Object Constraint Language (OCL)

- A UML diagram (e.g., a class diagram) does not provide all relevants aspects of a specification
- It is necessary to describe additional constraints about the objects in the model
- Constraints specify invariant conditions that must hold for the system being modeled
- Constraints are often described in natural language and this always result in ambiguities
- Traditional formal languages allow to write unambiguous constraints, but they are difficult for the average system modeler
- OCL: Formal language used to express constraints, that remains easy to read and write

Object Constraint Language (OCL)

- **Pure expression language**: expressions **do not have side effet**
  - ◇ when an OCL expression is evaluated, it returns a value
  - ◇ its evaluation cannot alter the state of the corresponding executing system
  - ◇ an OCL expression can be used to specify a state change (e.g., in a post-condition)
- **Not a programming language**
  - ◇ it is not possible to write program logic or flow of control in OCL
  - ◇ cannot be used to invoke processes or activate non-query operations
- **Typed language**: each expression has a type
  - ◇ well-formed expressions must obey the type conformance rules of OCL
  - ◇ each classifier defined in a UML model represents a distinct OCL type
  - ◇ OCL includes a set of supplementary predefined types
- The evaluation of an OCL expression is **instantaneous**
  - ◇ the state of objects in a model cannot change during evaluation

Where to Use OCL

- To specify invariants on classes and types in the class model
- To specify type invariants for stereotypes
- To describe pre- and post-conditions on operations and methods
- To describe guards
- As a navigation language
- To specify constraints on operations
- OCL is used to specify the well-formedness rules of the UML metamodel

Basic Values and Types

- A number of basic types are predefined in OCL
- Examples of basic types and their values

Type	**Values**
`Boolean`	`true, false`
`Integer`	`1, -5, 2564, ...`
`Real`	`1.5, 3.14, ...`
`String`	`‘To be or not to be’, ...`

- A number of operations are defined on the predefined types

Type	**Operations**
`Boolean`	`and, or , xor, not, implies, if-then-else-endif`
`Integer`	`+, -, *, /, abs, div, mod, max, min`
`Real`	`+, -, *, /, abs, floor, round, max, min, <, >, <=, >=`
`String`	`size, concat, substring, toInteger, toReal`

Collections

- `Collection`: an abstract type with four concrete collection types
  - `Set`: the mathematical set (without duplicate elements)
    `Set {1 , 2 , 5}`   `Set {‘apple’, ‘orange’, ‘strawberry’}`
  - `OrderedSet`: a set in which the elements are ordered by their position
    `OrderedSet {5, 4, 3, 2, 1}`
  - `Bag`: a set that may contain duplicate elements
    `Bag {1, 2, 5, 2}`
  - `Sequence`: a bag in which the elements are ordered
    `Sequence {1, 2, 5, 10}`   `Sequence {‘ape’, ‘nut’}`
- Notation ‘`..`’ used for a sequence of consecutive integers
  - `Sequence {1..5}` is the same as `Sequence {1, 2, 3, 4, 5}`
- Elements of collections may be collections themselves
  `Set { Sequence {1, 2, 3, 4}, Sequence {5, 6} }`
- Collections have a set of predefined operations
  - They are accessed using the `->` notation

Common Operations for All Collections

- C, C_1, C_2 are values of type `Collection(t)`, v is a value of type `t`

	Signature	**Semantics**
`size`	`Collection(t)` → `Integer`	$\mid C \mid$
`count`	`Collection(t)` × `t` → `Integer`	$\mid C \cap \{v\} \mid$
`includes`	`Collection(t)` × `t` → `Boolean`	$v \in C$
`excludes`	`Collection(t)` × `t` → `Boolean`	$v \notin C$
`includesAll`	`Collection(t)` × `Collection(t)` → `Boolean`	$C_2 \subseteq C_1$
`excludesAll`	`Collection(t)` × `Collection(t)` → `Boolean`	$C_2 \cap C_1 = \varnothing$
`isEmpty`	`Collection(t)` → `Boolean`	$C = \varnothing$
`notEmpty`	`Collection(t)` → `Boolean`	$C \neq \varnothing$
`sum`	`Collection(t)` → `t`	$\sum_{i=1}^{\mid C \mid} v_i$

Set Operations

- S, S_1, S_2 are values of type `Set(t)`, B is a value of type `Bag(t)`, v is a value of type `t`

