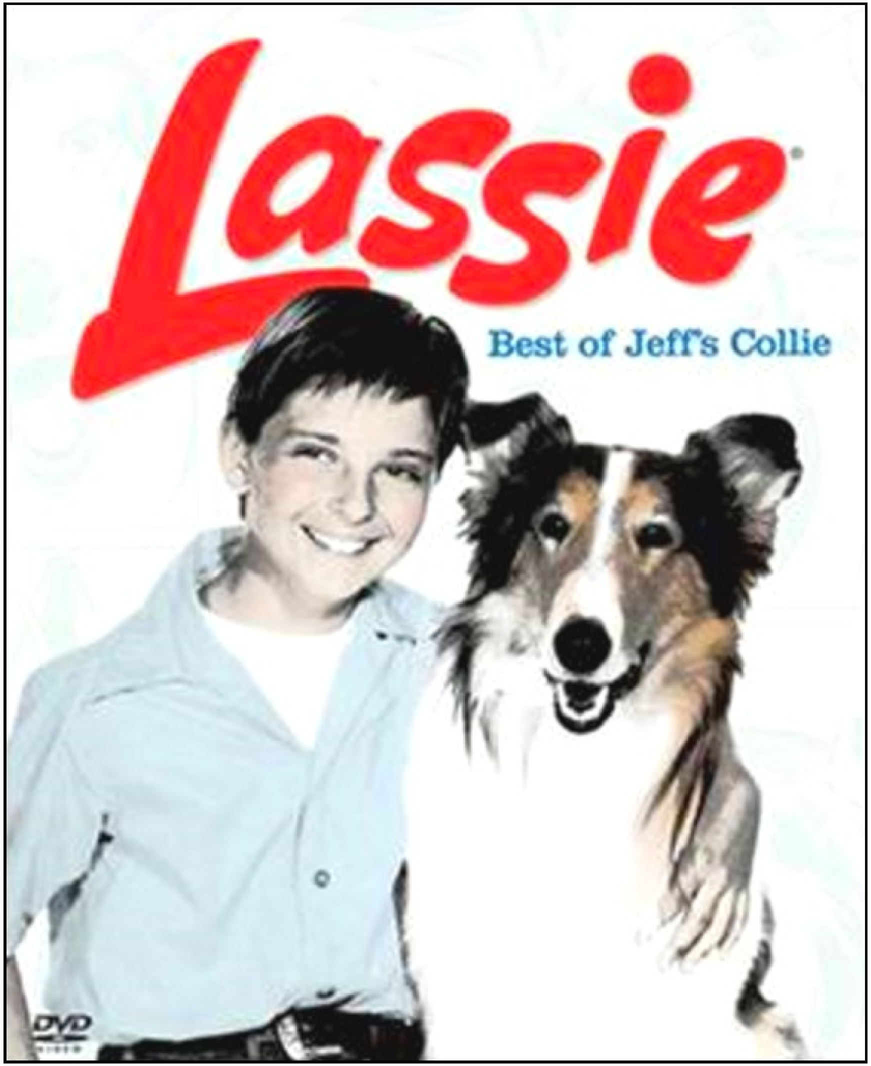
The OCL Datatypes Real, Integer, String, and Boolean together with their Operations

martin gogolla

University of Bremen

Summer semester 2020


Lassie
Best of Jeff's Collie
DVD

Possible questions regarding the DVD cover

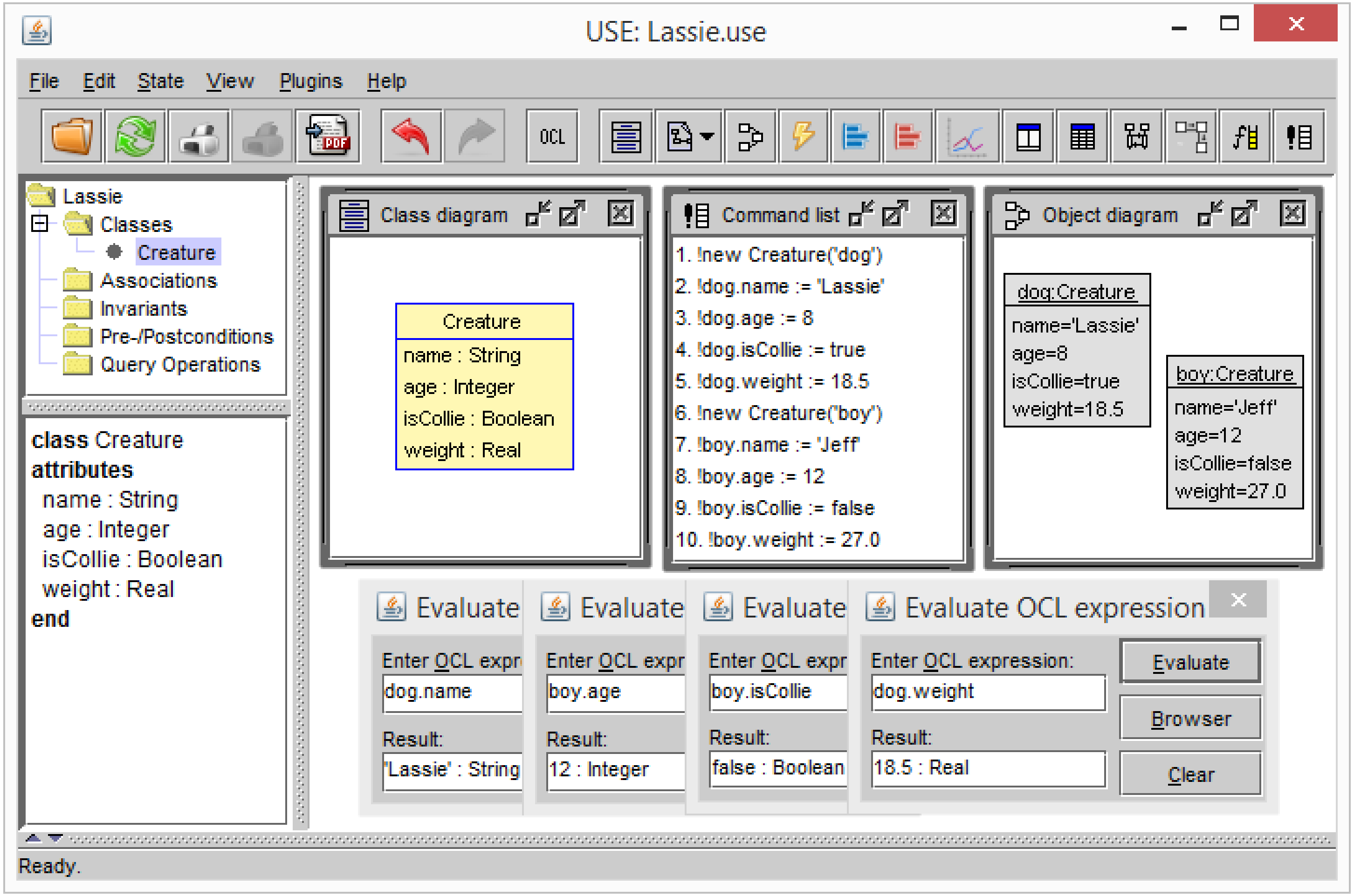
- What is the name of the dog?
- And the name of the boy?
- How old is the dog?
- And how old is the boy?
- The dog is a collie. Is that true or false?
- The boy is a collie. True or false?
- What is the weight of the dog?
- And the weight of the boy?
- Answers involve *different kind* of information

