Module 6

OCL

Vasco Amaral
vma@fct.unl.pt

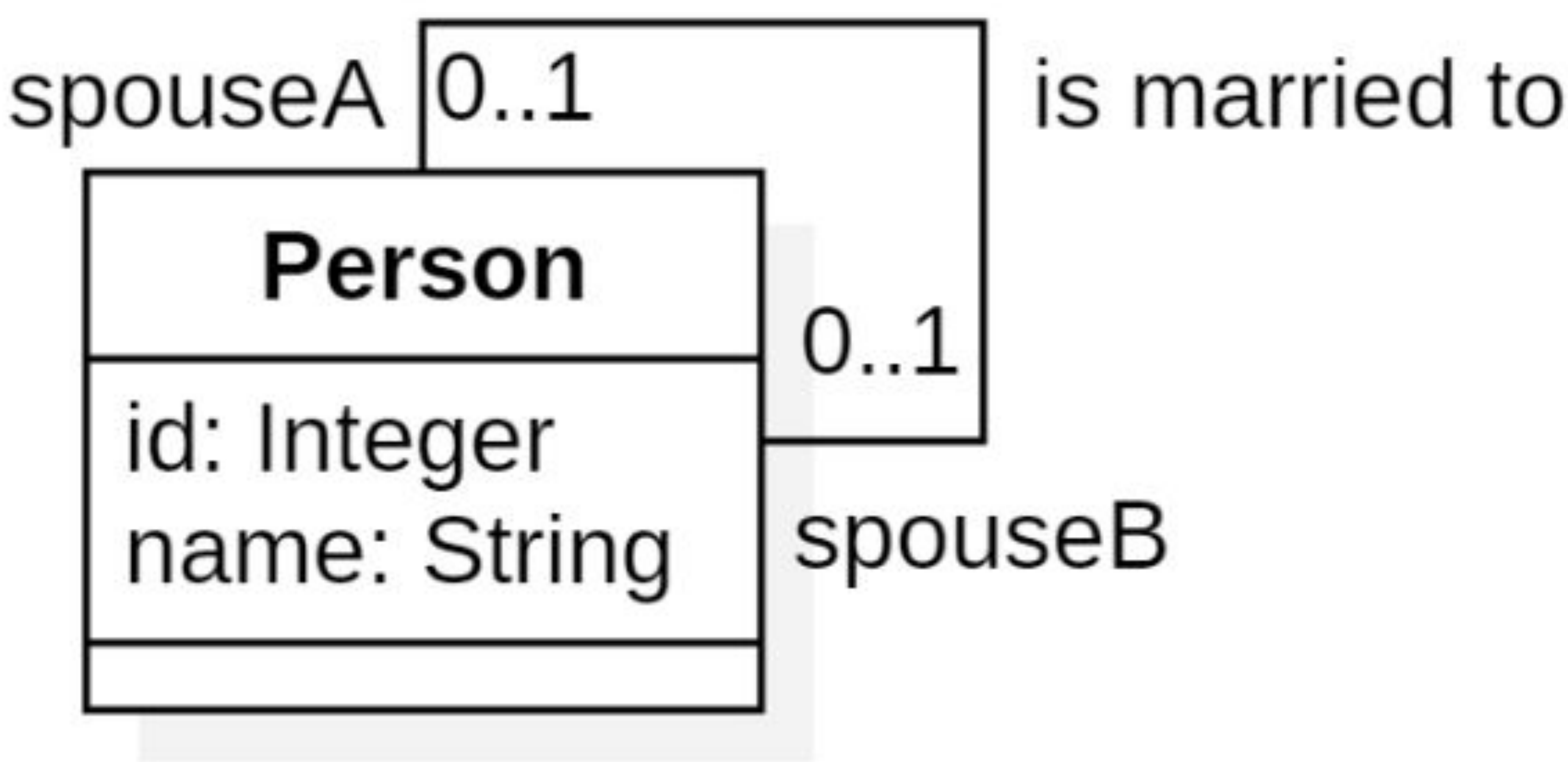
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
Departamento de Informática

Let us look at the trailer

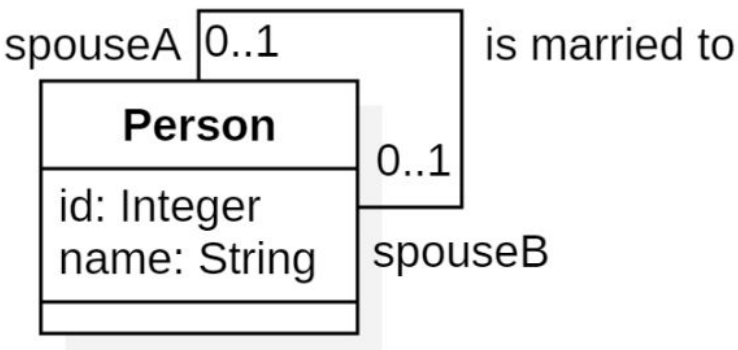
uhuuu! Carrie is getting married!

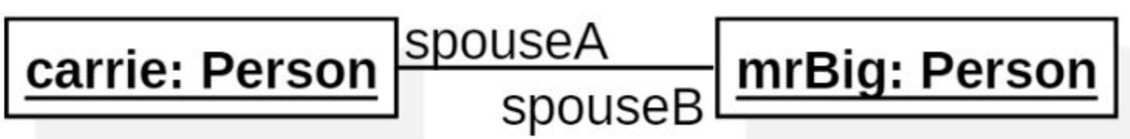
Let us model her marriage with a class diagram

Super easy, barely an inconvenience

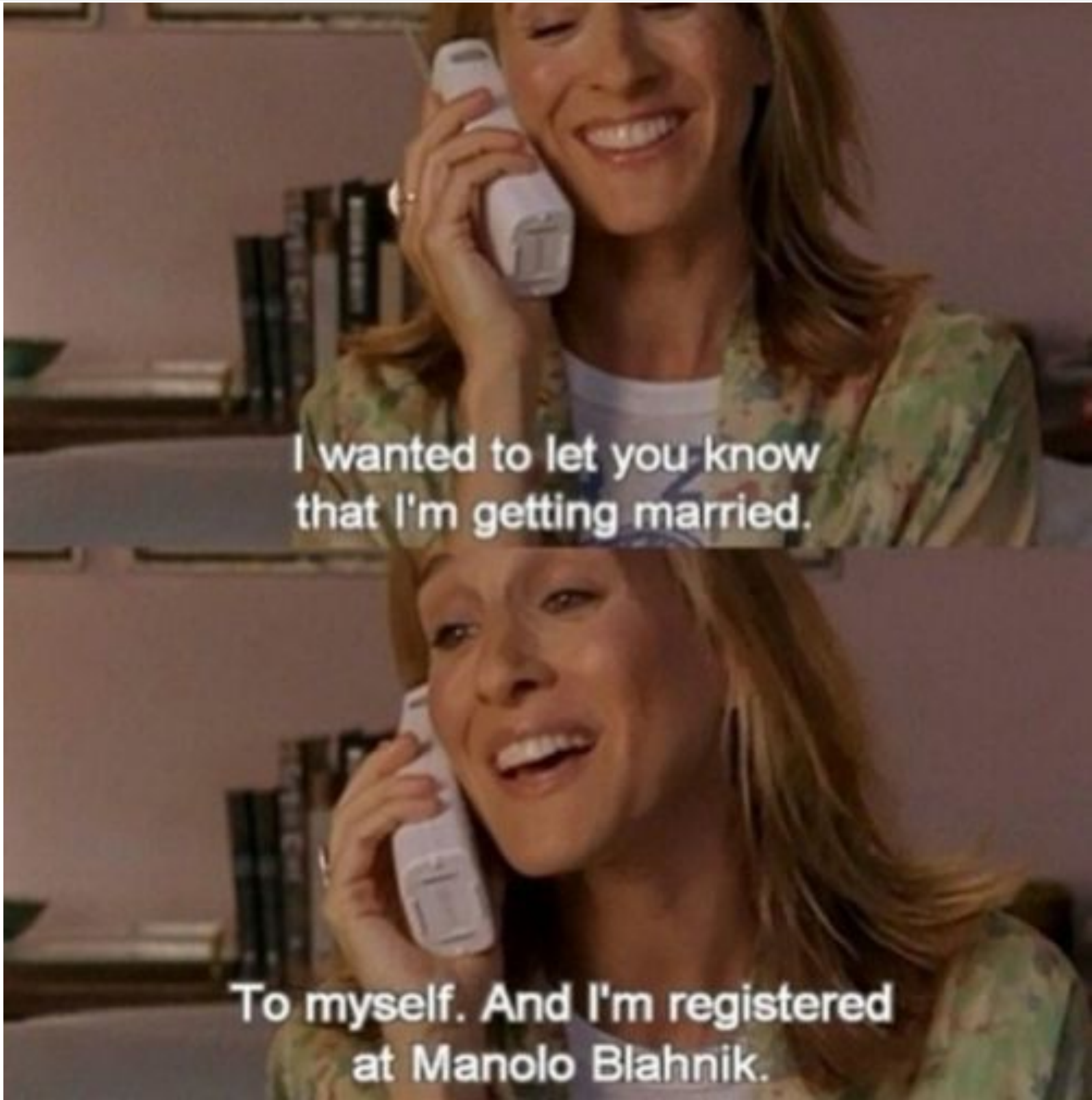


Carrie can marry Mr. Big!

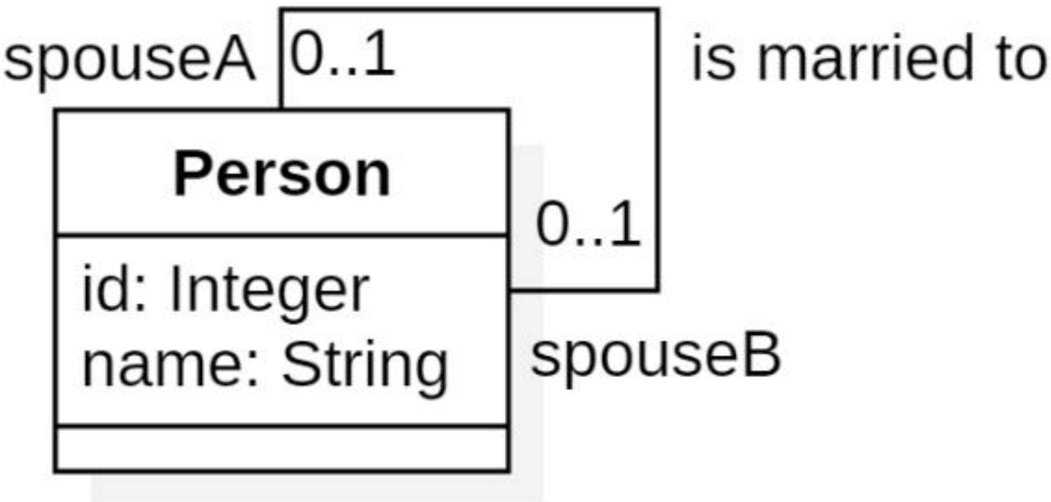


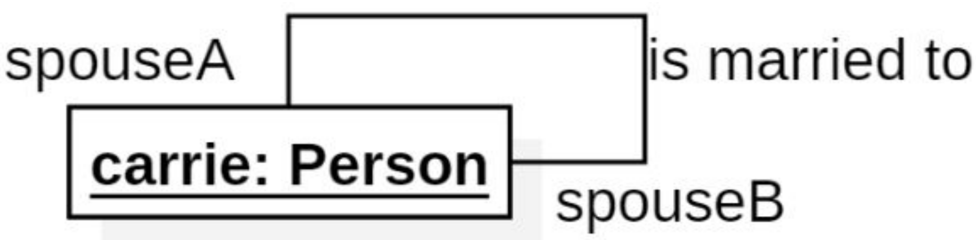


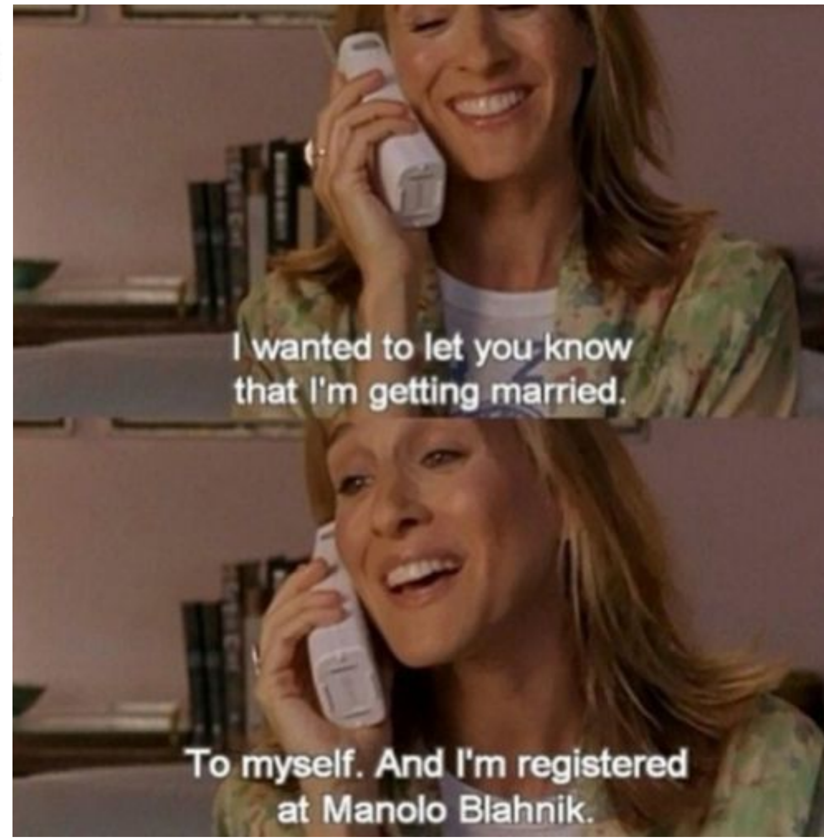
Ooopsy…


I wanted to let you know
that I'm getting married.
To myself. And I'm registered
at Manolo Blahnik.

Wrong episode...



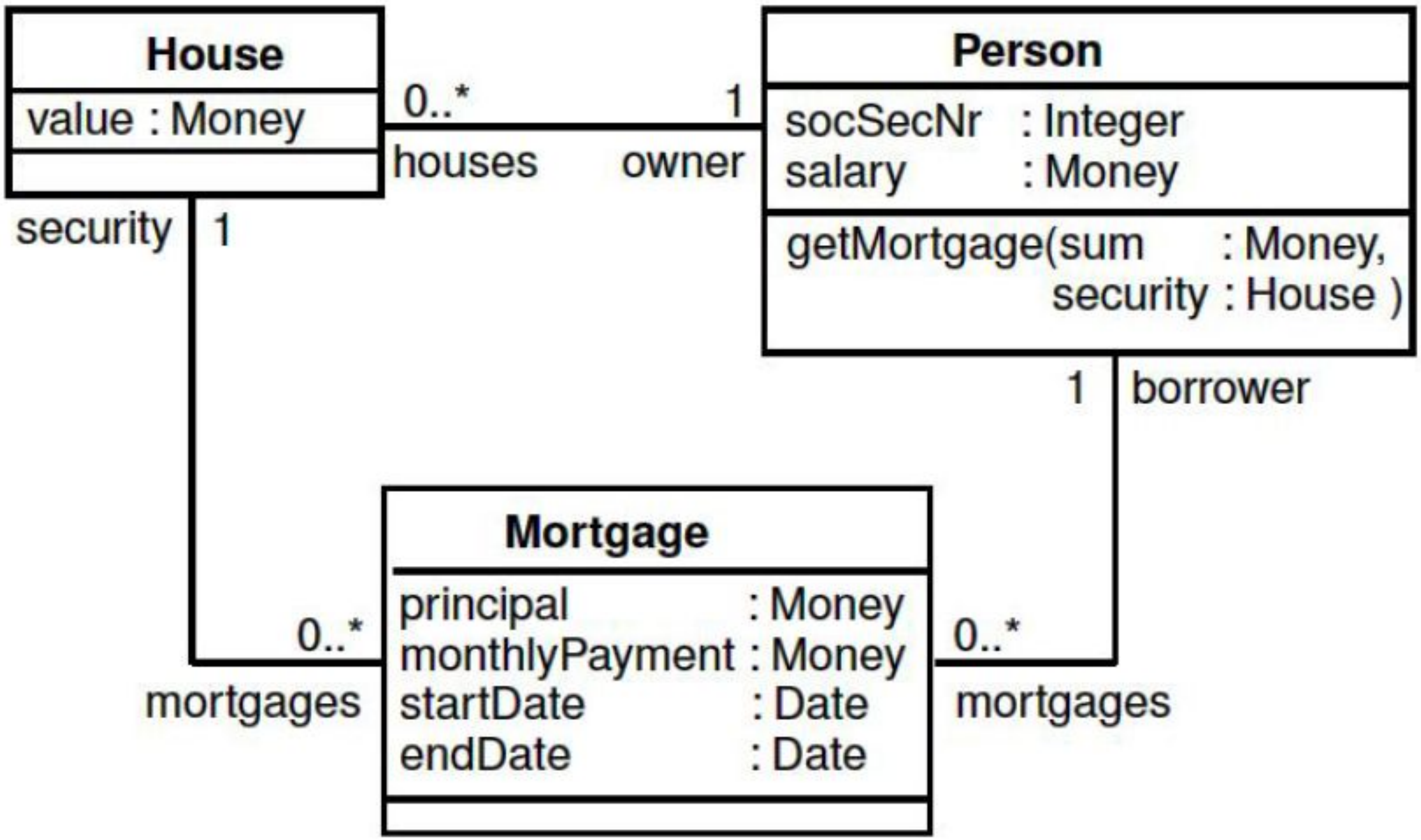




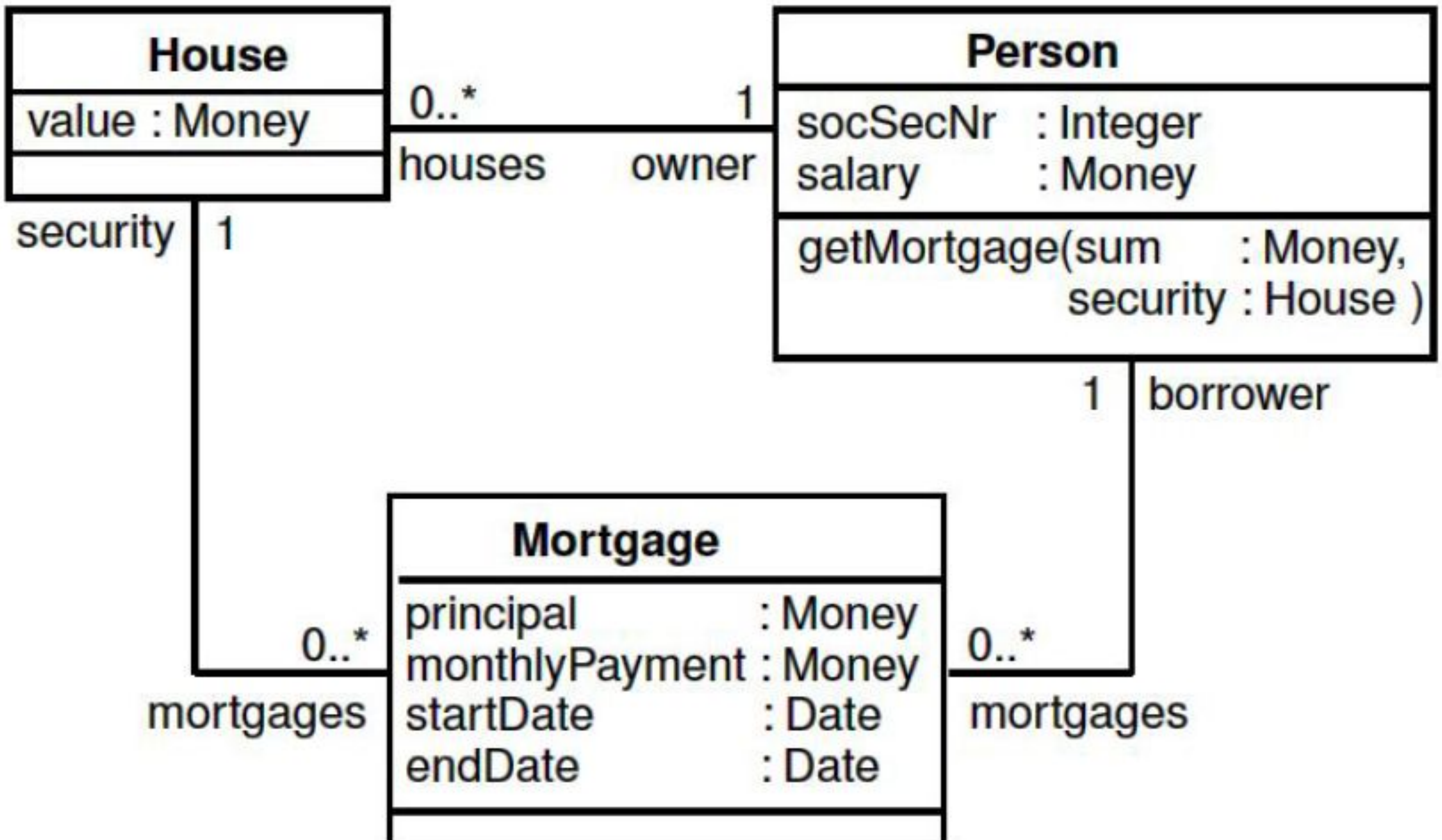
What if Carrie was in a country where *sologamy* (self-marriage) is not allowed?