	Signature	Semantics
`union`	`Set(t)` × `Set(t)` → `Set(t)`	$S_1 \cup S_2$
`union`	`Set(t)` × `Bag(t)` → `Bag(t)`	$S \cup B$
`intersection`	`Set(t)` × `Set(t)` → `Set(t)`	$S_1 \cap S_2$
`intersection`	`Set(t)` × `Bag(t)` → `Set(t)`	$S \cap B$
`-`	`Set(t)` × `Set(t)` → `Set(t)`	$S_1 - S_2$
`symmetricDifference`	`Set(t)` × `Set(t)` → `Set(t)`	$(S_1 - S_2) \cup (S_2 - S_1)$
`including`	`Set(t)` × `t` → `Set(t)`	$S \cup \{v\}$
`excluding`	`Set(t)` × `t` → `Set(t)`	$S - \{v\}$
`asSet`	`Set(t)` → `Set(t)`	
`asOrderedSet`	`Set(t)` → `OrderedSet(t)`	
`asBag`	`Set(t)` → `Bag(t)`	
`asSequence`	`Set(t)` → `Sequence(t)`	

- Operations `asOrderedSet` and `asSequence` are nondeterministic
  ⇒ Result contains the elements of the source set in arbitrary order

Bag Operations

- B, B_1, B_2 are values of type `Bag(t)`, S is a value of type `Set(t)`, v is a value of type `t`

	Signature	Semantics
`union`	`Bag(t)` × `Bag(t)` → `Bag(t)`	$B_1 \cup B_2$
`union`	`Bag(t)` × `Set(t)` → `Bag(t)`	$B \cup S$
`intersection`	`Bag(t)` × `Bag(t)` → `Bag(t)`	$B_1 \cap B_2$
`intersection`	`Bag(t)` × `Set(t)` → `Set(t)`	$B \cap S$
`including`	`Bag(t)` × `t` → `Bag(t)`	$S \cup \{\{v\}\}$
`excluding`	`Bag(t)` × `t` → `Bag(t)`	$S - \{\{v\}\}$
`asSet`	`Bag(t)` → `Set(t)`	
`asOrderedSet`	`Bag(t)` → `OrderedSet(t)`	
`asBag`	`Bag(t)` → `Bag(t)`	
`asSequence`	`Bag(t)` → `Sequence(t)`	

Sequence Operations

- S, S_1, S_2 are values of type `Set(t)`, v is a value of type `t`, operator $\circ$ denotes the concatenation of lists, $\pi_i(S)$ projects the ith element of a sequence S, $\pi_i^j(S)$ is the subsequence of S from the ith to the jth element

	Signature	Semantics
`union`	`Sequence(t)` × `Sequence(t)` → `Sequence(t)`	$S_1 \circ S_2$
`append`	`Sequence(t)` × `t` → `Sequence(t)`	$S \circ \langle v \rangle$
`prepend`	`Sequence(t)` × `t` → `Sequence(t)`	$\langle v \rangle \circ S$
`subSequence`	`Sequence(t)` × `Integer` × `Integer` → `Sequence(t)`	$\pi_i^j(S)$
`at`	`Sequence(t)` × `Integer` → `Sequence(t)`	$\pi_i(S)$
`first`	`Sequence(t)` → `t`	$\pi_1(S)$
`last`	`Sequence(t)` → `t`	$\pi_{\|S\|}(S)$
`including`	`Sequence(t)` × `t` → `Sequence(t)`	$S \circ \langle v \rangle$
`excluding`	`Sequence(t)` × `t` → `Sequence(t)`	$S - \{v\}$
`asSet`	`Sequence(t)` → `Set(t)`	
`asOrderedSet`	`Sequence(t)` → `OrderedSet(t)`	
`asBag`	`Sequence(t)` → `Bag(t)`	
`asSequence`	`Sequence(t)` → `Sequence(t)`	