Construction of an object-oriented *UML model* with USE

- Representing elements of the DVD cover (of a snapshot, in general)
- Class diagram as a general frame for object diagrams (snapshots)
- Command list for building an object diagram
- Object diagram for representation of a snapshot
- OCL term evaluation for retrieving snapshot properties

Live: *USE demo*

- textual USE model file: model Lassie class Creature ... end
- Shell SOIL Actions: !new Creature ('lassie') OR !lassie.isCollie:=true
- Shell SOIL Queries: ?lassie.name
- Shell commands on multiple lines:
```
\
?lassie.isCollie and
   jeff.isCollie
.
```
- Model browser, Class diagram, Command list, Object diagram, OCL expression evaluation, ...



USE: Lassie.use

```
class Creature
attributes
  name : String
  age : Integer
  isCollie : Boolean
  weight : Real
end
```

Creature
name : String
age : Integer
isCollie : Boolean
weight : Real

```
1. !new Creature('dog')
2. !dog.name := 'Lassie'
3. !dog.age := 8
4. !dog.isCollie := true
5. !dog.weight := 18.5
6. !new Creature('boy')
7. !boy.name := 'Jeff'
8. !boy.age := 12
9. !boy.isCollie := false
10. !boy.weight := 27.0
```

dog:Creature
name='Lassie'
age=8
isCollie=true
weight=18.5

boy:Creature
name='Jeff'
age=12
isCollie=false
weight=27.0

Enter OCL expression:	Result:
dog.name	'Lassie' : String
boy.age	12 : Integer
boy.isCollie	false : Boolean
dog.weight	18.5 : Real

Commmand list for generating the object diagram

```
!new Creature('dog')  !new Creature('boy')
!dog.name:='Lassie'   !boy.name:='Jeff'      String values
!dog.age:=8           !boy.age:=12           Integer
!dog.isCollie:=true   !boy.isCollie:=false   Boolean
!dog.weight:=18.5     !boy.weight:=27.0      Real
```

Simple example terms querying for object attribute values

```
?dog.name
?boy.age
?boy.isCollie
?dog.weight
```

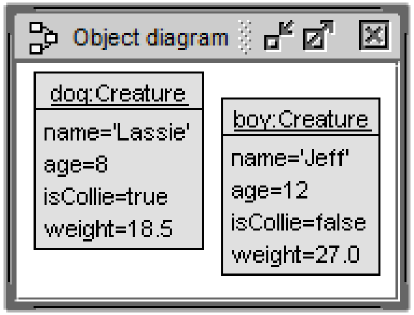
Example terms querying for values of combined properties

```
?dog.name.concat(' and ').concat(boy.name).concat(' are friends!')
?dog.weight + boy.weight
?(boy.age + dog.age) div 2
?boy.isCollie and dog.isCollie
```

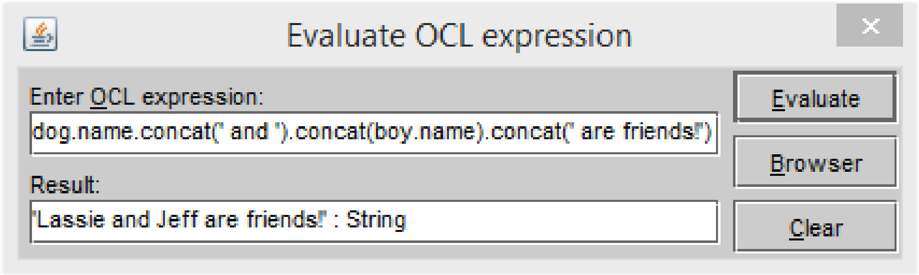
First general observations on terms

```
- Constants: 'Lassie', 8, true, 18.5
- Operations: concatenation of strings 'concat'
-             addition '+' on reals
-             division 'div' on integers
-             logical conjunction 'and' on booleans
```

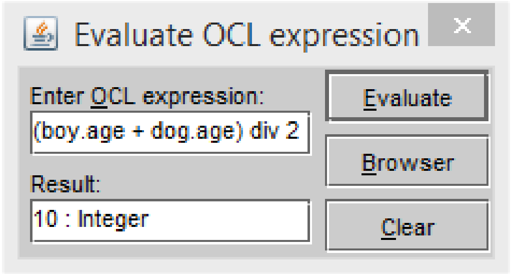
OCL terms for *combined properties*



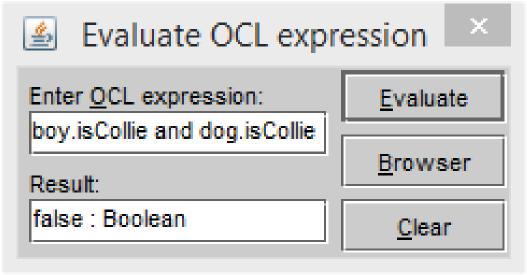
```
dog:Creature
name='Lassie'
age=8
isCollie=true
weight=18.5

boy:Creature
name='Jeff'
age=12
isCollie=false
weight=27.0
```



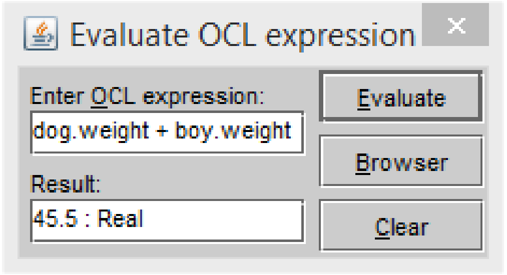
Enter OCL expression:
```
dog.name.concat(' and ').concat(boy.name).concat(' are friends!')
```
Result:
```
'Lassie and Jeff are friends!' : String
```



Enter OCL expression:
```
(boy.age + dog.age) div 2
```
Result:
```
10 : Integer
```



Enter OCL expression:
```
boy.isCollie and dog.isCollie
```
Result:
```
false : Boolean
```



Enter OCL expression:
```
dog.weight + boy.weight
```
Result:
```
45.5 : Real
```