How would you model such a restriction in a class diagram, if you had to?

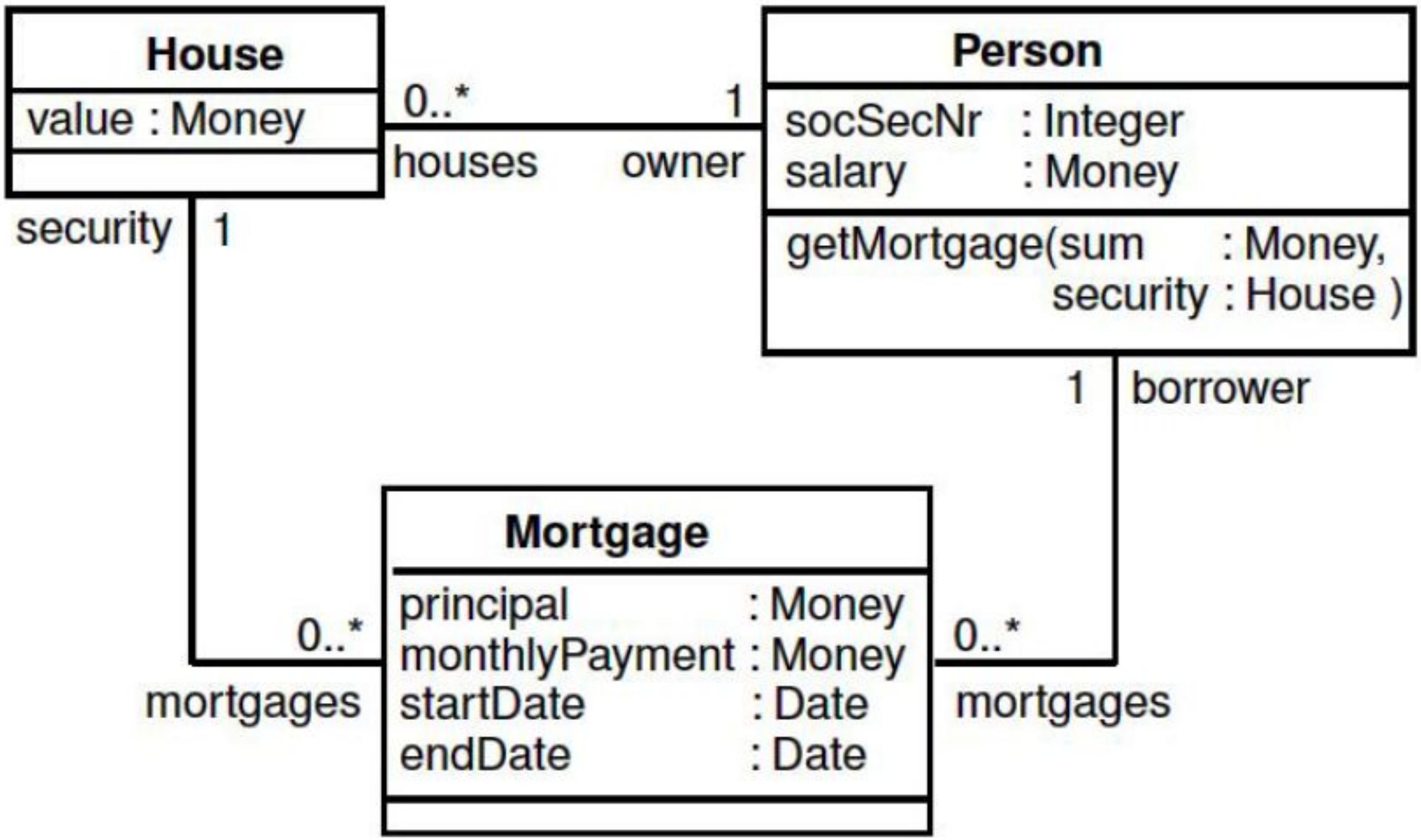
Yet another example

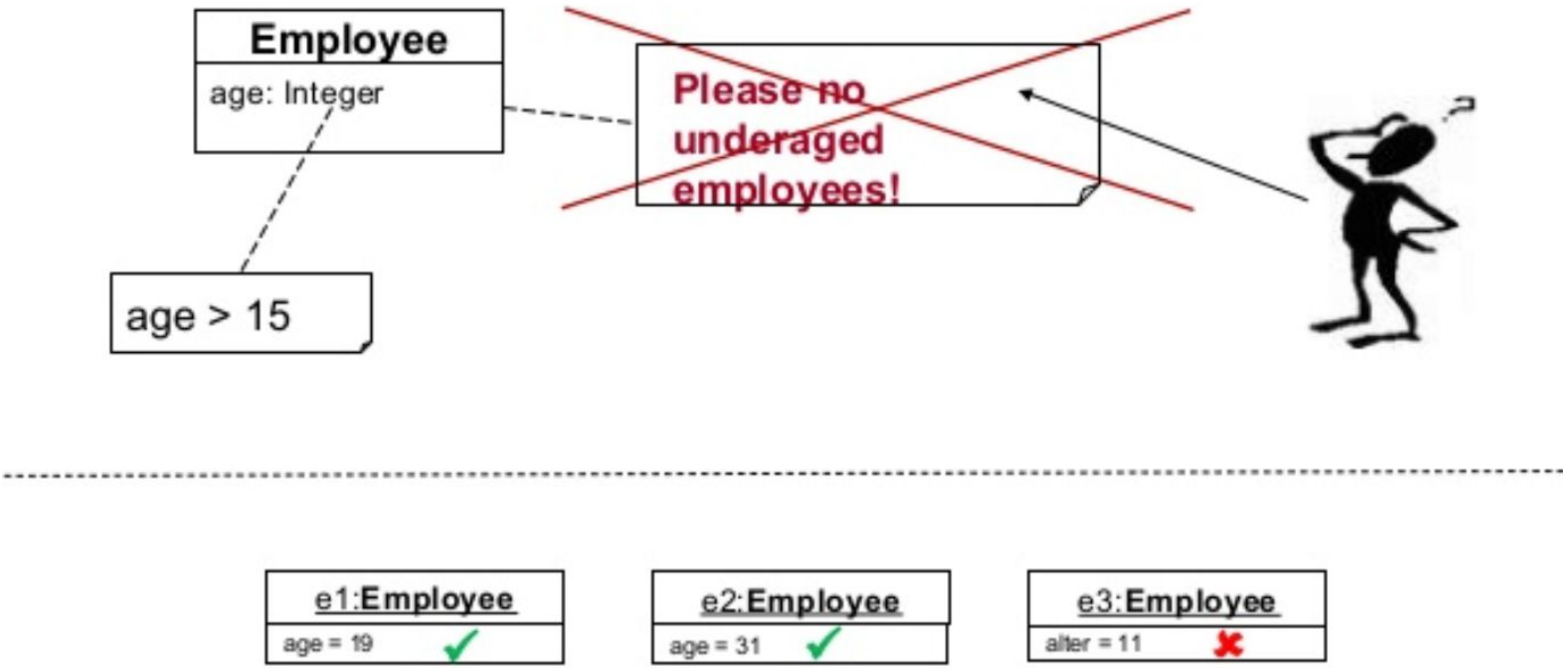


Can a person have a mortgage on a house she does not own?



Can the start of the mortgage be <u>after</u> its end?





Additional question: How do I get all Employees younger than 30 years old?

We need another language to enforce this kind of business rules

Class diagrams alone are not enough

Object Constraint Language
OCL can specify queries, constraints and query operations

About OCL

- First developed in 1995 as IBEL by IBM's Insurance division for business modelling
- IBM proposed it to the 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.
- Currently a standard extension of UML
  https://www.omg.org/spec/OCL/2.4/

UML Diagrams are <u>not</u> enough!

- We need a language to help specifying additional information in UML model
  - We look for some "add-on", not a new language with full specification capability
    - We need it to integrate smoothly with UML
  - Why not first order logic? – Not OO
- OCL is used to specify constraints on OO systems
  - OCL is not the only alternative...
  - ... but OCL is the only one that is standardized as a UML extension

OCL is not a programming language

- OCL expressions have no side effects
  - OCL can't change the value of a variable
    - You can only query values and state conditions on values!
  - OCL can only define query operations
  - OCL can only execute query operations with no side effects
  - OCL can't be used to specify business rules dynamically at runtime
    - OCL can specify business rules at modelling time

Why use OCL?

- OCL constraints add information about the model elements and their relationships to the UML models
- OCL offers a reasoning mechanism to UML models
  - Can be used for model consistency checking
- OCL expressions can be used in code generation
  - Enforcing constraints, pre- and postconditions
- OCL adds precision to the models
  - Reduces ambiguity in models - improves communication among stakeholders

Challenges in OCL adoption

- Tool support is not as good as for other parts of UML, in general
- OCL syntax is not too user friendly
- Many modellers are not familiar with it
    - Fewer programmers are familiar with it
- Depending on what you want the models for, it may be an overkill to use OCL

and yet...

- It is part of UML professional certification syllabus!
- It is an essential tool for automatic quality support for models
- It is an essential tool for automatic code generation
  - This is where the real money is in modelling
- It is an essential tool for software languages engineering
  - There is also a business opportunity there, with the increasing popularity of Domain-Specific Languages

So, where do we use OCL?

- as a query language
- to specify invariants on classes and types in the class model
- to specify type invariant for Stereotypes
- to describe pre- and post conditions on Operations and Methods
- to describe Guards
- to specify target (sets) for messages and actions
- to specify constraints on operations
- to specify derivation rules for attributes for any expression over a UML model

Combining UML and OCL

- Without OCL expressions, many models would be severely underspecified
- Without the UML diagrams, the OCL expressions would refer to non-existing model elements
  - there is no way in OCL to specify classes and associations
- Only when we combine the diagrams and the constraints can we completely specify the model

Introduction to the Object Constraint Language

OCL is a declarative language

- Most languages you are familiar with are procedural
  - C, C++, C#, Java, …
    - You describe, step by step, how the result you want may be achieved
- Because OCL is declarative…
  - You describe the result you want, rather than how to achieve that result

To be clear...

OCL is **not** a programming language:
You do not get to specify behaviors

OCL is a constraint language:
You specify queries and conditions, not behaviors

OCL expressions syntax

```
package <packagePath>
    context <contexualInstanceName>: <modelElement>
        <expressionType> <expressionName>:
            <expressionBody>
        <expressionType> <expressionName>:
            <expressionBody>
        ...
endpackage
```

Key:

- **boldface - OCL keyword**
- dark grey - mandatory
- blue - optional

An OCL expression are attached to UML model elements

```
package <packagePath>
   context <contexualInstanceName>: <modelElement>
      <expressionType> <expressionName>:
         <expressionBody>
      <expressionType> <expressionName>:
         <expressionBody>
      ...
endpackage
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
Departamento de Informática

OCL expressions may have a package context

```
package <packagePath>
    context <contexualInstanceName>: <modelElement>
        <expressionType> <expressionName>:
            <expressionBody>
        <expressionType> <expressionName>:
            <expressionBody>
        ...
endpackage
```

OCL expressions always have an expression context

```
package <packagePath>
    context <contexualInstanceName>: <modelElement>
        <expressionType> <expressionName>:
            <expressionBody>
        <expressionType> <expressionName>:
            <expressionBody>
        ...
endpackage
```

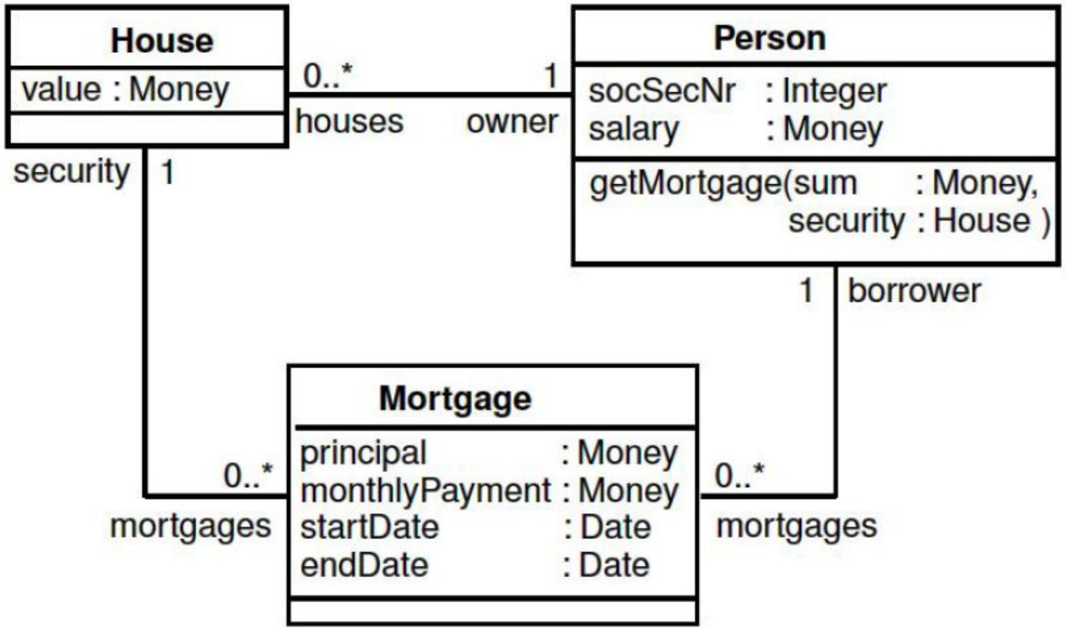
We can define one or more OCL expressions within a given context

```
package <packagePath>
    context <contexualInstanceName>: <modelElement>
        <expressionType> <expressionName>:
            <expressionBody>
        <expressionType> <expressionName>:
            <expressionBody>
        ...
endpackage
```

Back to our mortgage example

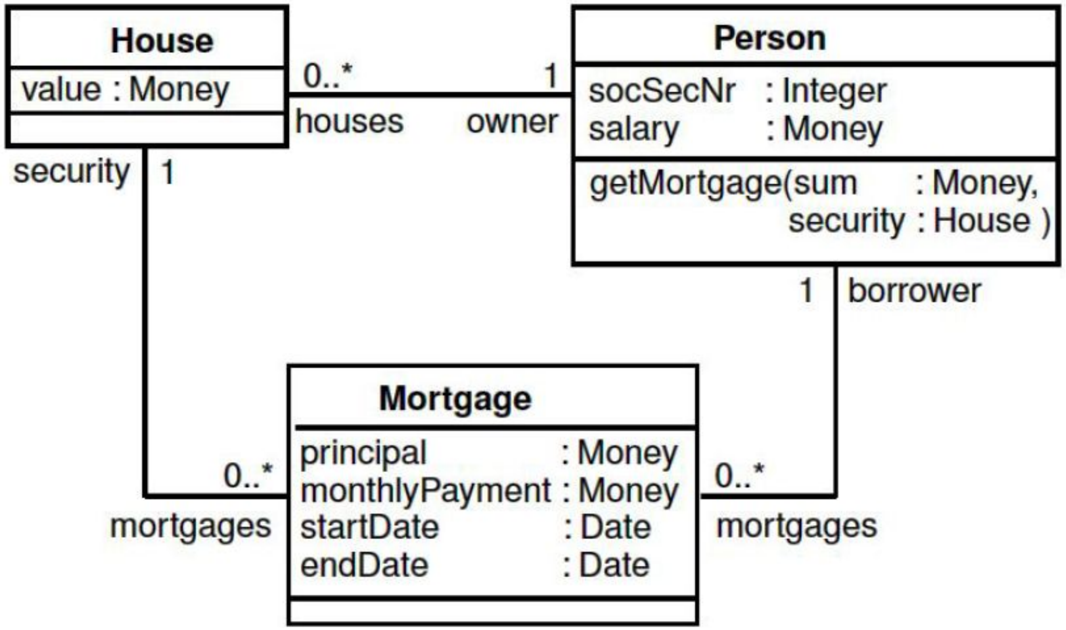
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
Departamento de Informática

Can a person have a mortgage on a house she does not own?

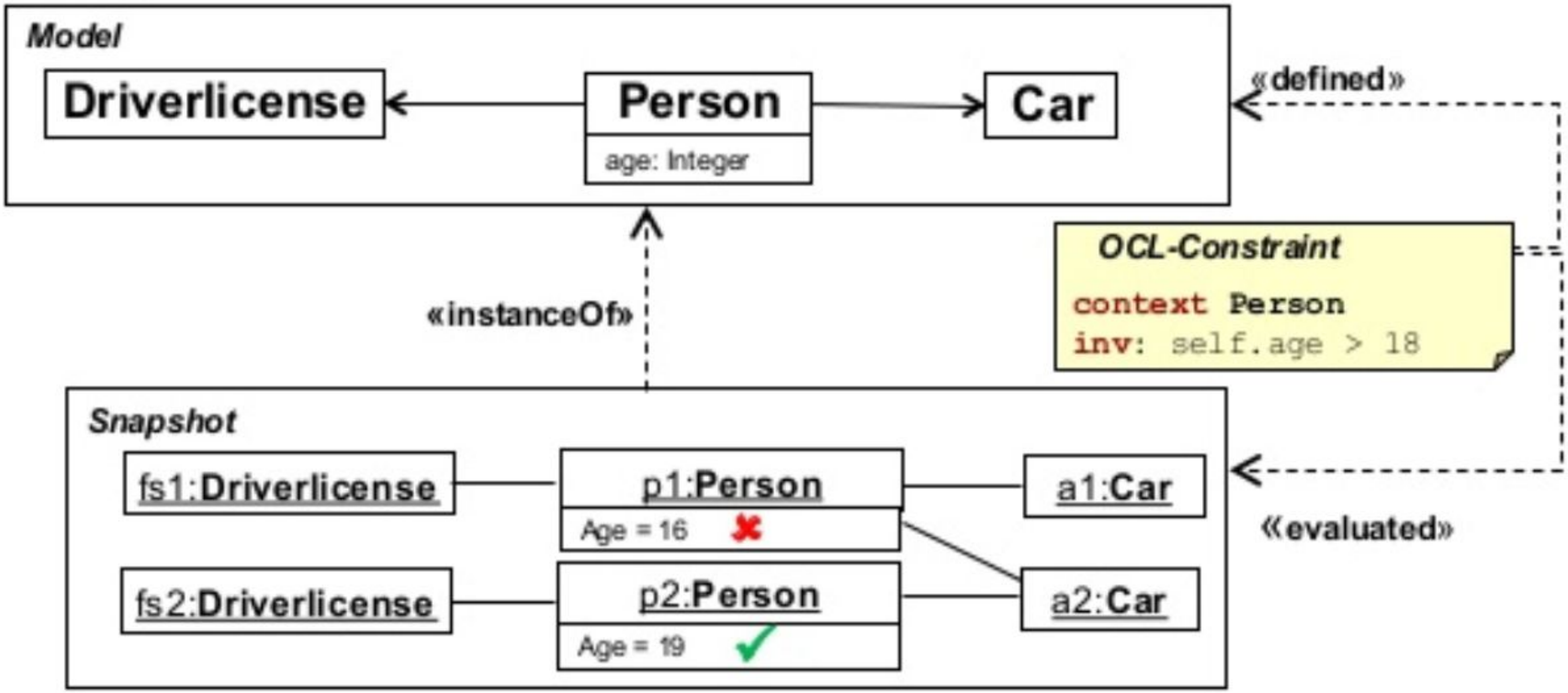


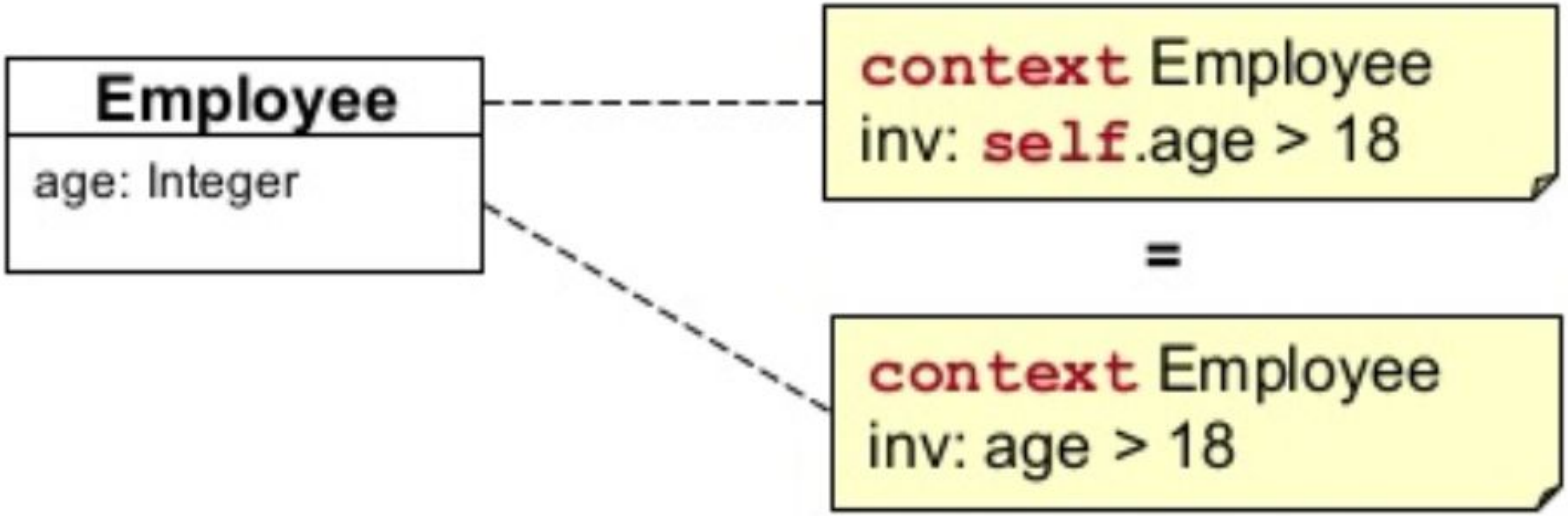