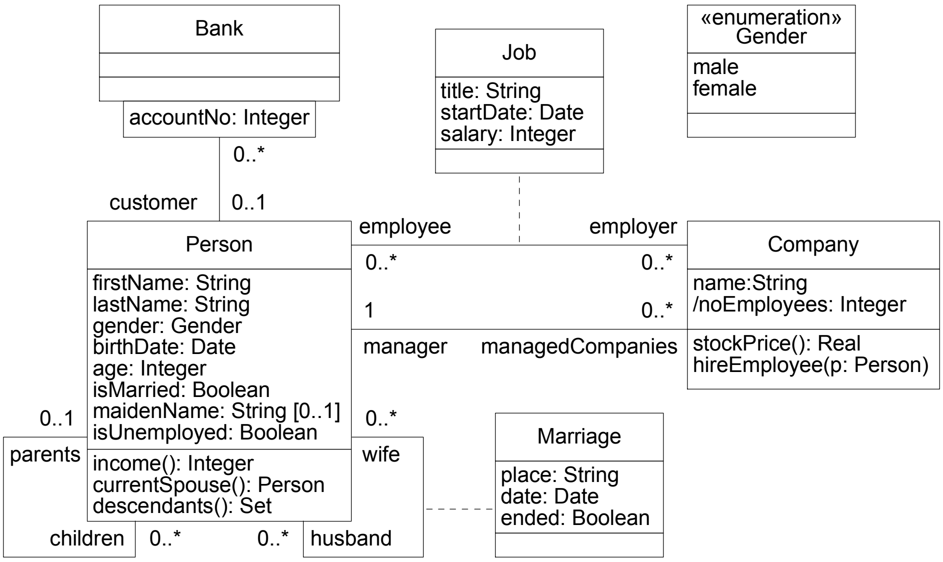
Type Conformance

- OCL is a typed language, the basic value types are organized in a type hierarchy
- The hierarchy determines conformance of the different types to each other
- Type `type1` **conforms** with type `type2` when an instance of `type1` can be substituted at each place where an instance of `type2` is expected
- **Valid expression**: OCL expression in which all types conform

Type Conformance Rules

- `Type1` conforms to `Type2` when they are identical
- `Type1` conforms to `Type2` when it is a subtype of `Type2`
- `Collection(Type1)` conforms to `Collection(Type2)` when `Type1` conforms to `Type2`
- Type conformance is transitive: if `type1` conforms with `type2` and `type2` conforms with `type3`, then `type1` conforms with `type3`
- Example: If `Bicycle` and `Car` are subtypes of `Transport`
  - `Set(Bicycle)` conforms to `Set(Transport)`
  - `Set(Bicycle)` conforms to `Collection(Bicycle)`
  - `Set(Bicycle)` conforms to `Collection(Transport)`
  - `Set(Bicycle)` **does not** conform to `Bag(Bicycle)`

Class Diagram Example



Comments, Infix Operators

Comments

- Denoted by `--`

```
-- this is a comment
```

Infix Operators

- Use of infix operators (e.g., `+`, `-`, `=`, `<`, ...) is allowed
- Expression `a + b` is conceptually equivalent to `a.+(b)`, i.e., invoking the `+` operation on `a` with `b` as parameter
- Infix operators defined for a type must have exactly one parameter

Context and Self

- All classifiers (types, classes, interfaces, associations, datatypes, ...) from an UML model are types in the OCL expressions that are attached to the model
- Each OCL expression is written in the context of an instance of a specific type

```
context Person
  ...
```

- Reserved word `self` is used to refer to the contextual instance
- If the context is `Person`, `self` referes to an instance of `Person`

Object and Properties

- All **properties** (attributes, association ends, methods and operations without side effects) defined on the types of a UML model can be used in OCL expressions
- The value of a property of an object defined in a class diagram is specified by a dot followed by the name of the property
- If the context is `Person`, `self.age` denotes the value of attribute `age` on the instance of `Person` identified by `self`
- The type of the expression is the type of attribute `age`, i.e., `Integer`
- If the context is `Company`, `self.stockPrice()` denotes the value of operation `stockPrice` on the instance identified by `self`
- Parentheses are mandatory for operations or methods, even if they do not have parameters

Invariants

- Determine a constraint that must be true for all instances of a type
- Value of attribute `noEmployees` in instances of `Company` must be less than or equal to 50

```
context Company inv:
  self.noEmployees <= 50
```

- Equivalent formulation with a `c` playing the role of `self`, and a name for the constraint

```
context c: Company inv SME:
  c.noEmployees <= 50
```

- The stock price of companies is greater than 0

```
context Company inv:
  self.stockPrice() > 0
```

Pre- and Post-conditions

- Constraints associated with an operation or other behavioral feature
- **Pre-condition**: Constraint assumed to be true **before** the operation is executed
- **Post-condition**: Constraint satisfied **after** the operation is executed
- Pre- and post-conditions associated to operation `income` in `Person`

```
context Person::income(): Integer
  pre:  self.age >= 18
  post: result < 5000
```

