

Modeling and Verification

Advanced Modeling Concepts



Luis Pedro

Courses Plan

- Object Constraint Language
 - 15th May - room 259
- Meta Modeling: concepts
 - 18th May - room 259
- Meta Modeling: tools and application
 - 22nd May - room 259



Modeling and Verification

Introduction to the Object Constraint Language



Luis Pedro

OCL Course Outline

- OCL Basis
- OCL Motivations
- OCL Types
- OCL Expressions
- Advanced topics
 - More about Types and Operations
 - Collections





OCL Basis

OCL Basis

- Object Constraint Language
- Standard “add-on” to UML
- OCL Expressions relies on types defined in the diagrams
 - Classes, Interfaces, etc
- OCL allows to define **queries, reference values, state conditions** and **business rules**



OCL Basis

- Expression can be written in a clear and unambiguous manner
- Uses **pre-** and **post-** conditions
 - Pre-condition is a **restriction** that **must be true by the time of operation execution**
 - Post-condition is a **restriction** that **must be true** at the moment that an **operation ended its execution**



OCL Basis

- Mathematically based language
 - No mathematical symbols are used
 - Based on mathematical set theory and predicate logic



OCL Basis

- Query and Constraint Language
 - A **Constraint** is a restriction on one or more values of (part of) an object-oriented model or system
 - Can be used to write not only constraints but any expression on the elements of the diagram
 - OCL expression can indicate any value or collection of values in a system - like SQL



OCL Basis

- Strongly Typed Language
 - It is possible to check an OCL expression without having to produce an executable version of the model
 - Expressions can be check during modeling process
 - Errors in the model can be removed at an early stage of Software Development Process



OCL Basis

- Declarative Language
 - An expression simply stresses **what** should be done, but not **how**
 - Evaluating an OCL expression **does not change** the state of the system





OCL Motivations

OCL Motivations

- Information covered by UML diagrams tend to be incomplete, informal and imprecise
 - Many of the flaws in the model are caused by limitations of the diagrams used
 - The preciseness of the diagrams can be achieved by using OCL expression associated with them



OCL Motivations

- Expressions can not be interpreted differently by different people
- Expressions can be checked by automated tools
 - To ensure that they are correct and consistent with other elements of the model
- Code generation becomes much more powerful



OCL Motivations

- Combination of UML and OCL offers a great possibility to make precise models of the system under specification
 - Without OCL the system might be severely under-specified
 - Without UML, OCL expressions would refer to non existing model elements





OCL Types

OCL Types

- OCL provides a hierarchical classification for its data types
- All the information manipulated in OCL must be of one of the following variable groups:
 - Basic Types
 - Collections
 - Special Types
 - UML Model Types

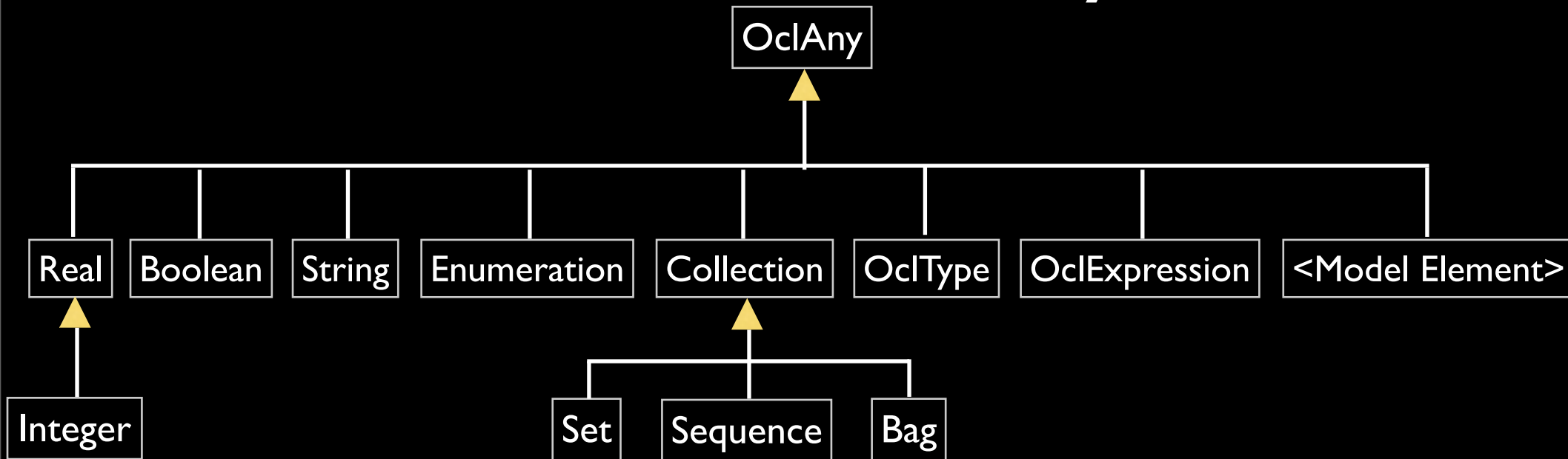


OCL Types

- Basic Types
 - Integer, Real, Boolean and String
- Collections
 - Collection, Set, Sequence, Bag
- Special Types
 - OclAny, OclType and OclExpression
- UML Models
 - All types defined in the UML model that defines the context can also be used



Class Hierarchy



- *OCLAny* is the superclass for all data types
- Each data type defines a set of operations that can be applied to them or to their children