Example for USE shell protocol

```
use> ?dog.name
-> 'Lassie' : String
use> ?boy.age
-> 12 : Integer
use> ?boy.isCollie
-> false : Boolean
use> ?dog.weight
-> 18.5 : Real

use> ?dog.name.concat(' and ').concat(boy.name).concat(' are friends!')
-> 'Lassie and Jeff are friends!' : String
use> ?dog.weight + boy.weight
-> 45.5 : Real
use> ?(boy.age + dog.age) div 2
-> 10 : Integer
use> ?boy.isCollie and dog.isCollie
-> false : Boolean

use> ?dog.name.oclIsTypeOf(String)
-> true : Boolean
use> ?dog.age.oclIsTypeOf(Integer)
-> true : Boolean
use> ?dog.isCollie.oclIsTypeOf(Boolean)
-> true : Boolean
use> ?dog.weight.oclIsTypeOf(Real)
-> true : Boolean
```

Example terms checking the type of attributes

```
dog.name.oclIsTypeOf(String)
dog.age.oclIsTypeOf(Integer)
dog.isCollie.oclIsTypeOf(Boolean)
dog.weight.oclIsTypeOf(Real)
```

Datatypes and values

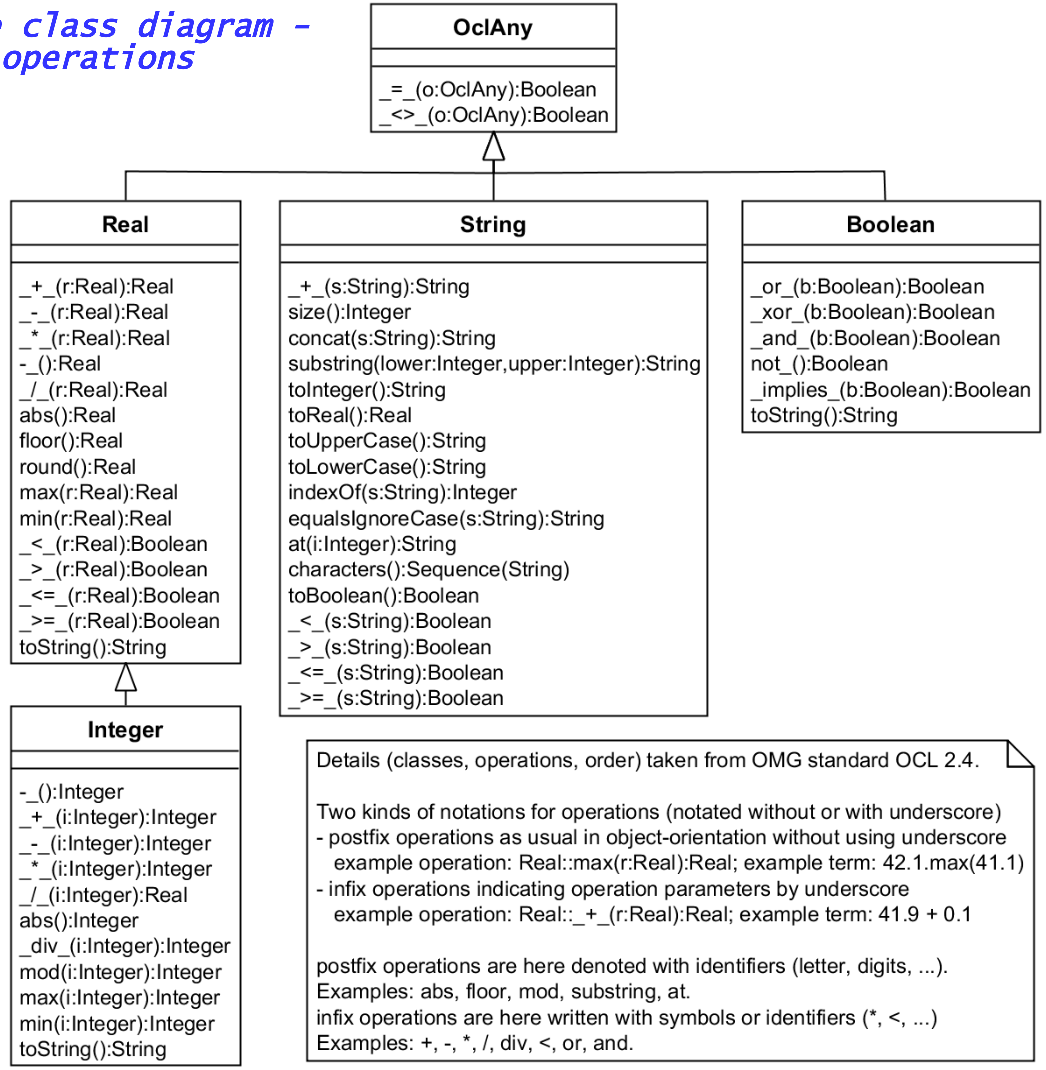
- Basic datatypes: Integer, Real, String, Boolean; constants available
- Integer values: 0, 1, 2, ..., -1, -2, ..., 365, 366, ...,
- Real values: 0, 1, 1.1, 3.14, -3.141, ..., 9.81, ..., 299792.458, ...
- String values: '', 'a', ..., 'z', '0', ..., '9', '1.1', 'Lassie', ...
- Boolean values: false, true, ...

[Definition] Datatype

A datatype has a name and has associated operations that possess arguments and a result being also associated with a datatype. A datatype is conntected to a set of values. The operations are associated with according functions on the value sets.

```
Example: Integer values: ... -2 -1  0  1  2 ...

         operation abs   ...  ↓  ↓  ↓  ↓  ↓ ...   Integer::abs():Integer

                         ...  2  1  0  1  2 ...   -2.abs() = 2
```

