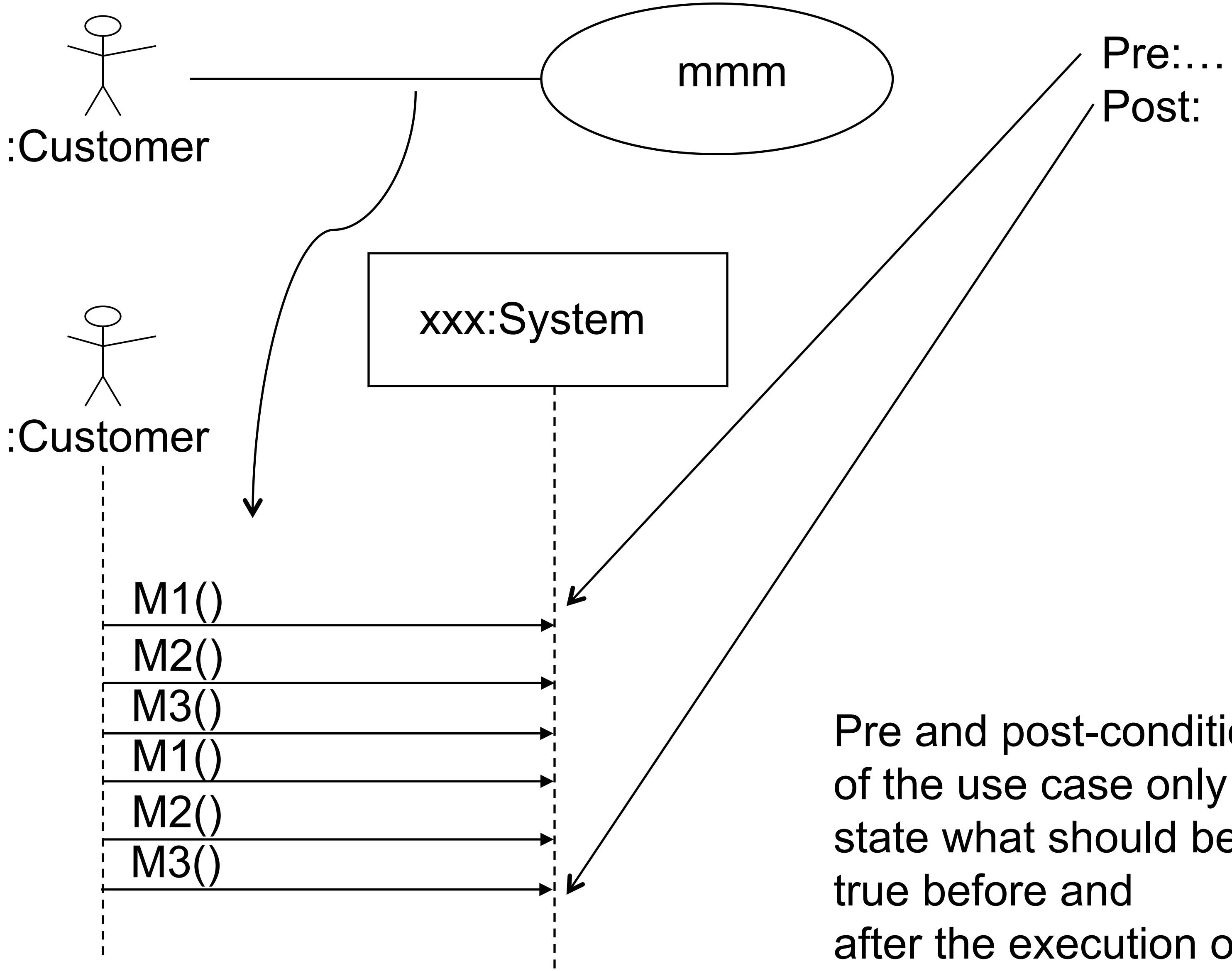
1. user identifies himself by a card
2. system reads the bank ID and account number from card and validates them
3. user authenticates by PIN
4. system validates that PIN is correct
5. user requests withdrawal of an amount of money
6. system checks that the account balance is high enough
7. system subtracts the requested amount of money from account balance
8. system returns card and dispenses cash

Suggested names for operations

System Sequence Diagram Withdraw Money



Syntax will be covered in later lectures in more detail!



Pre and post-condition of the use case only state what should be true before and after the execution of all the methods

Contract

System operations

Example: Contract

- Operation: withdraw(amount:int)
- Postcondition:
  - **If** account contains enough cash

  then the balance of the account for the inserted card
  is decreased by “amount” **AND**
  the card has been returned **AND**
  cash had been dispensed
  else the account balance has not been changed **AND**
  card had been returned

Contract Template

- The signature of the operation:
  - Name, parameters, return value
- Description of the operation (optional), for instance
  - Informal meaning of operation
  - Implementation in pseudo-code
- Description of the parameters (optional)
- Description of the operation’s result (optional)
- Cross-reference
- Precondition
- Postcondition

Use Domain Model to obtain pre- and post-conditions

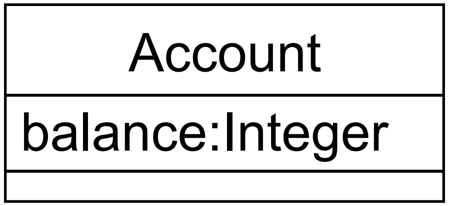
- Furthermore, the domain model can be used as the basis for the creation of the contracts.
  - The precondition specifies what has to hold in the domain model before the call to the operation.
  - The postcondition has to specify what has to hold in the domain model after the execution of the call.

Postcondition

- The postcondition has to specify the following things:
  - What instances have been created?
  - What attributes are modified?
  - What associations (to be precise, UML links) are formed and broken?
  - What value is returned from the operation?

Example: Withdraw Money

- What attributes are modified?
  - The balance attribute in the concept Account might be changed.



Account

balance:Integer

Problem

Write a contract for the operation authenticate.