```
context Mortgage inv: self.security.owner = self.borrower
```

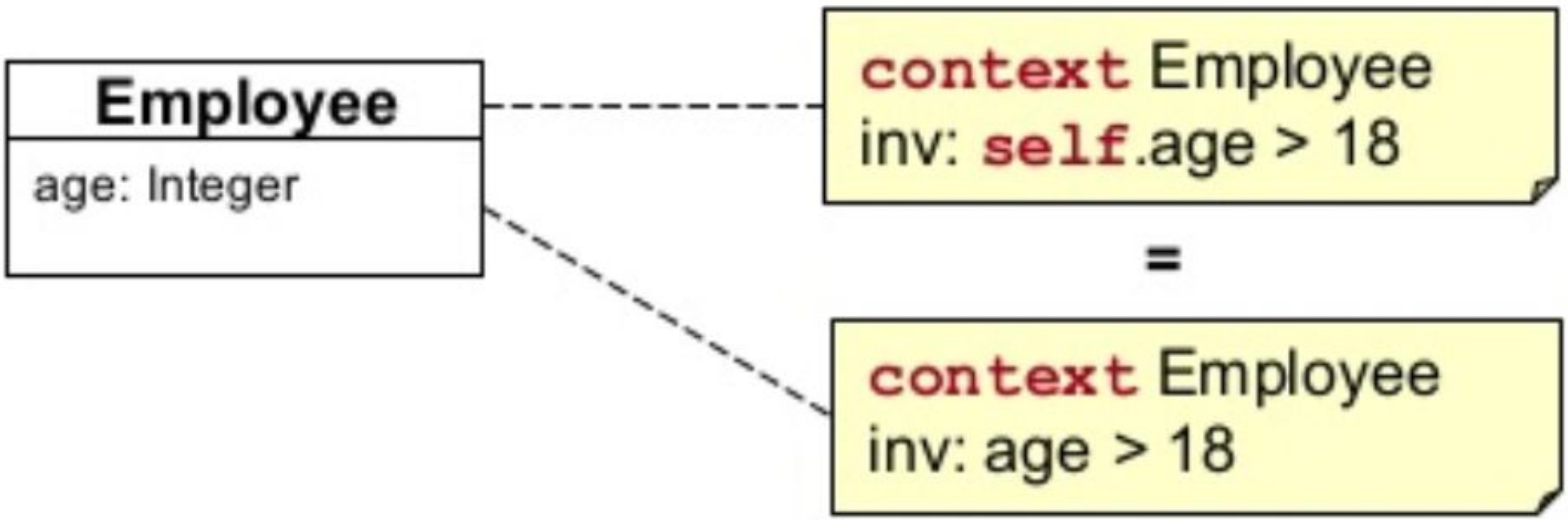
Can the start of the mortgage be after its end?




```
context Mortgage inv: self.startDate < self.endDate
```

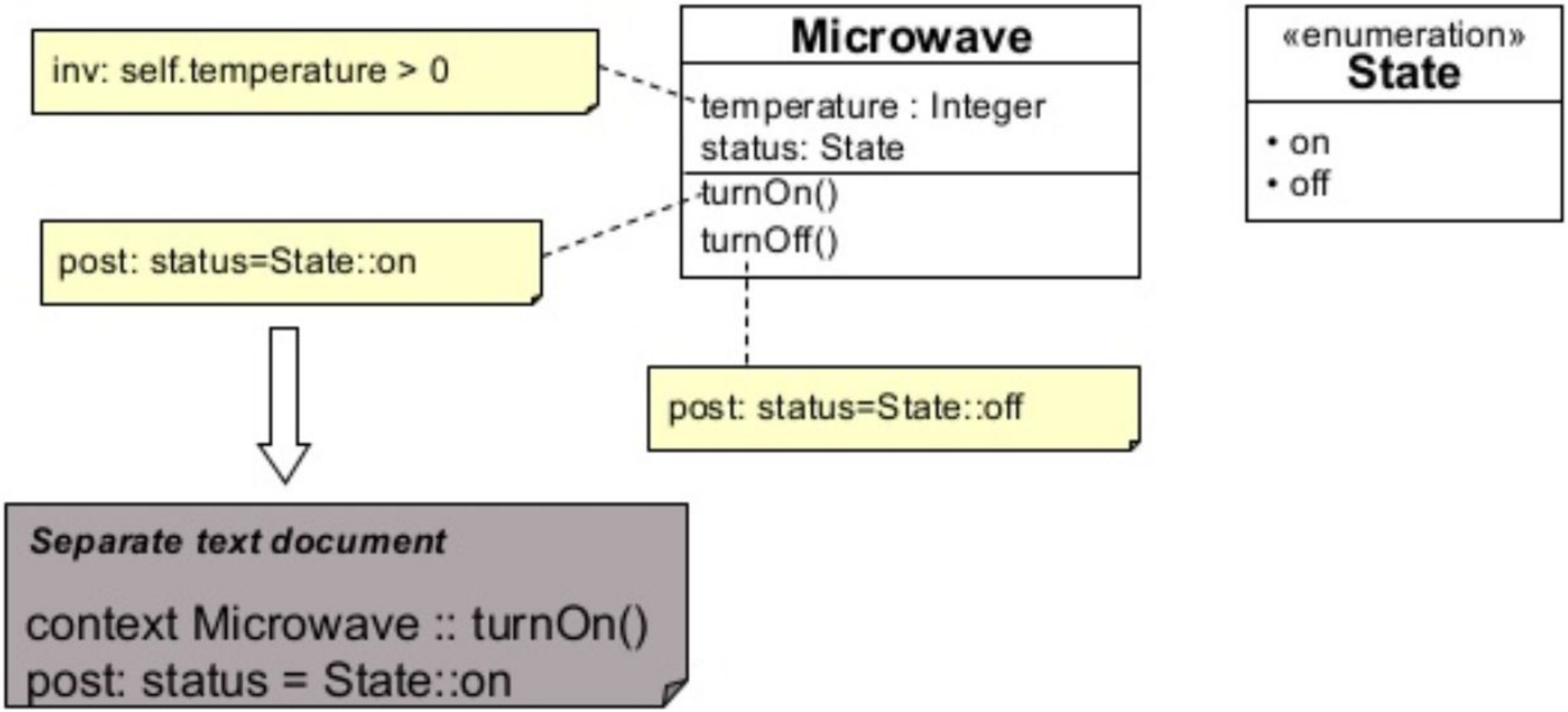

Model
Driverlicense
Person
age: Integer
Car
«defined»
«instanceOf»
OCL-Constraint
context Person
inv: self.age > 18
Snapshot
fs1:Driverlicense
p1:Person
Age = 16
a1:Car
fs2:Driverlicense
p2:Person
Age = 19
a2:Car
«evaluated»


Employee
age: Integer
context Employee
inv: self.age > 18
=
context Employee
inv: age > 18


Employee
age: Integer
context Employee
inv: self.age > 18
=
context Employee
inv: age > 18

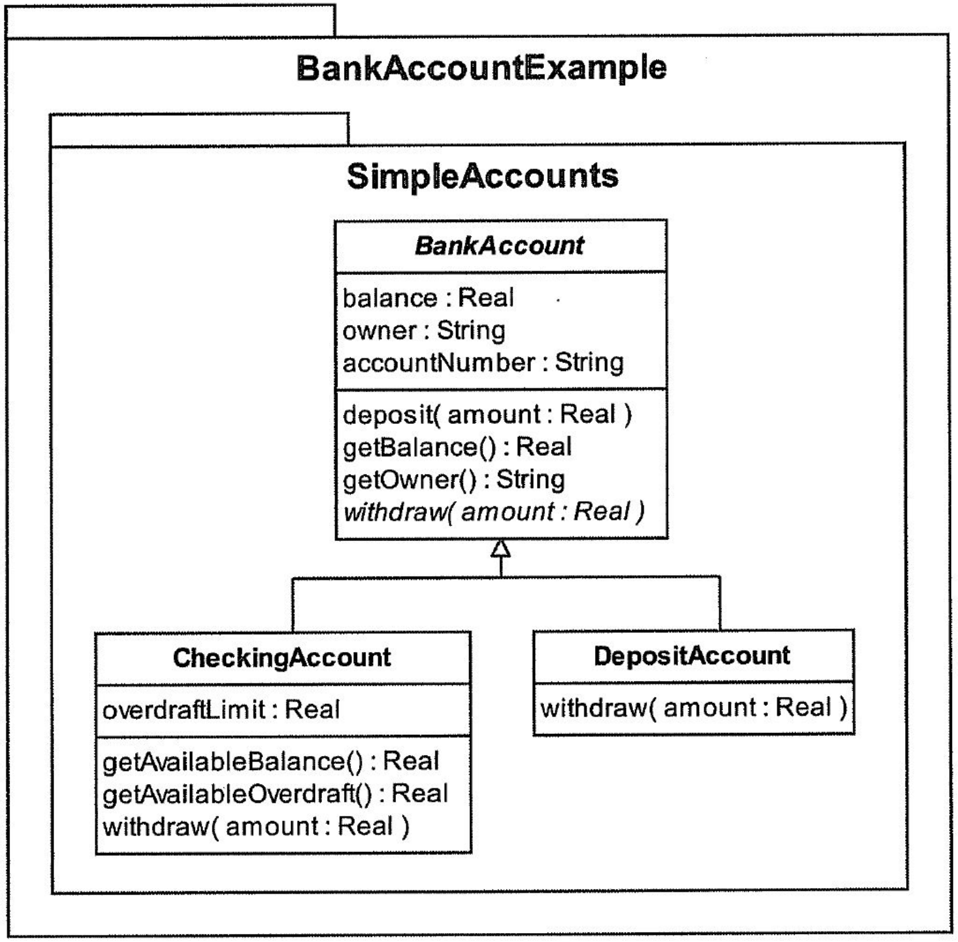
OCL can be specified in **two** different ways

- As a comment **directly** in the class diagram (context described by connection)
- Separate document file



Now that we have an intuition for OCL, let us study it in detail...

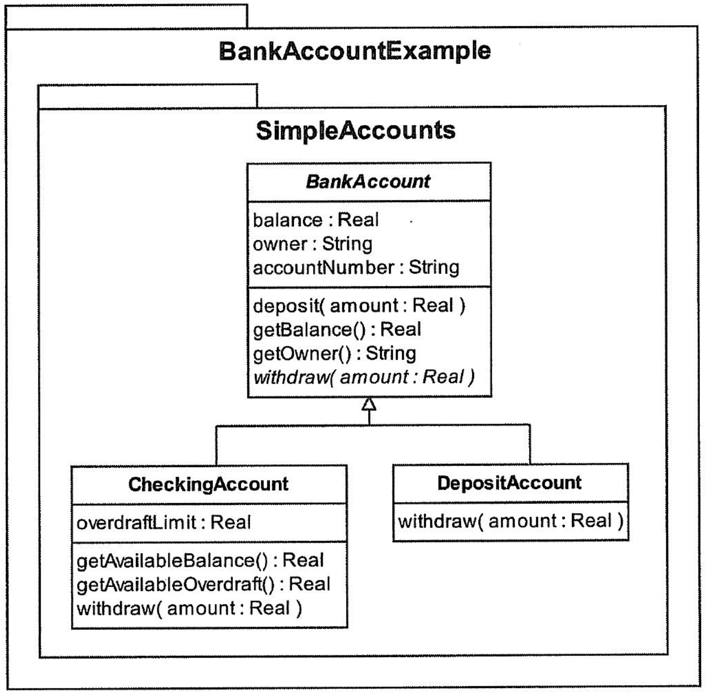
Running example



The package context defines the namespace for an OCL expression

```
package BankAccountExample::SimpleBankAccounts
    ...
endpackage
```

- If you don't specify a package's context, the namespace for the expression is the whole model
- If you attach an OCL expression directly to a model element, the namespace for the expression defaults to the package containing the element



How should we reference elements, then?

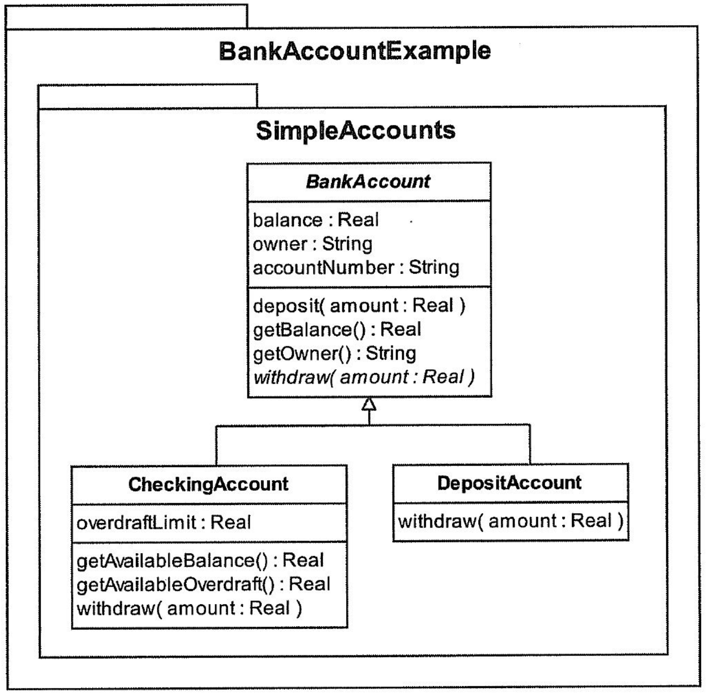
In general:

`Package1::Package2::...::PackageN:ElementName`

Example:

`BankAccountExample::SimpleBankAccounts::DepositAccount`

- If the element is unique, you can refer to it without the qualified name
- If elements in different packages have the same name:
  - Define a package context for each OCL expression referring to it, or
  - Refer to the elements using their full path name



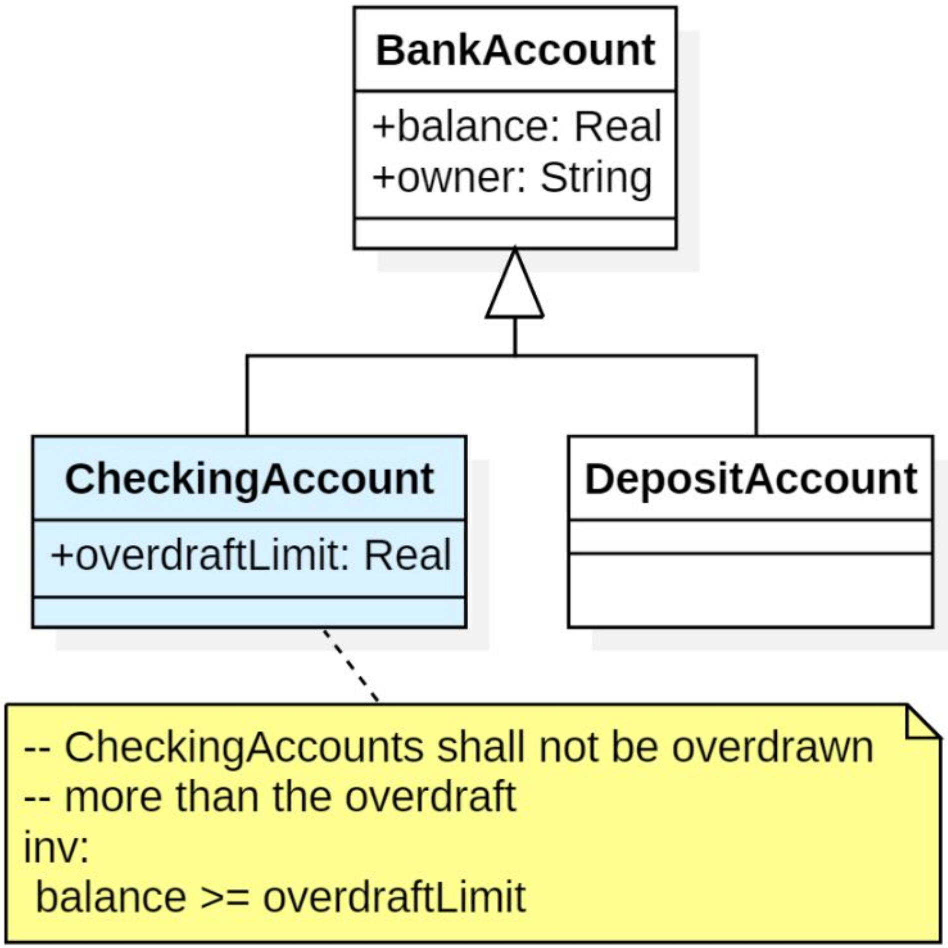
The expression context indicates the UML model element to which the OCL expression is attached

```
package BankAccountExample::SimpleBankAccounts
    context account:CheckingAccount
    ...
endpackage
```

- If the expression context is a classifier, the contextual instance is always an instance of that classifier
- If the expression context is an operation, or an attribute, the contextual instance is generally an instance of the classifier that owns the operation, or attribute

You can attach an OCL expression to a model element as a note - the context is the annotated element

In this particular case, the expression specifies an invariant, stating that it is not possible to overdraw from a CheckingAccount.



Types of OCL expressions

- Two categories of OCL expressions
  - OCL expressions specifying constraints
    - inv:
    - pre:
    - post:
  - OCL expressions specifying attributes, operation bodies, and local variables
    - init:
    - body:
    - def:
    - let:
    - derive:
- You can't give an expression name to the operations that define

Operations that constrain

Expression type	Syntax	Applies to	Contextual instance	Semantics
invariant	inv:	Classifier	An instance of the classifier	The invariant must be true for all instances of the classifier
precondition	pre:	Operation Behavioral feature	An instance of the classifier that owns the operation	The precondition must be true before the operation executes
postcondition	post:	Operation Behavioral feature	An instance of the classifier that owns the operation	The postcondition must be true after the operation executes The keyword result refers to the result of the operation

Operations that define

Expression type	Syntax	Applies to	Contextual instance	Semantics
query operation body	body:	Query operation	An instance of the classifier that owns the operation	Defines the body of a query operation
initial value	init:	Attribute Association end	The attribute The association end	Defines the initial value of the attribute or the association end
define	def:	Classifier	An instance of the classifier that owns the operation	Adds variables or helper operations to a context classifier These are used in OCL expressions on the context classifier
let	let	OCL expression	The contextual instance of the OCL expression	Adds local variables to OCL expressions
derived value	derive:	Attribute Association end	The attribute The association end	Defines the derivation rule for the derived attribute or association end

OCL precedence rules

Precedence decreasing

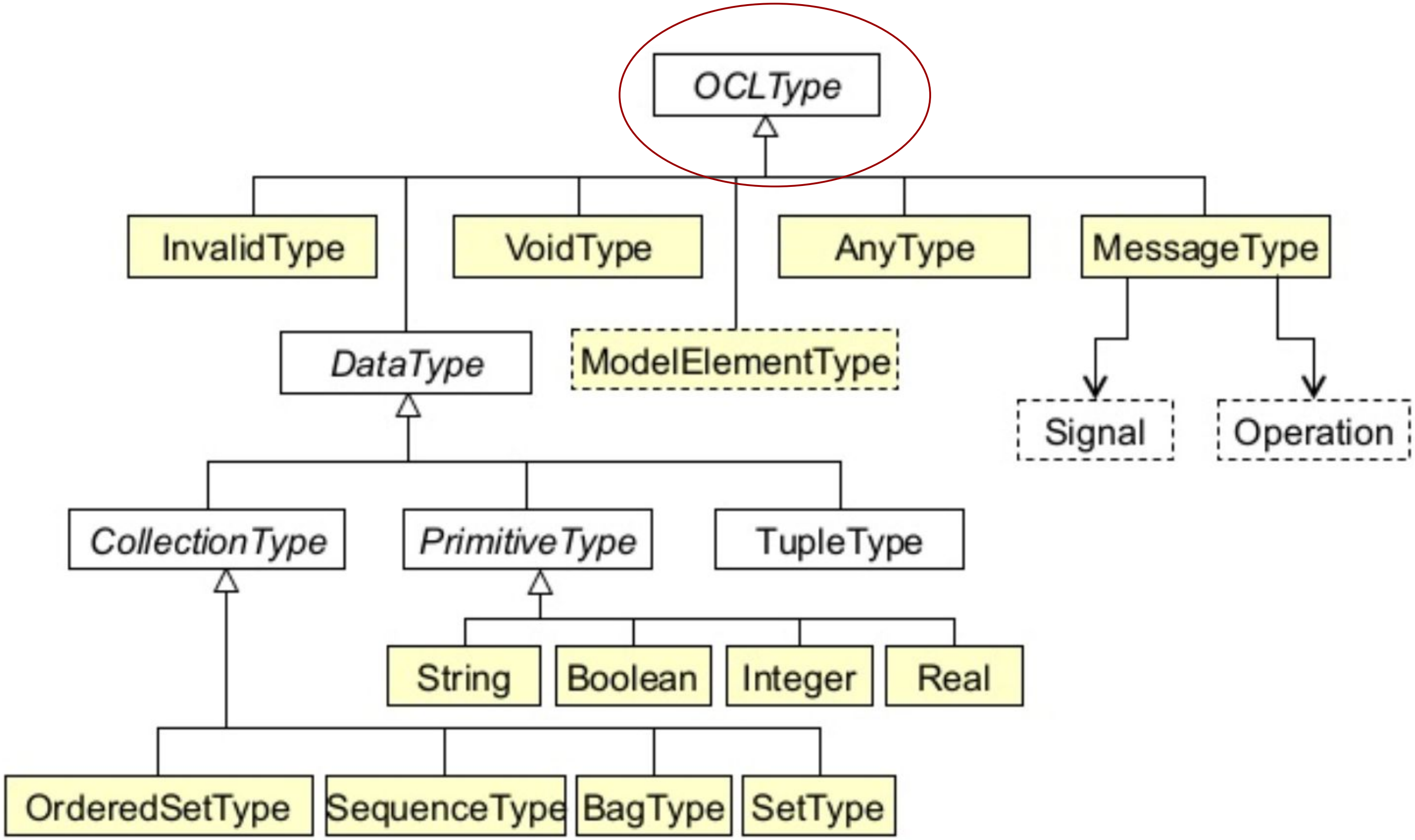
```
::
@pre
. ->
not - ^ ^^
* /
+ -
if … then … else … endif
> < <= >=
= <>
and xor or
implies
```