Java Mapping

- Different maps might be defined
- After mapping the types, the OCL Operations mapping must also be defined

| OCL Type | Java Type |
|----------|-----------|
| Integer | int |
| Real | float |
| String | String |
| Boolean | boolean |
| OclType | Class |
| OclAny | Object |



From UML

- Possible UML artifacts that can be used in OCL expressions
 - *Classifiers*: Attributes that are used to create Data Types. **Classes** of object are the most frequent occurrence of classifiers in OCL
 - *Attributes*: the structural properties that characterizes a classifier
 - *Associations*: an association “connects” one or more classifiers
 - *Operations*: all operations and methods can be used in OCL



Collections

- *Collection* is the abstract super-type of all OCL Collections.
- All object of a collection is called *Element*
 - If we have Associations derived from the UML models with a cardinality *, this represented as a collection



Collections

- Set
 - Mathematic set as we know it. Contains elements that can not be duplicated
- Bag
 - A collection of elements that can appear duplicated
 - There is no order defined
- Sequence
 - Same as *Bag* but there is a certain order associated to the elements



Collections

It is also possible to define Ordered Sets

- Set
 - Mathematic set as we know it. Contains elements that can not be duplicated
- Bag
 - A collection of elements that can appear duplicated
 - There is no order defined
- Sequence
 - Same as *Bag* but there is a certain order associated to the elements



Java Mapping

- Collection types mapping

| OCaml Type | Java Type |
|------------|-----------|
| Set | Set |
| Sequence | List |
| Bag | List |
| OrderedSet | List |





OCL Expressions

OCL Expressions

- All OCL expressions are declarative
- They present **what** the restriction represents and not **how** this restriction is implemented
- The evaluation of an expression always returns a boolean value
- Are used to define invariants in the model's components
- Used to define the **pre** and **post** conditions



OCL Expressions

- Can be used to perform **queries** to the system model
 - The result of OCL query expressions does not return a boolean value
 - Returns values of an OCL type that depends on the type of the executed query



OCL Expressions

- Types of expressions
 - Expressions that represent invariant conditions in classes or objects
 - Expressions that represent pre-conditions applied to a class or object
 - Expressions that define the operations associated to the post-conditions for a class or object



OCL Expressions

- Context of an Expression
 - OCL expression demand that restrictions are connected to a context in a model.
 - The context of an expression can be an object or an operation associated to an object
 - A context is represented by the reserved word: **context**