…

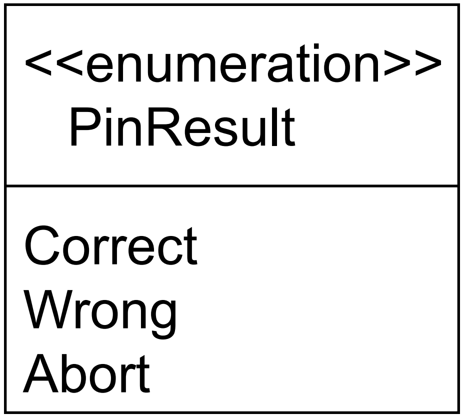
4. user authenticates himself by PIN
5. system validates that PIN is correct

…

- 4a. Wrong pin less than 3 times:
  - 1. System updates number of tries
  - 2. start from action step 3
- 4-8a. Wrong pin 3 times:
  - 1. System keeps the card

Part of the solution

- Operation: Authenticate (userPin: Integer):PinResult
- Cross-ref: Withdraw Money
- Result:
  - PinResult::Correct if authentication successful,
  - PinResult::Wrong if authentication failed, but further tries possible
  - PinResult::Abort if authentication failed
- post-condition: ?



<<enumeration>>
PinResult

Correct
Wrong
Abort

Solution

- Operation: Authenticate (userPin: Integer): PinResult
- …
- post-condition:

```
– if userPin was equal to the pin of the inserted card
  then PinResult::Correct has been returned
  else if tries was at most 3
       then tries has been incremented by 1 AND
            PinResult::Wrong has been returned
       else card has been kept AND
            PinResult::Abort has been returned
```

More details into Contracts

- In contracts, one often is more precise than in use cases, even formal.
- On the next slide we show a formal contract written in Object Constraint Language (OCL) for Withdraw Money.
- We might come back to OCL later in this course.

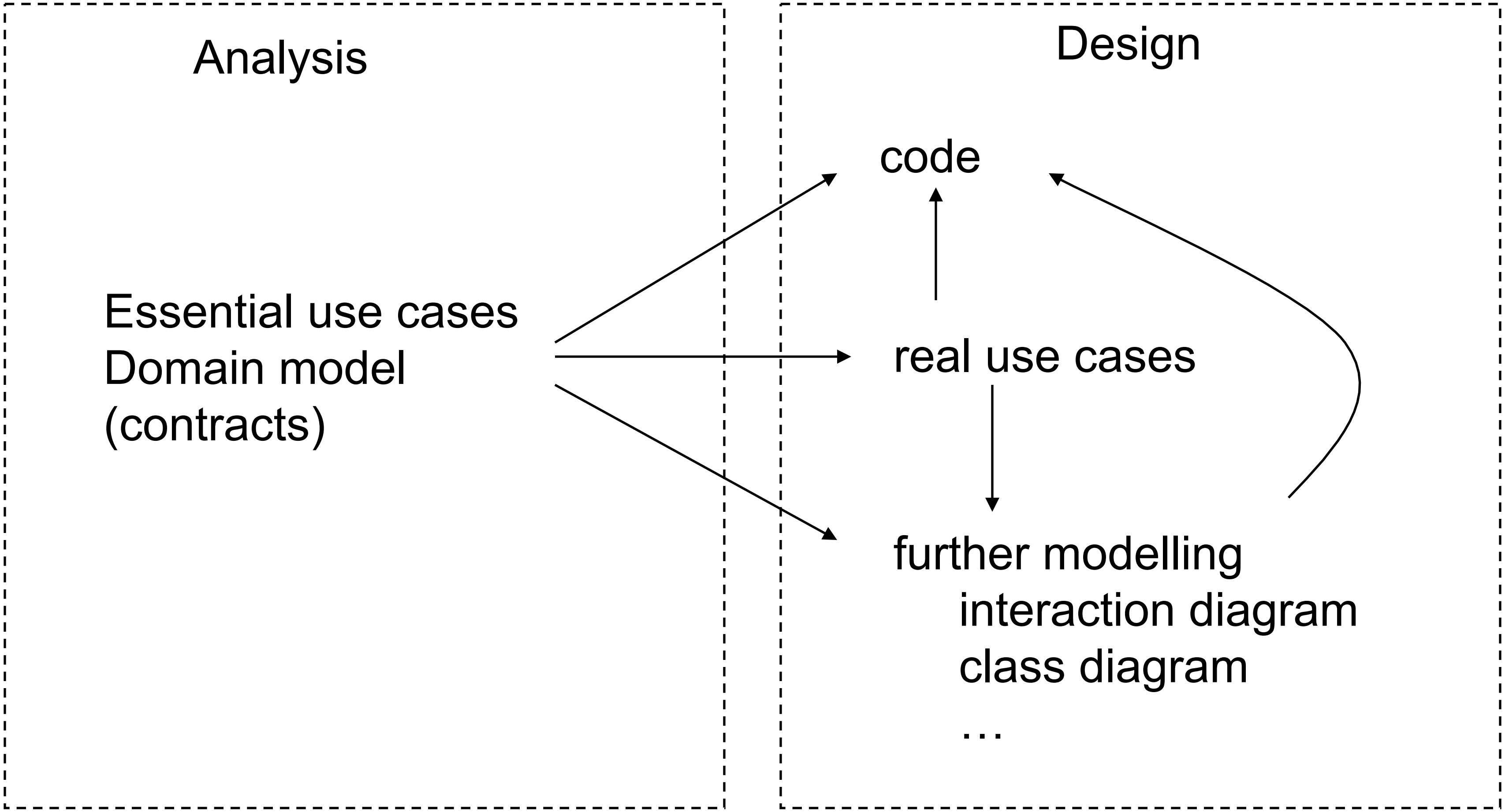
Formal Contract

```
Context ATMController::giveAmount(amount:long) post:
 if ( amount <= bank.getBalance(card.getID()) ) then
      cashDispenser^giveOutCash(amount)
   and   bank.getBalance(card.getID())
      = bank.getBalance@pre(card.getID()) - amount
   and card^returnCard()
 else
      not cashDispenser^giveOutCash(?)
   and   bank.getBalance(card.getID())
      = bank.getBalance@pre(card.getID())
   and card^returnCard()
```

Problem

- Write contract for the system operations obtained from “register on course”.

What next?