OCL datatype class diagram - Classes and operations

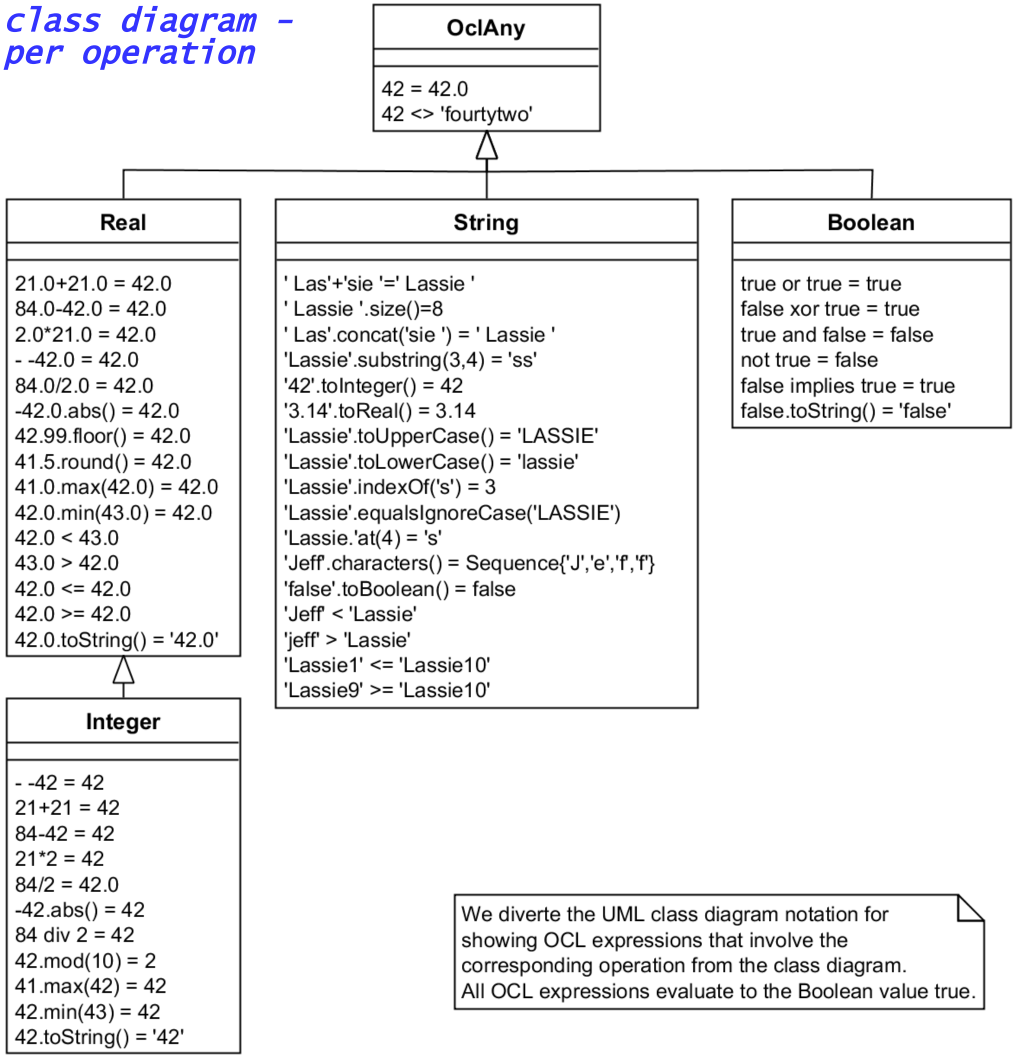


Details (classes, operations, order) taken from OMG standard OCL 2.4.

Two kinds of notations for operations (notated without or with underscore)
- postfix operations as usual in object-orientation without using underscore
  example operation: Real::max(r:Real):Real; example term: 42.1.max(41.1)
- infix operations indicating operation parameters by underscore
  example operation: Real::_+_(r:Real):Real; example term: 41.9 + 0.1

postfix operations are here denoted with identifiers (letter, digits, ...).
Examples: abs, floor, mod, substring, at.
infix operations are here written with symbols or identifiers (*, <, ...)
Examples: +, -, *, /, div, <, or, and.

OCL datatype class diagram – Example term per operation



OclAny

```
42 = 42.0
42 <> 'fourtytwo'
```

Real

```
21.0+21.0 = 42.0
84.0-42.0 = 42.0
2.0*21.0 = 42.0
- -42.0 = 42.0
84.0/2.0 = 42.0
-42.0.abs() = 42.0
42.99.floor() = 42.0
41.5.round() = 42.0
41.0.max(42.0) = 42.0
42.0.min(43.0) = 42.0
42.0 < 43.0
43.0 > 42.0
42.0 <= 42.0
42.0 >= 42.0
42.0.toString() = '42.0'
```

String

```
' Las'+'sie '=' Lassie '
' Lassie '.size()=8
' Las'.concat('sie ') = ' Lassie '
'Lassie'.substring(3,4) = 'ss'
'42'.toInteger() = 42
'3.14'.toReal() = 3.14
'Lassie'.toUpperCase() = 'LASSIE'
'Lassie'.toLowerCase() = 'lassie'
'Lassie'.indexOf('s') = 3
'Lassie'.equalsIgnoreCase('LASSIE')
'Lassie.'at(4) = 's'
'Jeff'.characters() = Sequence{'J','e','f','f'}
'false'.toBoolean() = false
'Jeff' < 'Lassie'
'jeff' > 'Lassie'
'Lassie1' <= 'Lassie10'
'Lassie9' >= 'Lassie10'
```

Boolean

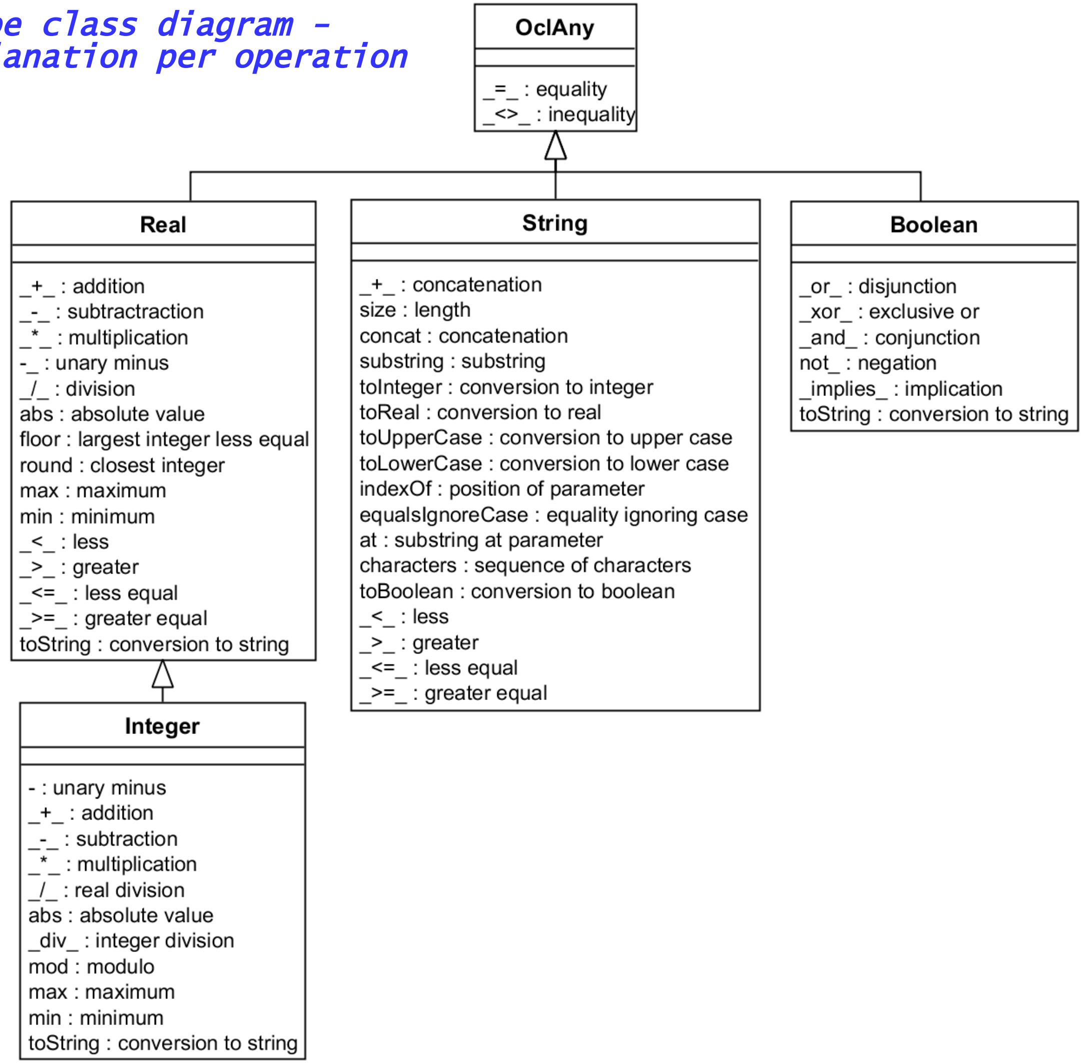
```
true or true = true
false xor true = true
true and false = false
not true = false
false implies true = true
false.toString() = 'false'
```