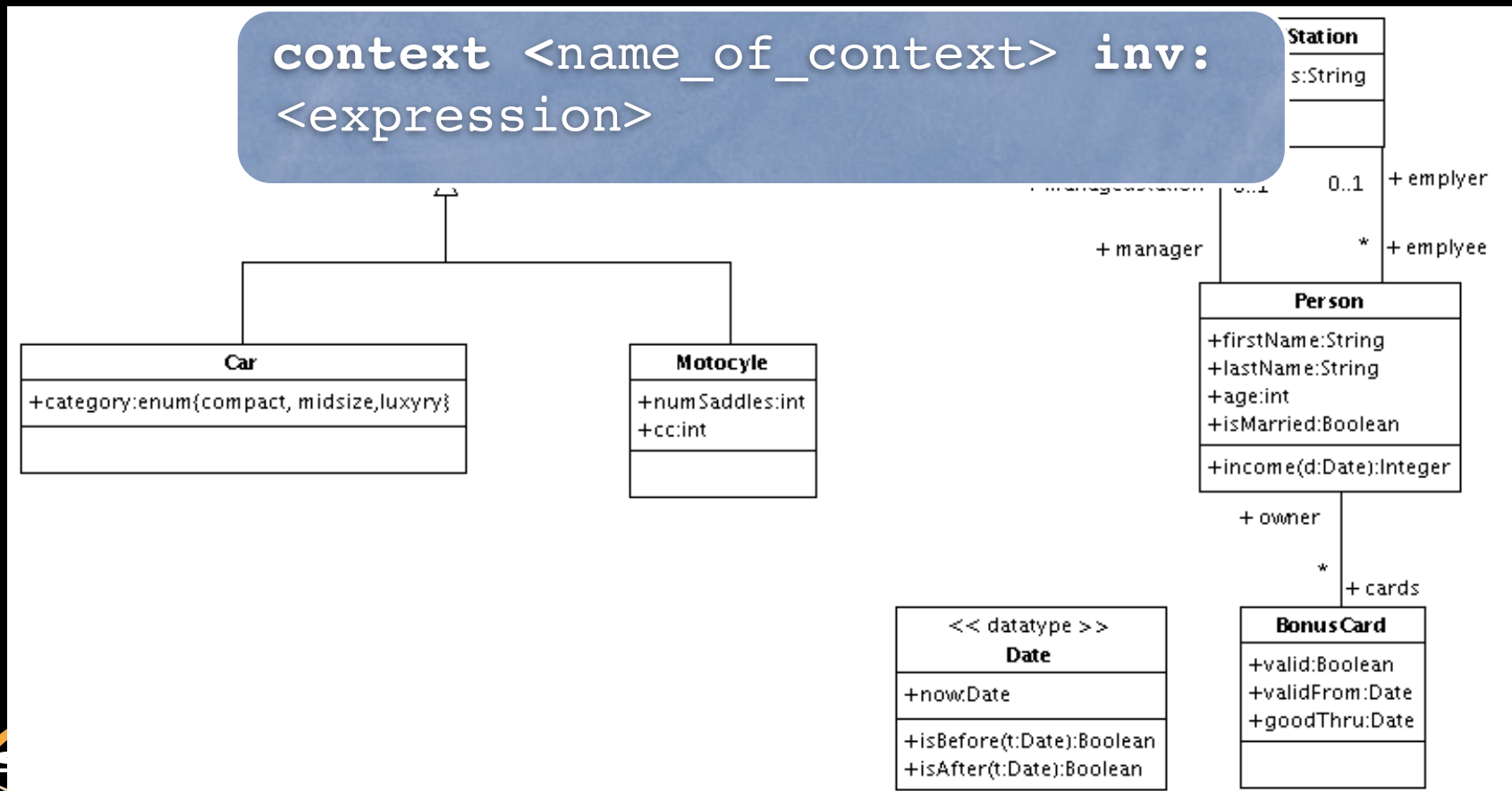
OCL Expressions

- The **Self** expression
 - Is used to make a reference to an instance of the context
- **Invariants**
 - Invariants are conditions that the objects must respect during their **complete life-time**
 - To indicate that an expression is an invariant the reserved word **inv** is used