Analysis

Essential use cases
Domain model
(contracts)

Design

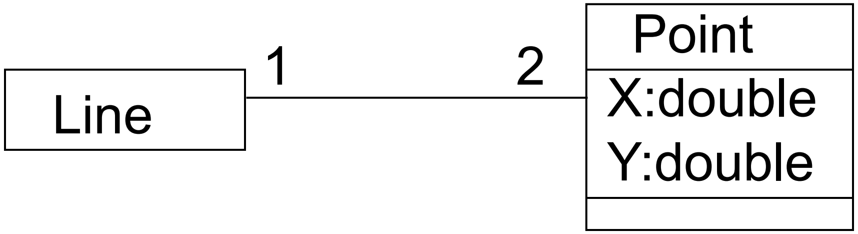
code

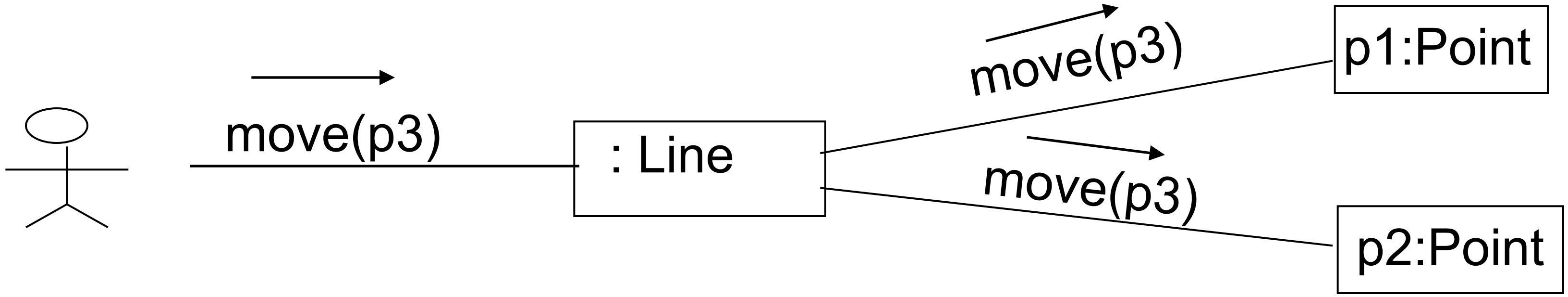
real use cases

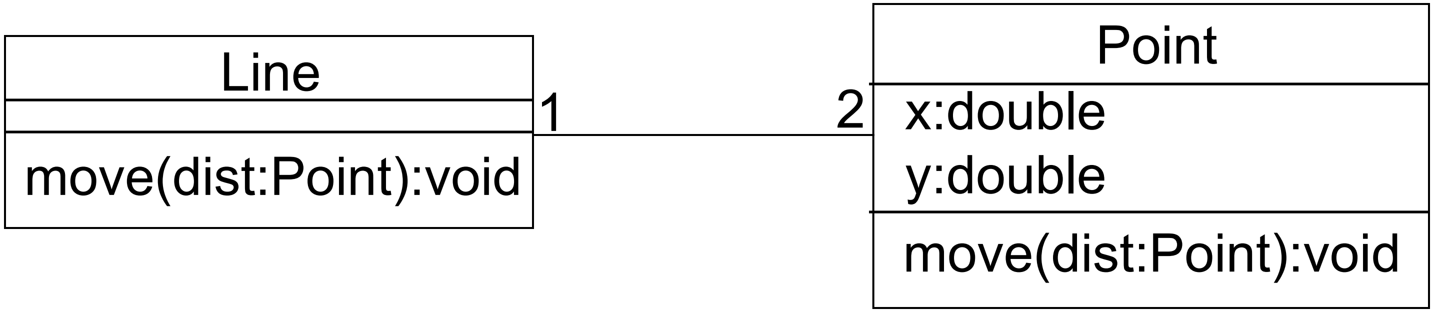
further modelling
interaction diagram
class diagram
…

Classes

Obtaining operations

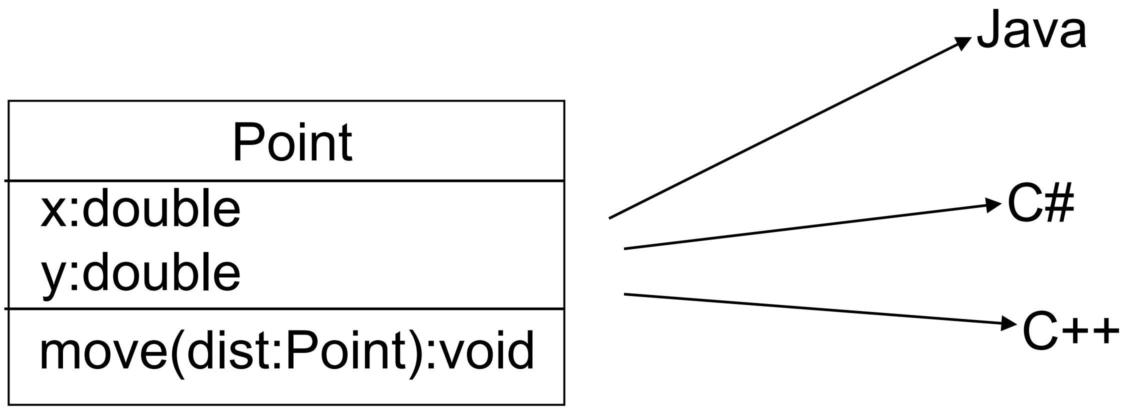






Mapping to code

- One can map a UML class to many different code skeletons in different programming languages such as:



Point

x:double
y:double

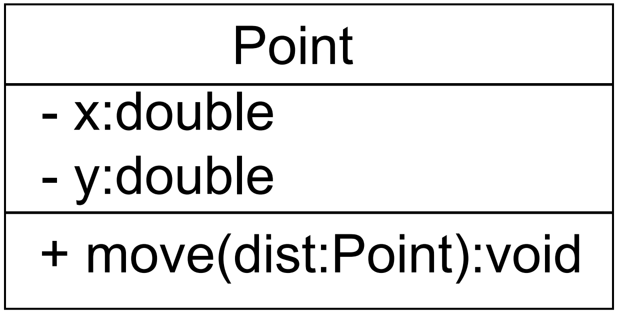
move(dist:Point):void

Java

C#

C++

UML Classes: Visibility



Point

- x:double
- y:double

+ move(dist:Point):void

Mapping visibility to java:

- **- -> private**
- **# -> protected**
- **+ -> public**
- **~ -> package**

(In this case the semantics of -,#,+,~ will be the one of Java.)

UML attribute

UML:

[visibility] name [multiplicity] [:type] [= initial value] [{properties}]

Properties could be:

- **changeable** (Variable may be changed.)
- **addOnly** (When **multiplicity** is bigger than one you can add more values, but not change or remove values.)
- **frozen** (Cannot be changed after it has been initialized.)