Integer

```
- -42 = 42
21+21 = 42
84-42 = 42
21*2 = 42
84/2 = 42.0
-42.abs() = 42
84 div 2 = 42
42.mod(10) = 2
41.max(42) = 42
42.min(43) = 42
42.toString() = '42'
```

We diverte the UML class diagram notation for showing OCL expressions that involve the corresponding operation from the class diagram.
All OCL expressions evaluate to the Boolean value true.

OCL datatype class diagram - Verbal explanation per operation



Generalization in the OCL datatype class diagram

Class Integer connected to class Real with unfilled arrowhead towards Real
alternative textual notation: Integer < Real
Real generalization of Integer; Integer specialization/subclass of Real

Real values: Set of Integer values is a subset of the set of Real values
Real operations: Operations from Real are also applicable on Integer

```
Examples: Real::_<_(r:Real):Boolean allows to compute:
          (2.5:Real  < 3:Integer):Boolean   =   true
          (2:Integer < 3:Integer):Boolean   =   true
```

Top-most type OclAny with Real < OclAny, String < OclAny, Boolean < OclAny

OclAny values: (Real values) U (String values) U (Boolean Values) ...
OclAny operations: equality   OclAny::_=_ (o:OclAny):Boolean
inequality OclAny::_<>_(o:OclAny):Boolean

```
Examples: 42 = 84 div 2    : Boolean   =   true
          3.14 <> 9.81     : Boolean   =   true
          'Lassie'<>'Jeff' : Boolean   =   true
          false = false    : Boolean   =   true

          Above the 2 operation parameters have identical datatypes
          Below 2 parameters of different datatypes being also of type OclAny

          42 <> 'fortytwo' : Boolean   =   true
          42 <> '42'       : Boolean   =   true
          42 =  84 / 2     : Boolean   =   true
          42 <> false      : Boolean   =   true
```

Example terms with Real operations

```
3.14+3.14
6.28-3.14
3.14*3.14
-3.14
3.14/2
(-3.14).abs()
(3.14).floor()
(3.14+0.5).floor()
(3.14+0.35).round()
(3.14+0.36).round()
2.14.max(3.14)
3.14.min(4.14)
3.14<3.14
3.14>3.14
3.14<=3.14
3.14>=3.14
3.14.toString()
```

Example terms with Integer operations

```
-42
21+21
84-42
6*7
84/2
(-42).abs
84 div 2
42.mod(10)
41.max(42)
42.min(43)
42.toString()
```

[Exercise] Parentheses in terms. Evaluate the following OCL expressions on the USE shell by using the verbose evaluation with '??'.

```
18/6/3
(18/6)/3
18/(6/3)

3+3*7
(3+3)*7
3+(3*7)

'La'+'ss'+'ie'
('La'+'ss')+'ie'
'La'+('ss'+'ie')

false implies false implies false
(false implies false) implies false
false implies (false implies false)
```

Example terms with String operations

```
'Lassie'+' is a '.concat('collie')
' is '.size()
'Lassie is a collie'.substring(7,10)
'42'.toInteger()
'3.14'.toReal()
'Lassie'.toUpperCase()
'Lassie'.toLowerCase()
'Lassie'.indexOf('s')
'Lassie'.equalsIgnoreCase('LASSIE')
'Lassie'.at(4)
'Lassie'.characters()
'false'.toBoolean()
'Jeff'<'Lassie'
'Jeff'>'Lassie'
'Jeff'<='Lassie'
'Jeff'>='Lassie'
```

Example terms with Boolean operations

```
boy.isCollie or dog.isCollie
boy.isCollie xor dog.isCollie
boy.isCollie and dog.isCollie
not boy.isCollie
boy.isCollie implies dog.isCollie
boy.isCollie.toString()
```

[Definition] Term

If we have in the class C an operation op returning the type D and having parameter p of type D1, we denote this by: C::op(p:D1):D. A term is a formation of constants and operations for building complex computations.

(1) If c is a ***constant*** for datatype D, then c is a term of (data-)type D, notated as c:D.

Examples: 42:Integer, 3.14:Real, 'Lassie':String, true:Boolean

(2) If t1:D1, ..., tn:Dn are terms of type Di and op:(D1,...Dn):D is an operation, then op(t1,...,tn):D is a term of type D. ***Operations*** are notated in two forms: with ***postfix*** and with ***infix*** notation.

(2a) If C::op(p:D1):D is a postfix operation in class C, c is an instance of class C, and t is a term for type D1, then c.op(t):D is a term in ***postfix*** notation.

Example: Given 'Lassie':String, String::at(i:Integer):String and 3:Integer, 'Lassie'.at(3):String is a postfix term of type String.

(2b) If C::_op_(p:D1):D is an infix operation in class C, c is an instance of class C, and t is a term of type D1, then (c op t):D is a term in ***infix*** notation. If the parentheses are ommitted, (complex) default rules apply.