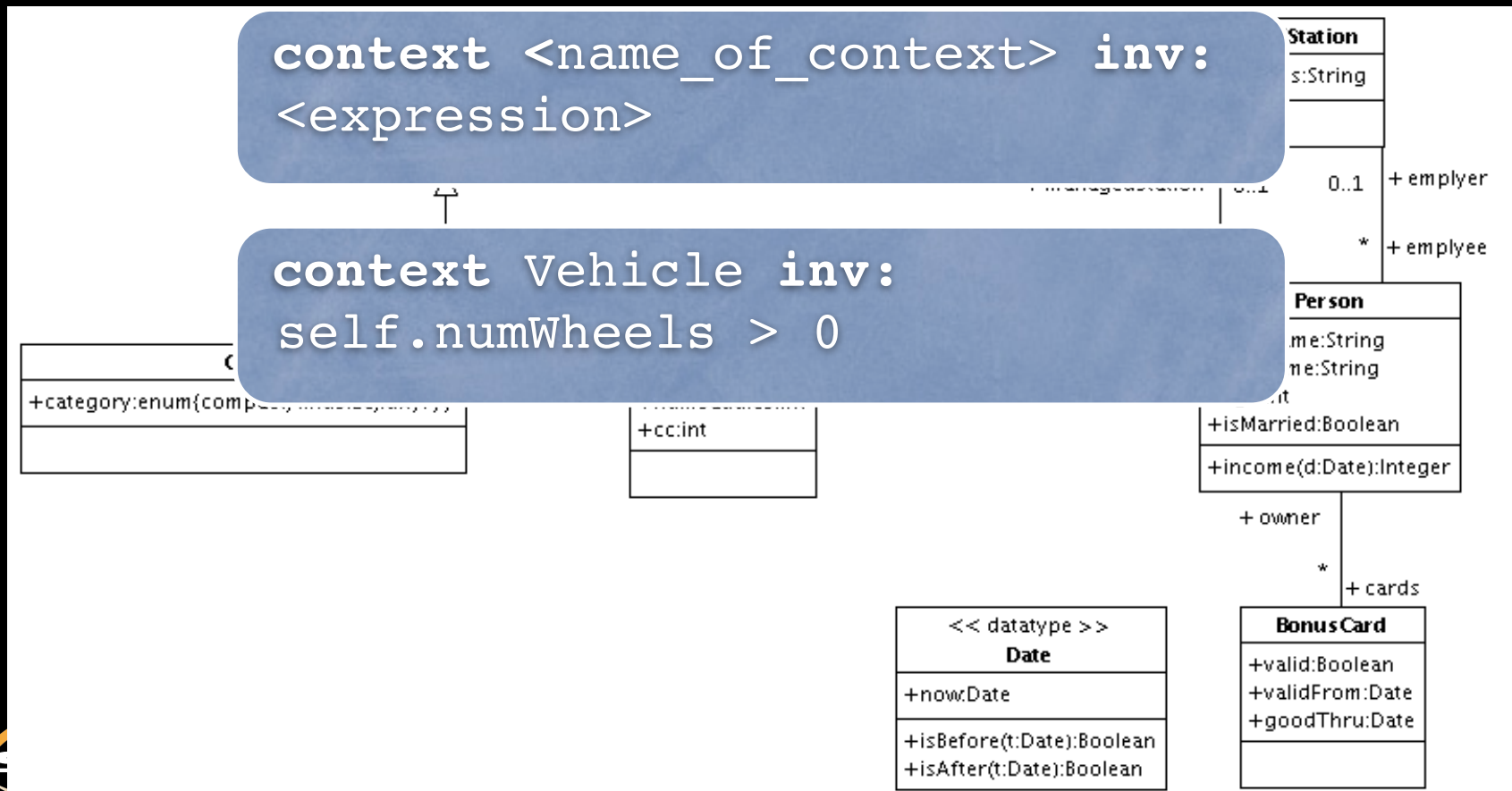
OCL Expressions

- Context, Self and Invariant example



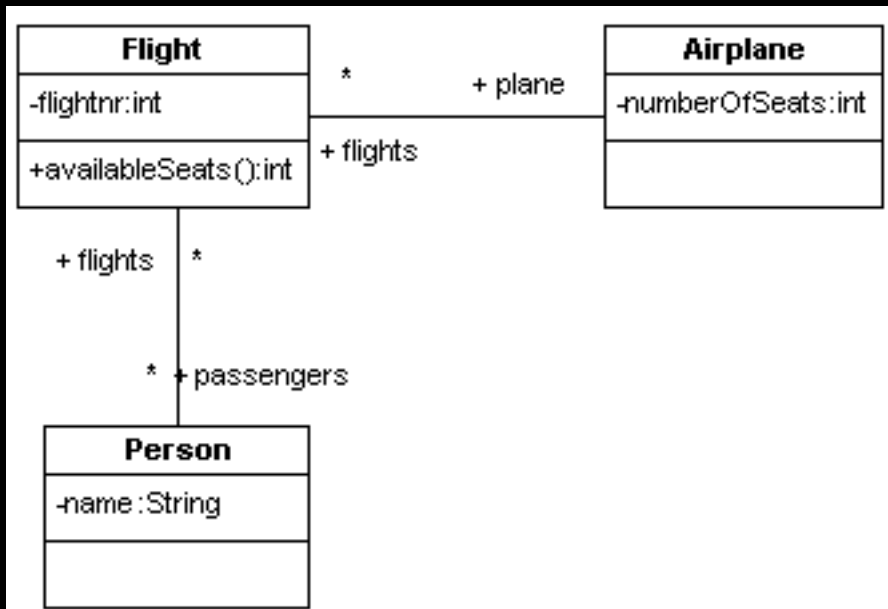
OCL Expressions

- Context, Self and Invariant example



OCL Expressions

- Another Example for *inv*

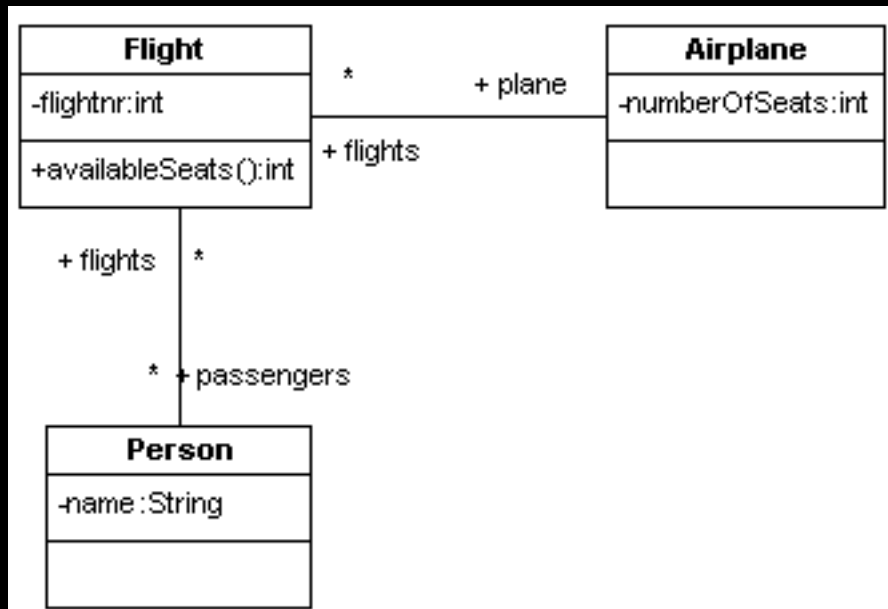


- Association between **Flight** and **Person** indicates that a certain group of persons are the passengers on a flight
 - Multiplicity many ($0..*$) on the side of **Person** class
 - Means that the **number of passengers is unlimited**
- In a realistic system, the number of passengers will be limited to the number of seats available on the airplane that is associated with the flight



OCL Expressions

- Another Example for *inv*



Context: Flight
inv: passengers->size()
 <= plane.numberOfSeats

- Association between **Flight** and **Person** indicates that a certain group of persons are the passengers on a flight
 - Multiplicity many (0..*) on the side of **Person** class
 - Means that the **number of passengers is unlimited**
 - In a realistic system, the number of passengers will be limited to the number of seats available on the airplane that is associated with the flight



OCL Expressions

- Initial values
 - The initial value of an attribute or association role can be given by an OCL expression



OCL Expressions

- Initial values
 - The initial value of an attribute or association role can be given by an OCL expression

```
context BonusCard::valid : Boolean
init: true
```



OCL Expressions

- Pre-Conditions
 - The conditions that reflect state in which the system **must** be before the execution of the operations in the objects
- In OCL it is used the reserved word **pre** after the context declaration, followed by “:”

```
context <context_name> pre:  
<expression>
```



OCL Expressions

- Pre-Conditions
 - The conditions that reflect state in which the system **must** be before the execution of the operations in the objects
- In OCL it is used the reserved word **pre** after the context declaration, followed by “:”

```
context <context_name> pre:  
<expression>
```

```
context Person:income(d: Date) pre:  
d > 01/01/1950
```



OCL Expressions

- Post-conditions
 - Expressions that define the system's state prior to the execution of operations in the objects
- In OCL it is used the reserved word **post** after the context declaration, followed by “:”

```
context <context_name> post:  
<expression>
```



OCL Expressions

- Post-conditions
 - Expressions that define the system's state prior to the execution of operations in the objects
- In OCL it is used the reserved word **post** after the context declaration, followed by “:”

