OCL Expressions

- Context
- Basic Elements
- Collection Types
- Summary

What is context?

The context of an OCL expression specifies the entity in the UML model for which the expression is defined.

Usually, the context is a class, interface, data type, component, or method.

Contextual Type

The contextual type is the type of the object for which the expression will be evaluated.

If the context itself is a type, the context is equal to the contextual type. If the context is an operation or attribute, the contextual type is the type for which the feature is defined.

Contextual Instance

An OCL expression is always evaluated for a single instance of the contextual type, called the contextual instance.

The contextual instance can be referred to by the keyword self, which is often omitted if obvious.

```
context Customer
inv: self.name = 'Edward'
```

Context - Classes

When the context is a class (or a classifier), the following types of OCL expressions can be used:

- ❑ Invariants
- ❑ Definitions of new attributes
- ❑ Definitions of new operations

Invariants

An invariant is a constraint that must hold for any instance upon completion of the constructor and completion of every public operation.

Note that an invariant does not have to hold during the execution of operations.

```
context Customer
inv myInvariant23: self.name = 'Edward'
```

New attributes

New attributes can be added to a class definition using OCL expressions.

```
context Customer
def: initial : String = name.substring(1, 1)
```

New Operations

Similarly, new operations can be added to a class definition. Note that all operations defined by OCL must be query operations.

```
context CustomerCard
def: getTotalPoints (d: Date) : Integer =
         transactions -> select( date.isAfter(d) ).points -> sum ()
```

Context – Attributes

When the context is an attribute, the following expressions can be used:

- ❑ Derivation rules
- ❑ Initial values

Derivation Rules

A derivation rule specifies that the value of the context element should always be equal to the value given by the evaluation of the rule.

```
context LoyaltyAccount::totalPointsEarned : Integer
derive: transactions -> select( oclIsTypeOf (Earning) ).points -> sum ()
```

Initial Values

An initial value is the value that the attribute or association end will have at the moment that the contextual instance is created.

```
context CustomerCard::valid : Boolean
init:  true
```

Context - Operations

When the context is an operation, the following types of OCL expressions can be used:

- ❑ Pre-/Post- conditions
- ❑ Body of query operations

Pre-/Post-conditions

A pre-condition is a boolean expression that must hold at the moment when the operation starts the execution;.

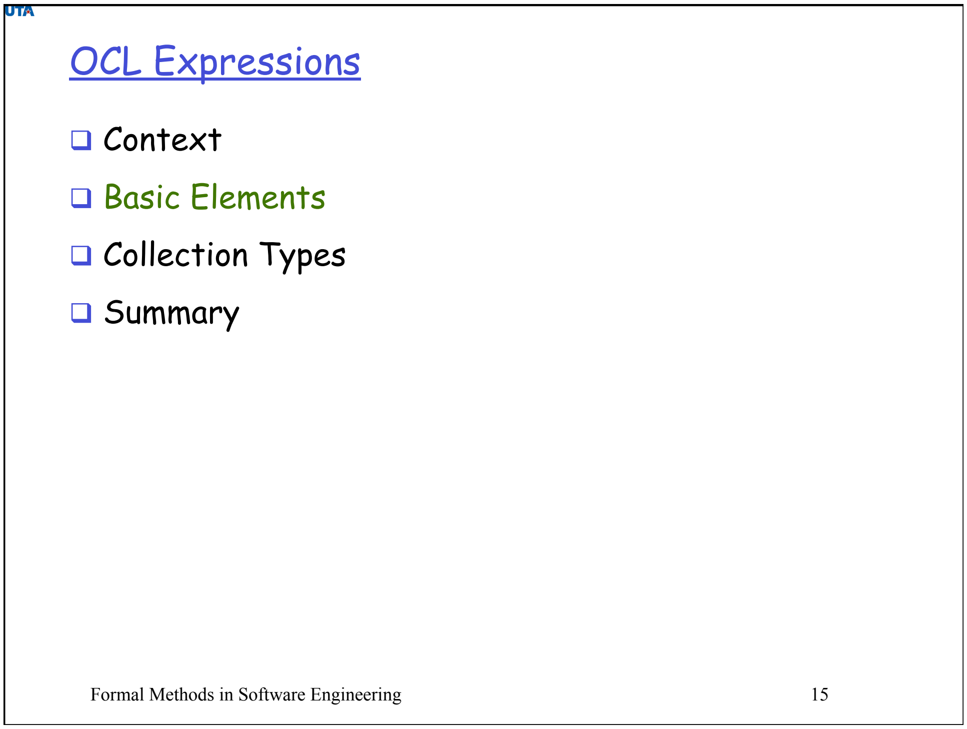
A post-condition is a boolean expression that must hold when the operation ends its execution.

```
context LoyaltyProgram::enroll (c: Customer)
pre:  c.name <> ''
post: participants -> including (c)
```