Example: Given 2.71:Real, 3.14:Real, 9.81:Real and Real::_+_(r:Real):Real, then 2.71+3.14*9.81:Real is a term of type Real using infix operations. Further example: (2.71+3.14)*9.81

Additionals matching parentheses may be introduced. Example: (84)div(3-1)

[Exercise] Syntactically *valid and invalid terms*

Consider the following list of pretended OCL terms. Check whether each term is syntactically valid or invalid w.r.t. the available predefined operations in the datatype class diagram.

```
3.14/0
(2.71+3.14).sqrt()
9.81/3.14/2.71

- - - - 42
40*41+42
42.max(41,42)

'Las'.concat('sie')
'Las'_+_('sie')
'Las'_concat_('sie')

(false or false) and (false and false)
2.71 < 3.14 < 9.81
'Lassie'.isSubstringOf('Jeff and Lassie are friends')
```

Terms with let, terms with if, set membership and supersets

Variable names in terms

let VARIABLE=TERM *in* TERM-OPTIONALLY-WITH-VARIABLE

Conditionals in terms

if BOOLTERM *then* THEN-TERM *else* ELSE-TERM *endif*

In OCL there is an if-then-else-endif. There is no 'if-then-endif', because a value has to be calculated when the BOOLTERM is true and when it is false.

Sets and set membership

X=E1 or X=E2 or X=E3 <=> Set{E1,E2,E3}->*includes*(X)

Test for supersets and for subsets

S->includes(E1) and S->includes(E2) <=> S->*includesAll*(Set{E1,E2})

Good way of reading (and thinking of) an OCL term like 'S1->includesAll(S2)' is 'S1 is a superset of S2 (with equality making the term also true)'.

```
S1->includesAll(S2)   <=>   S1 isSupersetOf S2   <=>   S1⊇S2
                      <=>   S2 isSubsetOf   S1   <=>   S2⊆S1
```

Examples: Area of a circle

```
let pi=3.14 in let radius=3.7 in pi*radius*radius
3.14*3.7*3.7
```

Value of a linear function

```
let slope=4 in let yAxis=2 in let ARG=10 in slope*ARG+yAxis
4*10+2
```

OCL terms with 'let' have the advantage that variable names can make the purpose of the term and role that particular term parts play more clear. Structure of the term 'pi*radius*radius' is close to a mathematical textbook formula for the area of a circle. In contrast to the concrete term '3.14*3.7*3.7', the 'let' term can be re-used with different substitutions for the variable 'radius'. Manipulating the concrete term bears the danger that, if one wants to re-calculate the area with a different radius, say '3.8', only one of the needed two substitutions is made, e.g., '3.14*3.8*3.7'. 'let' gives the opportunity to build abstractions. The term 'slope*ARG+yAxis' is more abstract than the concrete evaluation '4*10+2'. It makes the purpose of the term more clear by using good explaining identifiers. It can also be re-used by employing different substitutions for the variables. Both 'let' terms are however more complex than the concrete evaluation terms.

```
let ARG=-42 in if ARG>=0 then ARG else -ARG endif
```

Absolute value of an argument: (-42).abs()

```
I=2 or I=4 or I=8 or I=16 or I=32 or I=64
Set{2,4,8,16,32,64}->includes(I)
```

Check whether an Integer value is equal to a potency of 2 up to 64

```
Set{1,2,3,4,5,6,7,8,9,10,11,12}->includesAll(Set{4,5,6,7,8,9})
```

The set of all months is a superset of the set of summer months.

Complex example using let, if, includes and includesAll

Check whether a String represents a date in the format 'YYYY-MM-DD'

Solution 1

```
let DATE='2000-11-31' in
let YEAR=DATE.substring(1,4) in
let MONTH=DATE.substring(6,7) in
let DAY=DATE.substring(9,10) in
let DIGIT=Set{'0','1','2','3','4','5','6','7','8','9'} in
'0000'<=YEAR and YEAR<='2020' and '01'<=MONTH and MONTH<='12' and
'01'<=DAY and DAY<=if MONTH='02' then '29' else if Set{'04','06',
  '09','11'}->includes(MONTH) then '30' else '31' endif endif and
DIGIT->includesAll(Set{DATE.at(1),DATE.at(2),DATE.at(3),DATE.at(4),
  DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10)}) and
DATE.at(5)='-' and DATE.at(8)='-' and DATE.size()=10
```

It may be the case that in the argument set of the 'includesAll' two different terms (or even more) evaluate to the same String value. This is not a problem for the evaluation:

```
let DATE='2000-12-31' in Set{DATE.at(1),DATE.at(2),DATE.at(3),
  DATE.at(4),DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10)}
```

evaluates to

```
Set{'0','1','2','3'} : Set(String)
```

Instead of the restricting nested if conditions for the upper limit of DAY one can formulate this part with 3 implications that are connected though a conjunction.

```
'01'<=DAY and DAY<=if MONTH='02' then '29' else if Set{'04','06',
  '09','11'}->includes(MONTH) then '30' else '31' endif endif

<=>

'01'<=DAY and (MONTH='02' implies DAY<='29') and
(Set{'04','06','09','11'}->includes(MONTH) implies DAY<='30') and
(Set{'01','03','05','07','08','10','12'}->includes(MONTH)
   implies DAY<='31')
```

Solution 2

```
let DATE='2000-12-31' in
let YEAR=DATE.substring(1,4) in
let MONTH=DATE.substring(6,7) in
let DAY=DATE.substring(9,10) in
let DIGIT=Set{'0','1','2','3','4','5','6','7','8','9'} in
'0000'<=YEAR and YEAR<='2020' and '01'<=MONTH and MONTH<='12' and
'01'<=DAY and (MONTH='02' implies DAY<='29') and
(Set{'04','06','09','11'}->includes(MONTH) implies DAY<='30') and
(Set{'01','03','05','07','08','10','12'}->includes(MONTH)
   implies DAY<='31') and
DIGIT->includesAll(Set{DATE.at(1),DATE.at(2),DATE.at(3),DATE.at(4),
  DATE.at(6),DATE.at(7),DATE.at(9),DATE.at(10)}) and
DATE.at(5)='-' and DATE.at(8)='-' and DATE.size()=10
```

Glimpse of collections in OCL

Two kinds of collections above: *sequences and sets*

```
- 'Lassie'.characters() = Sequence{'L','a','s','s','i','e'}
- Set{1,2,3,4,5,6,7,8,9,10,11,12}->includesAll(Set{4,5,6,7,8,9})
```

Features of sequences and sets

- Both are collections; both able to contain elements; both typed
- A sequence has an order for its elements; an element may appear twice
- A set does not have an element order; an element appears 'de facto' once
- Operations can be appplied to collections (=, at, intersection, ...)
- Some operations can be applied to both sequences and sets (=, ...), some only to sequences (at, ...), some only to sets (intersection, ...)
- Collections essential OCL part; later discussed in detail

Example terms with collections and collection operations

```
Sequence{'J','e','f','f'} -> Sequence{'J','e','f','f'} : Sequence(String)
Set{'J','e','f','f'} -> Set{'J','e','f'} : Set(String)
Set{'L','a','s','s','i','e'} -> Set{'L','a','e','i','s'} : Set(String)

Set{'J','e','f','f'}=Set{'J','e','f'} -> true : Boolean
Set{'J','e','f','f'}=Set{'f','e','J'} -> true : Boolean
Sequence{'J','e','f','f'}=Sequence{'J','e','f'} -> false : Boolean

Sequence{'J','e','f','f'}->at(4) -> 'f' : String
Set{'J','e','f','f'}->
  intersection(Set{'L','a','s','s','i','e'}) -> Set{'e'} : Set(String)
```