```
1 + 2 * 3
7 (Integer)
(1 + 2) * 3
9 (Integer)
```

The OCL type system

- OCL is a strongly typed language
- OCL includes the following primitive types
  - **`Boolean`**, **`Integer`**, **`Real`** and **`String`**
- OCL also includes a set of built-in types
  - **`OclAny`**, **`OclType`**, **`OclState`**, **`OclVoid`**, **`OclMessage`**
- All the classifiers in the associated UML model become types in OCL
- OCL can directly refer to all those classifiers
  - This enables OCL as a constraint language

The OCL type system



OclAny is the supertype of all types in OCL and the associated UML model

- So, every type is a subtype of OclAny, including the classifiers imported from the UML model
- OCL needs this so that there is a common object protocol that it can use to manipulate any types in the UML model
- **OclAny supports** three kinds of operations;
  - **Comparison operations**
  - **Query operations**
  - **Conversion operations**

OclAny comparison operations

Comparison operations

a = b	Returns true if a is the same object as b, otherwise returns false
a <> b	Returns true if a is *not* the same object as b, otherwise returns false
a.oclIsTypeOf(b : OclType) : Boolean	Returns true if a is the same type as b, otherwise returns false
a.oclIsKindOf(b : OclType) : Boolean	Returns true if a is the same type as b, or a subtype of b
a.oclInState(b : OclState) : Boolean	Returns true if a is in the state b, otherwise returns false
a.oclIsUndefined() : Boolean	Returns true if a = OclUndefined

OclAny query operations

Query operations

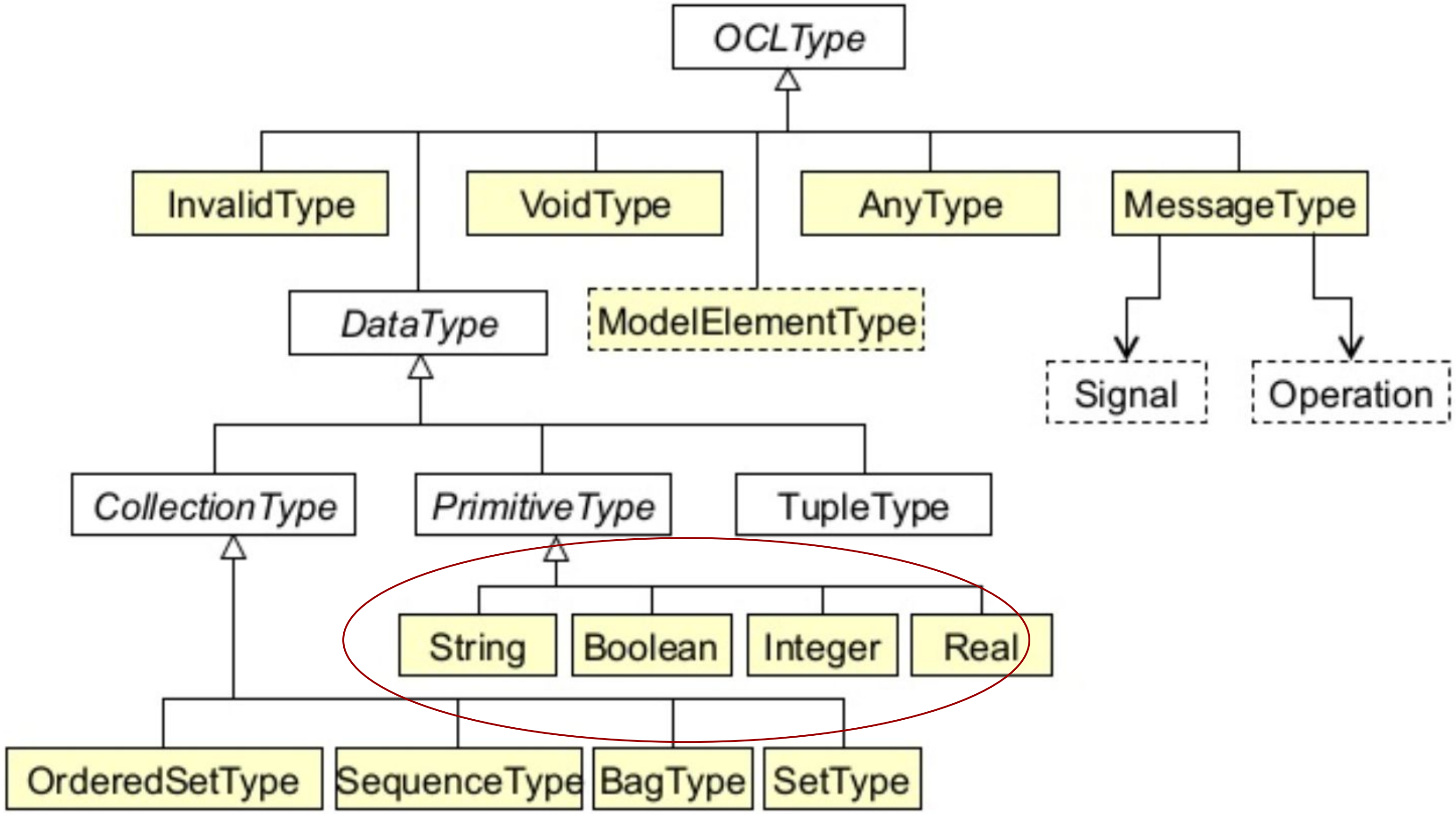
A::allInstances() : Set(A)	This is a class scope operation that returns a Set of all instances of type A
a.oclIsNew() : Boolean	Returns true if a was created by the execution of the operation Can only be used in operation postconditions

OclAny conversion operations

Conversion operations

a.oclAsType(SubType) : SubType	Evaluates to a retyped to SubType This is a casting operation, and a may only be cast to one of its subtypes or supertypes Casting to a supertype allows access to overridden supertype features

The OCL type system



OCL primitive types

OCL basic type	Semantics
Boolean	Can take the value true or false
Integer	A whole number
Real	A floating point number
String	A sequence of characters String literals are single quoted, e.g., 'Jim'

Boolean and its predefined operations

a	b	a = b	a<>b	a.and(b)	a.or(b)	a.xor(b)	a.implies(b)
true	true	true	false	true	true	false	true
true	false	false	true	false	true	true	false
false	true	false	true	false	true	true	true
false	false	true	false	false	false	false	true

a	not a
true	false
false	true

Boolean expressions can be used within other expressions (e.g. if… then… else)

```
if <booleanExpression> then
   <oclExpression1>
else
   <oclExpression2>
endif
```

Integer

- Represent whole numbers
- No lower and upper limits
- Supports the following set of infix operators
    - =, <>, <, >, <=, >=, +, -, *, /

Real

- Represent floating point numbers
- No lower and upper limits
- No precision limits
- Supports the following set of infix operators
  - =, <>, <, >, <=, >=, +, -, *, /

Operations for Integer and Real

Syntax	Semantics	Applies to
a.mod(b)	Returns the remainder after a is divided by b e.g., a = 3, b = 2, a.mod(b) returns 1	Integer
a.div(b)	The number of times that b fits completely within a e.g., a = 8, b = 3, a.div(b) returns 2	Integer
a.abs()	Returns positive a e.g., a = (–3), a.abs() returns 3	Integer and Real
a.max(b)	Returns the larger of a and b e.g., a = 2, b = 3, a.max(b) returns b	Integer and Real
a.min(b)	Returns the smaller of a and b e.g., a = 2, b = 3, a.min(b) returns a	Integer and Real

Operations for Integer and Real

Syntax	Semantics	Applies to
a.round()	Returns the Integer closest to a If there are two integers equally close, it returns the largest e.g., a = 2.5, a.round() returns 3 rather than 2 a = (–2.5), a.round() returns –2 rather than –3	Real
a.floor()	Returns the closest Integer less than or equal to a e.g., a = 2.5, a.floor() returns 2 a = (–2.5), a.floor() returns –3	Real

String

- Similar to strings from programming languages
- Same kind of operations
- OCL Strings are immutable
  - For example, the following returns a new String, leaving s1 and s2 unchanged

```
s1.concat(s2)
```

String operations

Syntax	Semantics
s1 = s2	Returns true if the character sequence of s1 matches the character sequence of s2, else returns false
s1 <> s2	Returns true if the character sequence of s1 does *not* match the character sequence of s2, else returns false
s1.concat(s2)	Returns a new String that is the concatenation of s1 and s2 e.g., 'Jim'.concat(' Arlow') returns 'Jim Arlow'
s1.size()	Returns the Integer number of characters in s1 e.g., 'Jim'.size() returns 3

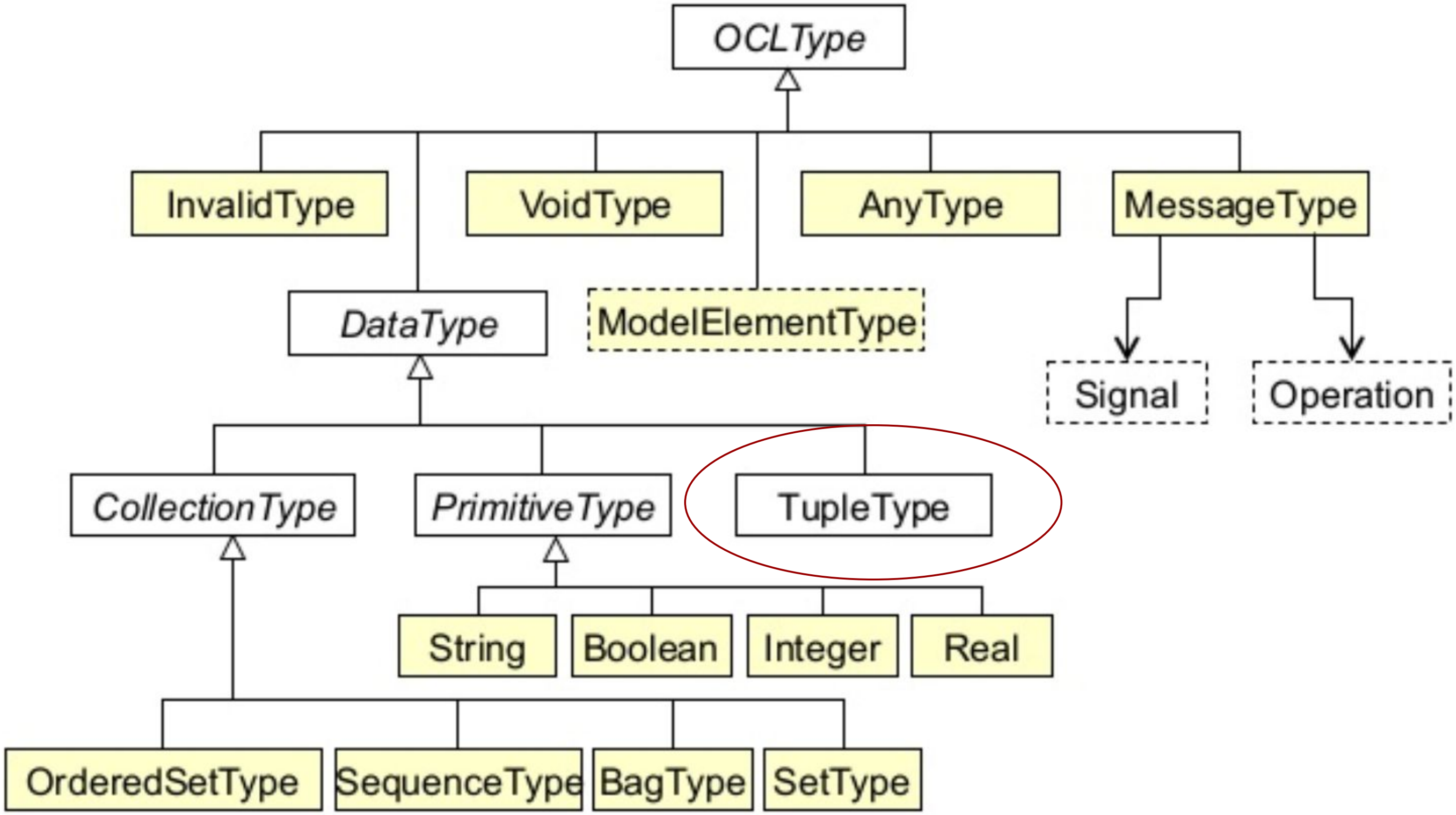
String operations

Syntax	Semantics
s1.toLower()	Returns a new String in lower case e.g., 'Jim'.toLower() returns 'jim'
s1.toUpper()	Returns a new String in upper case e.g., 'Jim'.toUpper() returns 'JIM'
s1.toInteger()	Converts s1 to an Integer value e.g., '2'.toInteger() returns 2
s1.toReal()	Converts s1 to a Real value e.g., '2.5'.toReal() returns 2.5

String operators

Syntax	Semantics
s1.substring(start, end)	Returns a new String that is a substring of s1 from the character at position start to the character at position end Notes: * start and end must be Integers * The first character in s1 is at index 1 * The last character in s1 is at index s1.size() e.g., 'Jim Arlow'.substring(5, 9) returns 'Arlow'

The OCL type system



Tuples are structured objects that have one or more named parts

- Tuples are needed as some OCL operations return multiple objects
- Tuple syntax (name and value are mandatory, type optional):

```
Tuple {partName1: partType1 = value1,
       partName2: partType2 = value2,
       ...}
```

Tuple parts can be initialized by any valid OCL expression

```
-- The following expression initializes a Tuple with
-- three parts: a course, a degree, and the number of
-- credits. The first two parts are represented by
-- Strings. The third one stores an integer value.
-- In this case, the tuple is initialized with literals.
Tuple {course: String = ‘Software Development Methods’,
       degree: String = ‘MIEI’,
       credits: Integer = 6}
```

You can access individual tuple parts with the dot operator

```
-- The following expression returns the value of
-- the degree part of this tuple.
Tuple {course: String = ‘Software Development Methods’,
       degree: String = ‘MIEI’,
       credits: Integer = 6}.degree
```

Each tuple must have a type

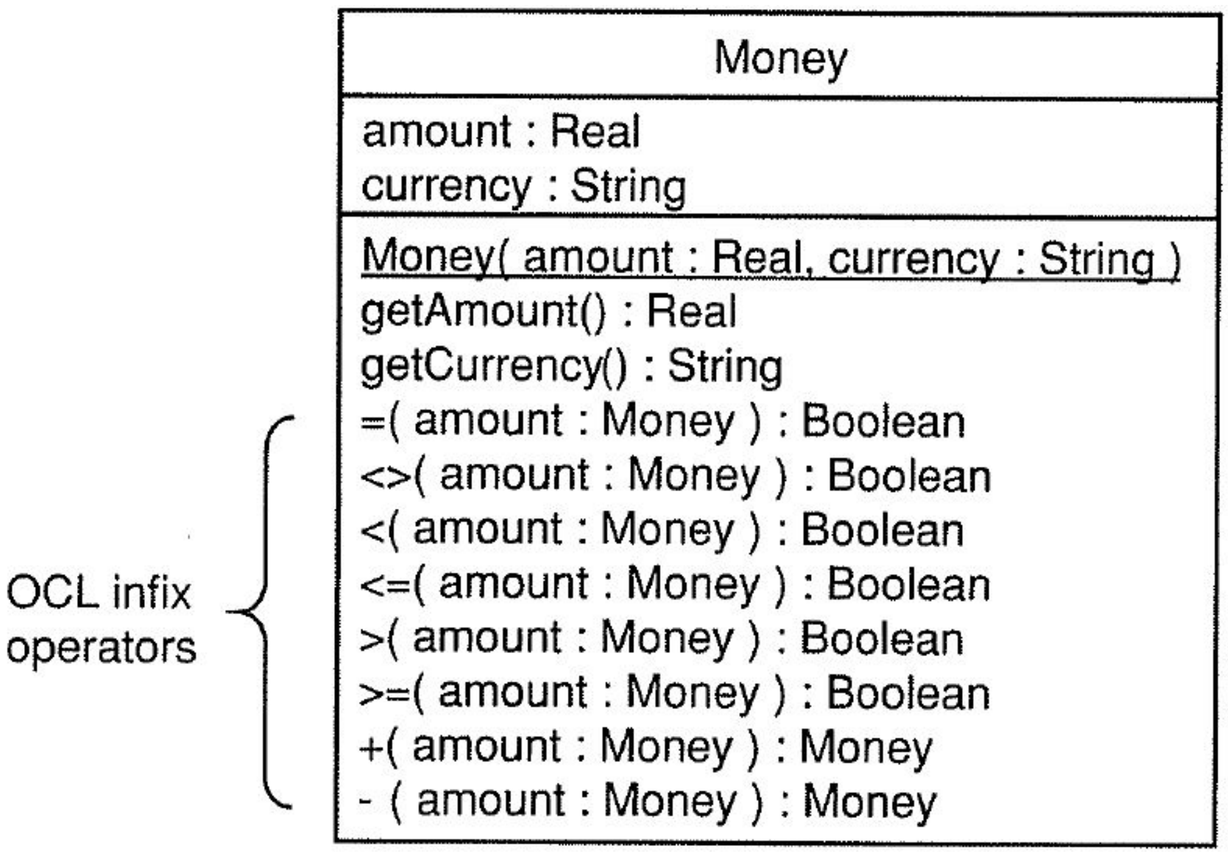
- Remember, OCL is a strongly typed language
- TupleTypes are anonymous
  - They have no name
  - They are implicitly defined (most of the times)
  - They can be explicitly defined, if necessary