- **Example:**
  - **x : int {frozen}**

Operations/methods

UML:

[visibility] name [(parameter list)] [: return type] [{properties}]

You can have zero or more parameters. Syntax for parameters:

[direction] name : type [= default value]

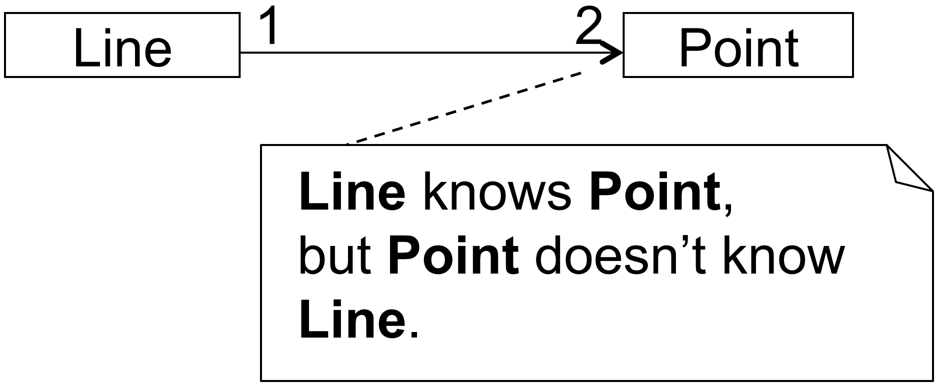
- direction: in, out, inout

- Example of a property
  - **isQuery** (no ”side effects”)

Relations

- All the associations we consider when drawing domain models can also be used in class diagrams.
- But there are some interesting issues to consider …

Navigability

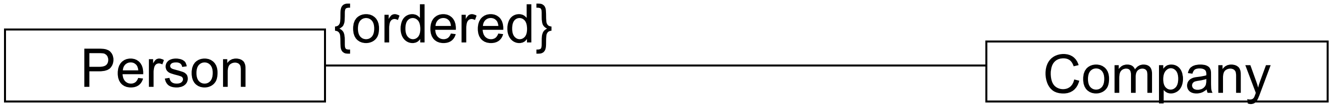


Line knows **Point**,
but **Point** doesn’t know **Line**.

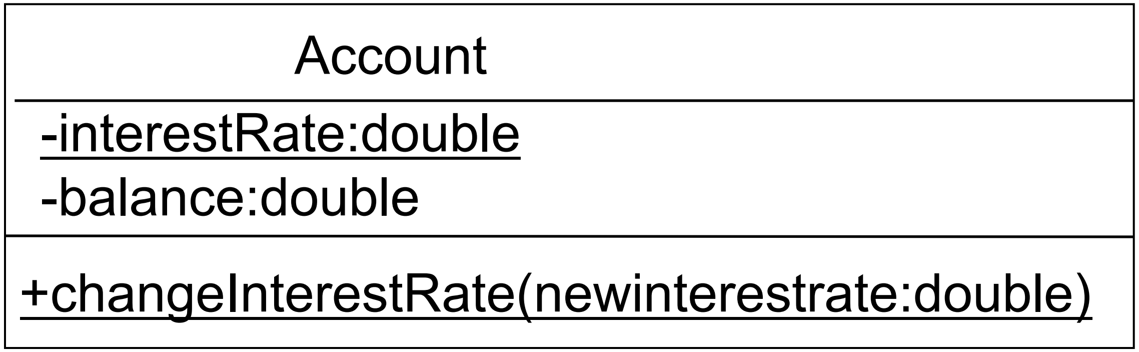
Association constraint

Constraint:

- **changeable** (Links may be changed.)
- **addOnly** (New links can be added by an object on the opposite side of the association.)
- **frozen** (When new links have been added from an object on the opposite side of the association, they cannot be changed.)
- **ordered** (Has a certain order)
- **bag** (multisets instead of sets)
- …



Class methods and class variables



Account

-interestRate:double

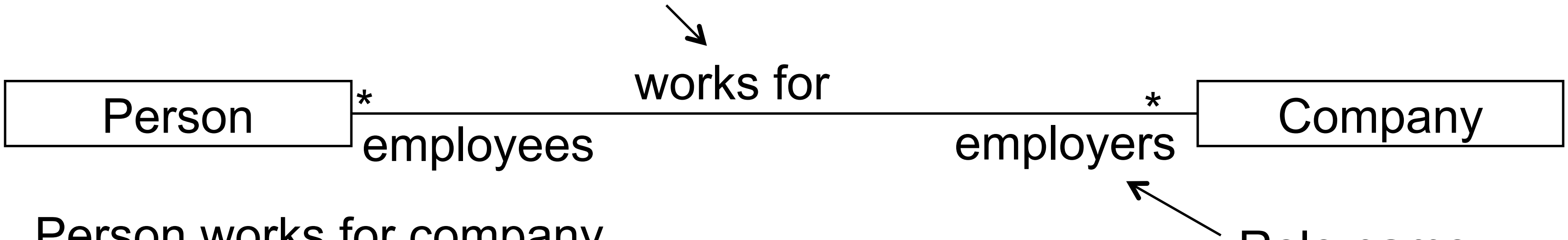
-balance:double

+changeInterestRate(newinterestrate:double)

Association names UML

UML:

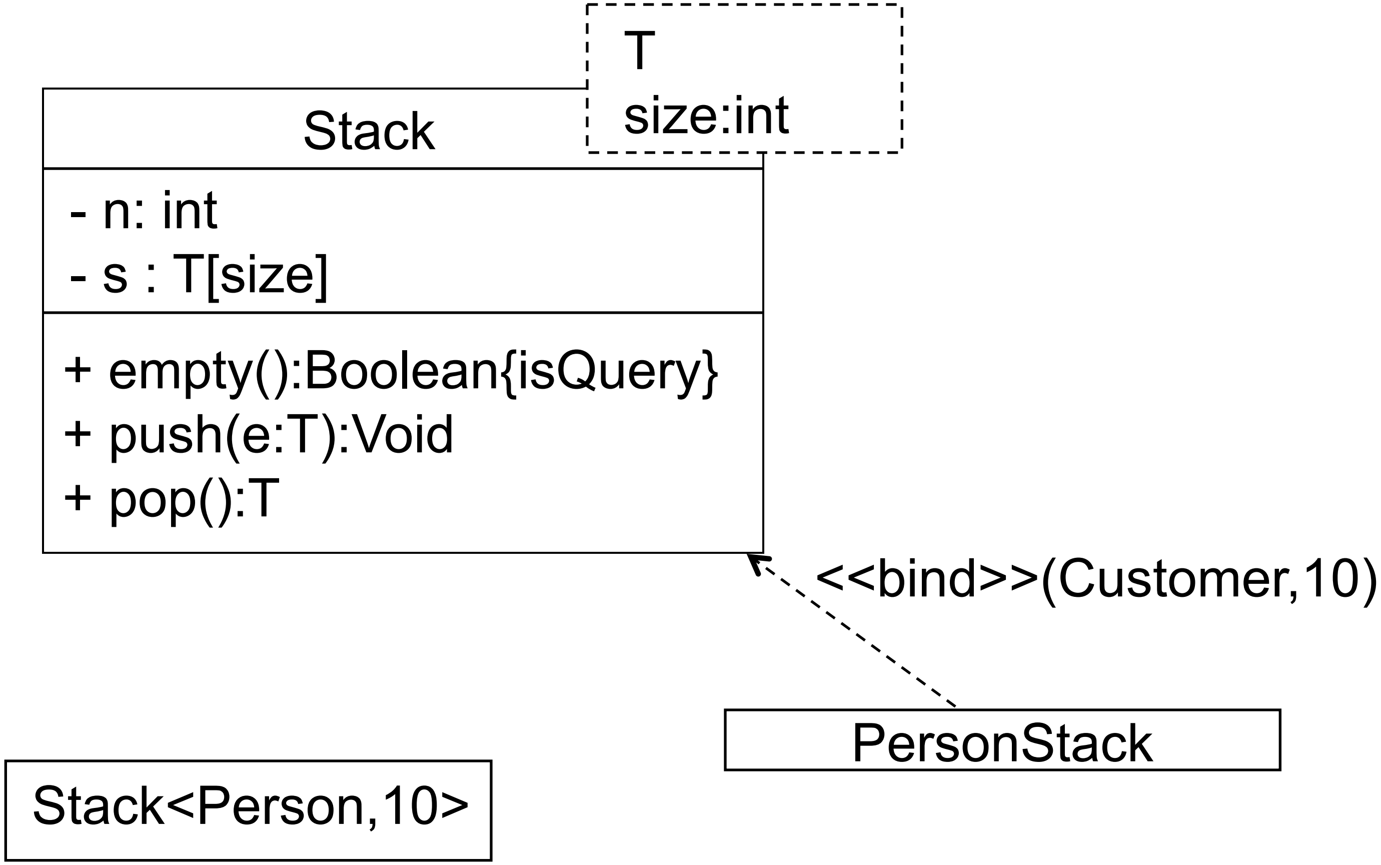
Association name, Verb phrase



Person works for company
Can be read only one way

Role name,
Noun phrase

Class templates



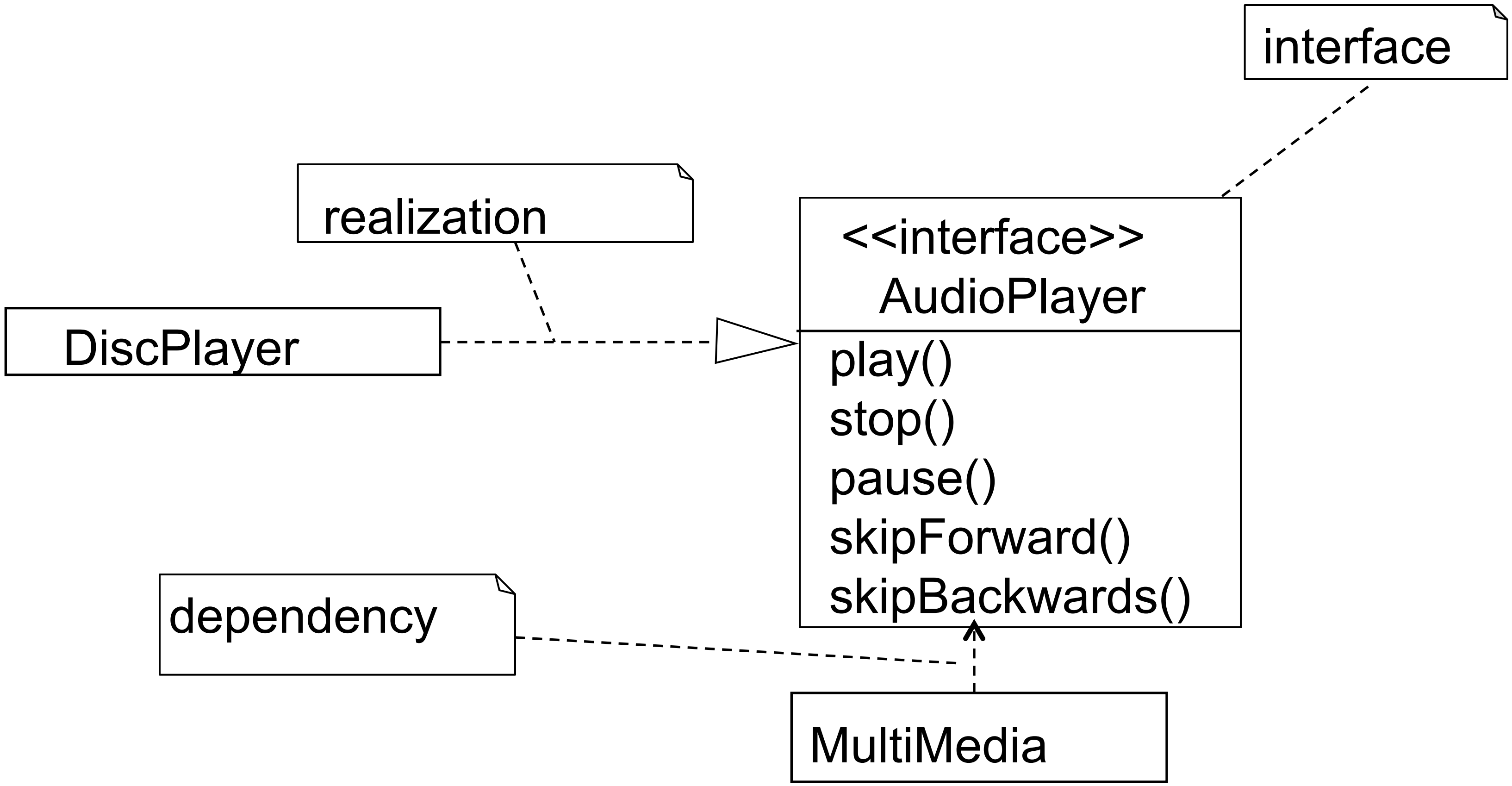
Interface

Interfaces

- Interfaces are very important. By using an interface you can separate implementation from specification.