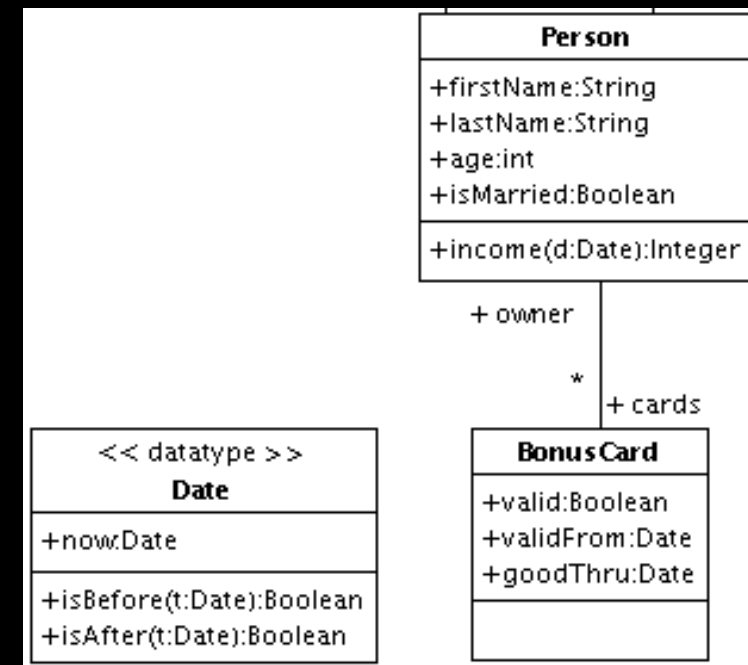
```
context <context_name> post:  
<expression>
```

```
context Person::income(d: Date) post:  
if d > "01/01/2001" then  
    result = 3000  
else  
    result = 2000
```



Let ... In

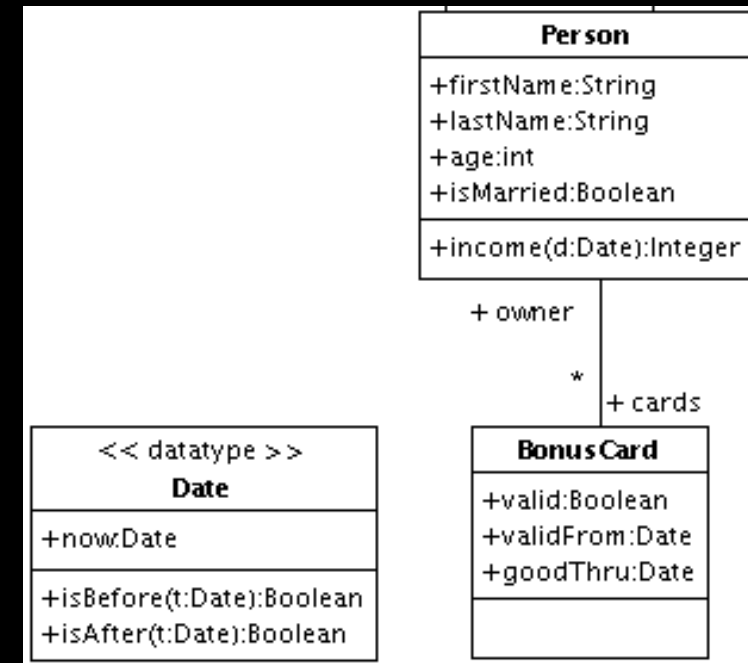
- Sometimes large expressions in which a sub-expression is used more than once are written. The *let* expression allows to define a variable that can be used instead of the sub-expression



Let ... In

- Sometimes large expressions in which a sub-expression is used more than once are written. The *let* expression allows to define a variable that can be used instead of the sub-expression

```
context BonusCard
inv: let correctDate : Boolean =
    self.validFrom.isBefore(Date::now) and
    self.goodThru.isAfter(Date::now)
    in
    if valid then
        correctDate = false
    else
        correctDate = true
    endif
```





Types and Operators

Boolean Type

| Operation | Notation | Result Type |
|--------------|------------------------|-------------|
| or | $a \text{ or } b$ | Boolean |
| and | $a \text{ and } b$ | Boolean |
| exclusive or | $a \text{ xor } b$ | Boolean |
| negation | $\text{not } a$ | Boolean |
| equals | $a = b$ | Boolean |
| not equals | $a \langle \rangle b$ | Boolean |
| implies | $a \text{ implies } b$ | Boolean |

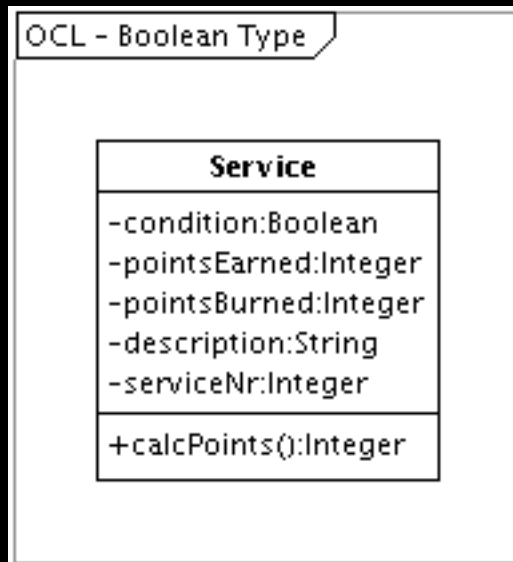


Boolean Type

- Operation defined on *Boolean* types return always a *Boolean*
 - *Boolean* can be one of two values: *true* or *false*
- The *implies* operation is not usually present in programming languages
 - States that the result of the total expression is true if it is the case that when the first *Boolean* operand is *true*, the second *Boolean* operand is also *true*



