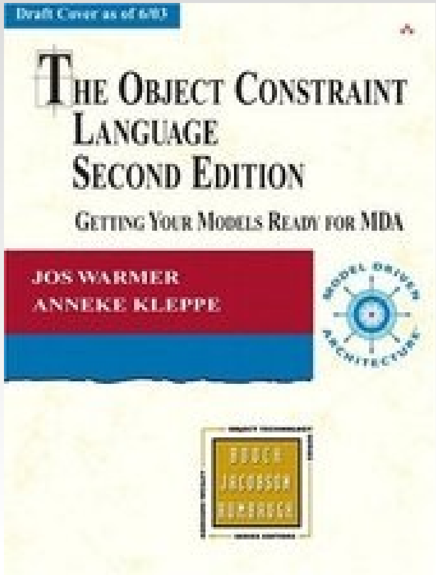
PA103 - Object-oriented Methods for Design of Information Systems

OCL – Object Constraint Language

© Radek Ošlejšek
Fakulta informatiky MU
oslejsek@fi.muni.cz

Literature



Draft Cover as of 6/03

The Object Constraint Language

Second Edition

Getting Your Models Ready for MDA

Jos Warmer

Anneke Kleppe

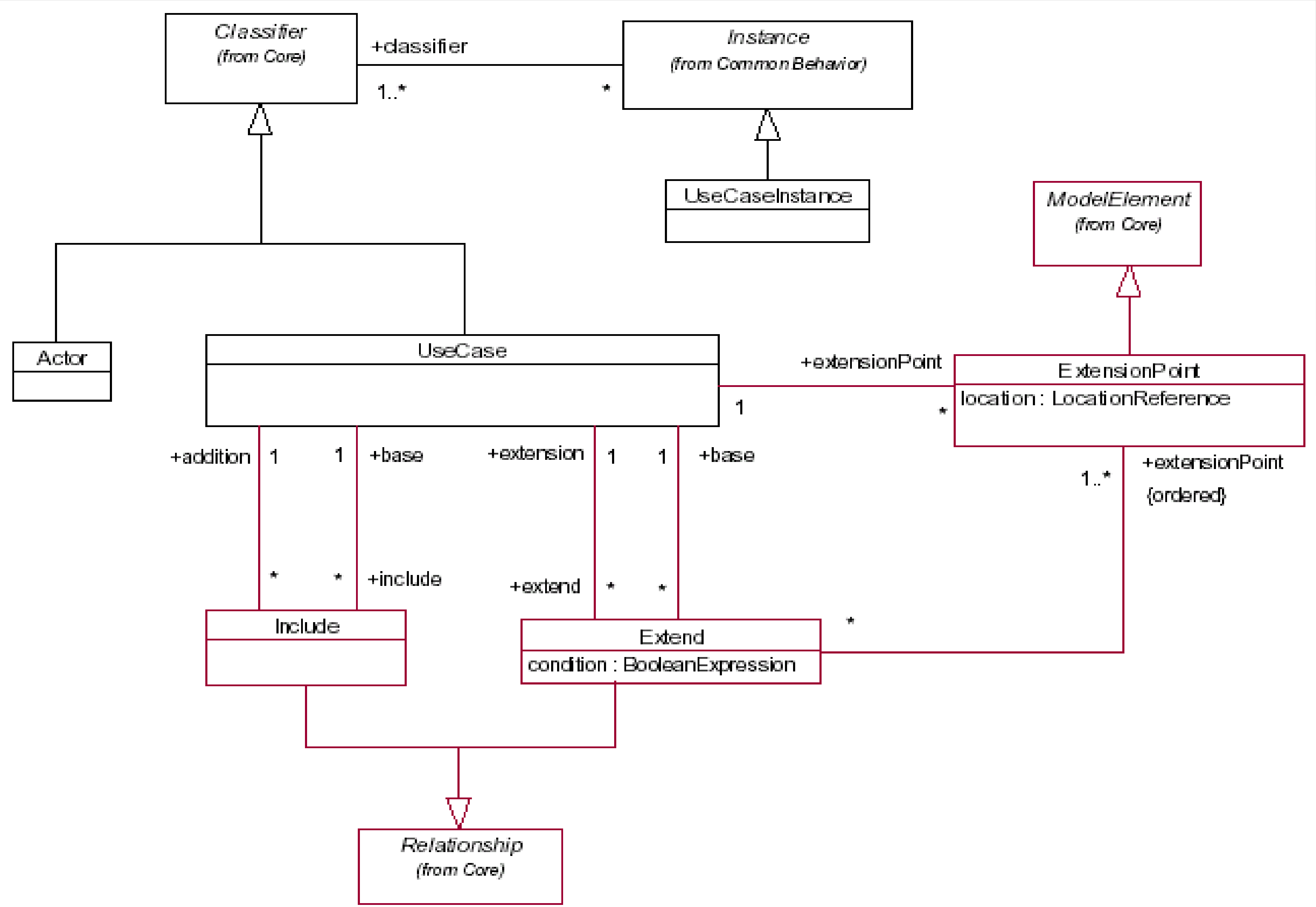
- The Object Constraint Language (Second Edition)
  - Author:J. Warner, A, Kleppe
  - Publisher: Addison-Wesley Professional
  - Copyright: 2003

Lecture 3 / Part 1:
Introduction to OCL

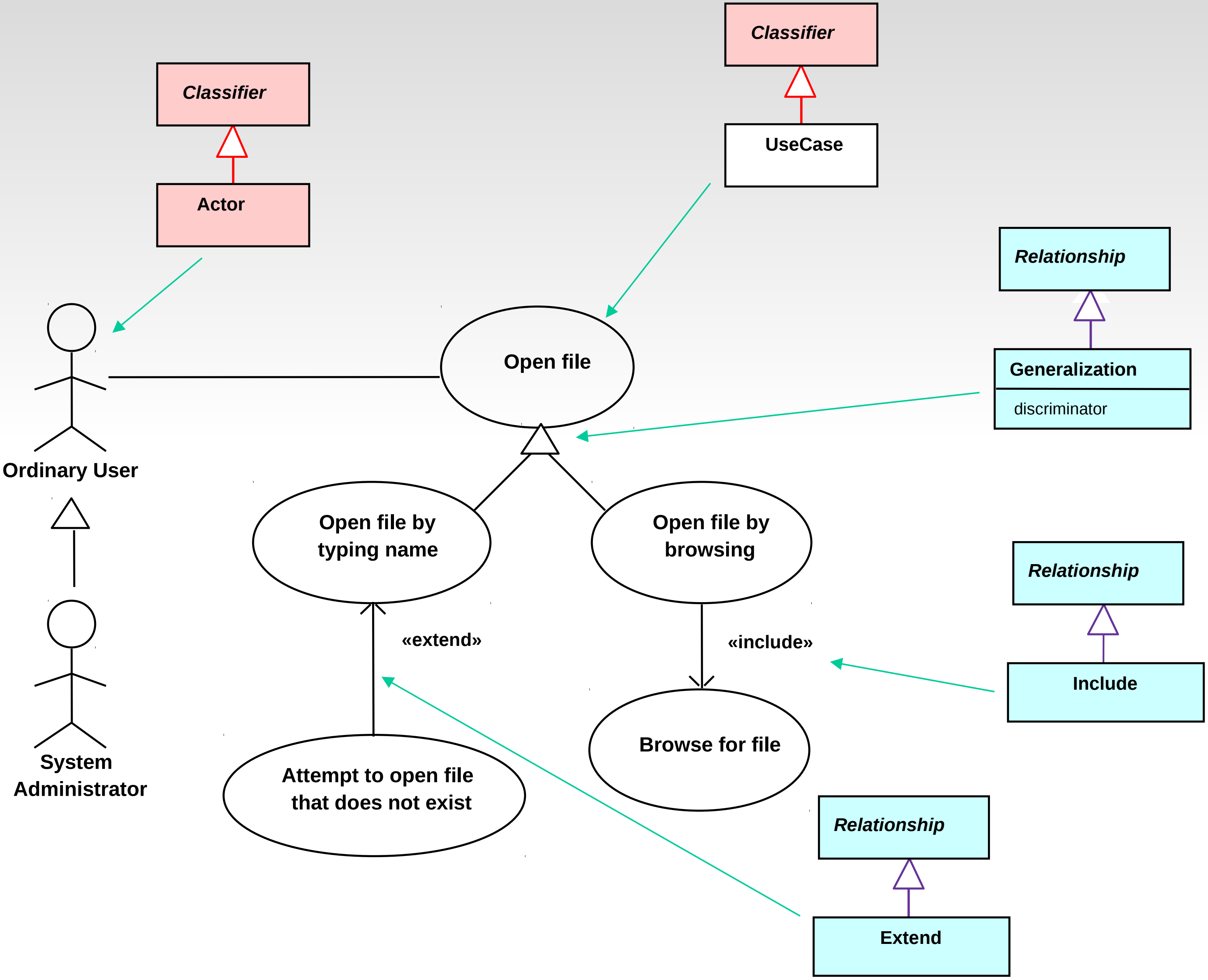
History

- First developed in 1995 as IBEL by IBM's Insurance division for business modeling.
- IBM proposed it to OMG's call for an object-oriented analysis and design standard. OCL was then merged into UML 1.1.
- OCL was used to define UML 1.2 itself (constraints in UML meta-models)
  - UML specification: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF
  - Example: p. 125

UML Metamodels – UC Diagram



UML Metamodels – UC Diagram (cont.)



Why use OCL?

- UML diagrams are not enough. We need a language to help with specification and semantics of UML models.
- We look for some “add-on” instead a brand new language with full specification capability.
- Q: Why not first order logic? A: Not object-oriented.
- OCL is not the only one, but is the only one that is standardized (OMG standard).

Advantages of Formal Constraints

Better documentation

- Constrains add information about model elements and their relationships to the visual models used in UML.
- It is a way of documenting UML models.

More precise

- OCL constrains have formal semantics. Can be used to reduce the ambiguity in the UML models.
- No side effects.
    - Evaluation of OCL cannot affect state of the running system.
    - It is not possible to assign values to attributes via OCL expression.

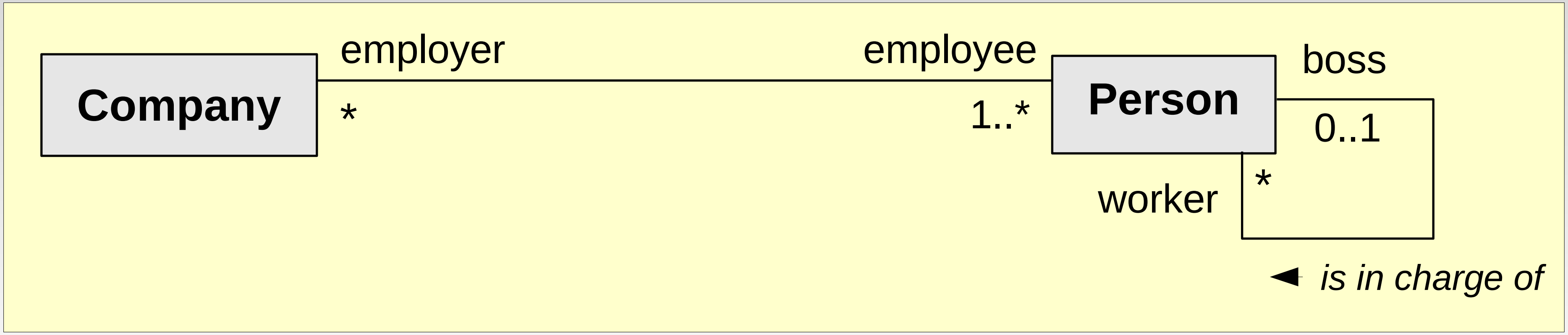
Communication without misunderstanding

- UML models are used to communicate between developers. Using OCL constraints modelers communicate unambiguously.