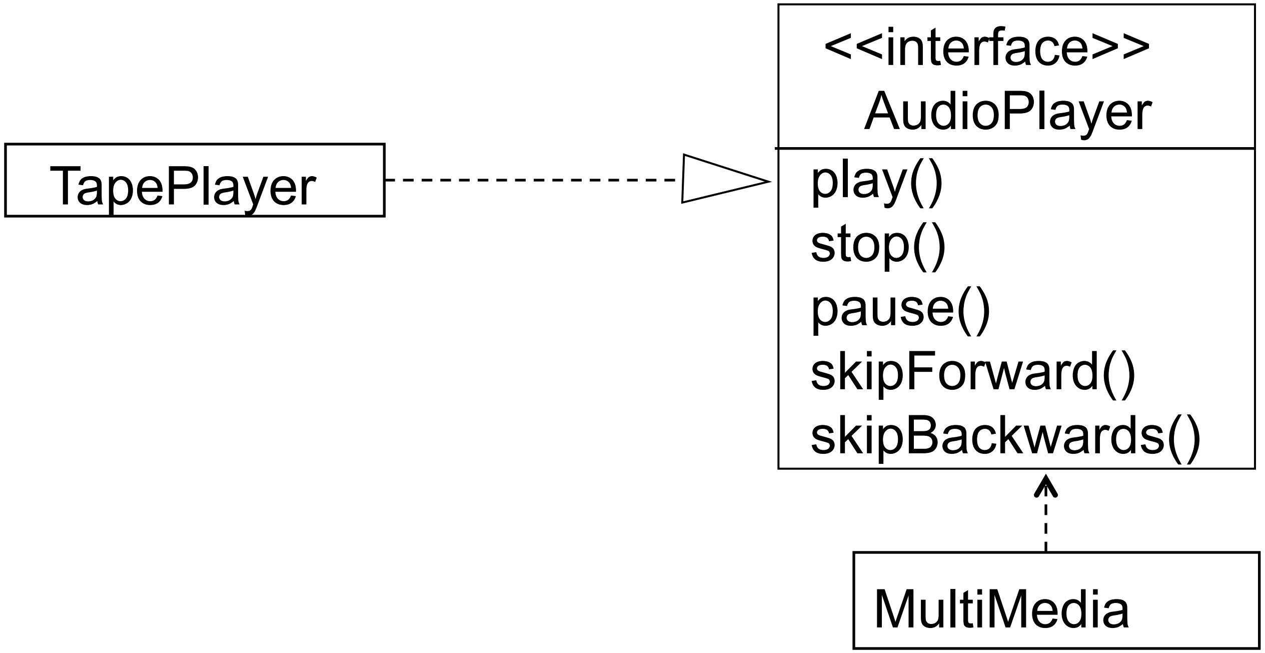
- An interface specifies a service of a class or component.

Interfaces in UML



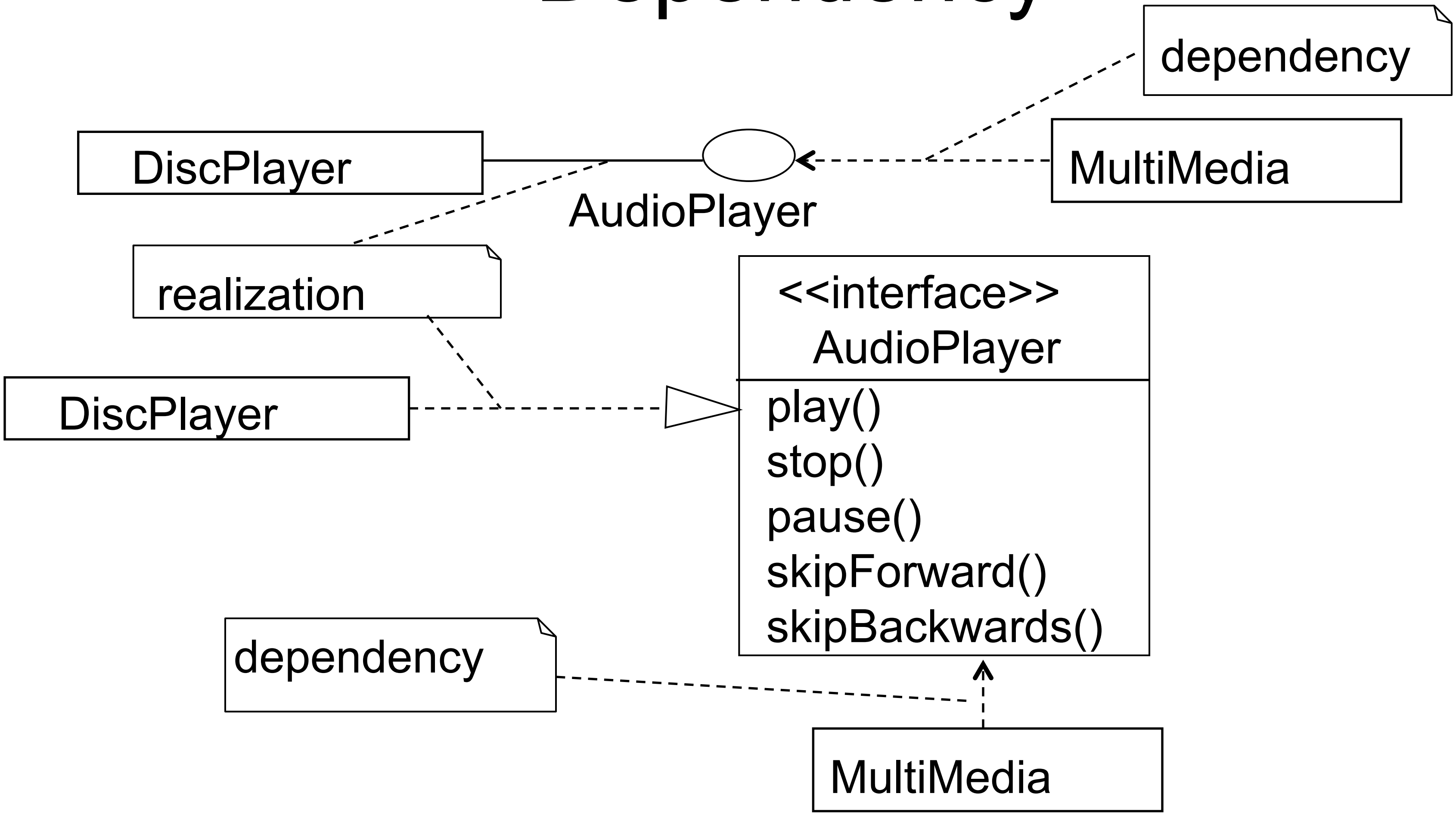
In this lecture we will just look at interfaces connected to classes, but later we will also look at interfaces connected to components.