```
-- The following expression creates set that can hold
-- multiple tuple objects, each representing a tuple
-- with a course, degree and its corresponding credits.
Set(TupleType {course: String,
               degree: String,
               credits: Integer})
```

You can access individual tuple parts with the dot operator

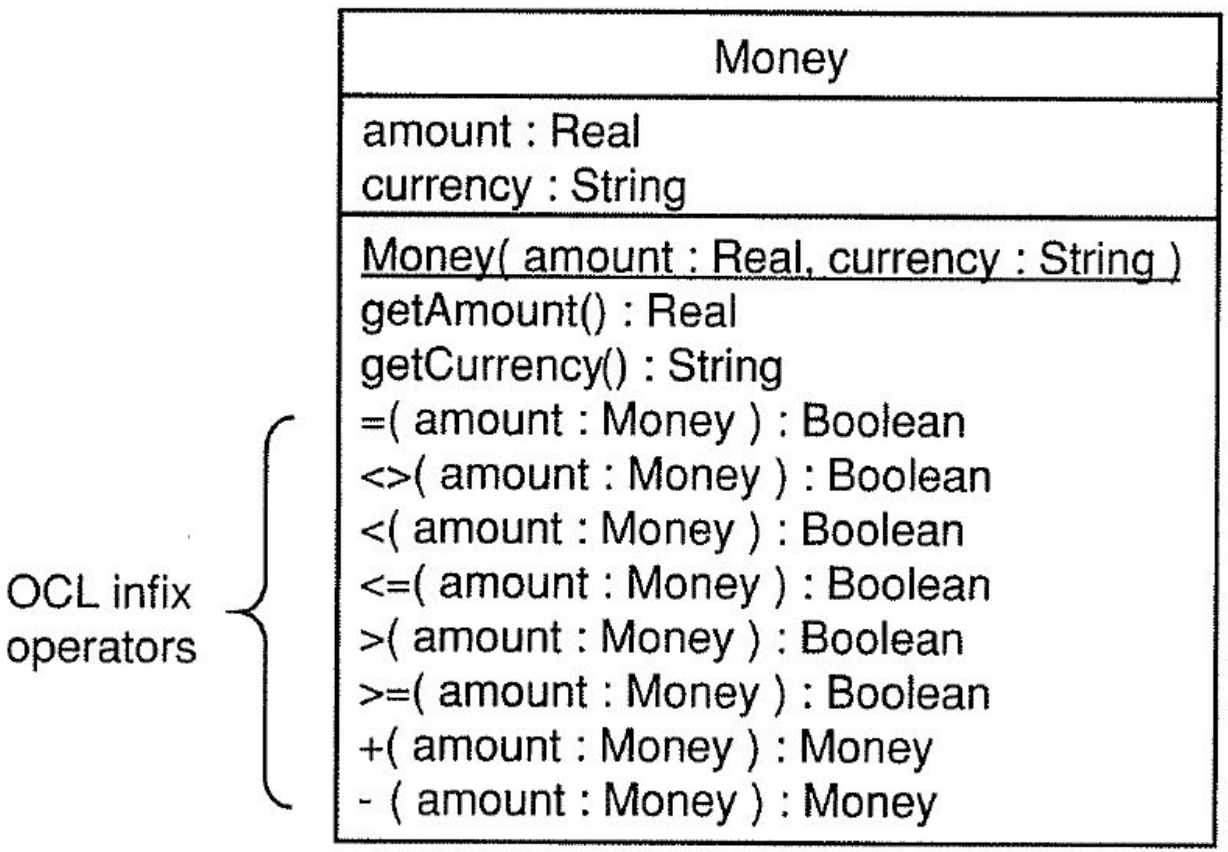
```
-- The following expression returns the value of
-- the degree part of this tuple.
Tuple {course: String = ‘Software Development Methods’,
       degree: String = ‘MIEI’,
       credits: Integer = 6}.degree
```

Infix operators are operators that are placed between their operands



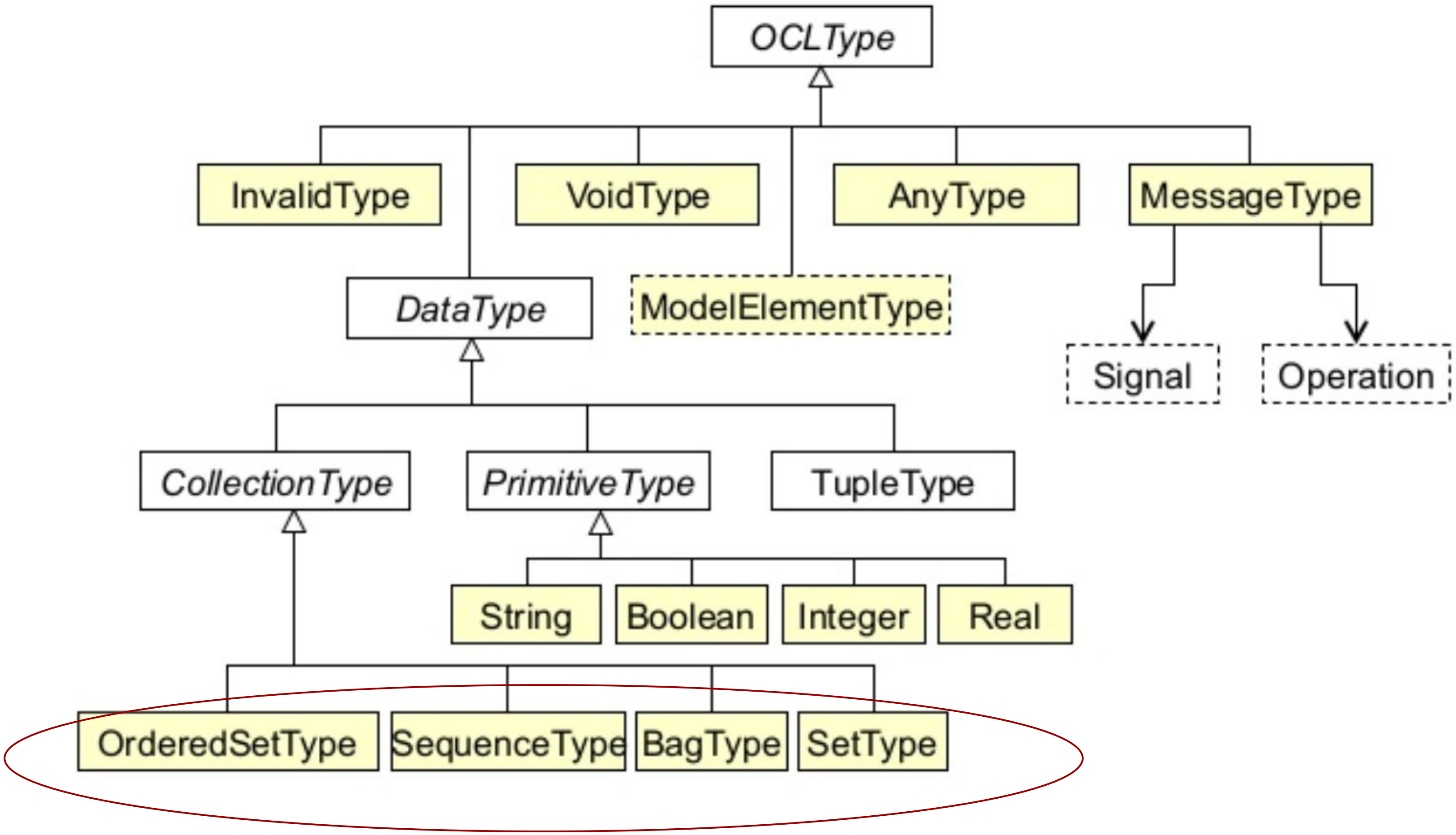

```
-- a and b are Money
-- the following calls
-- are some of the
-- supported calls.
-- Prefix operations
a.getAmount()
b.getCurrency()
-- Infix operators
a = b
a <> b
a + b
...
```

Explicit operation calls to infix operators are illegal in OCL




```
-- You cannot call
-- these operators
-- through explicit
-- operator calls.
-- It is illegal.
a.>=(b)
b.+(a)
```

The OCL type system



FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
Departamento de Informática

OCL collections

- OCL collections are immutable objects
  - When you add remove an item from a collection, this operation returns a new collection. The original collection remains untouched.
- Technically, collections are templates
  - They must be instantiated on a type (the type of their elements) before they can be used
    - E.g. `Set(Customer)` instantiates the `Set` template to hold elements of type `Customer`
- Collections can handle any element type

OCL collection types

OCL collection	Ordered	Unique (no duplicates allowed)	Association end properties
Set	No	Yes	{ unordered, unique } – default
OrderedSet	Yes	Yes	{ ordered, unique }
Bag	No	No	{ unordered, nonunique }
Sequence	Yes	No	{ ordered, nonunique }

How can you instantiate a collection?

```
Lets have the following elements: 1 5 5 3 3 4

Set{1,5,5,3,3,4}
-- evaluated Set{1,3,4,5} : Set(Integer)

OrderedSet{1,5,5,3,3,4}
-- evaluated OrderedSet{1,5,3,4} : OrderedSet(Integer)

Bag{1,5,5, 3,3,4}
-- evaluated Bag{1,3,3,4,5,5} : Bag(Integer)

Sequence{1,5,5, 3,3,4}
-- evaluated Sequence{1,5,5,3,3,4} : Sequence(Integer)
```

How can you instantiate a collection?

```
-- Enumerate the collection elements
OrderedSet{'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'}

-- Sequences of Integer literals have a special syntax
-- <start>...<end> means all the instances between <start> and
-- <end>. The following pairs of sequences are actually equivalent:
Sequence{1 .. 7}
Sequence{1, 2, 3, 4, 5, 6, 7}

Sequence{2 .. (4+3)}
Sequence{2, 3, 4, 5, 6, 7}
```

Collection elements may be other collections

```
OrderedSet{ OrderedSet{Monday, Tuesday},
            OrderedSet{Wednesday, Thursday, Friday} }
```

Collection operations

- Collections have several operations
- Collection operations must be invoked through the `->` operator
- Any single object can be treated as a collection (more precisely, a set with itself as its single element)

```
aCollection->collectionOperator(parameters...)
```

Collection conversion operations can convert a collection of one type into another of a different type

```
-- This example converts a collection of type Bag into an
-- OrderedSet. In this case, the target collection results
-- from removing duplicate elements and sorting the
-- remaining elements according to an arbitrary order.
Bag{‘Homer’, ‘Simpson’)->asOrderedSet()
```

Conversion operations: asSet(), asOrderedSet(), asBag(), asSequence()

Conversion operations

Collection operation	Semantics
X(T)::asSet() : Set(T) X(T)::asOrderedSet() : OrderedSet(T) X(T)::asBag() : Bag(T) X(T)::asSequence() : Sequence(T)	Converts a collection from one type of collection to another When a collection is converted to a Set, duplicate elements are discarded When a collection is converted to an OrderedSet or a Sequence, the original order (if any) is preserved, else an arbitrary order is established

Conversion operations: flatten()

Conversion operations	
Collection operation	**Semantics**
X(T)::flatten() : X(T2)	Results in a new flattened collection instantiated on T2 For example, if we have: Set{ Sequence{ 'A', 'B' }, Sequence{ 'C', 'D' } } the Set is instantiated on a Sequence that is instantiated on String – the result of flattening the Set is therefore a Set of String

Comparison operations compare the target collection with a parameter collection

- The comparison returns a Boolean
- The operations take into account the ordering of the collections

Comparison operations	
Collection operation	**Semantics**
X(T)::=(y : X(T)) : Boolean	Set and Bag – returns true if y contains the same elements as the target collection OrderedSet and Sequence – returns true if y contains the same elements in the same order as the target collection
X(T)::<>(y : X(T)) : Boolean	Set and Bag – returns true if y does *not* contain the same elements as the target collection OrderedSet and Sequence – returns true if y does *not* contain the same elements in the same order as the target collection

Query operations allow you to obtain information about the collection

Query operations

Collection operation	Semantics
X(T)::size() : Integer	Returns the number of elements in the target collection
X(T)::sum() : T	Returns the sum of all of the elements in the target collection Type T *must* support the + operator
X(T)::count(object : T) : Integer	Returns the number of occurrences of object in the target collection
X(T)::includes(object : T) : Boolean	Returns true if the target collection contains object
X(T)::excludes(object : T) : Boolean	Returns true if the target collection does *not* contain object
X(T)::includesAll(c : Collection(T)) : Boolean	Returns true if the target collection contains everything in c
X(T)::excludesAll(c : Collection(T)) : Boolean	Returns true if the target collection does *not* contain all of the elements in c
X(T)::isEmpty() : Boolean	Returns true if the target collection is empty, else returns false
X(T)::notEmpty() : Boolean	Returns true if the target collection is not empty else, returns false

Access operations allow accessing elements in a particular position within the collection

- This only works for ordered collections
  - For unordered collections, you need to iterate all their elements

Access operations	
Collection operation	**Semantics**
OrderedSet(T)::first() : T Sequence(T)::first() : T	Returns the first element of the collection
OrderedSet(T)::last() : T Sequence(T)::last() : T	Returns the last element of the collection
OrderedSet::at(i) : T Sequence::at(i) : T	Returns the element at position i
OrderedSet::indexOf(T) : Integer	Returns the index of the parameter object in the OrderedSet

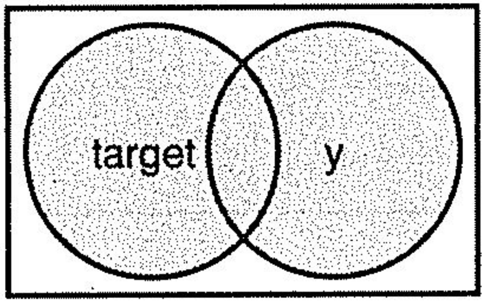
Selection operations return new collections which are supersets, or subsets of the target collection

- union
- intersection
- symmetricDifference
- - (note: this is read as the “complement” of a set)
- product
- including
- excluding
- subsequence
- subOrderedSet
- append
- prepend
- insertAt

X(T)::union(y: X(T)): X(T)

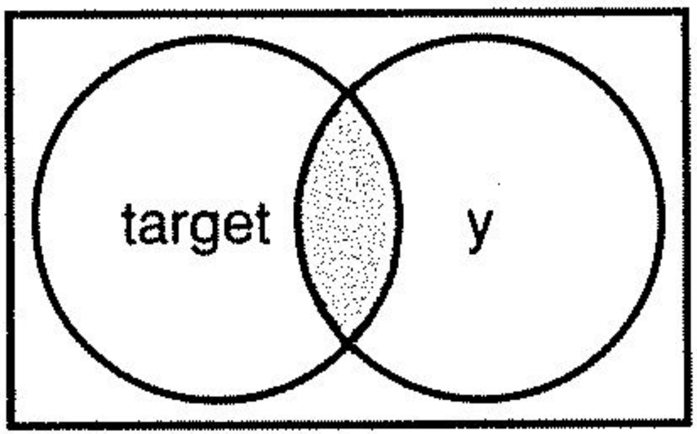
Returns a new collection that is the result of appending y to the target collection – the new collection is always of the same type as the target collection

Duplicate elements are removed and an order established as necessary



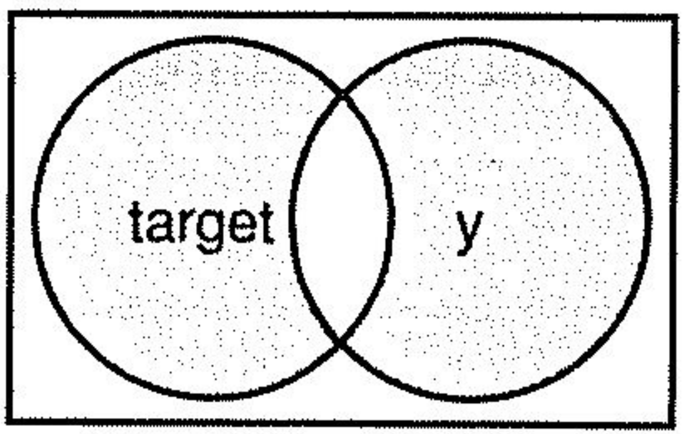
Set(T)::intersection(y: Set(T)): Set(T)
OrderedSet(T)::intersection(y: OrderedSet(T)): OrderedSet(T)

Returns a new collection containing elements common to y and the target collection



Set(T)::symmetricDifference(y: Set(T)): Set(T)
OrderedSet(T)::symmetricDifference(y: OrderedSet(T)): OrderedSet(T)

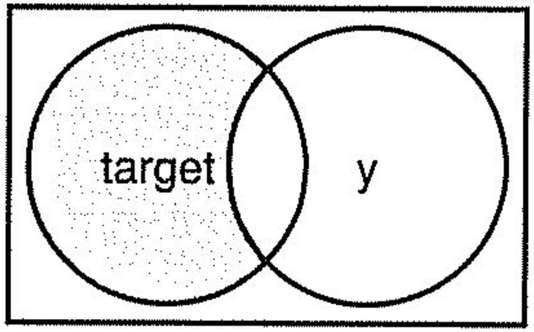
Returns a new Set that contains elements that exist in the target collection and y, but not in both



Set(T)::-(y: Set(T): Set(T)
OrderedSet(T)::-(y: OrderedSet(T): OrderedSet(T)

Returns a new Set that contains all elements of the target collection that are *not* also in y

In set theory, the return set is the *complement* of a with respect to b



X(T)::product(y: X(T2)): Set(Tuple(first: T, second: T2))

Returns the Cartesian product of the target collection and y – this is a Set of Tuple{ first=a,second=b } objects where a is a member of the target collection and b is a member of y

e.g., Set{ 'a',' b' }->product(Set{ '1','2' })

returns

Set{ Tuple{ first='a', second='1' },
Tuple{ first='a',second='2' }, Tuple{ first='b',
second='1' }, Tuple{ first='b',second='2' } }

X(T)::including(object: T): X(T)

Returns a new collection containing the contents of the target collection plus object

If the collection is ordered, object is appended

X(T)::excluding(object: T): X(T)

Returns a collection with all the copies of object removed

Sequence(T)::subsequence(i: Integer, j: Integer): Sequence(T)

Returns a new Sequence that contains elements from index i to index j of the target collection

OrderedSet::subOrderedSet(i: Integer, j: Integer): OrderedSet(T)

Returns a new OrderedSet that contains the elements from index i to index j of the target OrderedSet

OrderedSet(T)::append(object: T): OrderedSet(T)
Sequence(T)::append(object: T): Sequence(T)

Returns a new collection with object added on to the end

OrderedSet(T)::prepend(object: T): OrderedSet(T)
Sequence(T)::prepend(object: T): Sequence(T)

Returns a new collection with object added on to the beginning

OrderedSet(T)::insertAt(index: Integer, object: T): OrderedSet(T)
Sequence(T)::insertAt(index: Integer, object: T): Sequence(T)

Returns a new collection with object inserted at the index position

Iteration operations

- Allow to loop over the elements in a collection
- When each element of the collection is visited, all its features are automatically accessible to `iteratorExpression` and can be assessed by name
- The general format is as follows:

```
aCollection-><iteratorOperation>(<iteratorVariable>:<Type> |
                                     <iteratorExpression>
                                 )
```

Boolean iterator operations

Boolean iterator operations	Semantics
X(T)::exists(i : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for at least one value of i, else returns false
X(T)::forAll(i : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for all values of i, else returns false
X(T)::forAll(i : T, j : T ..., n : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for every { i, j ... n } Tuple, else returns false The set of { i, j ... n } pairs is the Cartesian product of the target collection with itself
X(T)::isUnique(i : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression has a unique value for each value of i, else returns false
X(T)::one(i : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for exactly one value of i, else returns false

Selection iterator operators

Selection iterator operations	Semantics
X(T)::any(i : T | iteratorExpression) : T	Returns a random element of the target collection for which iteratorExpression is true
X(T)::collect(i : T | iteratorExpression) : Bag(T)	Returns a Bag containing the results of executing iteratorExpression once for each element in the target collection (See Section 25.9.2 for a shorthand notation for collect(...))
X(T)::collectNested(i : T | iteratorExpression) : Bag(T)	Returns a Bag of collections containing the results of executing iteratorExpression once for each element in the target collection Maintains the nesting of the target collection in the result collection

More selection iterator operations

<table>
<tr><th>Selection iterator operations</th><th>Semantics</th></tr>
<tr><td>X(T)::select(i : T | iteratorExpression) : X(T)</td><td>Returns a collection containing those elements of the target collection for which the iteratorExpression evaluates to true</td></tr>
<tr><td>X(T)::reject(i : T | iteratorExpression) : X(T)</td><td>Returns a collection containing those elements of the target collection for which the iteratorExpression evaluates to false</td></tr>
<tr><td>X(T)::sortedBy(i : T | iteratorExpression) : X(T)</td><td>Returns a collection containing the elements of the target collection ordered according to the iteratorExpression
The iteratorVariable must be of a type that has the < operator defined</td></tr>
</table>

forall

- forall has two forms
  - With a single iteratorVariable
  - With nested forall operations

```
c->forAll(i | c->forAll(j | iteratorExpression))
```

can be written as

```
c->forAll(i, j | iteratorExpression)
```

`c->forAll(i, j | iteratorExpression)`

- Iterates over a set of `{i, j}` pairs
- These pairs are the Cartesian product of c with itself

```
c = Set{x, y, z}
```

The cartesian product of c with itself is the Set:

```
{ {x, x}, {x, y}, {x, z},
  {y, x}, {y, y}, {y, z},
  {z, x}, {z, y}, {z, z} }
```

iterate (How can we sum all numbers in a bag?)

Syntax:

```
aCollection->iterate( <iteratorVariable>:<Type>
                      <resultVariable>: <ResultType> =
                         <initializationExpression> |
                         <iteratorExpression>
                    )
```

Example:

```
Bag{1, 2, 3, 4, 5}->iterate(number: Integer;    -- This is the variable
                            sum: Integer = 0 | -- Initialization
                            sum + number       -- what is done each iter.
                           )
-- Result: 15 : Integer
Bag{1, 2, 3, 4, 5}->sum() -- equivalent to the above definition
-- Result: 15 : Integer
```

iterate (How can we filter only the positives?)

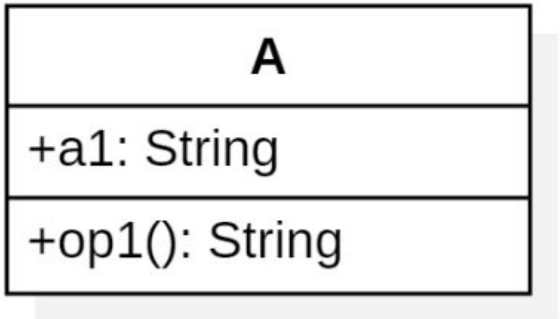
```
Set{-2, -3, 1, 2}->iterate(number: Integer;
                           positiveNumbers: Set(Integer) = Set{} |
                           if number >= 0 then
                              positiveNumbers->including(number)
                           else
                              positiveNumbers
                           endif
                          )
-- Result: Set{1,2} : Set(Integer)

-- The above code is equivalent to this one:
Set{-2, -3, 1, 2}->select(number: Integer | number >= 0)
-- Result: Set{1,2} : Set(Integer)
```

OCL navigation is the ability to get from a source object to one or more target objects

- Navigation expressions can refer to
  - Classifiers
  - Attributes
  - Association ends
  - Query operations
    - Operations where the property isQuery is set to true

Navigation within the contextual instance

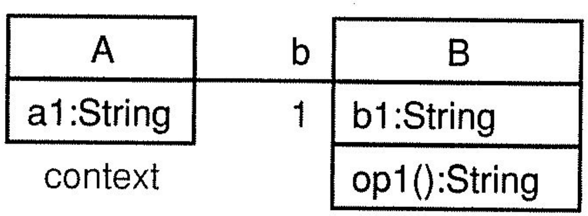
- Access the contextual instance with self
- Access properties of contextual difference directly
- Only query operations are accessible



Navigation expression	Semantics
self	The contextual instance – an instance of A
self.a1 a1	The value of attribute a1 of the contextual instance
self.op1() op1()	The result of op1() called on the contextual instance The operation op1() *must* be a query operation

Navigation across associations

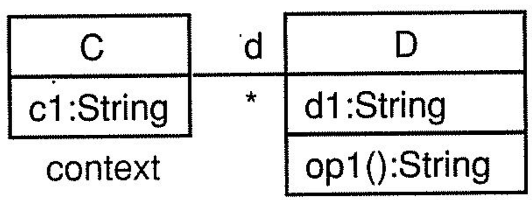
- Use the dot operator to navigate across associations

Example model	Navigation expressions (A is the expression context)	
	Expression	**Value**
	self	The contextual instance – an instance of A
	self.b	An object of type B
	self.b.b1	The value of attribute B::b1
	self.b.op1()	The result of operation B::op1()

Navigation across boundaries

- Navigation semantics depends on the multiplicity on

Example model



Navigation expressions	
Expression	**Value**
self	The contextual instance – an instance of C
self.d	A Set(D) of objects of type D
self.d.d1	A Bag(String) of the values of attribute D::d1 Shorthand for self.d->collect(d1)
self.d.op1()	A Bag(String) of the results of operation D::op1() Shorthand for self.d->collect(op1())

By default, the dot operator returns a set when the multiplicity is > 1.

- This default can be overridden using association end properties

OCL collection	Association end properties
Set	{ unordered, unique } – default
OrderedSet	{ ordered, unique }
Bag	{ unordered, nonunique }
Sequence	{ ordered, nonunique }

The collect shorthand notation

- Accessing a property of a collection is a shorthand for collect(...)