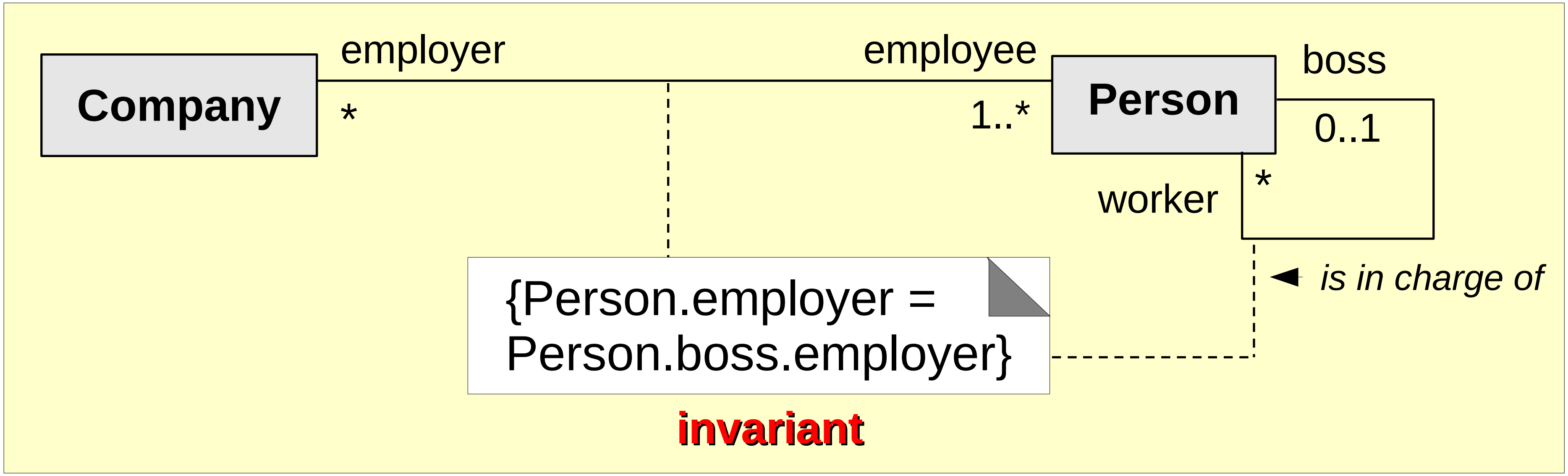
Where use OCL?

- To specify invariants for classes and types.
    - Constraint that must be always met by all instances of the class.
- To specify pre- and post-conditions of an operation.
    - Constraint that must be always true before/after the execution of the operation
- As a navigation language.
    - Syntax constructs enabling to navigate through object links.
- Test requirements and specifications.
- OCL expressions can be bound to any model element in UML.
    - Constrains may be denoted within the UML model or in a separate document

Example 1



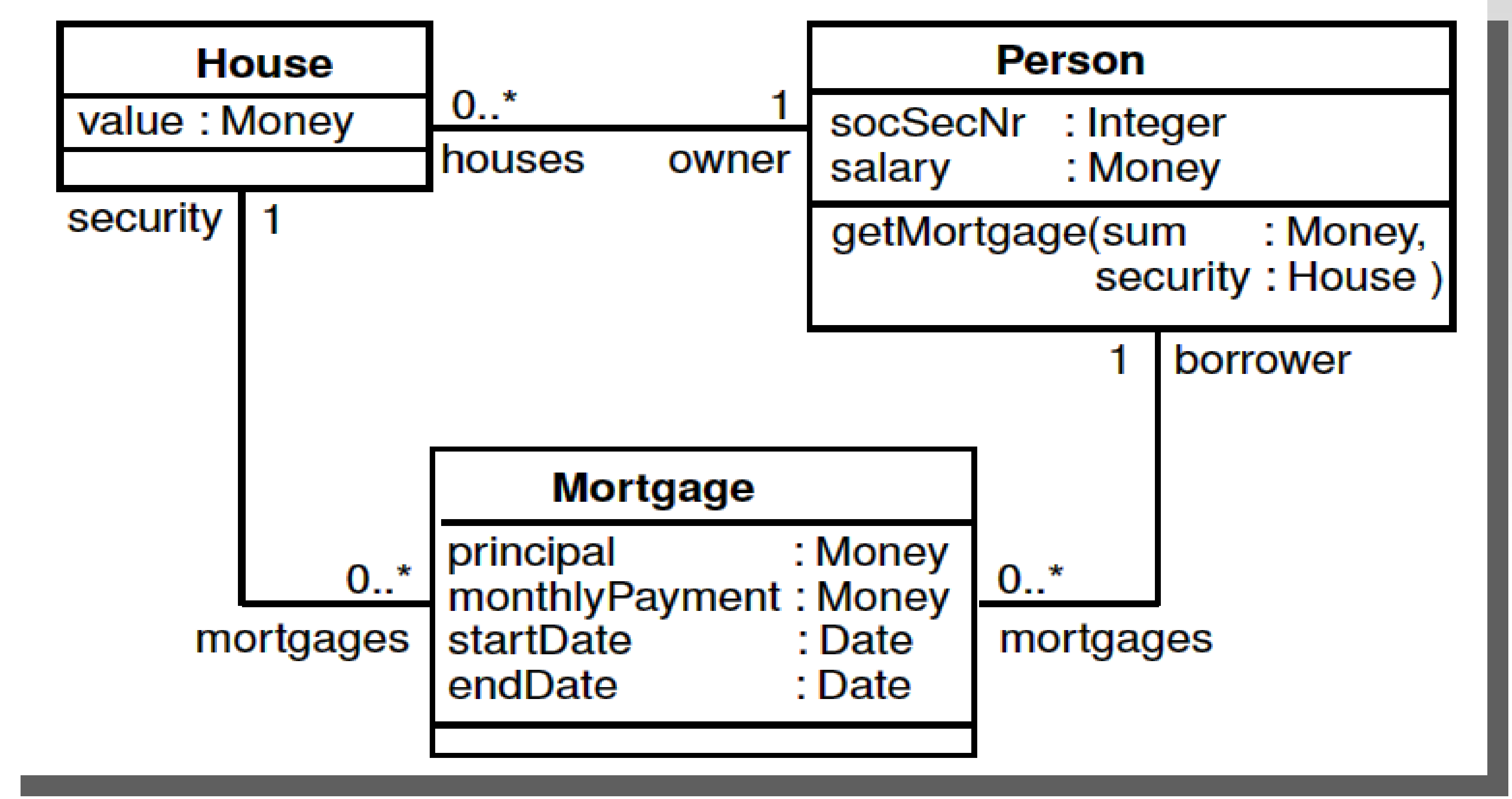
Boss cannot be in charge of employees from other company.



{Person.employer = Person.boss.employer}

invariant

Example 2 (cont.)



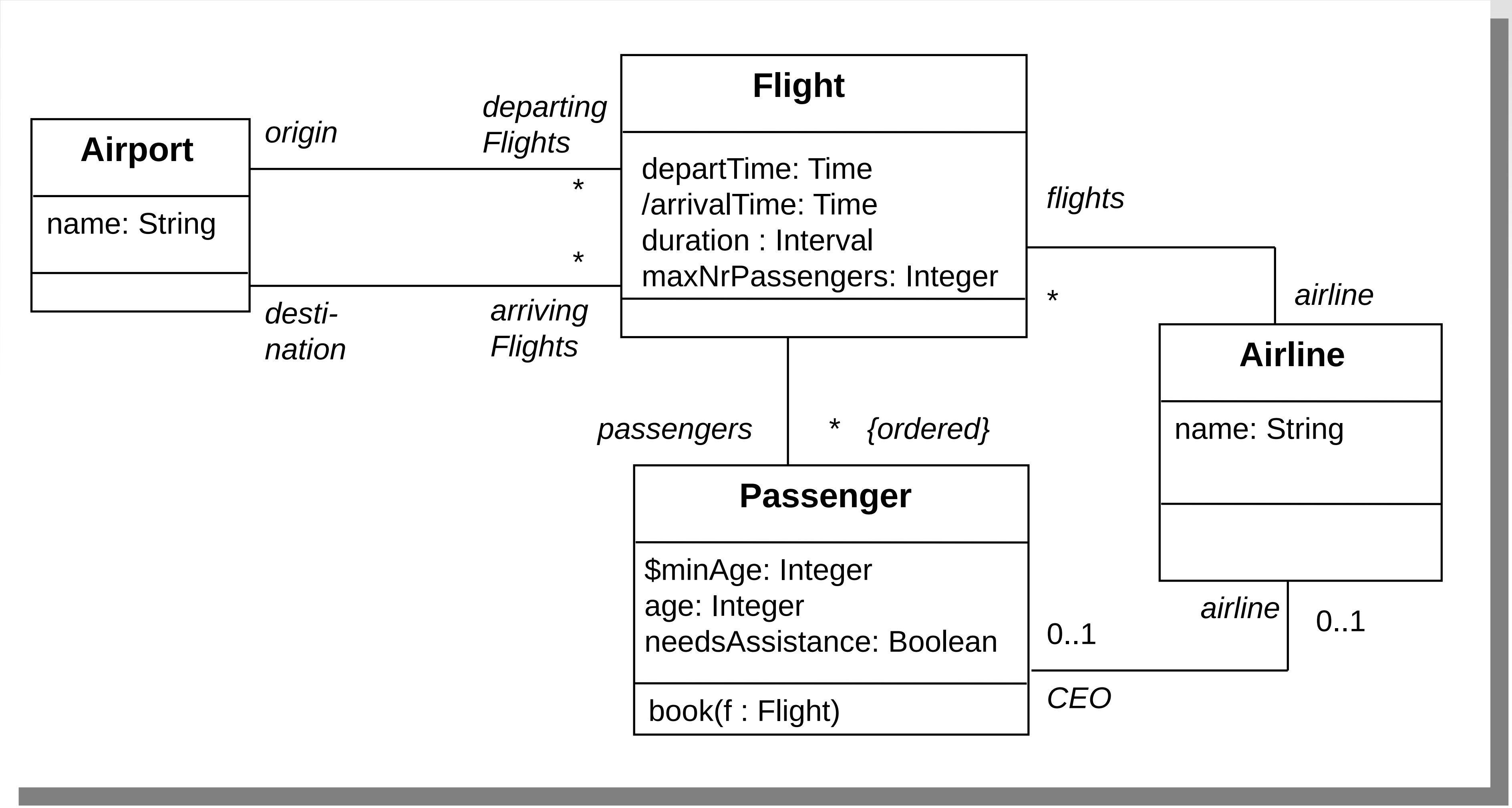
1. A person may have a mortgage only on a house he/she owns.
   context Mortgage **inv:** security.owner = borrower
2. The start date of any mortgage must be before the end date.
   context Mortgage **inv:** startDate < endDate
3. The social security number of all persons must be unique.
4. A new mortgage will be allowed only when the person's income is sufficient