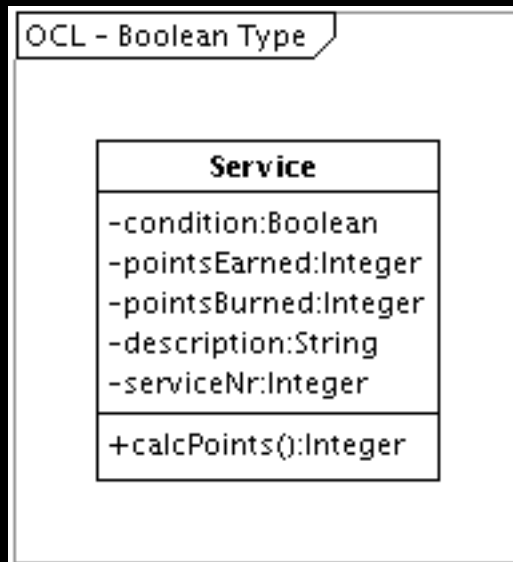
Boolean Type: Example



- The result of the expression is true if for every service it can be said that when it offers bonus points it never burns points
- A customer can't earn points when using a service that is bought with bonus points:



Boolean Type: Example



- The result of the expression is true if for every service it can be said that when it offers bonus points it never burns points
- A customer can't earn points when using a service that is bought with bonus points:

```
context Service
inv: self.pointsEarned > 0 implies not (self.pointsBurned = 0)
```



Boolean Type: Example 2

- If ... then ... else

```
if <boolean OCL expression>  
then <OCL expression>  
else <OCL expression>  
endif
```

- The result is either the OCL expression in the *then* clause or the OCL expression in the *else* clause
 - Depending on the result of the boolean expression in the *if* clause
- In OCL the *else* clause can not be omitted



Integer and Real Types

| Operation | Notation | Result Type |
|---------------|------------|--------------|
| equals | $a = b$ | Boolean |
| not equals | $a \neq b$ | Boolean |
| less | $a < b$ | Boolean |
| more | $a > b$ | Boolean |
| less or equal | $a \leq b$ | Boolean |
| more or equal | $a \geq b$ | Boolean |
| plus | $a + b$ | Integer/Real |
| minus | $a - b$ | Integer/Real |

| Operation | Notation | Result Type |
|------------------|--------------------|--------------|
| multiplication | $a * b$ | Integer/Real |
| division | a / b | Real |
| modulus | $a.\text{mod}(b)$ | Integer |
| integer division | $a.\text{div}(b)$ | Integer |
| absolute value | $a.\text{abs}()$ | Integer/Real |
| maximum | $a.\text{max}(b)$ | Integer/Real |
| minimum | $a.\text{min}(b)$ | Integer/Real |
| round | $a.\text{round}()$ | Integer |
| floor | $a.\text{floor}()$ | Integer |



Integer and Real Types

- Being OCL a modeling language, there are no restriction on the integer values
 - Maximum value does not exist
- Same for Real numbers
- Integer is a sub type of Real



Integer and Real Types

- Example of some operations

```
-1.abs()  
(3.4).abs()  
2.6.round()  
3.7.floor()  
2.1.floor() / 2  
12 > 23  
-3.2.floor() + 3  
13.mod(2)  
13.div(2)
```



Integer and Real Types

- Example of some operations

```
-1.abs()  
(3.4).abs()  
2.6.round()  
3.7.floor()  
2.1.floor() / 2  
12 > 23  
-3.2.floor() + 3  
13.mod(2)  
13.div(2)
```

```
1  
3.4  
3  
3  
1  
false  
0  
1  
2
```



String Type

| Operation | Notation | Result Type |
|---------------|------------------------------------|-------------|
| concatenation | <code>s.concat(s)</code> | String |
| size | <code>s.size()</code> | Integer |
| to lower case | <code>s.toLowerCase()</code> | String |
| to upper case | <code>s.toUpperCase()</code> | String |
| substring | <code>s.substring(int, int)</code> | String |
| equals | <code>s1 = s2</code> | Boolean |
| not equals | <code>s1 <> s2</code> | Boolean |



String Type

- Strings are sequences of characters
- Literal strings are written with enclosing single quotes
 - 'Modeling' or 'Modeling and Verification'
- Examples

```
'Model'.size()  
( 'Model' = 'Verification' )  
'Model'.concat(' and' )  
'Model'.toUpper  
'Model'.toLower()  
'Modeling and Verification'.substring(10,12)
```



String Type

- Strings are sequences of characters
- Literal strings are written with enclosing single quotes
 - 'Modeling' or 'Modeling and Verification'
- Examples

```
'Model'.size()  
( 'Model' = 'Verification' )  
'Model'.concat(' and')  
'Model'.toUpper  
'Model'.toLower()  
'Modeling and Verification'.substring(10,12)
```

```
5  
false  
'Model and'  
'MODEL'  
'model'  
'and'
```



Precedence for OCL Operations

- *Pathname* → ::
- *Time expression* → @pre
- *dot, arrow and message operations* → “.”, “->”, “^”, “^^”
- *Unary operations* → **not** and “-”
- *Multiplication and Division* → * and /
- *Addition and Subtraction* → + and -
- *Relational Operations* → <, >, <=, >=, =, <>
- *Logical Operations* → **and**, **or** and **xor**
- *Logical implies* → **implies**



Collection Types

- Collection Constants

```
Set{1, 2, 7, 95}  
Set{'apple', 'mango', 'orange'}  
OrderedSet{'apple', 'mango', 'orange', 'pear'}  
Sequence {1, 3, 44, 2, 1}  
Bag{1, 3, 4, 3, 5} Sequence{1..()}
```

- The following constructs are equivalent

```
Sequence{1..(2+3)}  
Sequence{1..5}  
Sequence{1, 2, 3, 4, 5}
```