- `self` is an instance of the type which owns the operation or method
- `result` denotes the result of the operation, if any
- Type of `result` is the result type of the operation (`Integer` in the example)
- A name can be given to the pre- and post-conditions

```
context Person::income(): Integer
  pre adult:  self.age >= 18
  post resultOK: result < 5000
```

Previous Values in Postconditions

- In a postcondition, the value of a property `p` is the value upon completion of the operation
- The value of `p` at the start of the operation is referred to as `p@pre`

```
context Person::birthDayHappens()
  post: age = age@pre + 1
```

- For operations, ‘`@pre`’ is postfixed to the name, before the parameters

```
context Company::hireEmployee(p: Person)
post: employee = employee@pre->including(p) and
      stockPrice() = stockPrice@pre() + 10
```

- The ‘`@pre`’ postfix is allowed only in postconditions
- Accessing properties of previous object values
  - `a.b@pre.c`: the new value of `c` of the old value of `b` of `a`
  - `a.b@pre.c@pre`: the old value of `c` of the old value of `b` of `a`

Body Expression

- Used to indicate the result of a query operation
- Income of a person is the sum of the salaries of her jobs

```
context Person::income(): Integer
body: self.job.salary->sum()
```

- Expression must conform to the result type of the operation
- Definition may be **recursive**: The right-hand side of the definition may refer to the operation being defined
- A method that obtains the direct and indirect descendants of a person

```
context Person::descendants(): Set
  body: result = self.children->union(
        self.children->collect(c | c.descendants()) )
```

- Pre-, and postconditions, and body expressions may be mixed together after one operation context

```
context Person::income(): Integer
pre: self.age >= 18
body: self.job.salary->sum()
post: result < 5000
```

Let Expression

- Allows to define a variable that can be used in a constraint

```
context Person inv:
  let numberJobs: Integer = self.job->count() in
  if isUnemployed then
    numberJobs = 0
  else
    numberJobs > 0
  endif
```

- A `let` expression is only known within its specific expression

Definition Expressions

- Enable to reuse variables or operations over multiple expressions
- Must be attached to a classifier and may only contain variable and/or operation definitions

```
context Person
def: name: String = self.firstName.concat(‘ ’).concat(lastName)
def: hasTitle(t: String): Boolean = self.job->exists(title = t)
```

- Names of the attributes/operations in a `def` expression must not conflict with the names of attributes/association ends/operations of the classifier

Initial and Derived Values

- Used to indicate the initial or derived value of an attribute or association end
- Attribute `isMarried` in `Person` is initialized to `false`

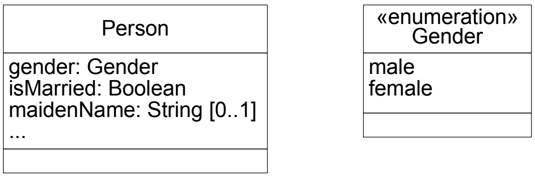
```
context Person::isMarried: Boolean
init: self.isMarried = false
```

- Attribute `noEmployees` in `Company` is a derived attribute

```
context Company::noEmployees: Integer
derive: self.employee->size()
```

- For an **attribute**: expression must conform to the attribute type
- For an **association end**: conformance depends on multiplicity
  - at most one: expression must conform to the classifier at that end
  - may be more than one: expression must conform to `Set` or `OrderedSet`

Enumeration Types


Person

gender: Gender
isMarried: Boolean
maidenName: String [0..1]
...

«enumeration»
Gender

male
female

- Define a number of literals that are the possible values of the enumeration
- An enumeration value is referred as in `Gender::female`
- Only married women can have a maiden name

```
context Person inv:
  self.maidenName <> ‘’ implies
  self.gender = Gender::female and self.isMarried = true
```

Packages

- Within UML, types are organized in packages
- Previous examples supposed that the package in which the classifier belongs is clear from the environment
- The `package` and `endpackage` statements can be used to explicitly specify this

```
package Package::SubPackage

context X inv:
  ... some invariant ...

context X::operationName(...): ReturnType
  pre: ... some precondition ...

endpackage
```