OCL is strongly typed language

- Well-defined OCL expressions have to satisfy type rules
  - e.g. it is not allowed to compare Integer and String
- Every classifier from UML model becomes OCL type
  - e.g. all classes from class diagram
- OCL predefines several basic types and collections

- Note: OCL is declarative language
- Note: text staring with two dashes „--“ is a comment
  - `-- this is uninterpreted comment in OCL`

Reference Model



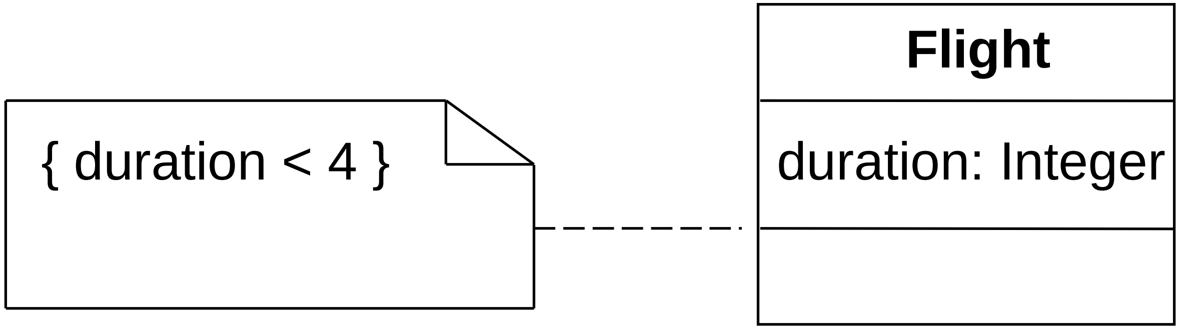
Lecture 3 / Part 2:
Constraints (invariants)

Constrains, Contexts and Self

- **Constraint** (**invariant**) is a boolean OCL expression, evaluates to true/false.
- **Context** links OCL constraint to specific type (class, association class, interface, etc.) in the UML model.
- Context object may be denoted within the expression using the keyword '**self**'.
  - 'self' is implicit in all OCL expressions
  - Similar to 'this' in C++ or Java
- Constraint should have a name followed by the '**invariant**' or '**inv:**' keyword

Context Notation

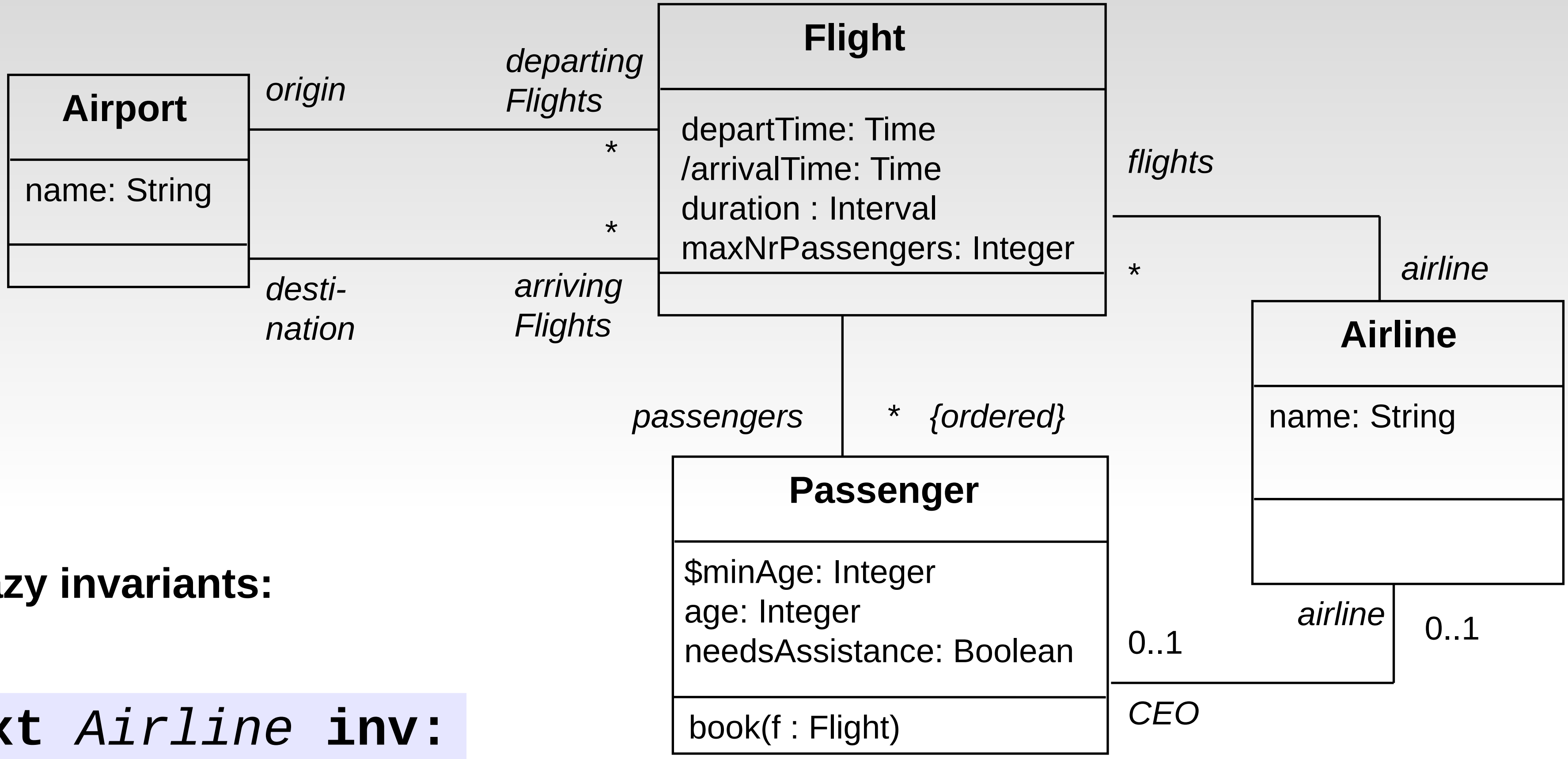
- Constraint may be denoted within the UML model or in a separate document.
- Expression:
    - **context** Flight **inv:** self.duration < 4
- is identical to:
    - **context** Flight **inv:** duration < 4
- is identical to:



Elements of an OCL expression

- **Basic types**:
    - Boolean (true, false),
        - **ops:** `and, or, xor, not, implies, if … then … else … endif`
    - Integer (1, -5, 2, 34, 26524, …),
        - **ops:** `*, +, -, /, abs`
    - Real (1.5, 3.14, …),
        - **ops:** `*, +, -, /, floor`
    - String (‘To be or not to be...’),
        - **ops:** `toUpper, concat, …`
- **Classifiers** from UML models and their features
    - Attributes
    - Query operations
- **Associations** from UML models
    - Including role names at either end of an association

Invariants with Basic Types

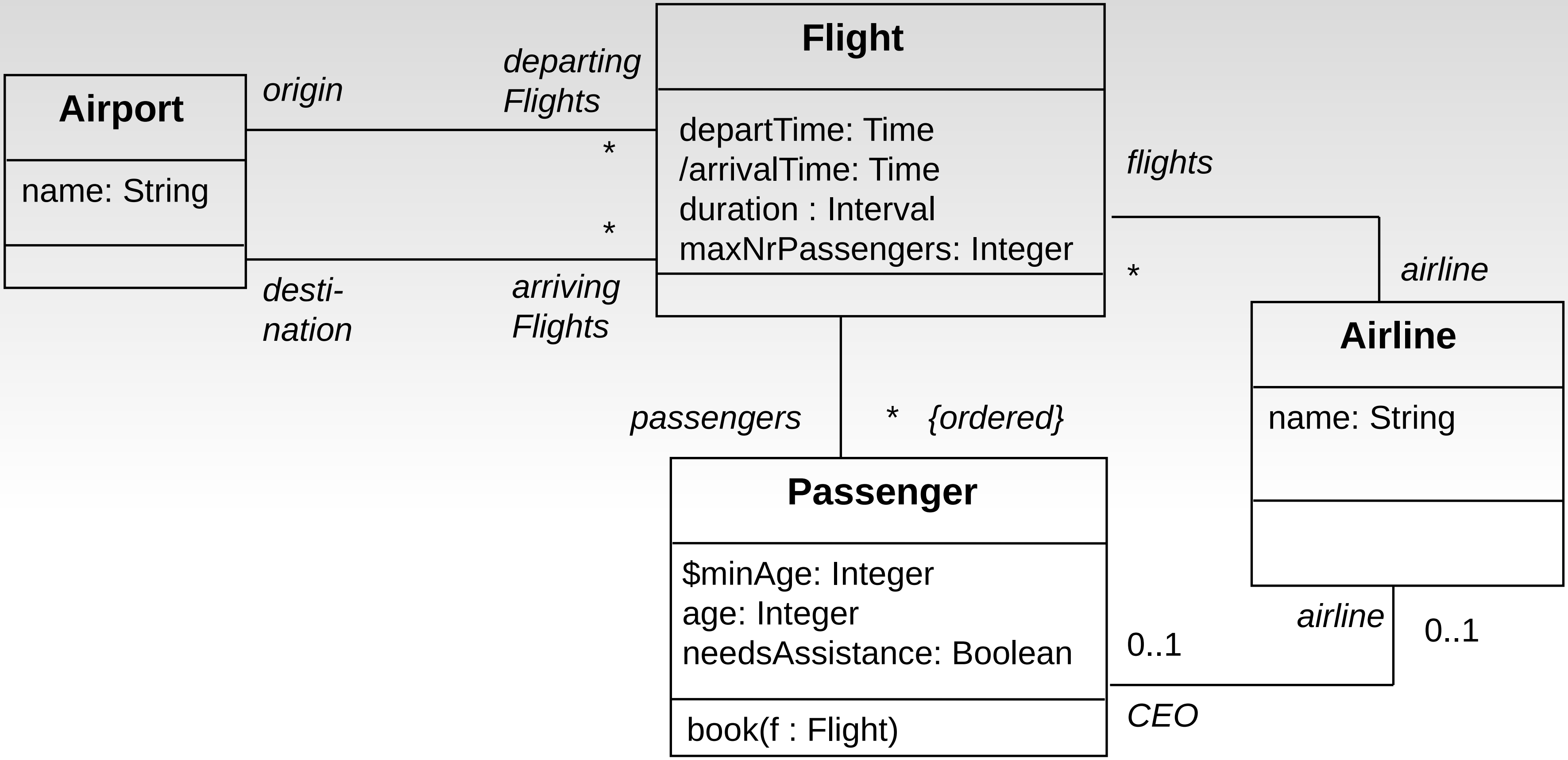


Some crazy invariants:

```
context Airline inv:
name.toLower = ‘klm’
```

```
context Passenger inv:
age >= ((9.6-3.5)*3.1).floor implies mature = true
```