The same interface



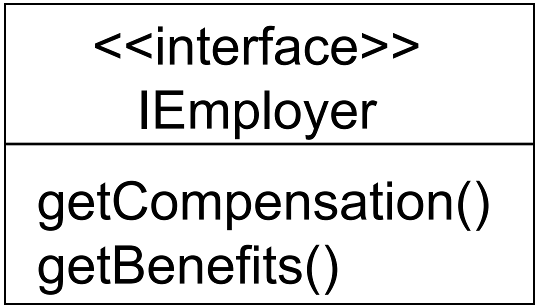
Here **TapePlayer** is a new implementation of **AudioPlayer**. If you have done everything correctly you only have to change the implementation of the methods in the interface, the rest of the program remains the same.
The **MultiMedia** doesn't need to be changed!

Dependency



The class **MultiMedia** uses the methods in the interface, which are implemented by **DiscPlayer**.

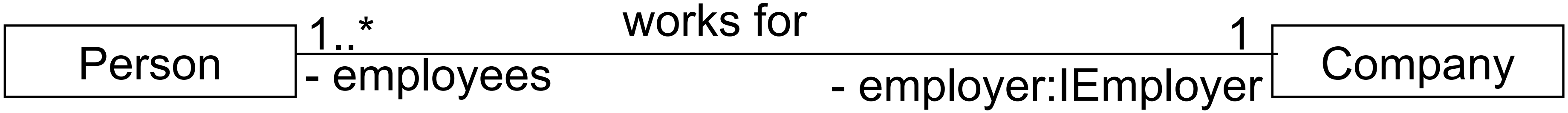
Interface Specifiers



<<interface>>
IEmployer

getCompensation()
getBenefits()

Roles can be shown using interfaces.

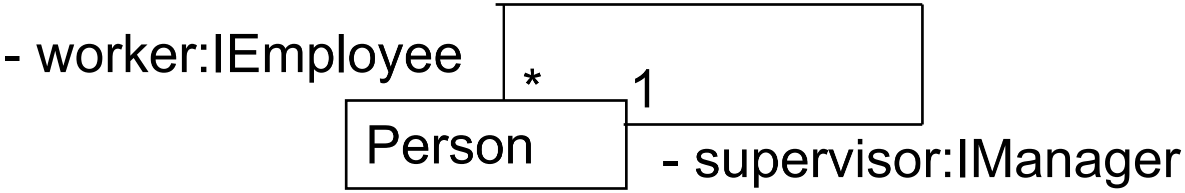


Person 1..* works for 1 Company

- employees

- employer:IEmployer

A person can have many other roles, such as customer, boss, father, pilot etc.