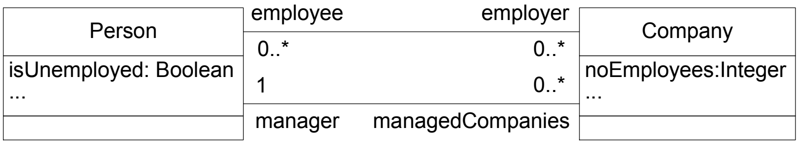
- For referring to types in other packages the following notations may be used

```
Packagename::Typename
Packagename1::Packagename2::Typename
```

Undefined Values

- One or more subexpressions in an OCL expression may be undefined
- In this case, the complete expression will be undefined
- Exceptions for Boolean operators
  - `true or` anything is `true`
  - `false and` anything is `false`
  - `false implies` anything is `true`
  - anything `implies true` is `true`
- The first two rules are valid irrespective of the order of the arguments and whether or not the value of the other sub-expression is known
- Exception for `if-then-else` expression: it will be valid as long as the chosen branch is valid, irrespective of the value of the other branch

Navigating Associations (1)



- From an object, an association is navigated using the opposite role name

```
context Company
  inv: self.manager.isUnemployed = false
  inv: self.employee->notEmpty()
```

- Value of expression depends on maximal multiplicity of the association end
  - 1: value is an object
  - *: value is a `Set` of objects (an `OrderedSet` if association is `{ordered}`)
- If role name is missing, the name of the type at the association end starting with a lowercase character is used (provided it is not ambiguous)

```
context Person
  inv: self.bank.balance >= 0
```

Navigating Associations (2)

- When multiplicity is at most one, association can be used as a single object or as a set containing a single object
- `self.manager` is an object of type `Person`

```
context Company inv:
  self.manager.age > 40
```

- `self.manager` as a set

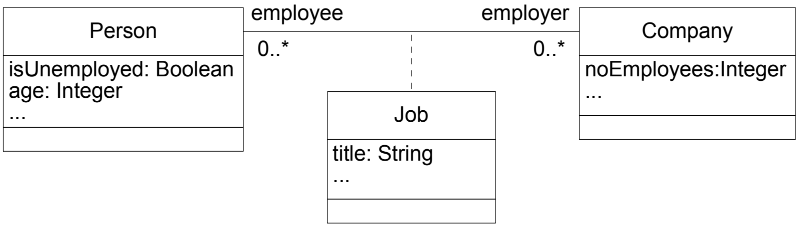
```
context Company inv:
  self.manager->size() = 1
```

- For optional associations, it is useful to check whether there is an object or not when navigating the association

```
context Person inv:
  self.wife->notEmpty() implies self.gender = Gender::male and
  self.husband->notEmpty() implies self.gender = Gender::female
```

- OCL expressions are read and evaluated from left to right

Association Classes



- For navigating **to** an association class: a dot and the name of the association class starting with a lowercase character is used

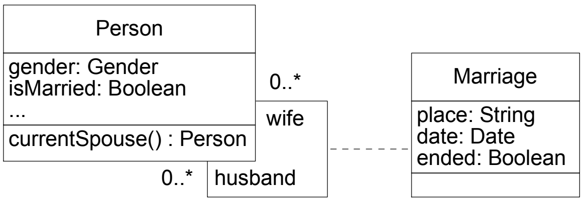
```
context Person
  inv: self.isUnemployed = false implies self.job->size() >= 1
```

- For navigating **from** an association class to the related objects: a dot and the role names at the association ends is used

```
context Job
  inv: self.employer.noEmployees >= 1
  inv: self.employee.age >= 18
```

- This always results in exactly **one** object

Recursive Association Classes (1)



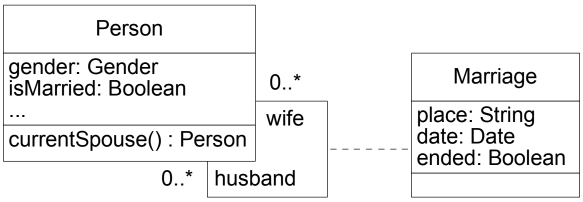
- Direction in which a **recursive** association is navigated is required
- Specified by enclosing the corresponding role names in square brackets
- A person is currently married to at most one person

```
context Person inv:
  self.marriage[wife]->select(m | m.ended = false)->size()=1 and
  self.marriage[husband]->select(m | m.ended = false)->size()=1
```