Invariants on Attributes



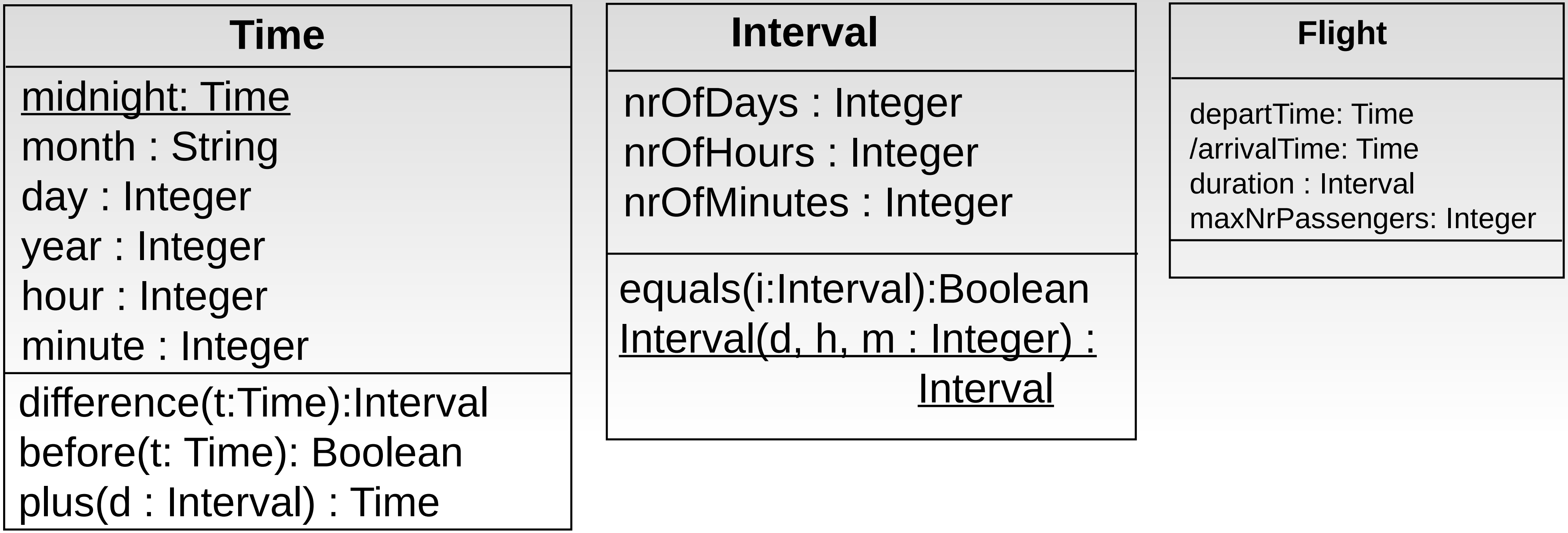
Normal attributes:

```
context Flight inv:
self.maxNrPassengers <= 1000
```

Class (static) attributes:

```
context Passenger inv:
age >= Passenger.minAge
```

Invariants with Query Operations


Time

midnight: Time
month : String
day : Integer
year : Integer
hour : Integer
minute : Integer

difference(t:Time):Interval
before(t: Time): Boolean
plus(d : Interval) : Time

Interval

nrOfDays : Integer
nrOfHours : Integer
nrOfMinutes : Integer

equals(i:Interval):Boolean
Interval(d, h, m : Integer) : Interval

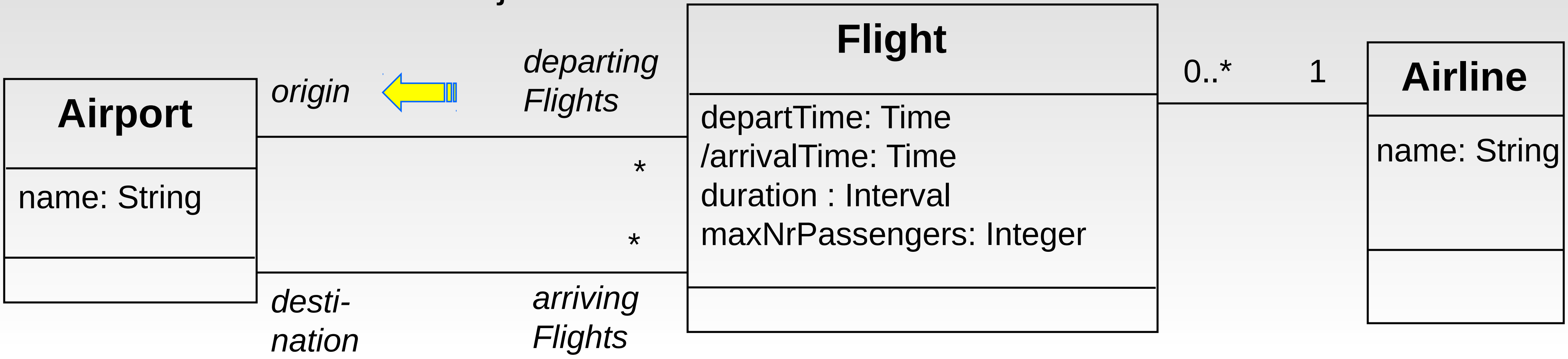
Flight

departTime: Time
/arrivalTime: Time
duration : Interval
maxNrPassengers: Integer

```
context Flight inv:
self.departTime.difference(self.arrivalTime).
   equals(self.duration)

-- Flight duration is just the difference between
-- arrival and departure time.
-- Invariant have to be boolean.
```

Navigation over Association Ends

- Navigation over associations is used to refer to associated objects, starting from the context object:



```
context Flight
inv: origin <> destination
inv: origin.name = 'Amsterdam'
```

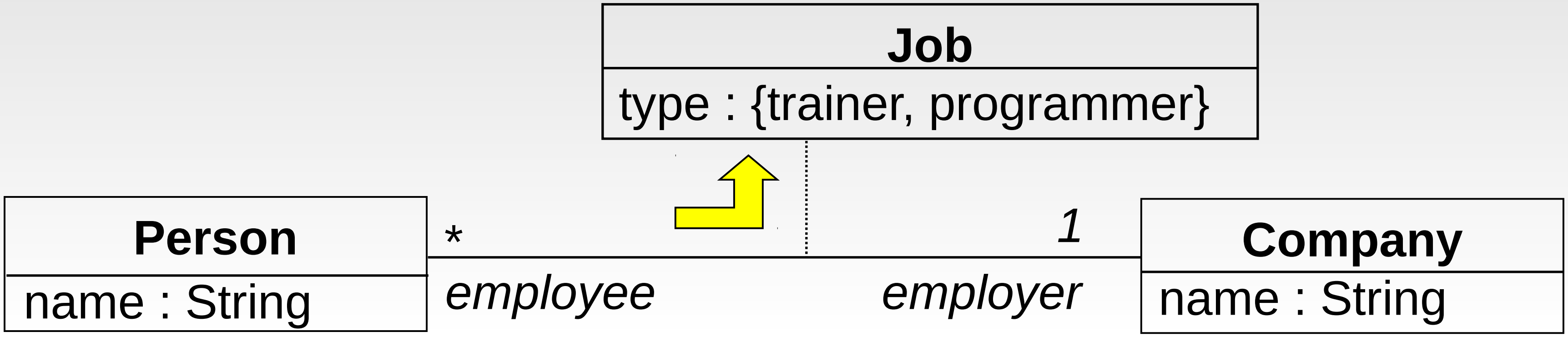
```
context Flight
inv: airline.name = 'KLM'
```

If the role name is missing use class name at the other end of the association, starting with a lowercase letter.

Preferred: Always give role names.

Navigation to Association Classes

- Association classes have no role names. OCL expression therefore has to use class name, starting a lowercase letter:

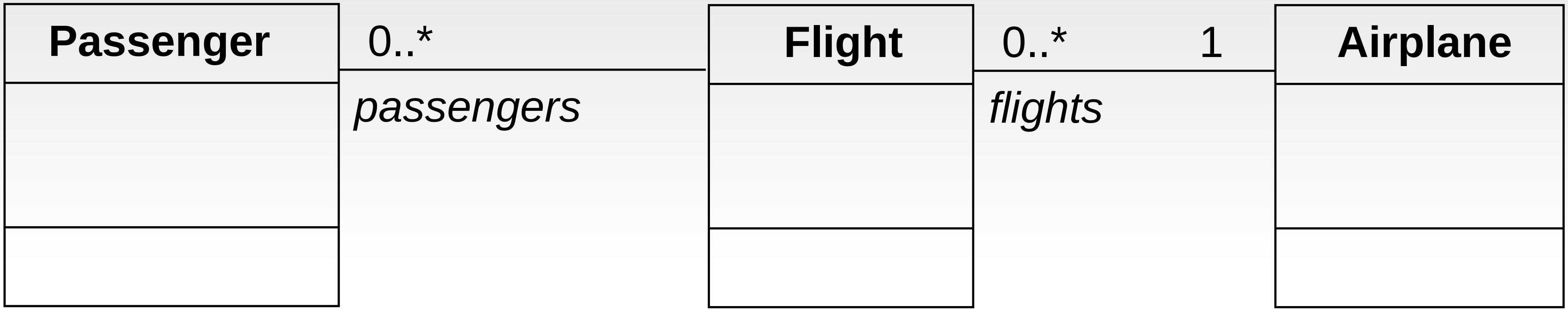


```
context Person inv:
if self.name = ‘Ivan Hrozny’ then
    job.type = #trainer
else
    job.type = #programmer
endif

-- Ivan Hrozny is trainer, other employees are programmers
```