Writing down sets more tricky than expected; good way of thinking about sets: distinguish (in mind) a set representation and the set itself

```
Example: (Set{9,8,7} = Set{7,8,9}) = (Set{9,8,8,7,7,7} = Set{7,9,8}) -> true
```

Special cases and limitations requiring special attention

Arithmetic works correctly only up to certain bounds

```
42=42.00000000000001  -> false
42=42.000000000000001 -> true
Above USE evaluation; same behavior in Java; 13 zeros VS 14 zeros;
relevant number of digits, not number of zeros

2147483647+1 -> -2147483648 : Integer
With markers for easier reading:
2.147.483.647+1 -> -2.147.483.648 : Integer
Typical Integer overflow in USE, as present in Java
```

Operation 'mod' in OCL needs getting used to

```
7 div 3     ->   2 : Integer
7 / 3       ->   2.3333333333333335 : Real
7 mod 3     ->   Error:line 1:2 missing EOF at 'mod'
7.mod(3)    ->   1 : Integer
```

ASCII ordering of String values with partly unexpected behavior

```
'0'<'9' and '9'<'A' and 'A'<'Z' and 'Z'<'a' and 'a'<'z'   ->   true
'a..z_A..Z_'<'A..Z_a..z'                                  ->   false
'9'<'10'                                                  ->   false

'ASCII order'<>'Lexicographical order'                    ->   true
'Lexicographical order'<>'Numerical order'                ->   true
```

[End of 'Content Presentation' - Beginning of 'Exercises']

[Exercise] Build an OCL term that *converts* between Kilometer and English Miles

```
let K2M=true in let ARG=2 in
if K2M then ARG/1.609344 else ARG*1.609344 endif
```

Analogously build an OCL term to *convert* between degree Celsius and degree Fahrenheit

[Exercise] Build at least 3 syntactically different terms that *reverse* a string with fixed length 4. By 'syntactically different' we mean terms that, considered as sequences of characters, are different. Example: Syntactically different terms that evaluate to 42.0 are 42.0, 042.0, 21.0+21.0, 15.0+14+13, (15+14.0)+13, 15+(14+13.0)

[Exercise] Test whether a String with up to 3 characters represents a *programming language identifier* (first a letter or underscore, optionally followed by up to 2 characters being a letter, a digit or the underscore). A regular expression (i.e., a pattern over characters) for such a test is:
^[a-zA-Z_][a-zA-Z_0-9]?[a-zA-Z_0-9]?$

For the solution another operation on sets may be useful. The elements of a first set S1 and a second set S2 may be combined in that the result consists of all elements that are either in the first or the second set, the union of S1 and S2: S1->union(S2).

Example:

```
let ARG='Ab' in let abc=Set{'a','b','c'} in let ABC=Set{'A','B','C'} in
ABC->includes(ARG.at(1)) and ABC->union(abc)->includes(ARG.at(2))
```

The first letter of the String ARG is one of A, B, C and the second letter is one of A, B, C, a, b, c.

[Exercise] Precision in formulating descriptions

Consider the two following descriptions.

Description A:

```
let ARG=... in let abc=Set{'a','b','c'} in let ABC=Set{'A','B','C'} in
ABC->includes(ARG.at(1)) and ABC->union(abc)->includes(ARG.at(2))
```

The first letter of the String ARG is one of A, B, C and the second letter is one of A, B, C, a, b, c.

Description B:

```
let ARG=... in let abc=Set{'a','b','c'} in let ABC=Set{'A','B','C'} in
ABC->includes(ARG.at(1)) and ABC->union(abc)->includes(ARG.at(2))
```

The first letter of the String ARG is one of 'A', 'B', 'C' and the second letter is one of 'A', 'B', 'C', 'a', 'b', 'c'.

Discuss whether these two statements show a difference w.r.t. precision.

[Exercise] Construct an OCL term that checks a String for the *time format* 'HH:MM' with 00<=HH<=23 and 00<=MM<=59.

[Exercise] Modify the OCL term checking a date in order to cover *leap* years.

[Exercise] Complete each of the following (partial) OCL terms showing *placeholders* ___ with missing items (values, terms or operation calls) so that a Boolean term evaluating to the Boolean value true arises.

Examples: OCL term (a): let I=___ in I+I = 42. The placeholder ___ can be substituted by 21 in order for the term to become true.

OCL term (b): let I=___ in I.___(2)=1. One placeholder substitution is: let I=3 in I.mod(2)=1. Another substitution is let I=5 in I.mod(2)=1. A third substitution is let I=1 in I.min(2)=1.

Two different placeholders may be substituted differently. In general, the solutions, i.e., the substitutions for the placeholders, are not unique. In other words, there may be more than one solution for each term.

```
___.___(3,5)='wxy'

___.size()=0

(___+___+___).size()=0

let V=___ in let W=___ in V.___(W).size=0

let V=___ in V.___(1,3)=V.___(3)+V.___(2)+V.___(1)

let V=___ in V.___()=Sequence{'A','B','B','A'}

let V=___ in V.toUpperCase()=V.toLowerCase()

let V=___ in let W=___ in V+W=W*V and V div W = V/W
let V=___ in V.round()=V.floor()+1

let V=___ in V.round()=V.floor()

let V=___ in V.abs()=V.abs().abs()

let V=___ in let W=___ in V.max(W)=V.min(W)

let V=___ in let W=___ in (V implies W) = (W implies V)

let V=___ in let W=___ in (V<>W) and (V xor W) and (W xor V)

let V=___ in let W=___ in (V implies W) = (not false)

let V=___ in let W=___ in (V and W) = (V or W)
```

[Exercise] In OCL, the comparison operations less <, greater >, less-equal <= and greater-equal >= are not defined on the datatype Boolean. Define these operations either by (partly) filling in the following table or/and by (partly) defining these operations by other operations or terms like 'V op W' :<=> 'TERM-OPTIONALLY-WITH-V-AND-W'.

		<	>	<=	>=
false	false				
false	true				
true	false				
true	true				

Give a justification for your decisions.

Alternative display of table:

A	B	A<B	A>B	A<=B	A>=B
false	false				
false	true				
true	false				
true	true				

[Exercise] Construct 4 Real, 4 Integer, 4 String and 4 Boolean OCL terms (pairwise different) each containing at least 2 different operations (from all available operations). The 4 terms for each datatype have to yield the same result. Example: let I=42 in I, let I=42 in I+0, let I=42 in 0+I, let I=42 in 0+I+0.