```
self.d.d1 -- is a shorthand for
self.d->collect(d1)

self.d.op1() --is a shorthand for self.d->collect(d.op1())
-- returns a Bag containing the return values of
-- operation op1, applied to each D object in the
-- Set(D) obtained by traversing self.d
```

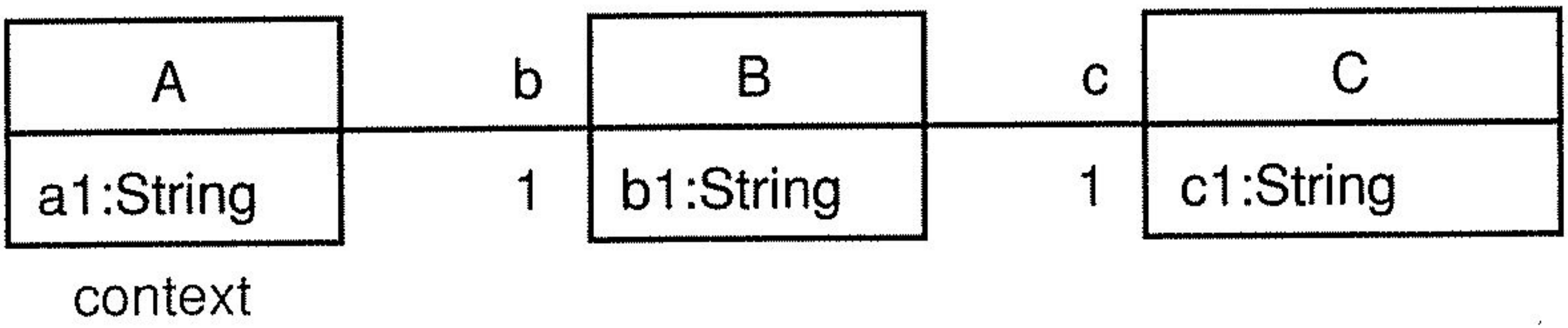
Navigation across multiple associations

- Navigation beyond a relationship end of multiplicity > returns a Bag

self.s.t.m -- is equivalent to
self.s->collect(t)->collect(m)

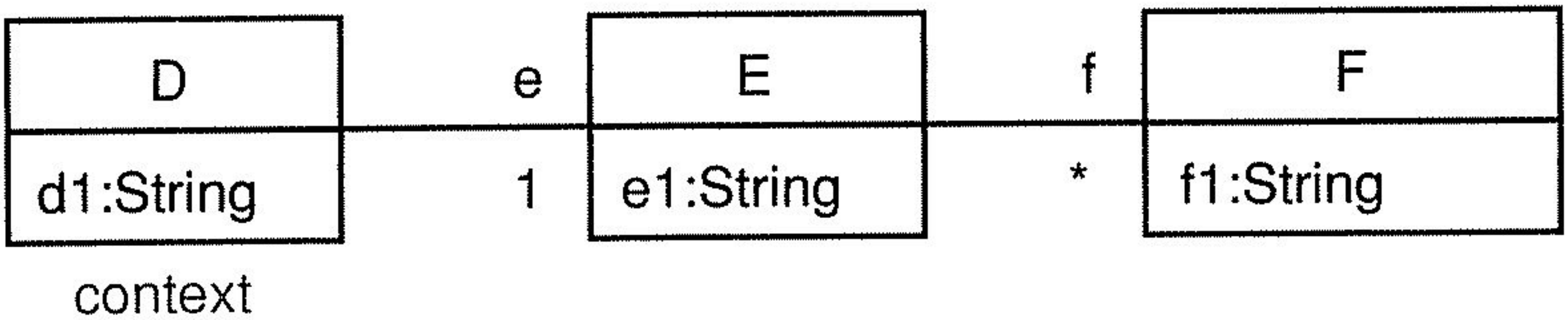
- Although you can navigate across as many associations as you want to, it is usually a good idea to keep it simple (up to two navigations) for the sake of the understandability of your expressions

Navigation examples (¼)



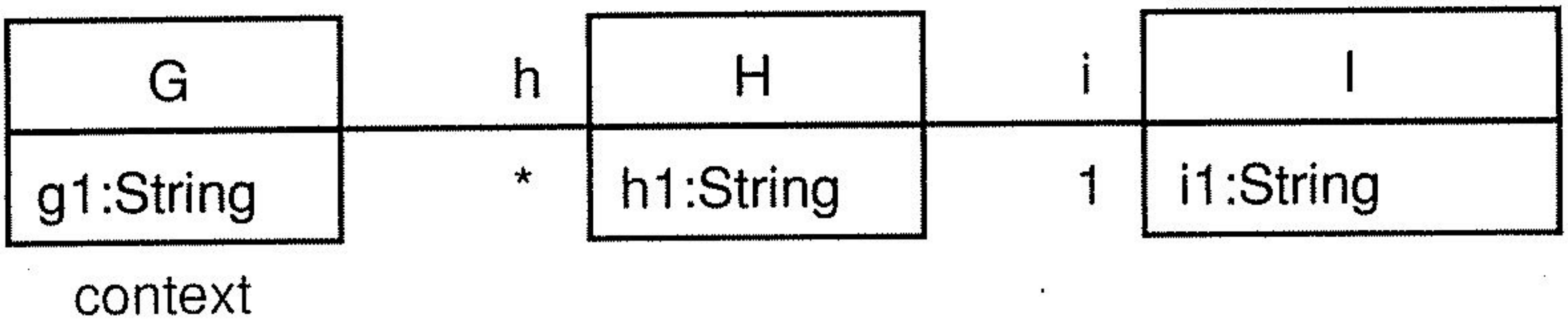
self	The contextual instance – an instance of A
self.b	An object of type B
self.b.b1	The value of attribute B::b1
self.b.c	An object of type C
self.b.c.c1	The value of attribute C::c1

Navigation examples (2/4)



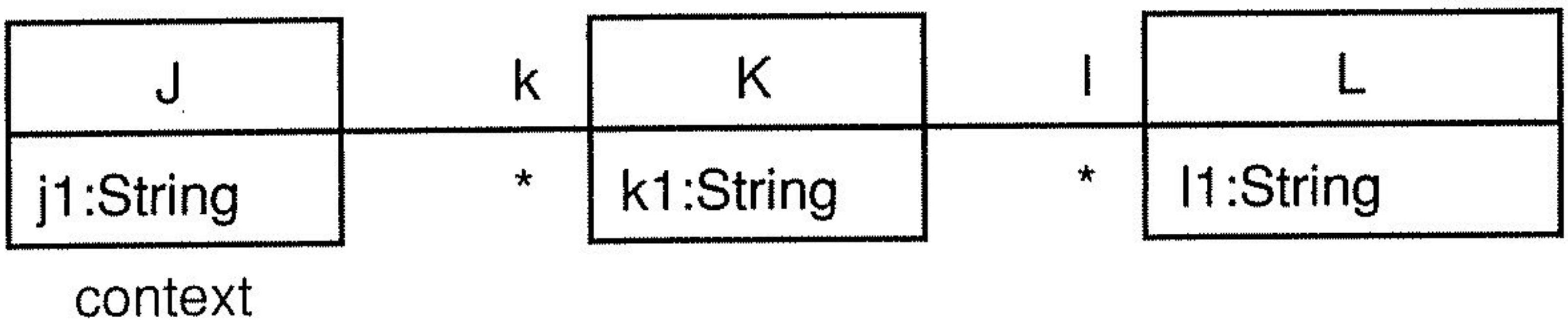
self	The contextual instance – an instance of D
self.e	An object of type E
self.e.e1	The value of attribute E::e1
self.e.f	A Set(F) of objects of type F
self.e.f.f1	A Bag(String) of values of attribute F::f1

Navigation examples (¾)



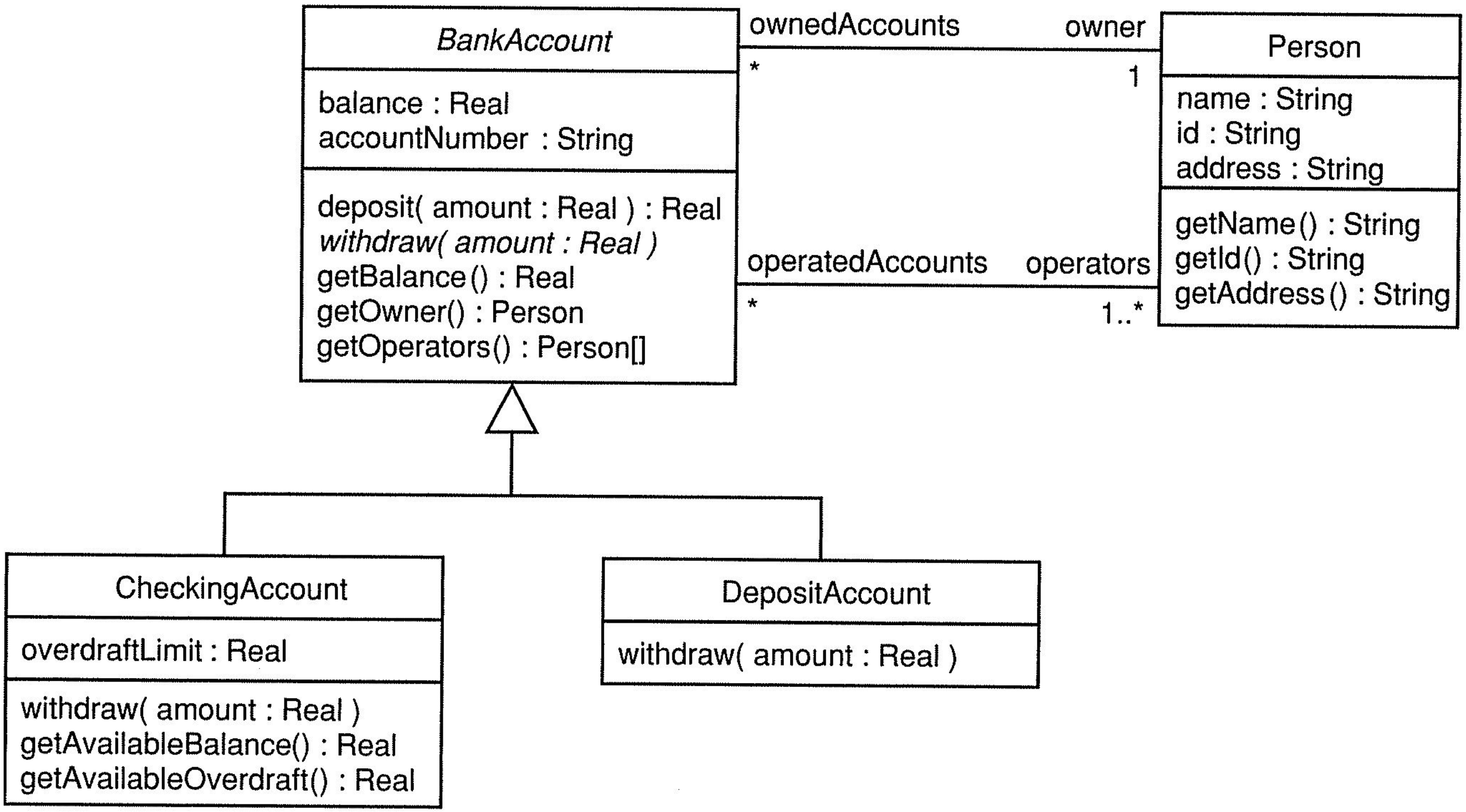
self	The contextual instance – an instance of G
self.h	A Set(H) of objects of type H
self.h.h1	A Bag(String) of values of attribute H::h1
self.h.i	A Bag(I) of objects of type I
self.h.i.i1	A Bag(String) of values of attribute I::i1

Navigation examples (4/4)

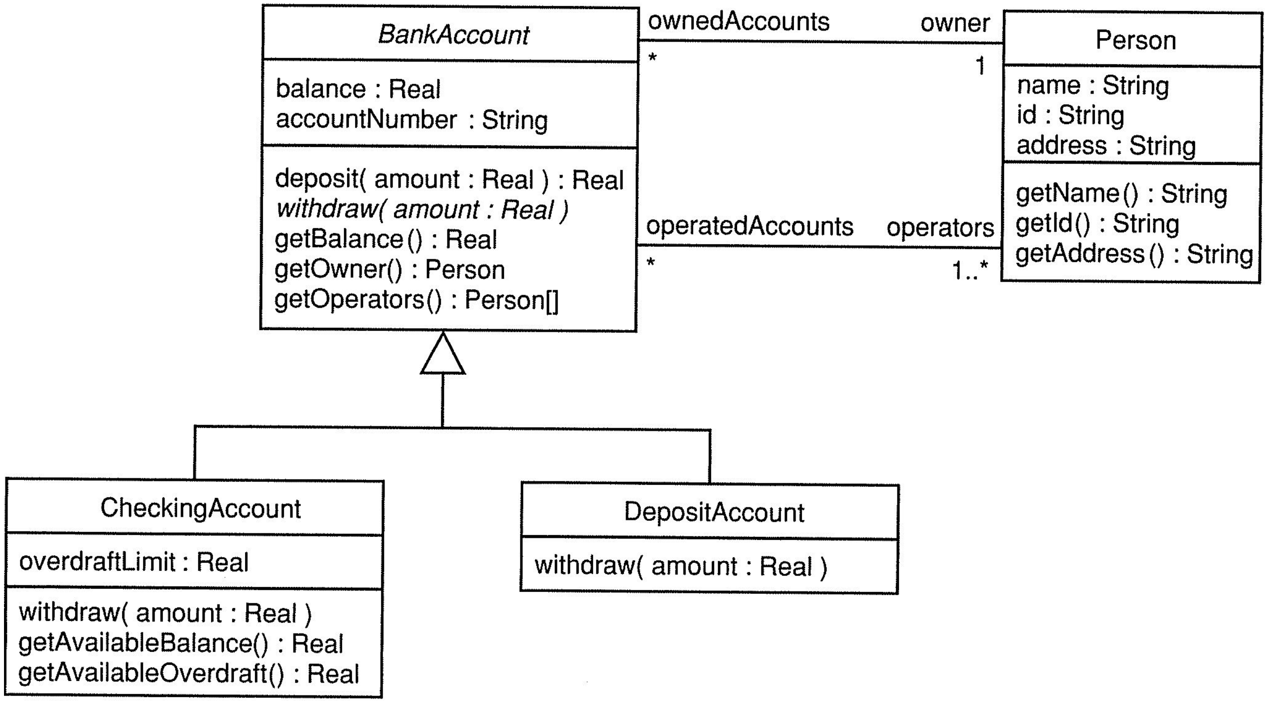


self	The contextual instance – an instance of J
self.k	A Set(K) of objects of type K
self.k.k1	A Bag(String) of values of attribute K::k1
self.k.l	A Bag(L) of objects of type L
self.k.l.l1	A Bag(String) of values of attribute L::l1

inv: an invariant is something that must be true for all instances of its context classifier

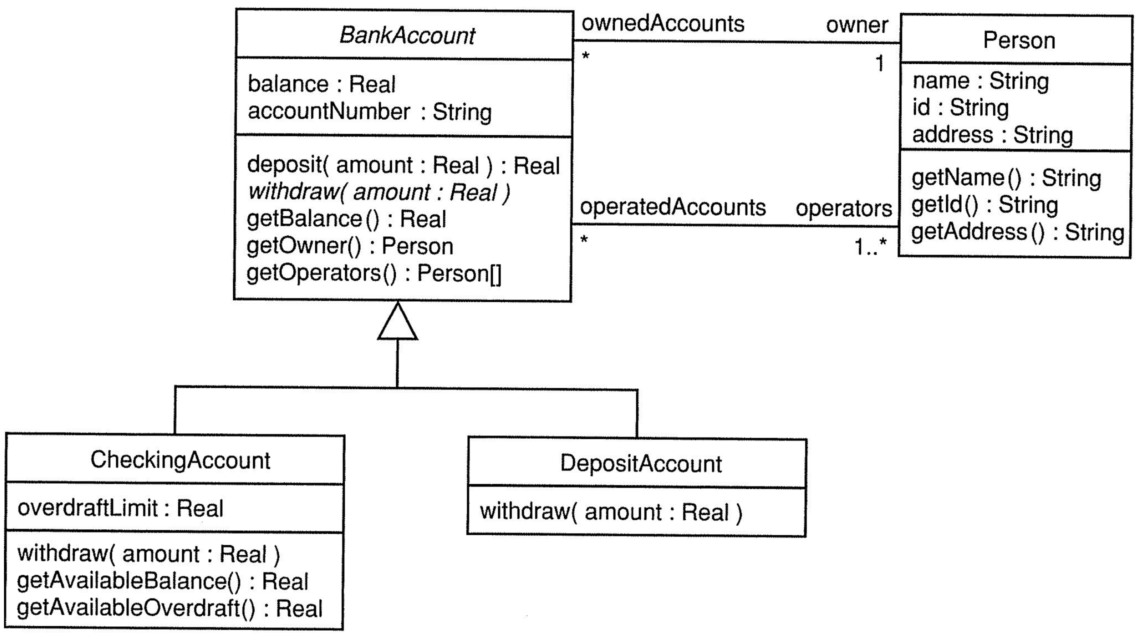


inv: No account shall be overdrawn by more than $1000.



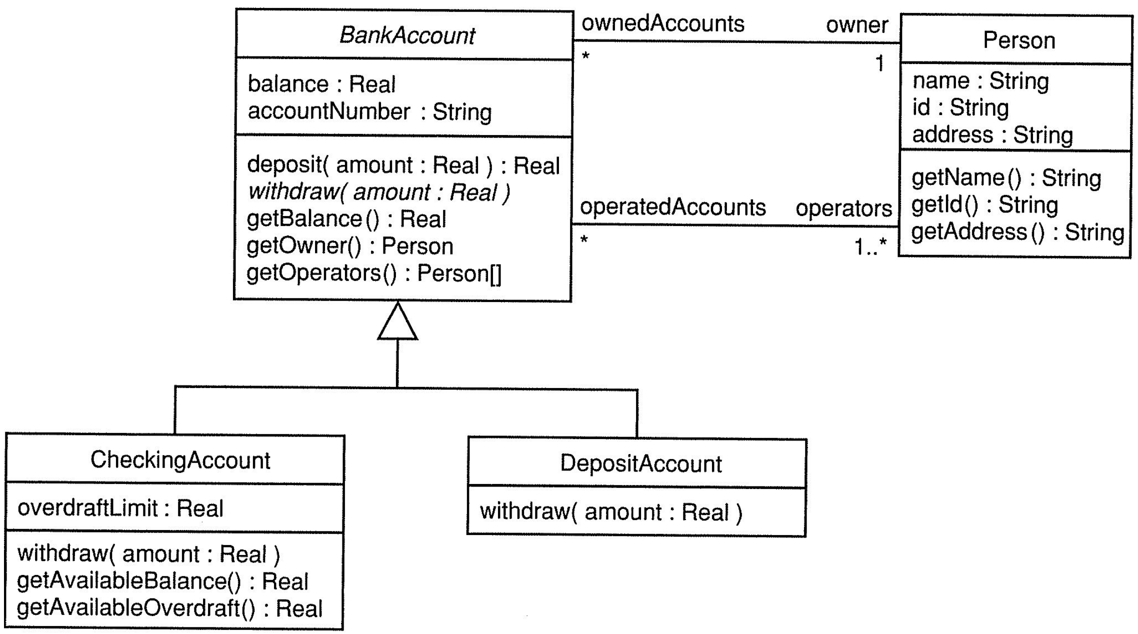