Lecture 3 / Part 3: **Collections**

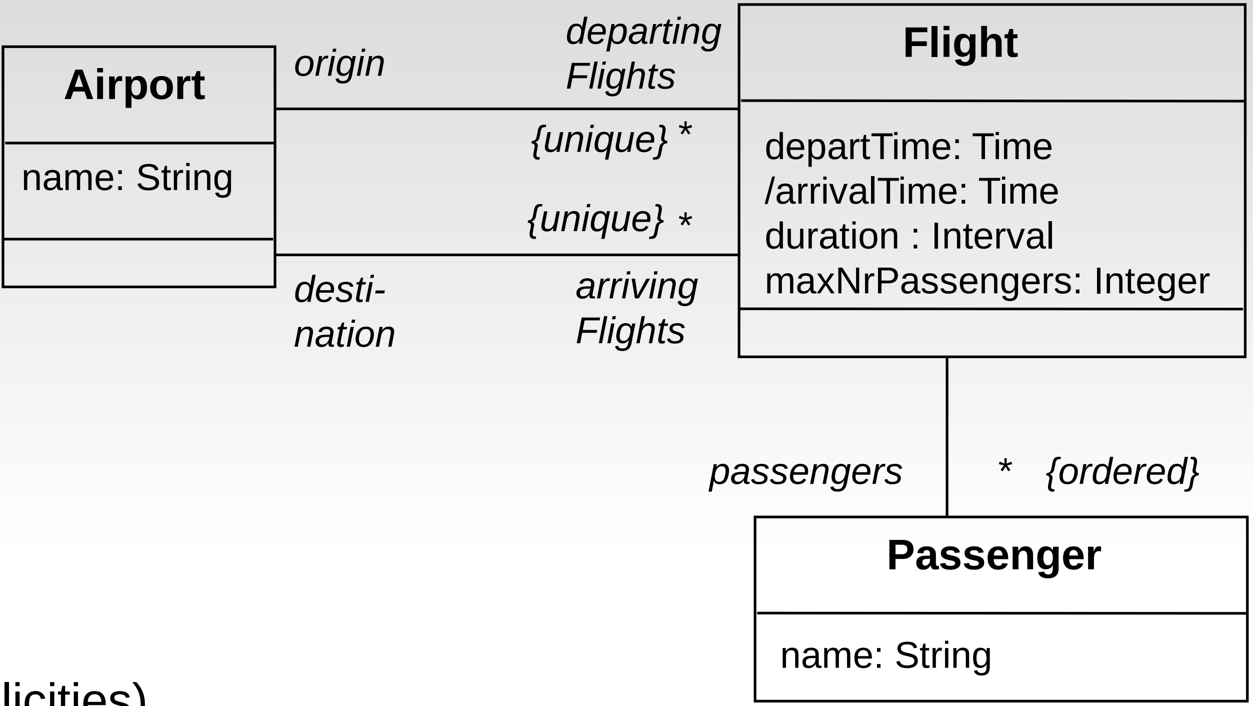
OCL Collections

- Most navigations return collections rather than single elements



```
context Flight inv: airplane. --> single airplane

context Airplane inv: flights. --> collection of flights

context Airplane inv: flights.passengers --> ???
```

OCL Collections (cont.)



- **Set** (non-ordered, no duplicities)

  `context` *`Airport`* **`inv:`** `self.arrivingFlights`

- **Bag** (non-ordered, duplicities)

  `context` *`Airport`* **`inv:`** `self.arrivingFlights.passengers.name`

- **Sequence** (ordered, duplicities)

  `context` *`Flight`* **`inv:`** `self.passengers`

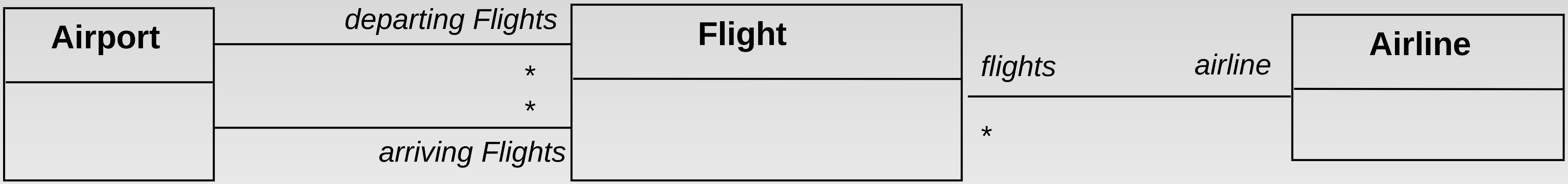
The *collect* Operation

Here can be an arbitrary name of the collection

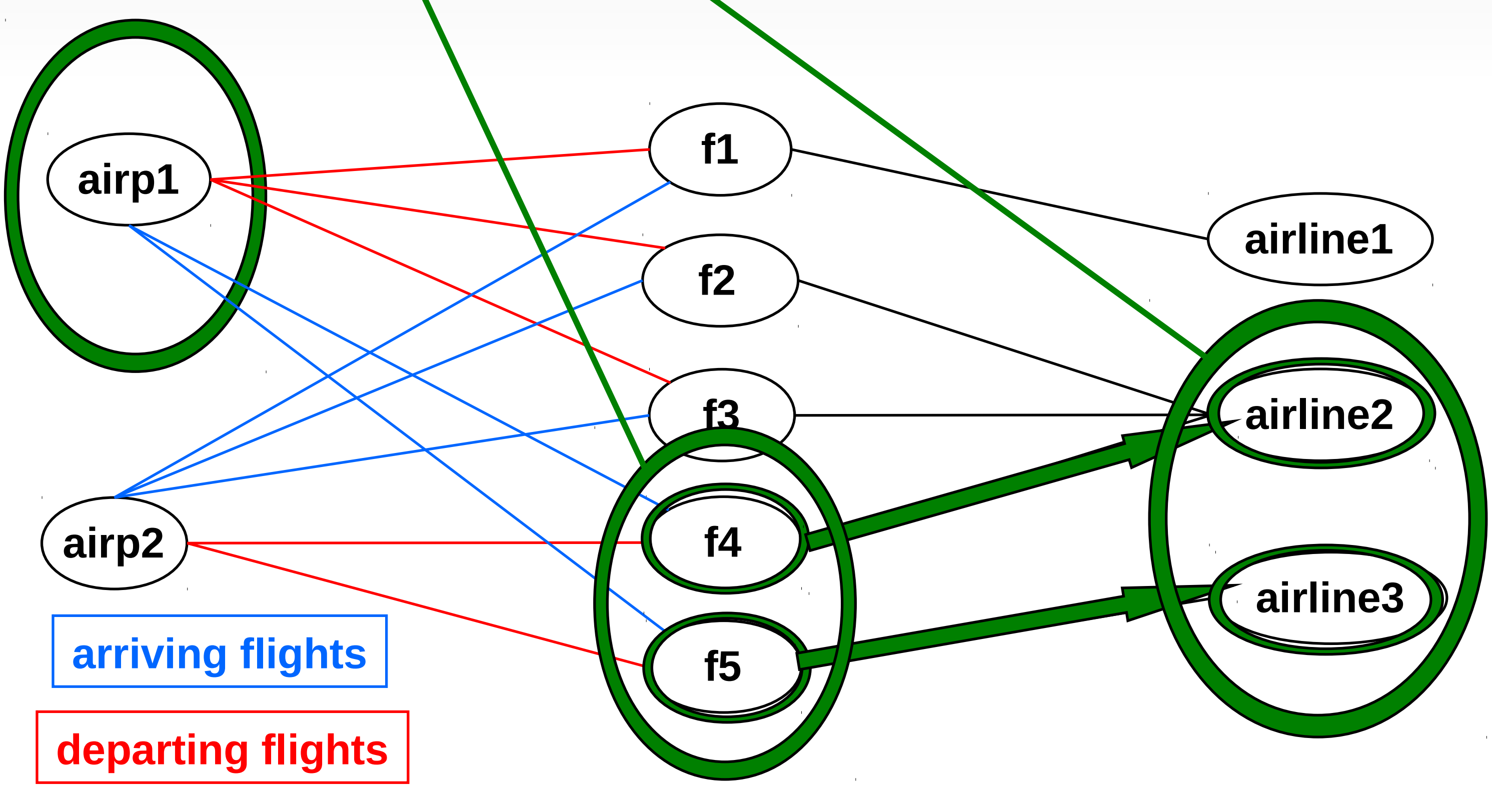
Arrow "->" is used for predefined operations instead of the dot "." operator

- Syntax:
  - `collection->collect(elem : T | expr)`
  - `collection->collect(elem | expr)`
  - `collection->collect(expr)`
  - `collection.expr    -- abbreviated syntax`
- The *collect* operation returns a *bag* containing the value of the expression *expr* for each of the items in the *collection.*
- For instance, it is used to get all values for certain attribute of all objects in a collection.
- Similar to *projection* in relational algebra (SQL).

Example: *collect* Operation



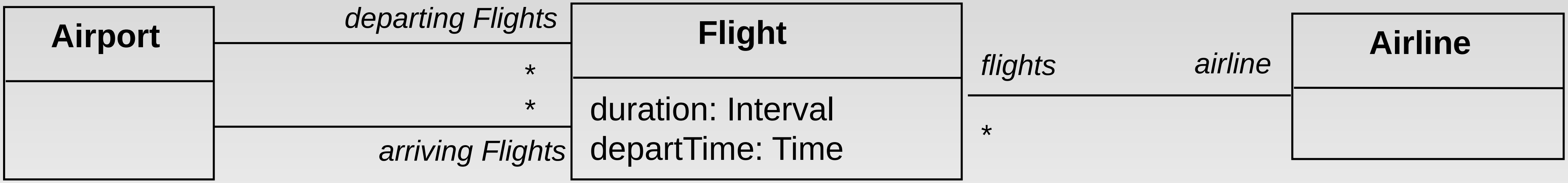
```
context Airport inv:
self.arrivingFlights -> collect(airline) -> notEmpty
```