[Exercise] In OCL the binary operations max and min are available on Real and Integer: Real::max(r:Real):Real, Real::min(r:Real):Real, Integer::max(i:Integer):Integer, Integer::min(i:Integer):Integer. Is there any advantage for defining max and min again on Integer? Or is this redundant and without any effect?

[Exercise] Construct an OCL term such that the term uses a Real, an Integer, a String and a Boolean operation or an operation with such a return type. Example: The used types in the term 'Lassie'.size() would be String and Integer.

[Exercise] Construct 12 pairs of syntactically different OCL terms (3 Real, 3 Integer, 3 String, 3 Boolean pairs) with appropriate variables such that the terms from the term pair are equivalent, i.e., evaluate to identical values for all substitutions for the variables.

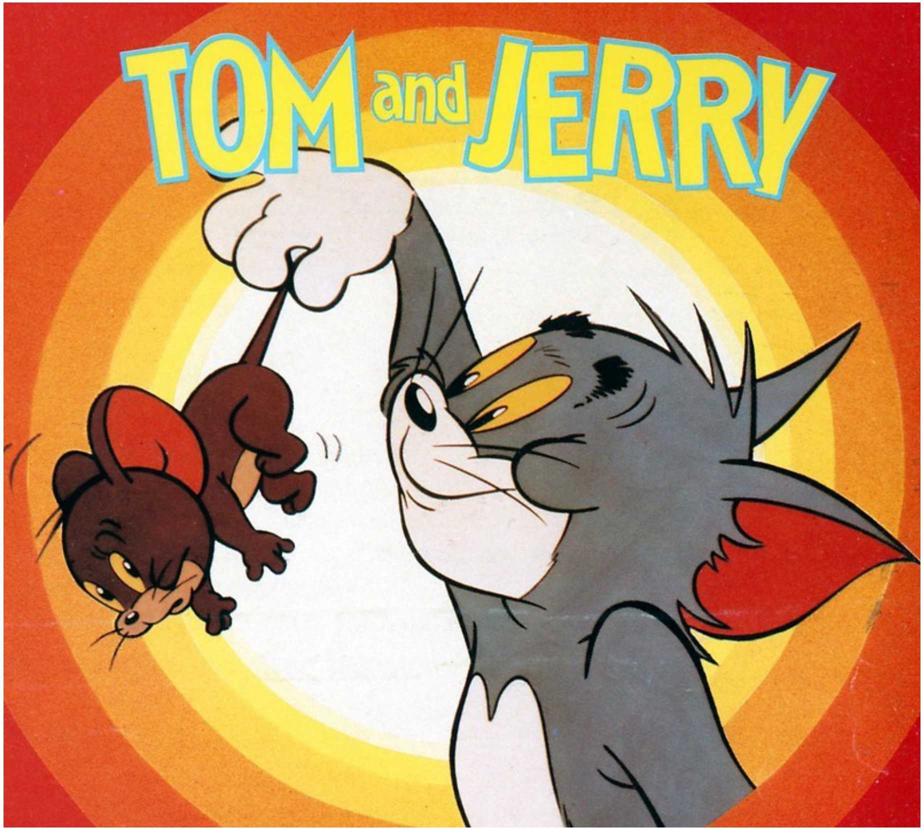
Example pairs: I+0, 0+I for I:Integer; I+J, J+I for I,J:Integer

[Exercise] Fill out the following matrixes evaluating the comparisons < and = between the given OCL terms. State whether the comparision is true with T, the comparison is false with F or the comparison yields a syntax error with -. Only the lower half and the diagonal have to be filled out.

<	42	042	42.0	042.0	'42'	'042'	'42.0'	'042.0'
42		...	....	.....	....	.....	......	....
042			....	.....	....	.....	......	....
42.0				.....	....	.....	......	....
042.0					....	.....	......	....
'42'						.....	......	....
'042'							......	....
'42.0'								....
'042.0'								

=	42	042	42.0	042.0	'42'	'042'	'42.0'	'042.0'
42		...	....	.....	....	.....	......	....
042			....	.....	....	.....	......	....
42.0				.....	....	.....	......	....
042.0					....	.....	......	....
'42'						.....	......	....
'042'							......	....
'42.0'								....
'042.0'								

[Exercise] Consider the figure below and the table below that shows possible command lists for a construction of an object model in the USE class model discussed before. Each command list is trying to represent the circumstances from the figure.



(A)	(B)	(C)	(D)	(E)

new Creature('tom')
tom.name:='Tom'
tom.age:=8
tom.isCat:=true
tom.weight:=6.5
 	
new Creature('jerry')
jerry.name:='Jerry'
jerry.age:=3
jerry.isMouse:=true
jerry.weight:=0.1
	new Creature('tom')
	tom.name:='Tom'
	tom.age:=8
	tom.isMouse:=true
	tom.weight:=6.5
	
	new Creature('jerry')
	jerry.name:='Jerry'
	jerry.age:=3
	jerry.isCat:=true
	jerry.weight:=0.1
		new Creature('tom')
		tom.name:='Tom'
		tom.age:=8
		tom.isCollie:=false
		tom.weight:=6.5
		
		new Creature('jerry')
		jerry.name:='Jerry'
		jerry.age:=3
		jerry.isCollie:=false
		jerry.weight:=0.1
			new Creature('tom')
			tom.name:=Tom
			tom.age:=8.25
			tom.isCollie:=FALSE
			tom.weight:=6.5kg
			
			new Creature('jerry')
			jerry.name:=Jerry
			jerry.age:=3.5
			jerry.isCollie:=FALSE
			jerry.weight:=50g
				new Creature('tom')
				tom.name:='Tom'
				tom.age:='8.25'
				tom.isCollie:='FALSE'
				tom.weight:='6.5kg'
				
				new Creature('jerry')
				jerry.name:='Jerry'
				jerry.age:='3.5'
				jerry.isCollie:='FALSE'
				jerry.weight:='50g'

1. Which of the command lists from (A)-(D) would be syntactically correct w.r.t. the USE class model? If errors during execution would occur, describe the nature of the error.

2. In your opinion, which elements of the command lists meet the circumstances presented in the previous figure most accurately?

3. How does one has to modify the USE class model in order to get the command list (E) getting accepted without syntax errors?

4. Define 6 additional, different Boolean attributes in the class Creature for the USE model, such that the attribute evaluations for the models { Lassie-Jeff, Tom-Jerry } show a wide spectrum of value combinations. For example:

	Lassie	Jeff	Tom	Jerry
Att1	false	false	false	false
Att2	false	false	false	true
Att3	false	false	true	false
Att4	false	false	true	true
Att5	false	true	false	false
Att6	false	true	false	true

Ideally, the six lines with the false-true tuples show six different tuples, not necessarily the ones displayed.