- worker:IEmployee

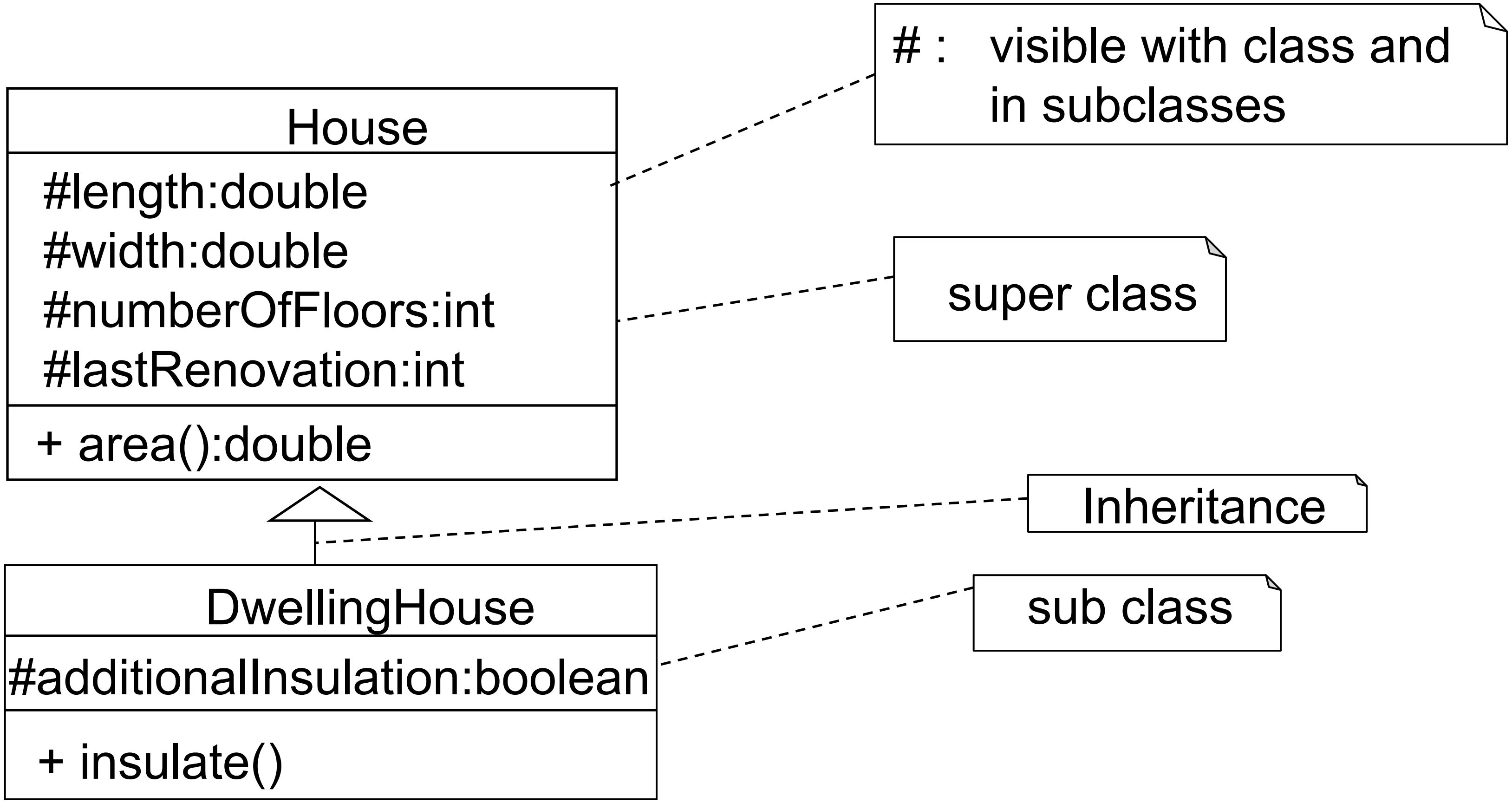
* 1

Person

- supervisor:IManager

Inheritance

Example: Dwelling-house



House

#length:double
#width:double
#numberOfFloors:int
#lastRenovation:int

+ area():double

DwellingHouse

#additionalInsulation:boolean

+ insulate()

# : visible with class and in subclasses

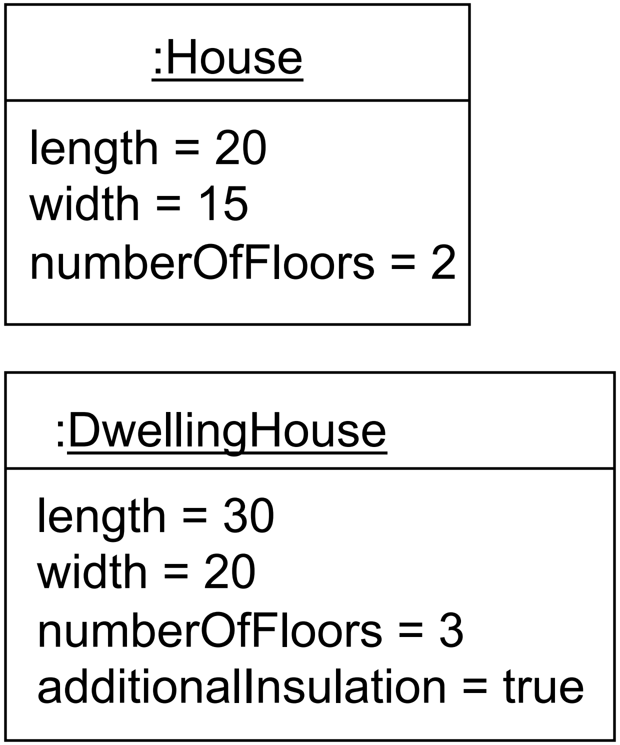
super class

Inheritance

sub class

Instances

Sometimes you want to work with instances of House and sometimes with instances of DwellingHouse etc.



:House

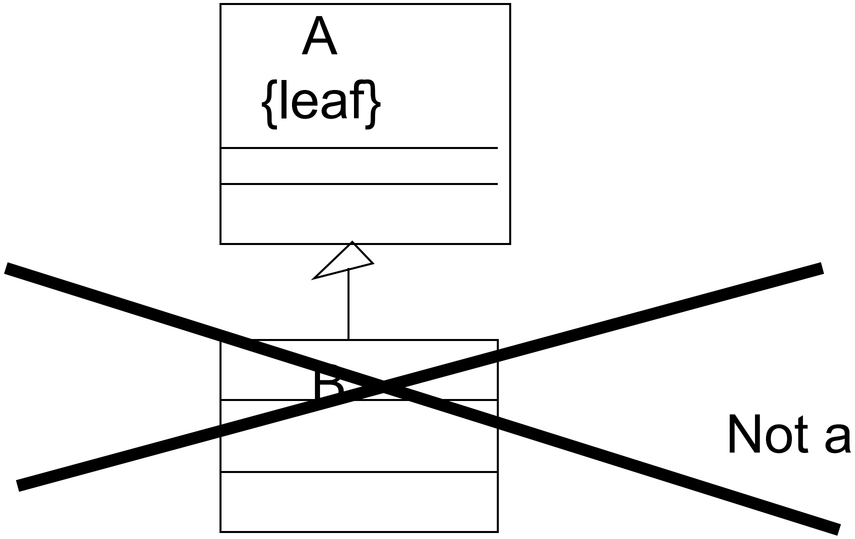
length = 20
width = 15
numberOfFloors = 2

:DwellingHouse

length = 30
width = 20
numberOfFloors = 3
additionalInsulation = true

leaf: stops inheritance

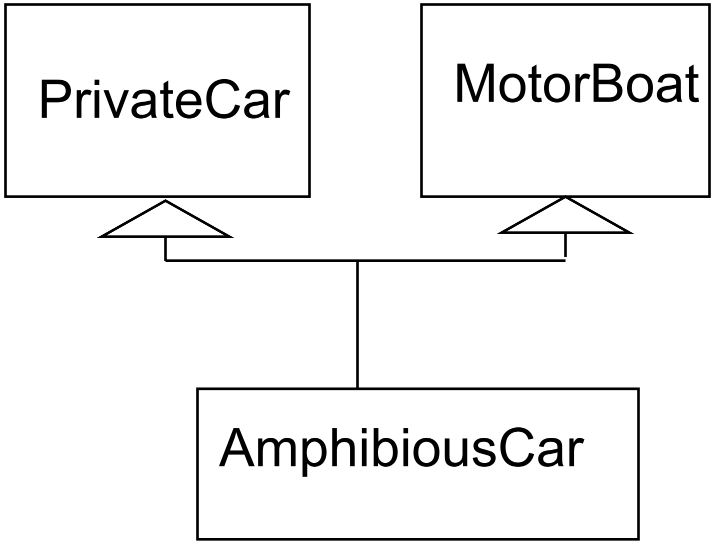
```
public final class A {
      …
}
```



Note that also a method can be final. Then the method must not be changed in the sub classes, e.g.

```
public final int test (int x) {
      …
}
```

Multiple inheritance



- This is allowed in C++, but not in Java.

(But: For interfaces in Java multiple inheritance is allowed)