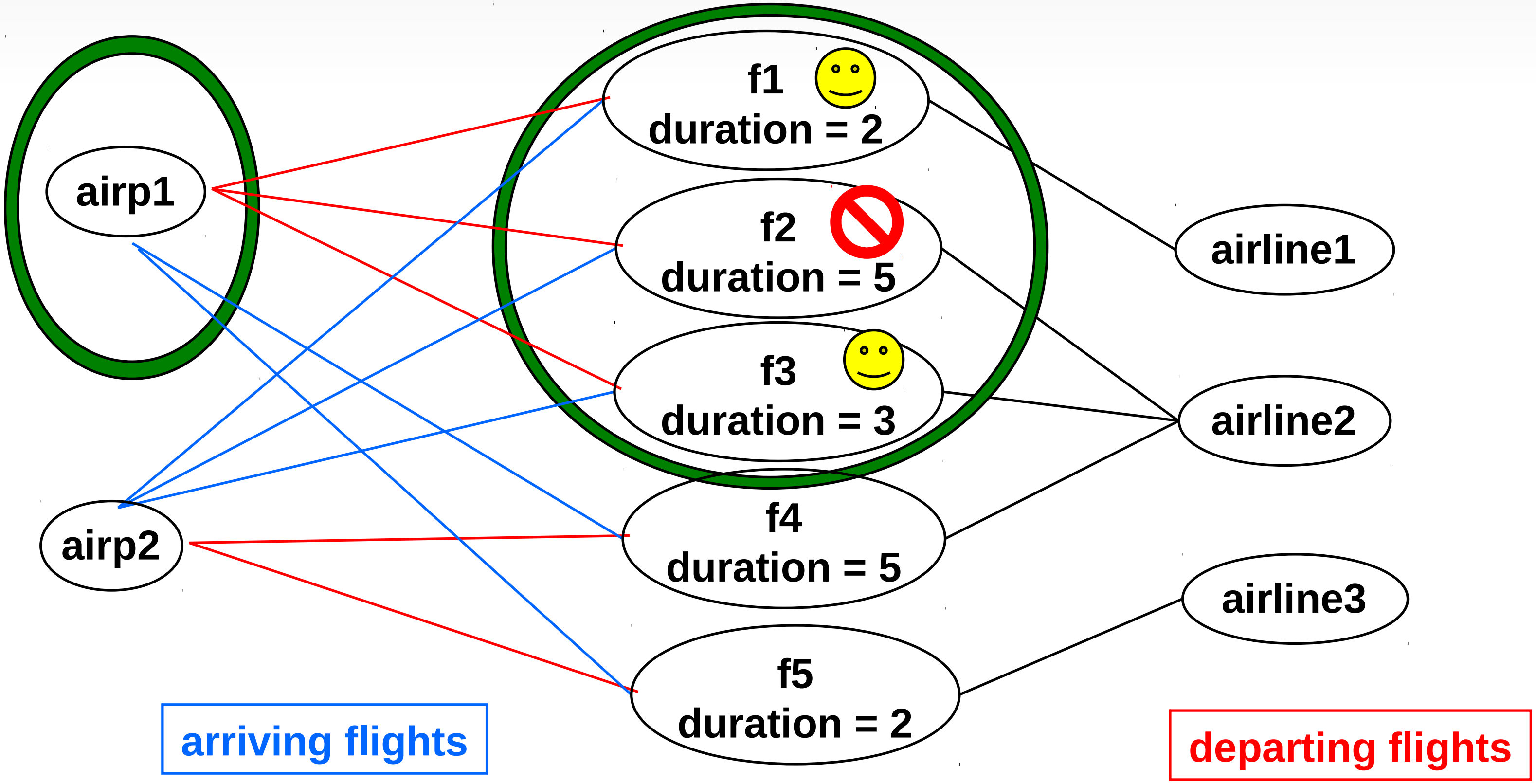
The *select* and *reject* Operations

- Syntax:
  - `collection->select(elem : T | expr)`
  - `collection->select(elem | expr)`
  - `collection->select(expr)`
- The *select* operation results in the subset of elements for which *expr* is true.
- Similar to *selection* in relational algebra (SQL).
- *reject* is the complementary operation to *select*.

Example: *select* Operation



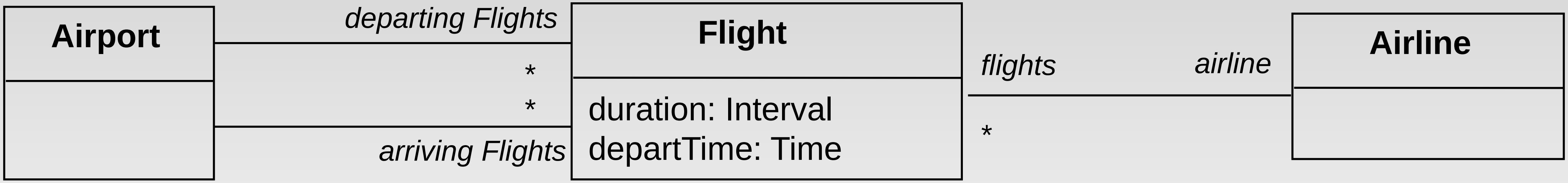
```
context Airport inv:
self.departingFlights->select(duration<4)->notEmpty
```



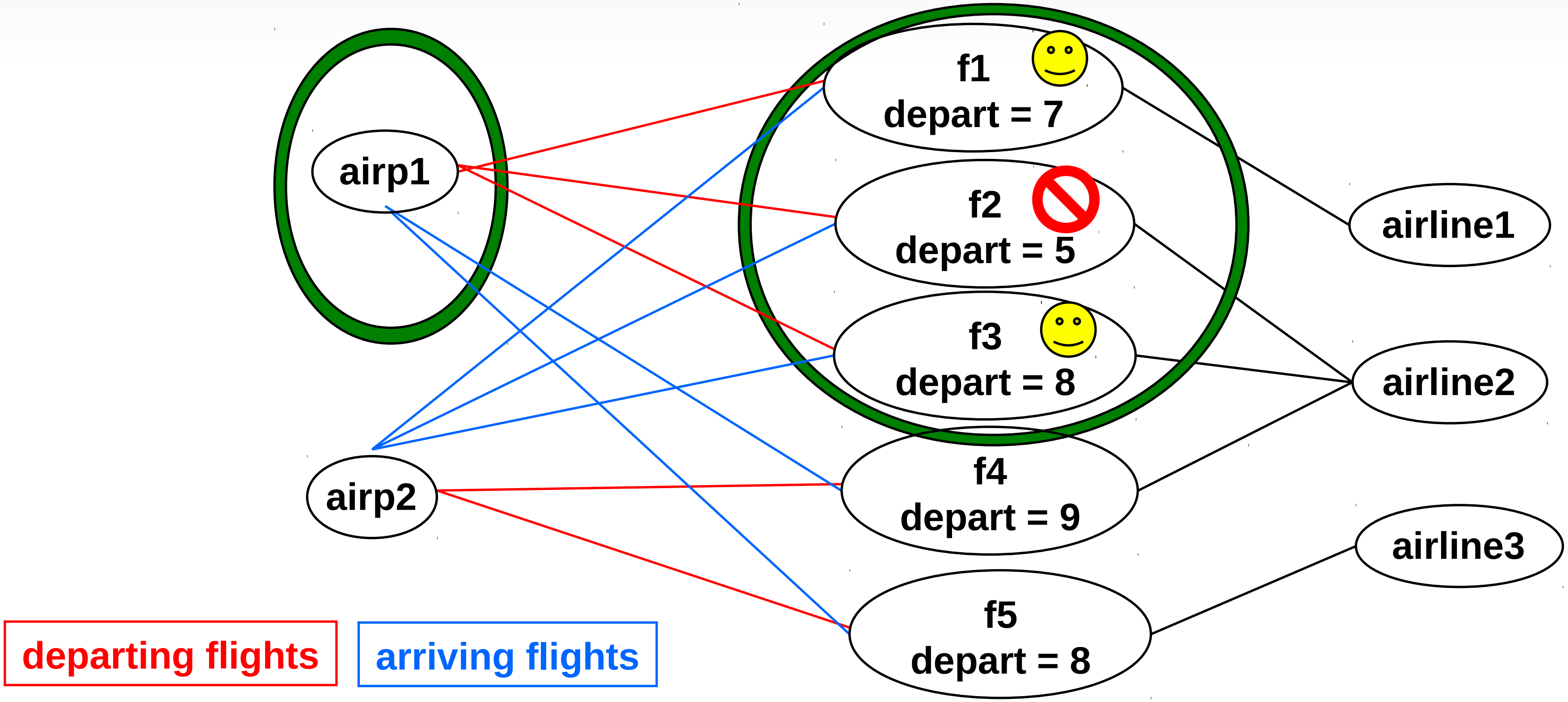
The *forAll* Operation

- Syntax:
  - `collection->forAll(elem : T | expr)`
  - `collection->forAll(elem | expr)`
  - `collection->forAll(expr)`
- The *forAll* operation results in true if *expr* is true for all elements of the *collection*.

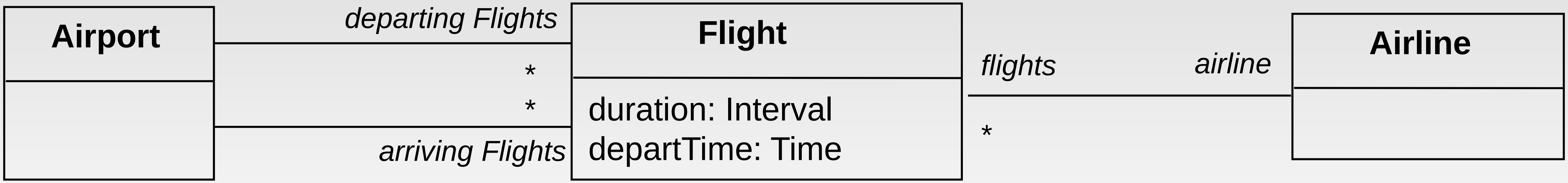
Example: *forAll* Operation



```
context Airport inv:
self.departingFlights->forAll(departTime.hour>6)
```



forAll Operation with two variables

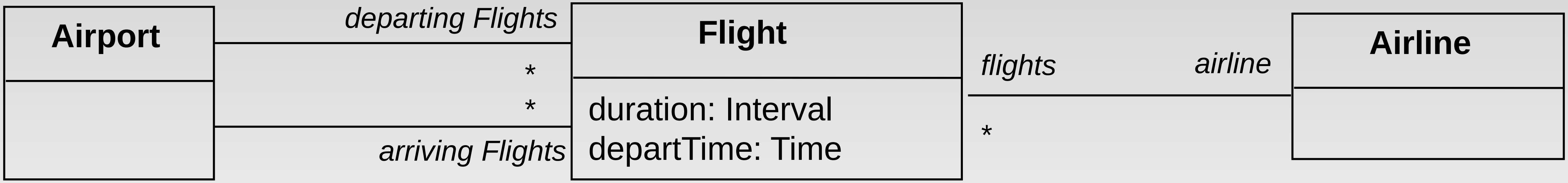


```
context Airport inv:
self.departingFlights->forAll(f1, f2 |
  f1.departTime <> f2.departTime)


-- all flights differ in their departure time
```