- May also be used for non-recursive associations, but it is not necessary

```
context Person inv:
  self.job[employer] ...
```

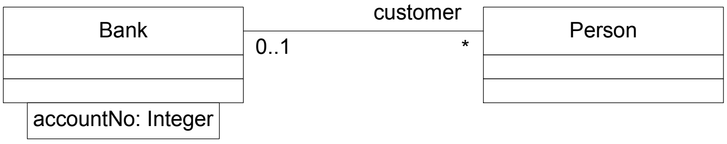
Recursive Association Classes (2)



- Operation that selects the current spouse of a person

```
context Person::currentSpouse() : Person
pre: self.isMarried = true
body:
  if gender = Gender::male
    self.marriage[wife]->select(m | m.ended = false).wife
  else
    self.marriage[husband]->select(m | m.ended = false).husband
  end
```

Qualified Associations



- Qualified associations use one or more qualifier attributes to select the objects at the other end of the association
- A bank can use the `accountNumber` attribute to select a particular customer
- Using qualifier values when navigating through qualified associations

```
context Bank
  inv: self.customer[12345] ...
  -- results in one Person, having account number 12345
```

- Leaving out the qualifier values

```
context Bank
  inv: self.customer ...
  -- results in a Set(Person) with all customers of the bank
```

Re-typing or Casting

- Allows an object to be re-typed as another type
- Expression `o.oclAsType(Type2)` re-types an object `o` of type `Type1` into a another type `Type2`
- Suppose `Super` is a supertype of type `Sub`
- Allows one to use a property of an object defined on a subtype of the currently known type of the object

```
context Super inv:
  self.oclAsType(Sub).p -- accesses the p property defined in Sub
```

- Can be used to access a property of a superclass that has been overriden

```
context Sub inv:
  self.p
  -- accesses the p property defined in Sub
  self.oclAsType(Super).p
  -- accesses the p property defined in Super
```

Predefined Properties on All Objects

- Several properties apply to all objects
  - ◇ `oclIsTypeOf(t: Type): Boolean` is `true` if the type of `self` and `t` are the same
  - ◇ `oclIsKindOf(t: Type): Boolean` is `true` if `t` is a direct/indirect type of `self`
  - ◇ `oclInState(s: State): Boolean` is `true` if `self` is in the state `s`
  - ◇ `oclIsNew: Boolean`, in a postcondition, is `true` if `self` has been created while performing the operation
- Example

```
context Person
  inv: self.oclIsTypeOf(Person)  -- is true
  inv: self.oclIsTypeOf(Company) -- is false
```

Class Features

- Features of a **class**, not of its instances
- They are either used-defined or predefined
- Predefined feature `allInstances` holds on all types
- There are at most 100 persons

```
context Person inv:
  Person.allInstances()->size() <= 100
```

- A user-defined feature `averageAge` of class `Person`

```
context Person inv:
  Person.averageAge =
    Person.allInstances()->collect(age)->sum()/
    Person.allInstances()->size()
```

Select Operation on a Collection

- Obtains the subset of elements of a collection satisfying a Boolean expression
- Alternative expressions for the `select` operation
  - ◇ `collection->select(Boolean-expression)`
  - ◇ `collection->select(v | Boolean-expression-with-v)`
  - ◇ `collection->select(v: Type | Boolean-expression-with-v)`
- A company has at least one employee older than 50

```
context Company inv:
    self.employee->select(age > 50)->notEmpty()
context Company inv:
    self.employee->select(p | p.age > 50)->notEmpty()
context Company inv:
    self.employee->select(p: Person | p.age > 50)->notEmpty()
```

Reject Operation on a Collection

- Obtains the subset of all elements of the collection for which a Boolean expression evalutes to `False`
- Alternative expressions for the `reject` operation
  - ◇ `collection->reject(Boolean-expression)`
  - ◇ `collection->reject(v | Boolean-expression-with-v)`
  - ◇ `collection->reject(v: Type | Boolean-expression-with-v)`
- The collection of employees of a company who have **not** at least 18 years old is empty

```
context Company inv:
    self.employee->reject(age>=18)->isEmpty()
```

- A `reject` expression can always be restated as a `select` with the negated expression