Collection Operations

- Various standard operations are defined on the collection types
- All operations in collections are denoted in OCL using an arrow
 - The operation following the arrow is applied to the collection before the arrow
- All collection types are defined as value types
 - the value of an instance cannot be changed



Collection Types

- Collections of Collections

```
Set{Set{1, 2}, Set{7, 95}, Set{3,5}}
```

- In OCL, collections are automatically flattened
 - The result of the previous collection of collection being:

```
Set{1, 2, 7, 95, 3, 5}
```



Operations on Collections

| Operation | Description |
|---------------------------------|---|
| count (obj) | The number of occurrences of the object in the collection |
| excludes (obj) | True if the object is not an element of the collection |
| excludesAll (collection) | True if all elements of the parameter collection are not present in the current collection |
| includes (object) | True if the object is an element of the collection |
| includesAll (collection) | True if all elements of the parameter collection are present in the current collection |
| isEmpty () | True if the collection contains no elements |
| notEmpty () | True if the collection contains at least one element |
| size () | The number of elements of the collection |
| sum () | The addition of all elements in the collection. The elements must be type supporting addition |

Collection Operations

- The *equals* and *notEquals* Operations
 - the *equals* operator evaluates to *true* if all elements in two collections are the same
 - For *sets* this means that all elements present in the first set must be present in the second and vice-versa
 - For *ordered sets* in addition, the order in which the elements appear must also be the same
 - For two *bags* to be equal, not only must all elements be present in both, but the number of times of an element must be the same



Collection Operations

- Many other operations have been defined for collection (see references for details)
 - Some of these operations might be different meanings for the different types of collections
 - Examples in next slide
 - *including* and *excluding* operations
 - *flatten operation*



Collection Operations

- *including*
 - results in a new collection with a new element added to the original collection
 - No problem for *Bag*
 - For *Set* and *Ordered Set* the element is added only if it is not already present
 - Otherwise, the result is a collection equal to the previous one



Collection Operations

- *Excluding*
 - Results in a new collection with an element removed from the original one
 - From a *Set* or *Ordered Set*, it removes only one element
 - From *Bag* and *Sequence* removes all occurrences of a given object



Collection Operations

- *Flatten*
 - Modifies a collection of collection into a collection of single objects
 - When applied to a *Bag* the result is also a *Bag*
 - When an object is in more than one sub-collection it will be included in the resulting *Bag* more than once
 - When applied to a *Set* the result is also a *Set*
 - If an object is in more than one sub-collection it is included in the result only once



Collection Operations

- *Flatten in a Set*

`Set{Set{1, 2}, Set{2, 3}, Set{4, 5, 6}}`

- The result of the Flatten Operation

`Set{1, 2, 3, 4, 5, 6}`

- *Flatten in a Bag*

`Bag{Set{1, 2}, Set{1, 2}, Set{4, 5, 6}}`

- The result of the Flatten Operation

`Bag{1, 1, 2, 2, 4, 5, 6}`



Collection Operations

- More operations available
 - *union*
 - combines two collections in one
 - Set with Set results in a Set
 - Set with Bag results in a Bag
 - Sequence or ordered Set can only be combine with each other
 - *intersection*
 - results in another collection containing the elements of both collections



Collection Operations

- More operations available
 - *minus*
 - results in a new *Set* containing all elements in the *Set* on which the operation is called

```
Set(1, 4, 7, 10) - Set(4,7)  
OrderedSet{12, 9, 6, 3} - Set{1, 3, 2}
```

- *simetricDifference*
 - results in a *Set* containing all elements in the *Set* on which the operation is called or in the parameter *Set* but not in both

```
Set(1,4,7,10).symetricDifference(Set{4,5,7})
```



Collection Operations

- More operations available
 - *minus*
 - results in a new *Set* containing all elements in the *Set* on which the operation is called

```
Set(1, 4, 7, 10) - Set(4,7)  
OrderedSet{12, 9, 6, 3} - Set{1, 3, 2}
```

```
Set{1, 10}  
OrderedSet{12, 9, 6}
```

- *simetricDifference*
 - results in a *Set* containing all elements in the *Set* on which the operation is called or in the parameter *Set* but not in both

```
Set(1,4,7,10).symetricDifference(Set{4,5,7})
```



Collection Operations

- More operations available
 - *minus*
 - results in a new *Set* containing all elements in the *Set* on which the operation is called

```
Set(1, 4, 7, 10) - Set(4,7)  
OrderedSet{12, 9, 6, 3} - Set{1, 3, 2}
```

```
Set{1, 10}  
OrderedSet{12, 9, 6}
```

- *simetricDifference*
 - results in a *Set* containing all elements in the *Set* on which the operation is called or in the parameter *Set* but not in both

```
Set(1,4,7,10).symetricDifference(Set{4,5,7})
```

```
Set{1, 5, 10}
```