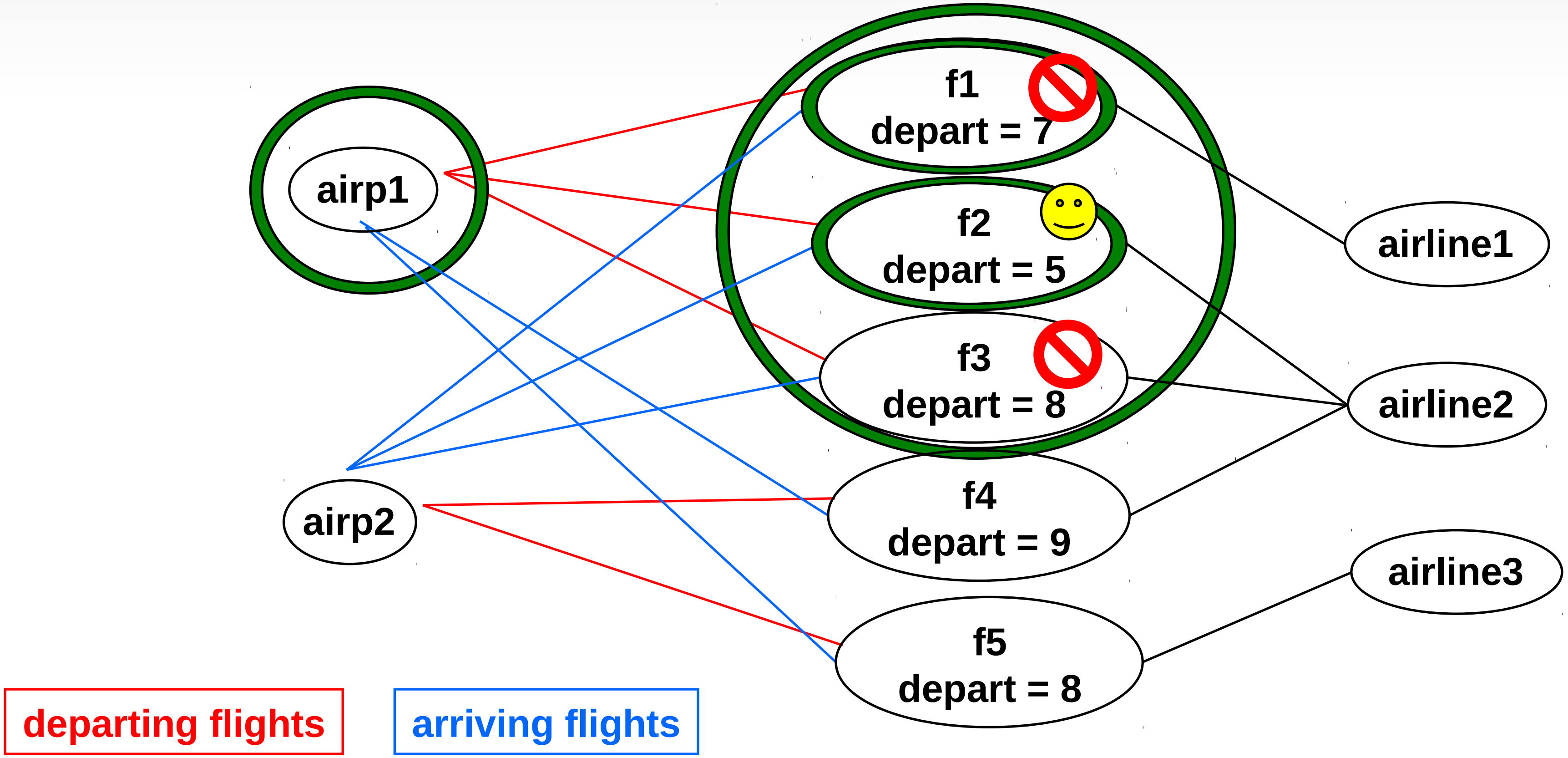
The *exists* Operation

- Syntax:
  - `collection->exists(elem : T | expr)`
  - `collection->exists(elem | expr)`
  - `collection->exists(expr)`
- The *exists* operation results in true if there is at least one element in the *collection* for which the *expr* is true.

Example: *exists* Operation



```
context Airport inv:
self.departingFlights->exists(departTime.hour<6)
```



The *iterate* Operation

- Syntax:
  - `collection->iterate(elem : Type; answer : Type = <value> | <expression-with-elem-and-answer>)`
- Example:

```
context Airline inv:
flights->iterate (f : Flight;
                  answer : Set(Flight) = Set{ } |
                  if f.maxNrPassengers > 150 then
                        answer->including(f)
                  else
                        answer
                  endif)->notEmpty
```

- is identical to:

```
context Airline inv:
flights->select(maxNrPassengers > 150)->notEmpty
```

Other collection Operations

- *c->isEmpty*: true if collection has no elements.
- *c->notEmpty*: true if collection has at least one element.
- *c->size*: number of elements in collection.
- *c->sum*: summation of numerical elements in collection.
- *c->count(elem)*: number of occurrences of elem in collection.
- *c->includes(elem)*: true if elem is in collection.
- *c->excludes(elem)*: true if elem is not in collection.
- *c->includesAll(coll)*: true if every element of *coll* is found in *c*.
- *c->excludesAll(coll)*: true if no element of *coll* is found in *c*.

Other collection Operations (cont.)

For bags and sequences:

- *c-> asSet*: transforms *bag* or *sequence* collection to *set*, i.e. removes duplicities.

For sets:

- *s1->intersection(s2)*: returns set of those elements found in *s1* and also in *s2*.
- *s1->union(s2)*: returns set of those elements found in either *s1* or *s2*.
- *s1->excluding(x)*: returns *s1* with object *x* omitted.

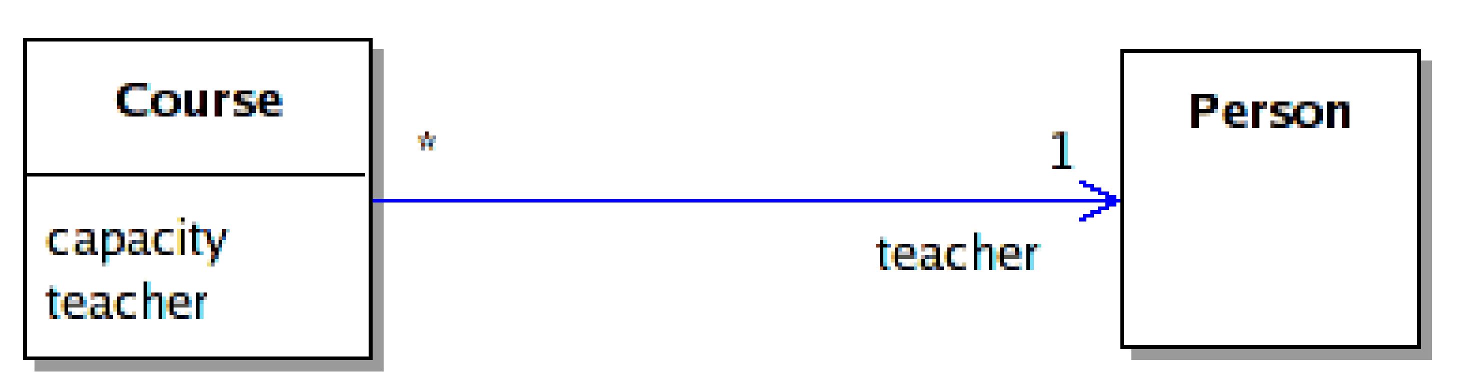
For sequences:

- *s->first()*

Predefined set-theoretic predicates:

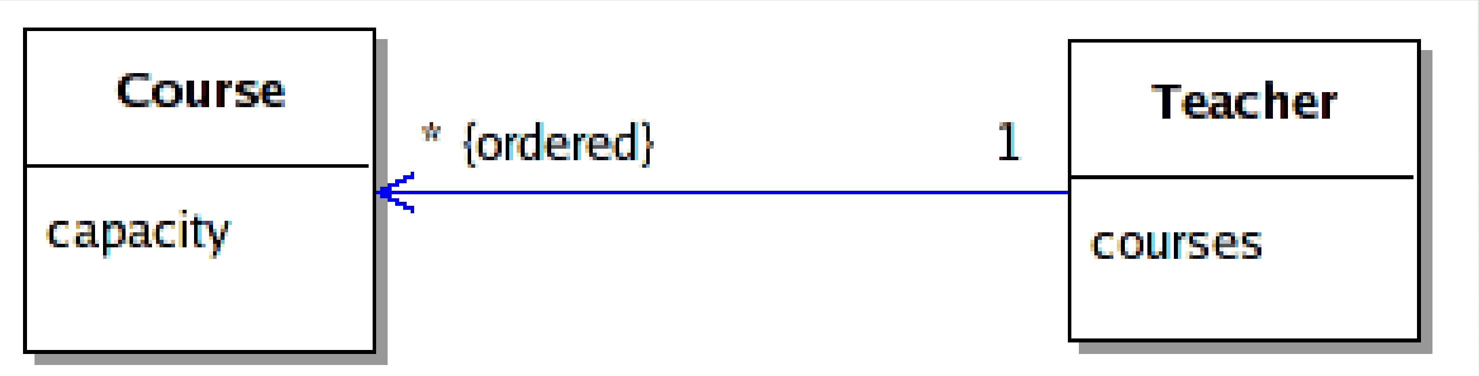
- *size(set), sum(set), max(set), average(set), ...*

Exercise



What returns `self.capacity` for context *Course*?
What returns `self.teacher` for context *Course*?

Exercise (cont.)



What returns `self.courses` for context *Teacher*?
What returns `self.courses->first().capacity` for context *Teacher*?
What returns `self.courses.capacity` for context *Teacher*?
What is the result of:

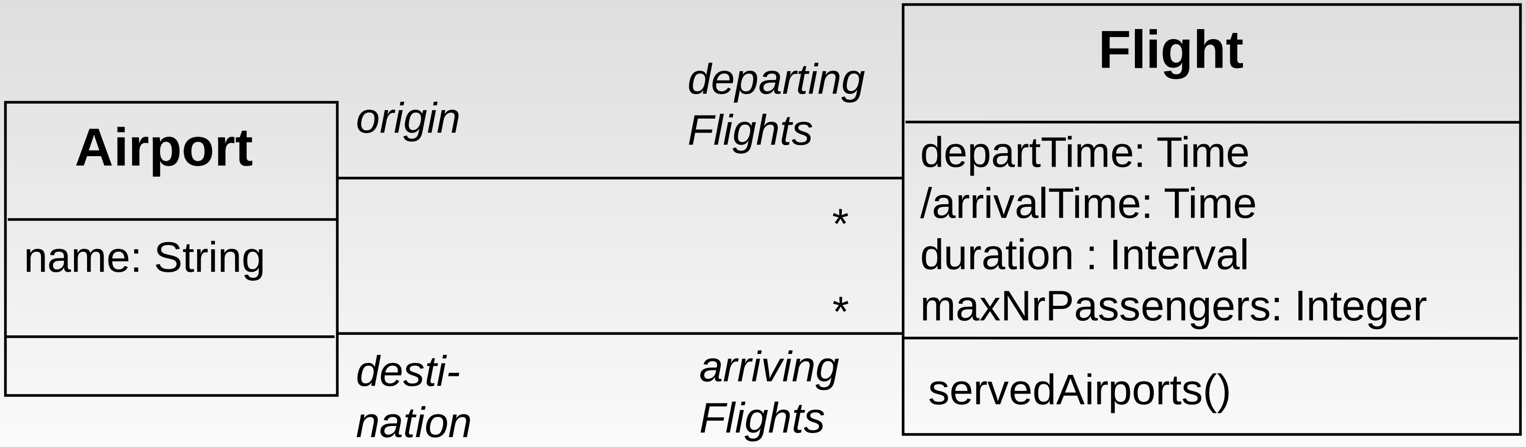
```
    context Teacher inv: self.courses->size() > 0
```

What is the result of:

```
    context Teacher inv: max(self.courses.capacity) < 150
```

Lecture 3 / Part 4:
Advanced OCL Concepts

Pre- and Post-conditions

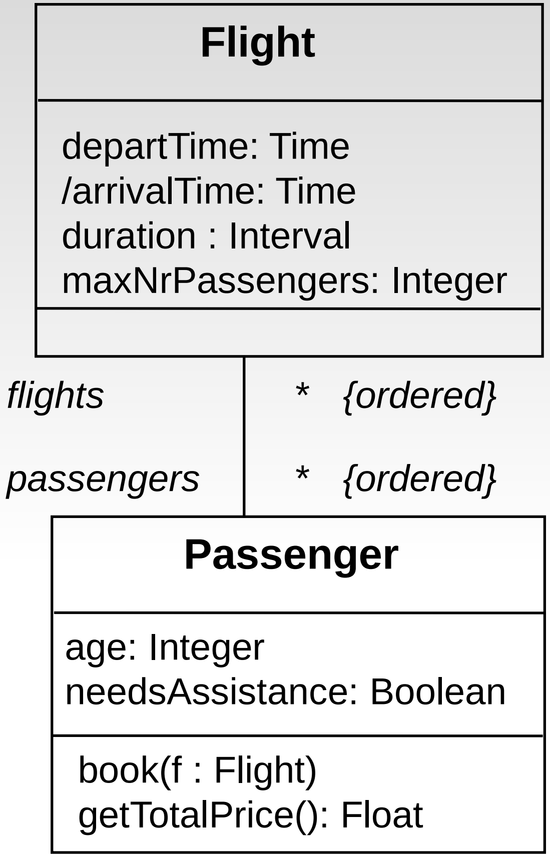


Context of a method

```
context Flight::servedAirports() : Set(Airport)
pre :  -- none
post: result = flights.destination->asSet
```