Collect Operation on a Collection

- Derives a collection from another collection, but which contains different objects from the original collection
- Alternative expressions for the `collect` operation
  - `collection->collect(expression)`
  - `collection->collect(v | expression-with-v)`
  - `collection->collect(v: Type | expression-with-v)`
- Collect of birth dates for all employees in the context of a `Company` object
  - `self.employee->collect(birthDate)`
  - `self.employee->collect(p | p.birthDate)`
  - `self.employee->collect(p:Person | p.birthDate)`
- Resulting collection above is a `Bag`: some employees may have the same birth date

ForAll Operation on a Collection

- Specifies a Boolean expression that must be true for **all elements** in a collection
- Alternative expressions for the `forall` operation
  - `collection->forAll(Boolean-expression)`
  - `collection->forAll(v | Boolean-expression-with-v)`
  - `collection->forAll(v: Type | Boolean-expression-with-v)`
- The `age` of each employee is less than or equal to `65`

```
context Company
  inv: self.employee->forAll(age <= 65)
  inv: self.employee->forAll(p | p.age <= 65)
  inv: self.employee->forAll(p: Person | p.age <= 65)
```

- More than one iterator can be used in the `forAll` operation
- All instances of persons have unique names

```
context Person inv:
  Person.allInstances()->forAll(p1, p2 |
          p1 <> p2 implies p1.name <> p2.name )
```

Exists Operation on a Collection

- Specifies a Boolean expression that must be true for **at least one element** in a collection
- Alternative expressions for the `exists` operation
  - ◇ `collection->exists(Boolean-expression)`
  - ◇ `collection->exists(v | Boolean-expression-with-v)`
  - ◇ `collection->exists(v: Type | Boolean-expression-with-v)`
- The `firstName` of at least one employee is equal to `‘Jack’`

```
context Company
  inv: self.employee->exists(firstName = ‘Jack’)
  inv: self.employee->exists(p | p.firstName = ‘Jack’)
  inv: self.employee->exists(p: Person | p.firstName = ‘Jack’)
```

Iterate Operation on a Collection

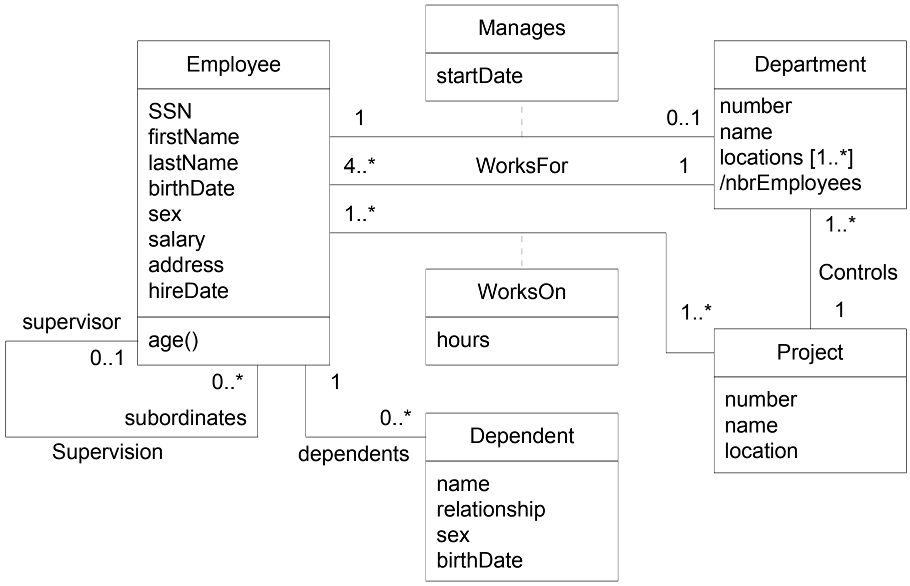
- Provides a generic mechanism to iterate over a collection
- Syntax

```
collection->iterate(elem: Type; acc: Type = <expression> |
   expression-with-elem-and-acc)
```

  - ◇ `elem`: the iterator as in `select`, `forAll`, etc.
  - ◇ `acc`: the accumulator with an initial value `<expression>`
  - ◇ `expression-with-elem-and-acc`: is evaluated for each `elem` and its value is assigned to `acc`
- The operations `select`, `reject`, `forAll`, `exists`, `collect`, can all be described in terms of `iterate`
- For example, `collection->collect(x: T | x.property)` is identical to

```
collection->iterate(x: T; acc: T2 = Bag{} |
   acc->including(x.property))
```