```
context BankAccount
   inv balanceValue:
      self.balance >= (-1000.0)
```

A subclass may strengthen a superclass invariant, but can't weaken it, to preserve the substitutability principle



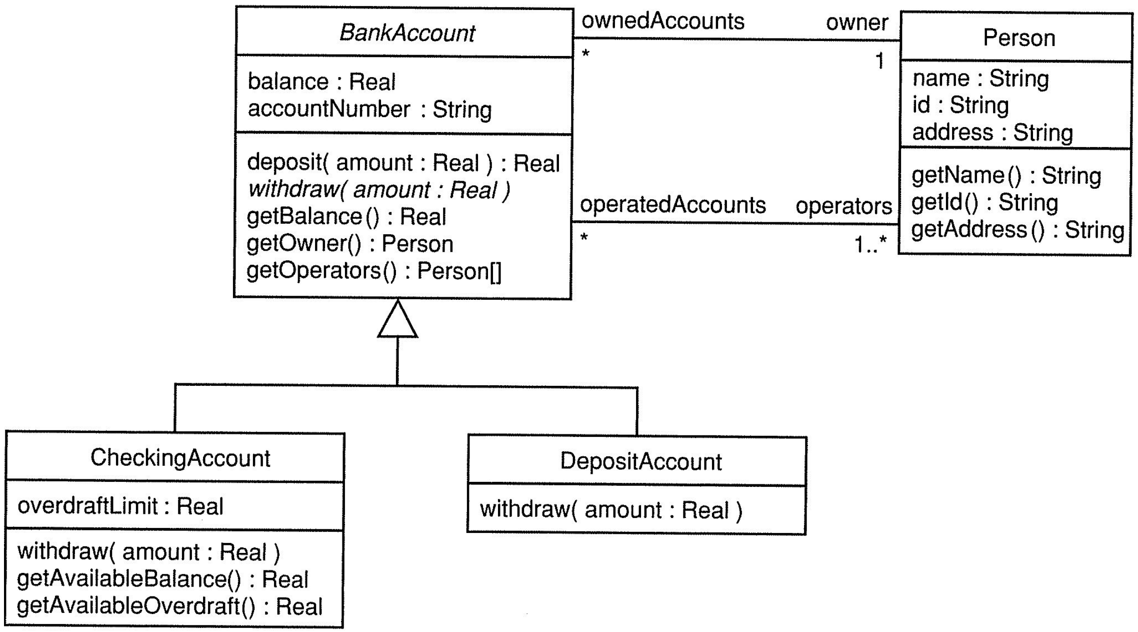

```
context CheckingAccount
   inv balanceValue:
      self.balance >= (-overdraftLimit)
      self.overdraftLimit <= (-1000.0)
```

DepositAccounts shall have a balance of zero or more (this is also stronger than in the superclass)



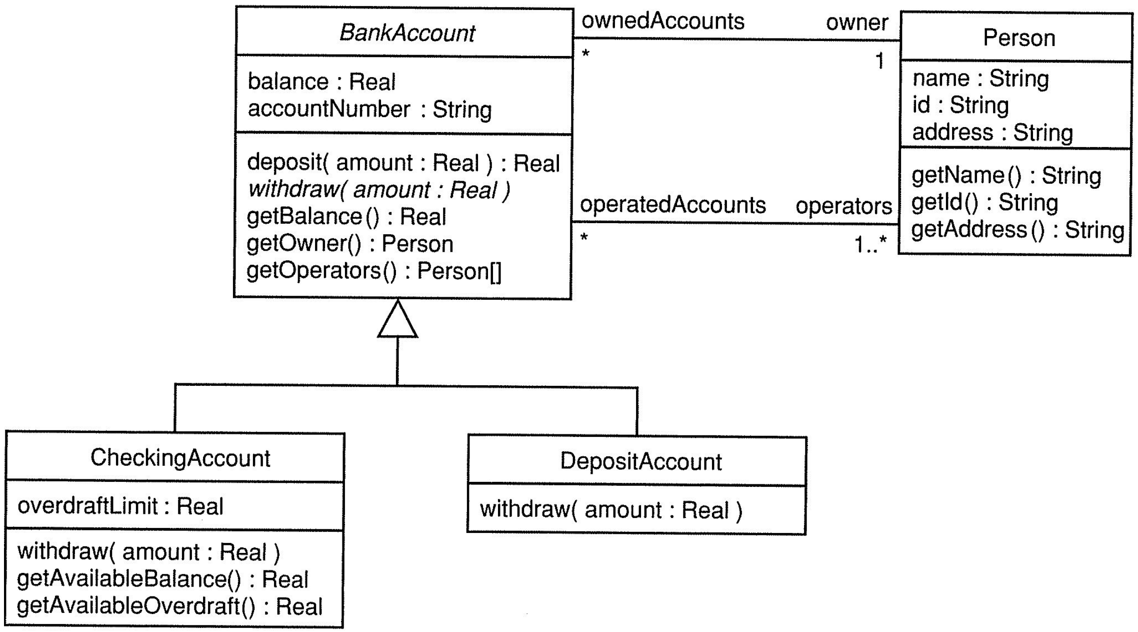

```
context DepositAccount
   inv balanceValue:
      self.balance > 0.0
```

Each BankAccount must have a unique account number



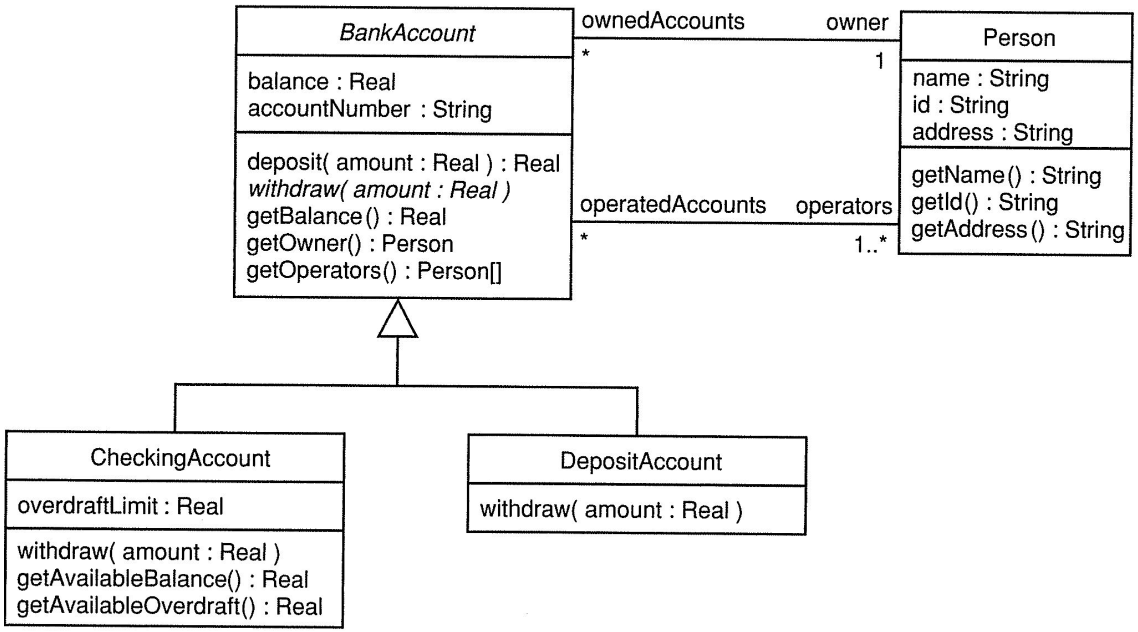

```
context BankAccount
   inv uniqueAccountNumber:
   BankAccount::allInstances()->isUnique(account
                                          |account.accountNumber)
```

The owner of the bank account must be




```
context BankAccount
   inv ownerIsOperator:
   self.operators->includes(self.owner)
```

A Person's owned accounts shall be a subset of a person's operated accounts




```
context Person
   inv ownedAccountsSubsetOfOperatedAccounts:
   self.operatedAccounts->includesAll(self.ownedAccounts)
```

Here, there be dragons!

- When comparing objects of the same type, remember they may be:
    - **identical** - each object refers to the same region of memory (same object reference)
    - **equivalent** - each object has the same set of attribute values, but different object references
- This is just like comparing objects in Java:
  == (identical) is not the same as equals() (equivalent)

Preconditions and postconditions apply to operations

- Preconditions state things that must be true before an operation executes
- Postconditions state things that must be true after an operation executes

Back to BankAccount, and to the operation

- In the deposit() operation, there are two rules:
    - The amount to be deposited shall be greater than 0
    - The final balance shall be the original balance plus the amount

```
context BankAccount::deposit(amount: Real): Real
   pre amountToDepositGreaterThanZero:
      amount > 0
   post depositSucceeded:
      self.balance = self.balance@pre + amount
```

Still in BankAccount, now for the withdraw()

- In the deposit() operation, there are two rules:
  - The amount to withdraw shall be greater than 0
  - The final balance shall be the original balance minus the amount

```
context BankAccount::withdraw(amount: Real): Real
   pre amountToWithdrawGreaterThanZero:
      amount > 0
   post withdrawalSucceeded:
      self.balance = self.balance@pre - amount
```

What if the class is within an inheritance hierarchy?

- Redefined operations may only weaken the pre-conditions
- Redefined operations may only strengthen the post condition
- Remember the principle of substitutibility

body: where you specify the result of a query operation? Examples from BankAccount

```
context BankAccount::getBalance(): Real
  body:
    self.balance

context BankAccount::getOwner(): Person
  body:
    self.owner

context BankAccount::getOperators(): Set(Person)
  body:
    self.operators
```

body: where you specify the result of a query operation? Examples from CheckingAccount

```
context CheckingAccount::getAvailableBalance(): Real
  body:
    self.balance + self.overdraftLimit

context CheckingAccount::getAvailableOverdraft(): Real
  body:
    if self.balance >= 0 then
       self.overdraftLimit
    else
       self.balance + self.overdraftLimit
    endif
```

init: Initialize attributes

- OCL can set the initial value of the variables
- This is mostly used for complex initializations
  - Otherwise, it is more common to make these initializations directly in the class attribute compartment

```
context BankAccount::balance
   init:
      0
```

def: lets you define variables and helper operations on a classifier for use in other OCL expressions

```
context CheckingAccount::getAvailableBalance() : Real
  body:
    -- you can withdraw an amount to take your account down to your overdraft limit
    balance + overdraftLimit

context CheckingAccount::getAvailableOverdraft() : Real
  body:
    if balance >= 0 then
      -- the full overdraft facility is available
      overdraftLimit
    else
      -- you have used up part of the overdraft facility
      balance + overdraftLimit
    endif
```

def: lets you define variables and helper operations on a classifier for use in other OCL expressions

```
context CheckingAccount
  def:
    availableBalance = balance + overdraftLimit
context CheckingAccount::getAvailableBalance() : Real
  body:
    -- you can withdraw an amount to take your account down to your overdraft limit
    availableBalance

context CheckingAccount::getAvailableOverdraft() : Real
  body:
    if balance >= 0 then
      -- the full overdraft facility is available
      overdraftLimit
    else
      -- you have used up part of the overdraft facility
      availableBalance
    endif
```

def: You can also define helper functions

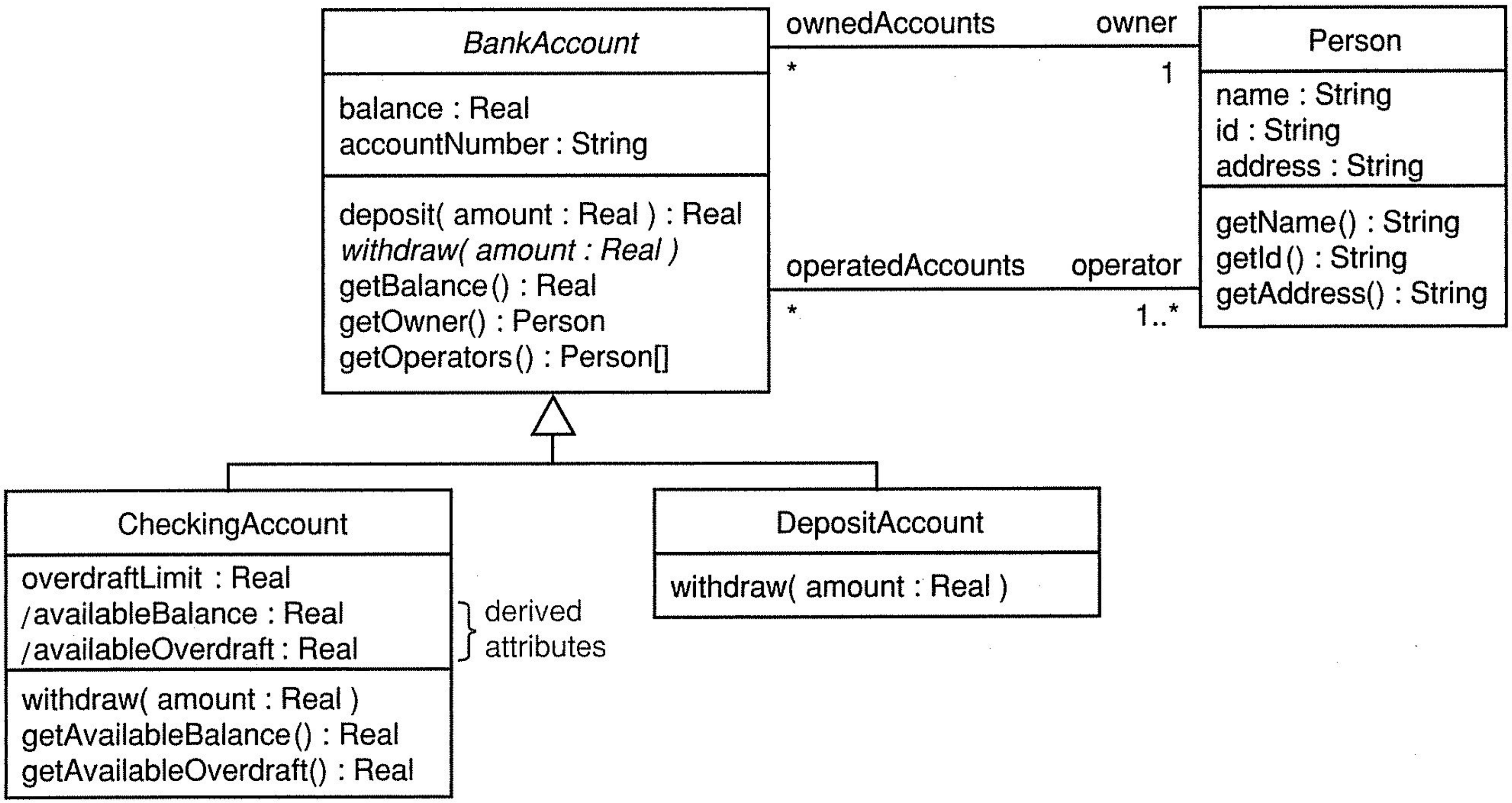
```
context CheckingAccount
  def:
    canWithdraw( amount : Real ) : Boolean = ( (availableBalance – amount ) >= 0 )
```

let defines a variable local to an OCL expression

```
let <variableName>:<variableType> = <letExpression> in
  <usingExpression>
```

```
context BankAccount::withdraw( amount : Real )
  post withdrawalSucceeded:
    let originalBalance : Real = self.balance@pre in
      -- the final balance is the original balance minus the amount
      self.balance = originalBalance - amount
```

derive: We can use OCL to derive attributes



derive: Derivation rules can define derived attributes

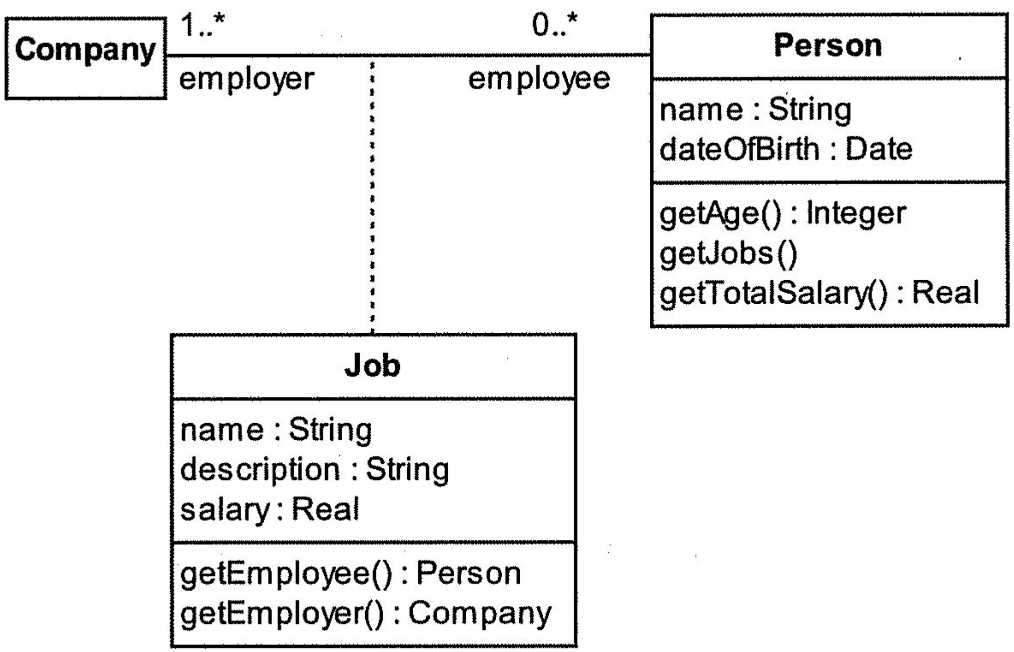
```
context CheckingAccount::availableBalance : Real
  derive:
    -- you can withdraw an amount up to your overdraft limit
    balance + overDraftLimit

context CheckingAccount::availableOverdraft : Real
  derive:
    if balance >= 0 then
      -- the full overdraft facility is available
      overdraftLimit
    else
      -- you have used part of the overdraft facility
      overdraftLimit + balance
    endif
```

body: You can then use these derived attributes to simplify other operations

```
context CheckingAccount::getAvailableBalance() : Real
  body:
    availableBalance

context CheckingAccount::getAvailableOverdraft() : Real
  body:
    availableOverdraft
```

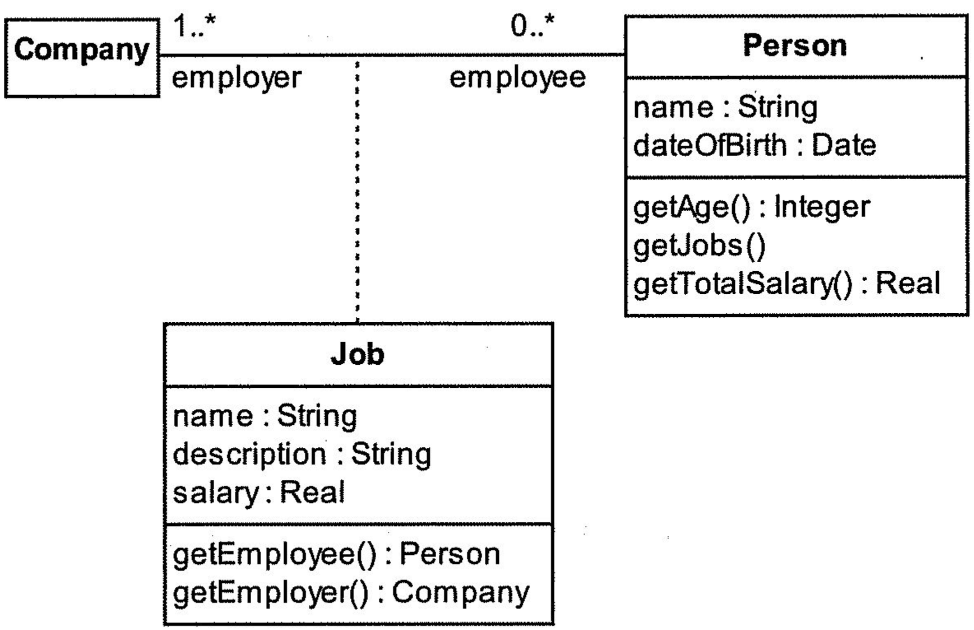
body: Use the association class name to navigate to an association