Loop Operations or Iterators

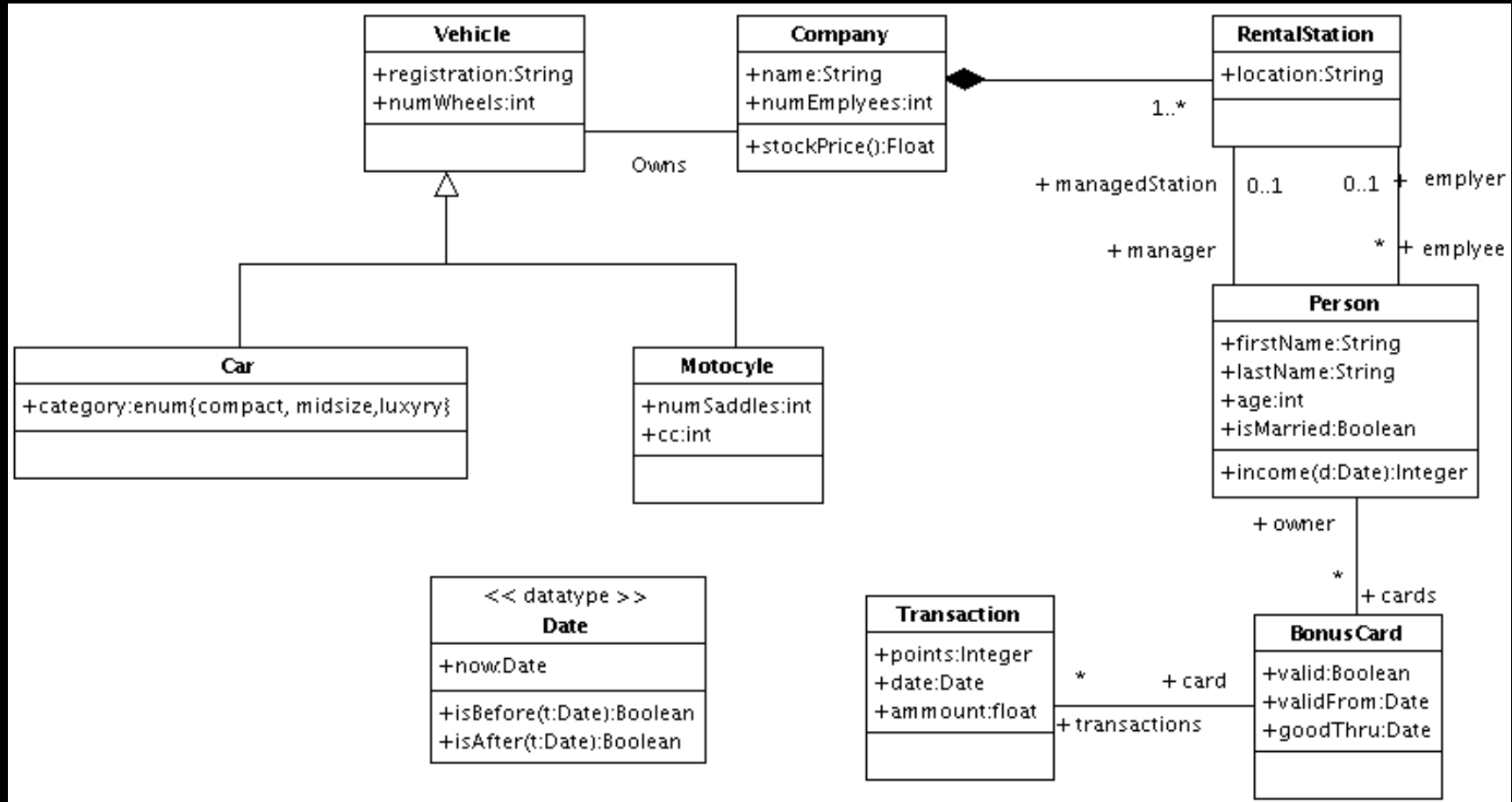
- A number of standard OCL operations allow to loop over the elements in a collection
- These operations take each element in the collection and evaluate an expression on it
- Every loop operation has an OCL expression as parameter
 - Body of the operation



Loop Operations

| Operation | Description |
|--------------------------------------|---|
| any (<i>expr</i>) | Random element of the source collection for which <i>expr</i> is true |
| collect (<i>expr</i>) | Collection of objects that result from evaluating <i>expr</i> for each element in the source collection |
| collectNested (<i>expr</i>) | Collection of collections that result from evaluating <i>expr</i> for each element in the source collection |
| exists (<i>expr</i>) | <i>True</i> if there is at least one element in the source collection for which <i>expr</i> is true |
| forAll (<i>expr</i>) | <i>True</i> if <i>expr</i> is true for all elements in the source collection |
| isUnique (<i>expr</i>) | <i>True</i> if <i>expr</i> has a unique value for all elements in the source collection |
| iterate (...) | Iterates over all elements in the source collection |
| one (<i>expr</i>) | <i>True</i> if there is exactly one element in the source collection for which <i>expr</i> is true |
| reject (<i>expr</i>) | Subcollection of the source collection containing all elements for which <i>expr</i> is <i>False</i> |
| select (<i>expr</i>) | Subcollection of the source collection containing all elements for which <i>expr</i> is <i>True</i> |
| sortedBy (<i>expr</i>) | Collection containing all elements of the source collection ordered by <i>expr</i> |

Select Operation



context BonusCard

inv: self.transactions->select(points > 100)->notEmpty()



Select Operation

```
context BonusCard
```

```
inv: self.transactions->select(points > 100)->notEmpty()
```

- **Operationally it means**

```
element = collection.firstElement();  
while (collection.notEmpty()) do  
  if ( <expression-with-element> )  
  then  
    result.add(element);  
  endif  
  element = collection.nextElement();  
endwhile  
return result;
```



References

- The Object Constraint Language Second Edition
 - *Jos Warner, Anneke Kleppe*
- OCL Specification
 - Object Management Group (<http://www.omg.org/docs/ptcl/05-06-06.pdf>)
- Understanding OCL
 - Daniel Henrique Alves Lima, Rafael Musial
- OCL
 - João Pascoal Faria