Company Example: Class Diagram


Manages
startDate
Employee
SSN
firstName
lastName
birthDate
sex
salary
address
hireDate
age()
Department
number
name
locations [1..*]
/nbrEmployees
1
0..1
4..*
WorksFor
1
1..*
WorksOn
hours
1..*
Controls
1
1..*
Project
number
name
location
supervisor
0..1
0..*
subordinates
Supervision
1
0..*
dependents
Dependent
name
relationship
sex
birthDate

Company Example: Integrity Constraints (1)

- The age of employees must be greater than or equal to 18

```
context Employee inv:
  self.age() >= 18
```

- The supervisor of an employee must be older than the employee

```
context Employee inv:
  self.supervisor->notEmpty() implies
    self.age() > self.supervisor.age()
```

The condition `notEmpty` must be tested since the multiplicity of the role is not mandatory

- The salary of an employee cannot be greater than the salary of his/her supervisor

```
context Employee inv:
  self.supervisor->notEmpty() implies
    self.salary < self.supervisor.salary
```

- The hire date of employees must be greater than their birth date

```
context Employee inv:
  self.hireDate > self.birthDate
```

Company Example: Integrity Constraints (2)

- The start date of an employe as manager of a department must be greater than his/her hire date

```
context Employee inv:
  self.manages->notEmpty() implies
  self.manages.startDate > self.hireDate
```

- A supervisor must be hired before every employee s/he supervises

```
context Employee inv:
  self.subordinates->notEmpty() implies
  self.subordinates->forall( e | e.hireDate > self.hireDate )
```

- The manager of a department must be an employee of the department

```
context Department inv:
  self.worksFor->includes(self.manages.employee)
```

- The SSN of employees is an identifier (or a key)

```
context Employee inv:
  Employee.allInstances->forAll( e1, e2 |
    e1 <> e2 implies e1.SSN <> e2.SSN )
```

Company Example: Integrity Constraints (3)

- The name and relationship of dependents is a partial identifier: they are unique among all dependents of an employee

```
context Employee inv:
  self.dependents->notEmpty() implies
  self.dependents->forAll( e1, e2 | e1 <> e2 implies
    ( e1.name <> e2.name or e1.relationship <> e2.relationship ))
```

- The location of a project must be one of the locations of its department

```
context Project inv:
  self.controls.locations->includes(self.location)
```

- The attribute nbrEmployees in Department keeps the number of employees that works for the department

```
context Department inv:
  self.nbrEmployees = self.worksFor->size()
```

- An employee works at most in 4 projects

```
context Employee inv:
  self.worksOn->size() <= 4
```

Company Example: Integrity Constraints (4)

- An employee may only work on projects controlled by the department in which s/he works

```
context Employee inv:
  self.worksFor.controls->includesAll(self.worksOn.project)
```

- An employee works at least 30h/week and at most 50 h/week on all its projects

```
context Employee inv:
  let totHours: Integer = self.worksOn->collect(hours)->sum() in
  totHours >= 30 and totHours <=50
```

- A project can have at most 2 employees working on the project less than 10 hours

```
context Project inv:
  self.worksOn->select( hours < 10 )->size() <= 2
```

Company Example: Integrity Constraints (5)

- Only department managers can work less than 5 hours on a project

```
context Employee inv:
  self.worksOn->select( hours < 5 )->notEmpty() implies
  Department.allInstances()->collect(manages.employee)->
    includes(self)
```

  If the manager of a department must be an employee of the department (previous contraint), this constraint can be specified as follows

```
context Employee inv:
  self.worksOn->select( hours < 5 )->notEmpty() implies
  self.worksFor.manages.employee=self
```

- Employees without subordinates must work at least 10 hours on every project they work

```
context Employee inv:
  self.subordinates->isEmpty() implies
  self.worksOn->forAll( hours >=10 )
```

Company Example: Integrity Constraints (6)

- The manager of a department must work at least 5 hours on all projects controlled by the department

```
context Department inv:
  self.controls->forall( p:Project | self.manages.
    employee.worksOn->select(hours >= 5)->contains(p) )
```

- An employee cannot supervise him/herself

```
context Employee inv:
  self.subordinates->excludes(self)
```

- The supervision relationship must not be cyclic

```
context Employee
  def: allSubordinates = self.subordinates->union(
    self->subordinates->collect(e:Employee | e.allSubordinates))
  inv: self.allSubordinates->exludes(self)
```