@pre in Post-conditions

- @pre refer to the previous value of some property



Flight

departTime: Time
/arrivalTime: Time
duration : Interval
maxNrPassengers: Integer

flights * *{ordered}*

passengers * *{ordered}*

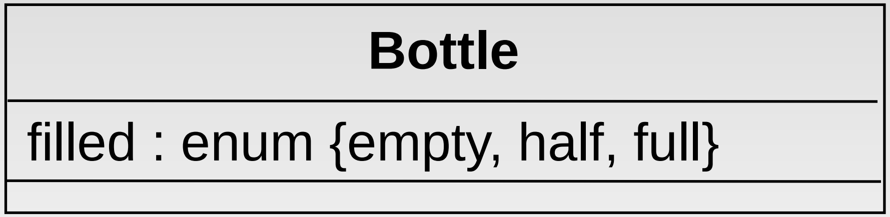
Passenger

age: Integer
needsAssistance: Boolean

book(f : Flight)
getTotalPrice(): Float

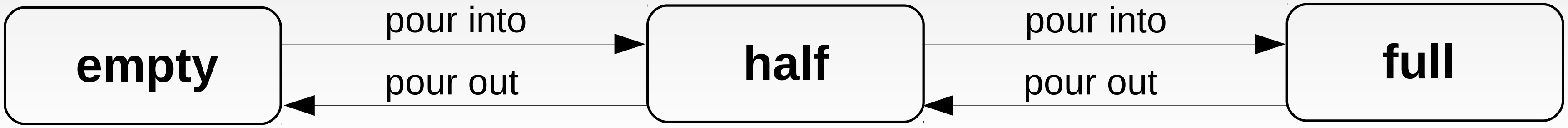
```
context Passenger::book(f : Flight)
pre : not flights->include(f)
post: flights->include(f) and
      getTotalPrice() = getTotalPrice@pre() + f.ticketPrice()

-- getTotalPrice() include the price of booked flight
```

Reference to State



Bottle

filled : enum {empty, half, full}



empty

pour into

pour out

half

pour into

pour out

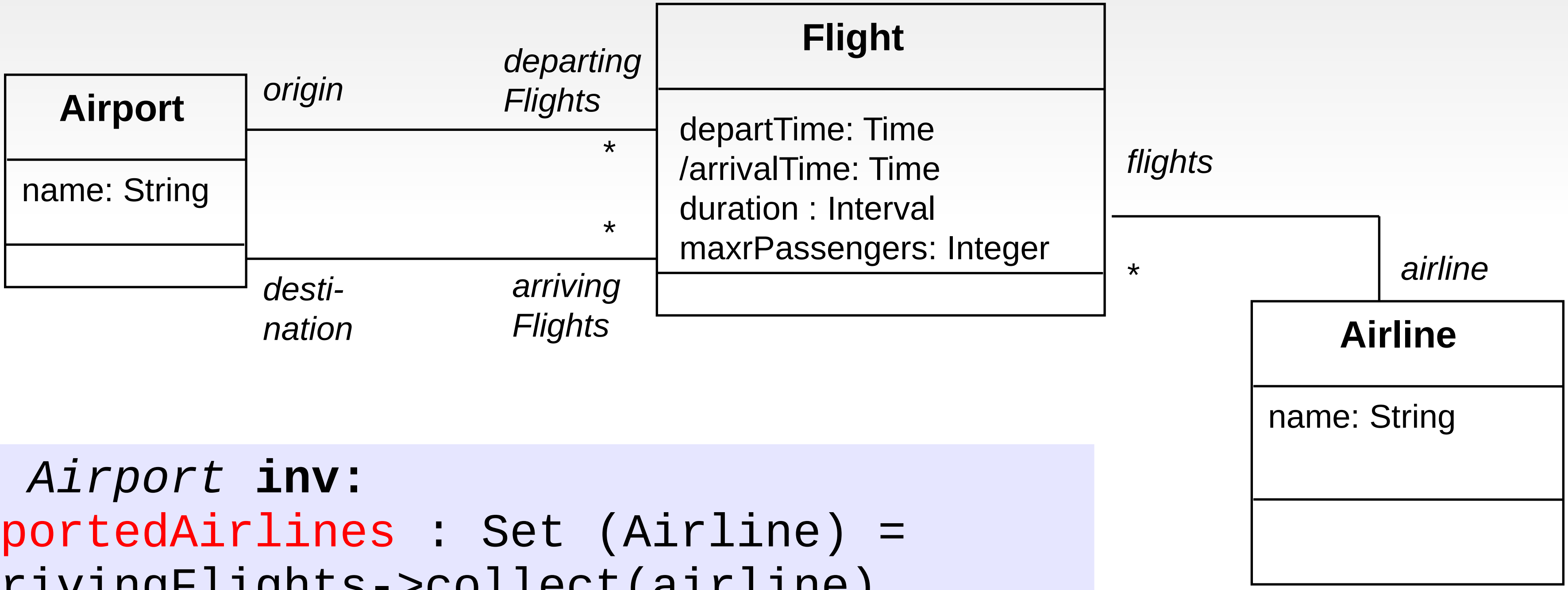
full

```
context Bottle inv:
self.oclInState(just_bought) implies filled = #full
```

- *oclInState* returns true if the object is in the specified state

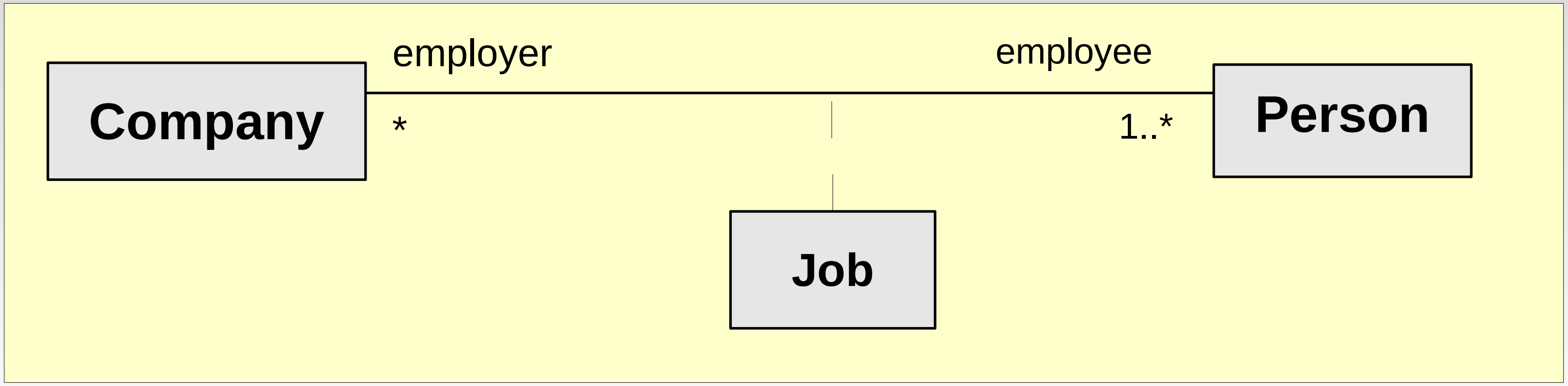
Local Variables

- *let* construct defines variables local to one constraint:
  - `Let var : Type = <expression1> in <expression2>`



```
context Airport inv:
let supportedAirlines : Set (Airline) =
self.arrivingFlights->collect(airline)
  in (supportedAirlines->notEmpty)
     and (supportedAirlines->size < 500)
```

Local Variables – Advanced Example



```
context Person inv:
let income : Integer = self.job.salary->sum
let hasTitle(t:String) : Boolean = self.job->exists(title=t) in
     if self.hasTitle(‘manager’) then
        self.income >= 1000
     else
        self.income >= 100
     endif
```

Inheritance of Constrains

Inheritance principle:

- Whenever an instance of a class is expected, one can always substitute an instance of any of its subclasses

Consequences for invariants:

- An invariant is always inherited by each subclass.
- Subclasses may strengthen the invariant.

Consequences for pre- and post-conditions:

- A precondition may be ***weakened*** (contravariance) in subclass
  - `context SuperClass::foo(i : Integer) pre: i > 1000`
  - `context SubClass::foo(i : Integer) pre: i > 0`
  - => ok, because the sub-class is able to process the same input values as its super-class.
- A postcondition may be ***strengthened*** (covariance) in subclass
  - `context SuperClass::foo() : Integer post: result > 0`
  - `context SubClass::foo() : Integer post: result > 1000`
  - => ok, because a caller gets always number > 0, even from the sub-class

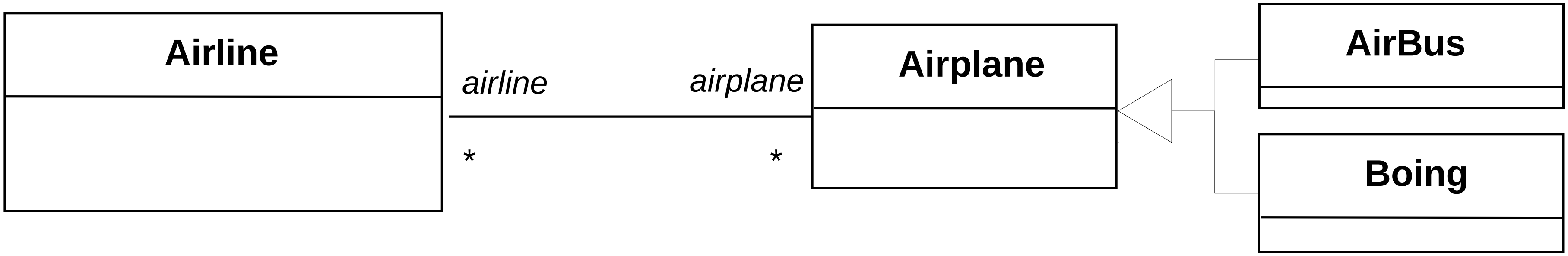
Type Checking and Casting Operations

- *oclType represents the type of „self“ object*
- *oclIsTypeOf(t : OCLType)* returns true if „self“ and „t“ are of the same type.

```
context AirBus inv: self.oclIsTypeOf(Airplane) -- is false
```

- *oclIsKindOf(t : OCLType)* returns true if „self“ and “t” are of the same type or if „t“ is super-type of „self“.

```
context AirBus inv: self.oclIsKindOf(Airplane) -- is true
```



```
context Airline inv:
self.airplane->select(oclType = Airbus)->notEmpty

-- Every airline has at least one Airbus in its airplane fleet.
-- Hide this constraint from EU representatives as they could
   adopt this idea ;)
```

Questions?