```
context Person::getJobs() : Set(Job)
  body:
    self.Job
```



inv: What if a person can't have more than one job at a time?

context Person
inv:
-- a person can't have the same Job more than once
self.Job->isUnique(j : Job | j.name)



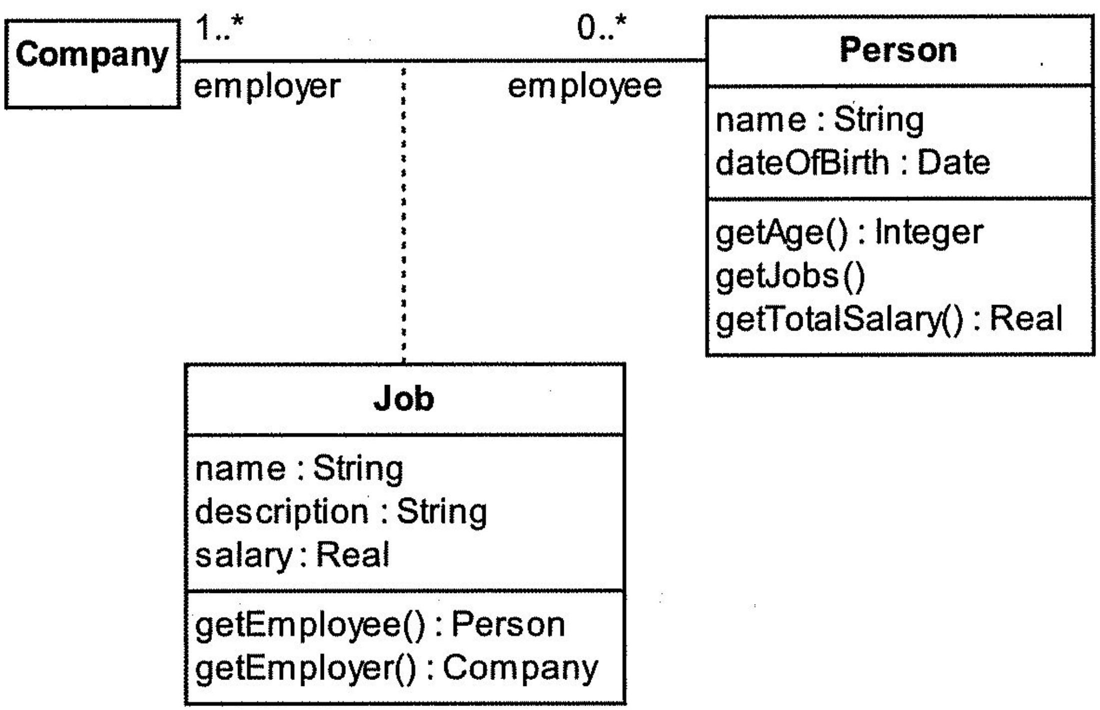
inv: What if the “only one” job restriction only applies to senior citizens over 60 years old (and every 60+ years old citizen has a job)?

context Person

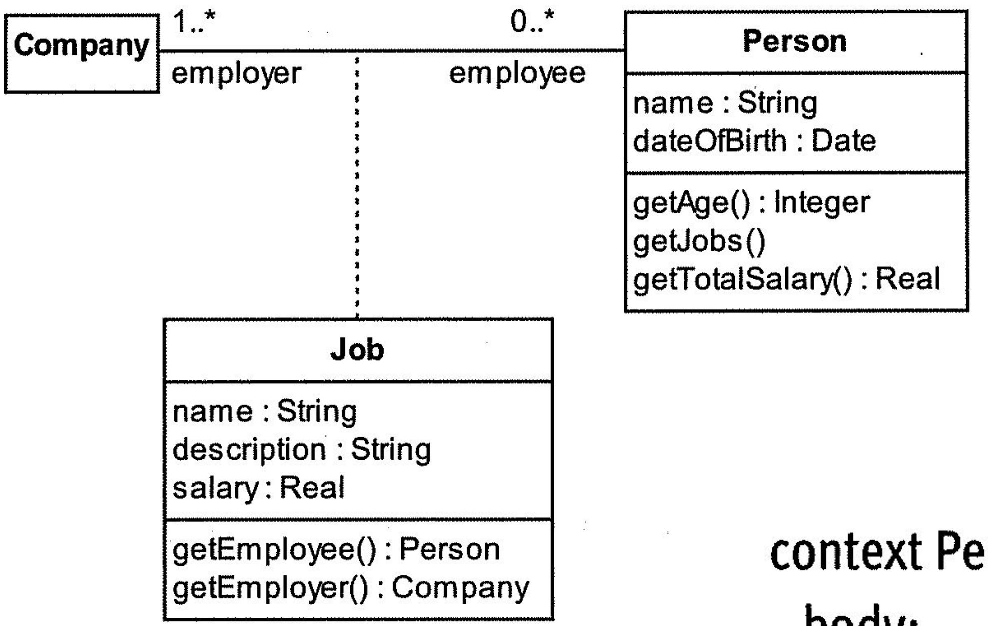
inv:

-- people over 60 can only have one Job

(self.getAge() > 60) implies (self.Job->count() = 1)

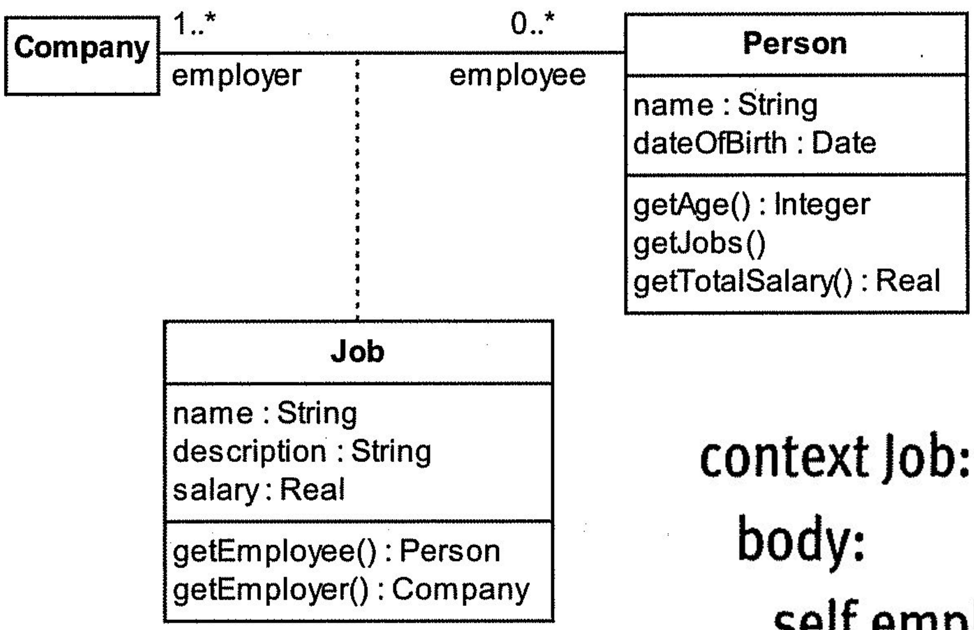


body: Navigating from one of the participants of an association class to the association class




```
context Person::getTotalSalary() : Real
  body:
    -- return the total salary for all Jobs
    self.Job.salary->sum()
```

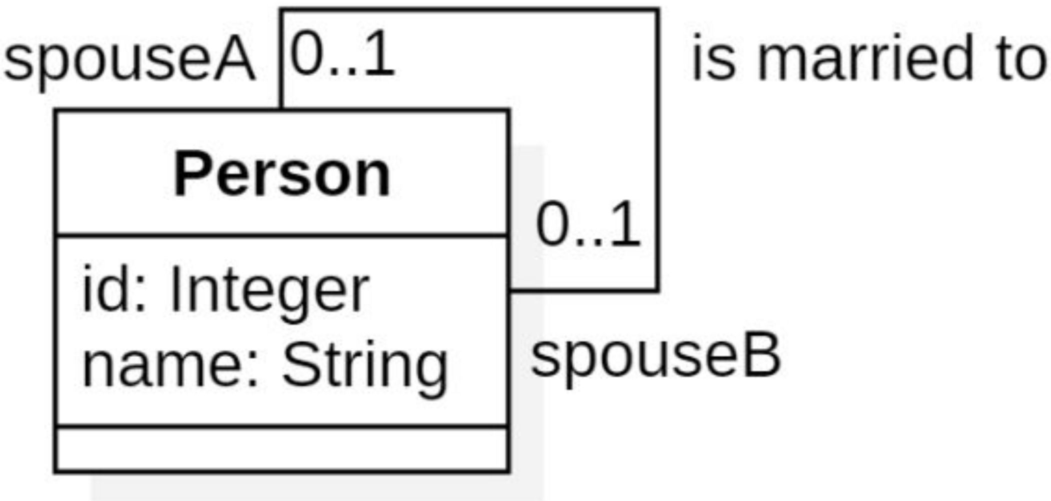
body: Navigation from the association class to the association participants

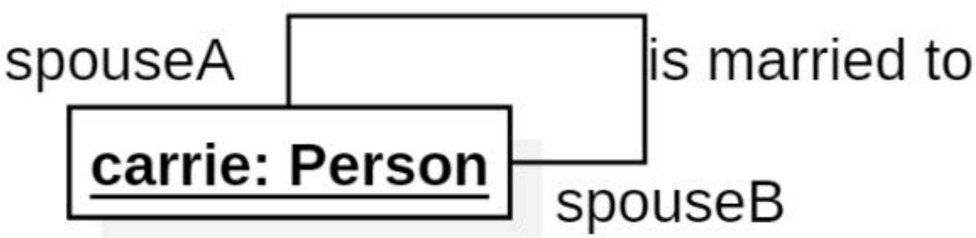


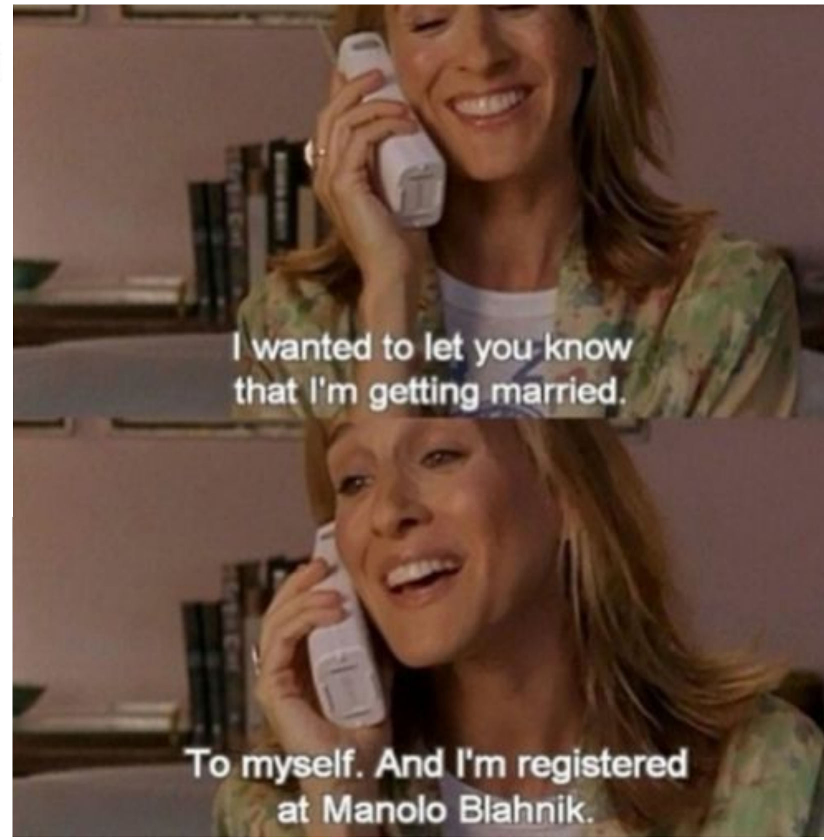

```
context Job::getEmployee() : Person
  body:
    self.employee
```

```
context Job::getEmployer() : Company
  body:
    self.employer
```

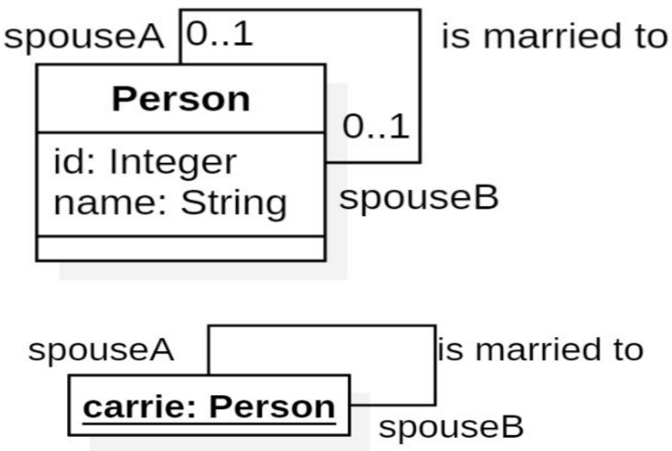
Back to Carrie...





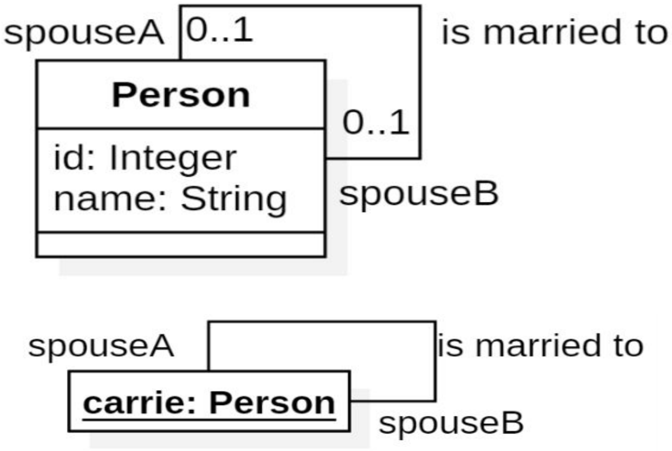


Let us define this in OCL, using the USE tool



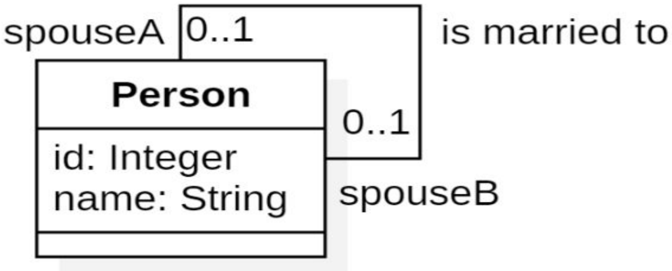

```
model Marriage
-- classes
class Person
attributes
  id: Integer
  name : String
end
```

Add the isMarriedTo association



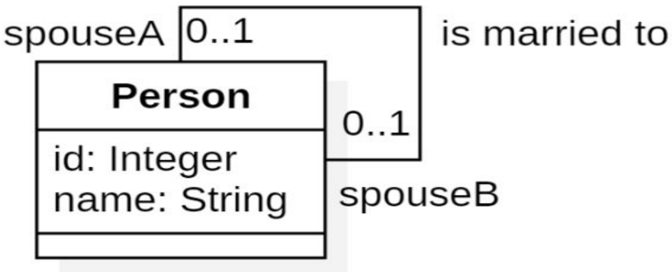

```
model Marriage
-- classes
class Person
attributes
  id: Integer
  name : String
end
--associations
association isMarriedTo between
  Person [0..1] role spouseA
  Person [0..1] role spouseB
end
```

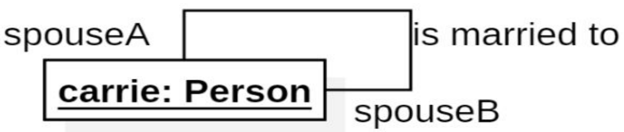
How can we make it so that Carrie marries herself?




```
model Marriage
-- classes
class Person
attributes
  id: Integer
  name : String
end
--associations
association isMarriedTo between
  Person [0..1] role spouseA
  Person [0..1] role spouseB
end
```

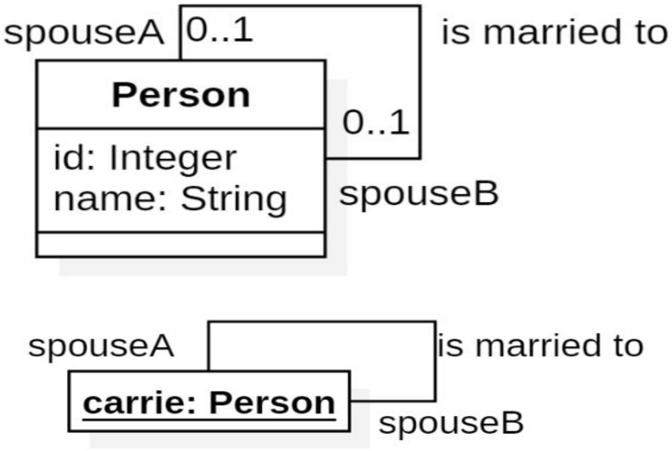
We can make selfMarriageMandatory





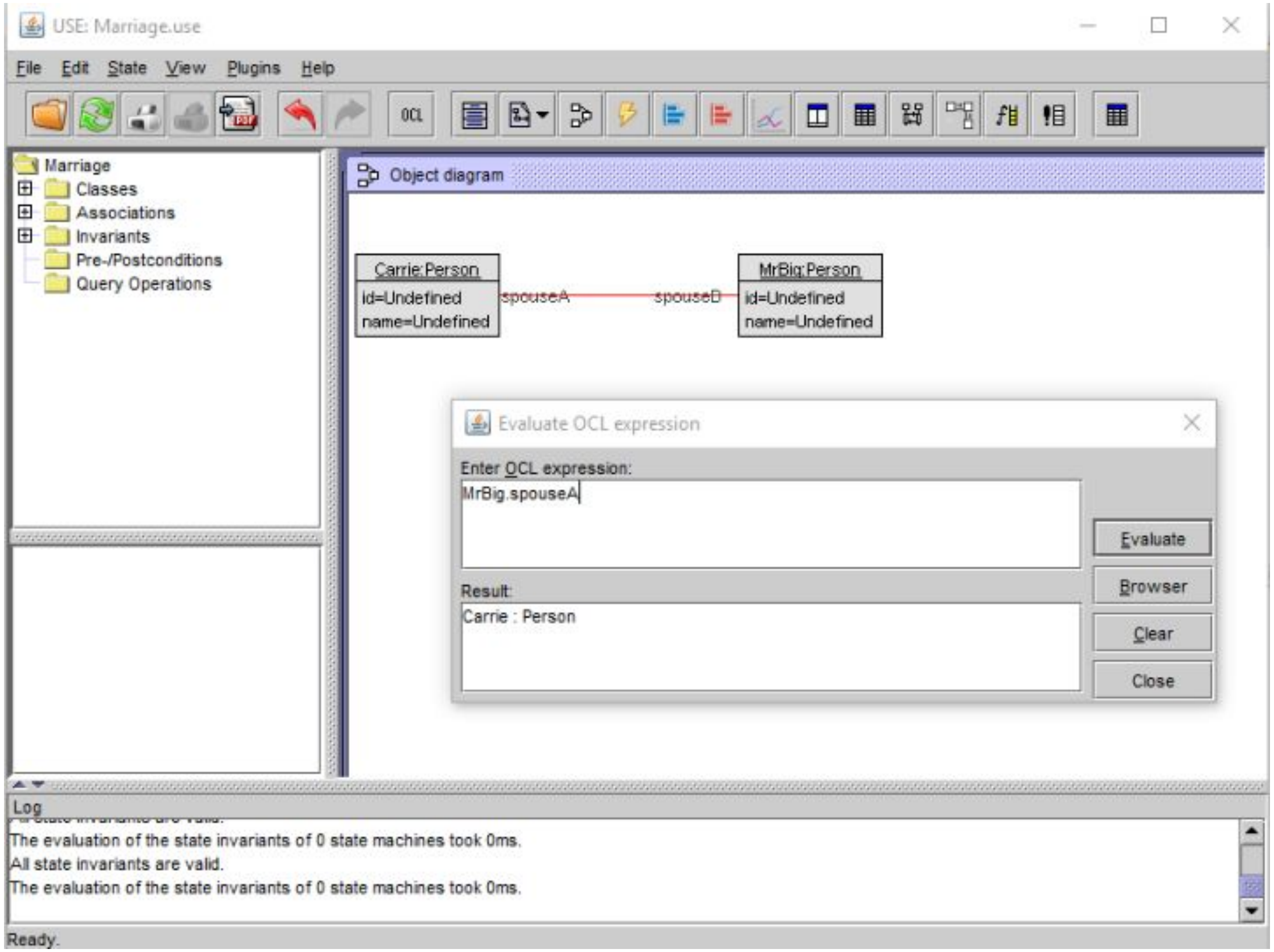

```
model Marriage
-- classes
class Person
attributes
  id: Integer
  name : String
end
--associations
association isMarriedTo between
  Person [0..1] role spouseA
  Person [0..1] role spouseB
end
constraints
-- selfMarriage is mandatory
context Person
  inv selfMarriageMandatory:
    self = spouseA and self = spouseB
```

We can also make it illegal




```
model Marriage
-- classes
class Person
attributes
  id: Integer
  name : String
end
--associations
association isMarriedTo between
  Person [0..1] role spouseA
  Person [0..1] role spouseB
end
constraints
-- selfMarriage is illegal
context Person
  inv selfMarriageIllegal:
    self = spouseA and self <> spouseB
```

Checking invariants with USE



Bibliography

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”, Second Edition, Addison-Wesley 2006

- Chapter 25