Body of Query Operations

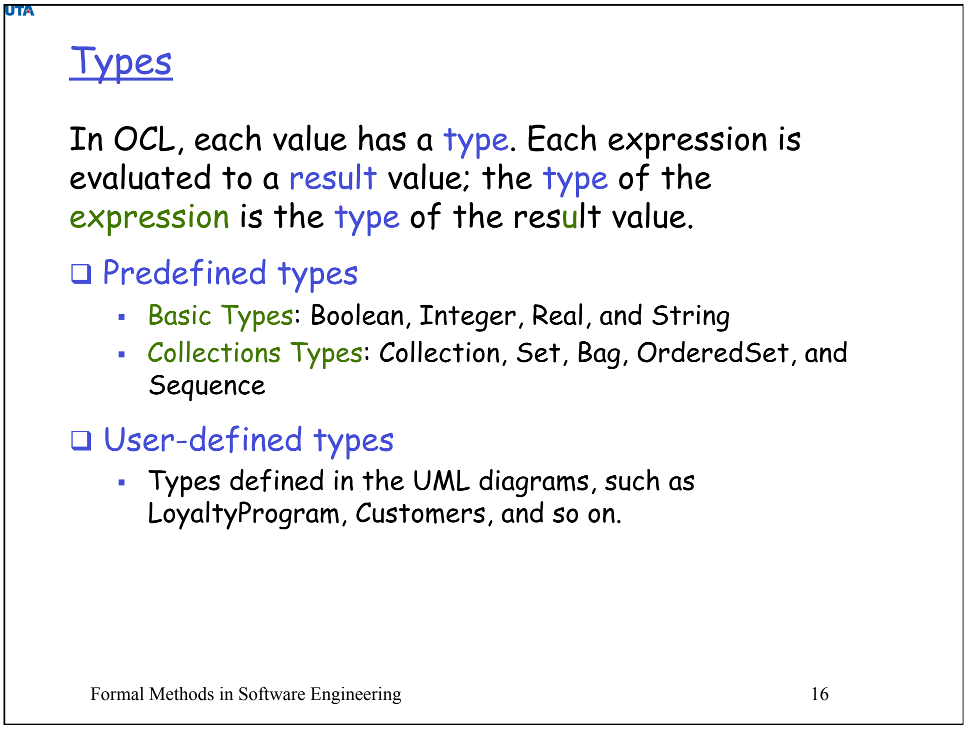
Query operations can be fully defined by specifying the result of the operation in a single expression.

```
context CustomerCard::getTransactions (
            from : Date, until : Date) : Set (Transaction)
body:  transactions -> select (date.isAfter(from) and
                              date.isBefore(until))
```



OCL Expressions

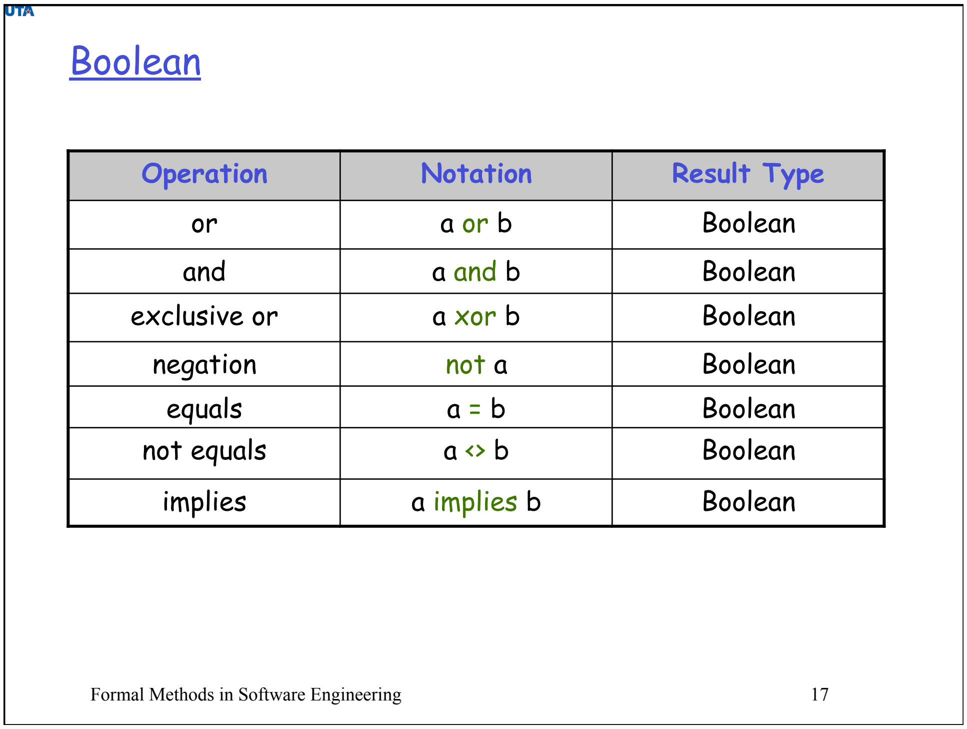
- ❑ Context
- ❑ Basic Elements
- ❑ Collection Types
- ❑ Summary



Types

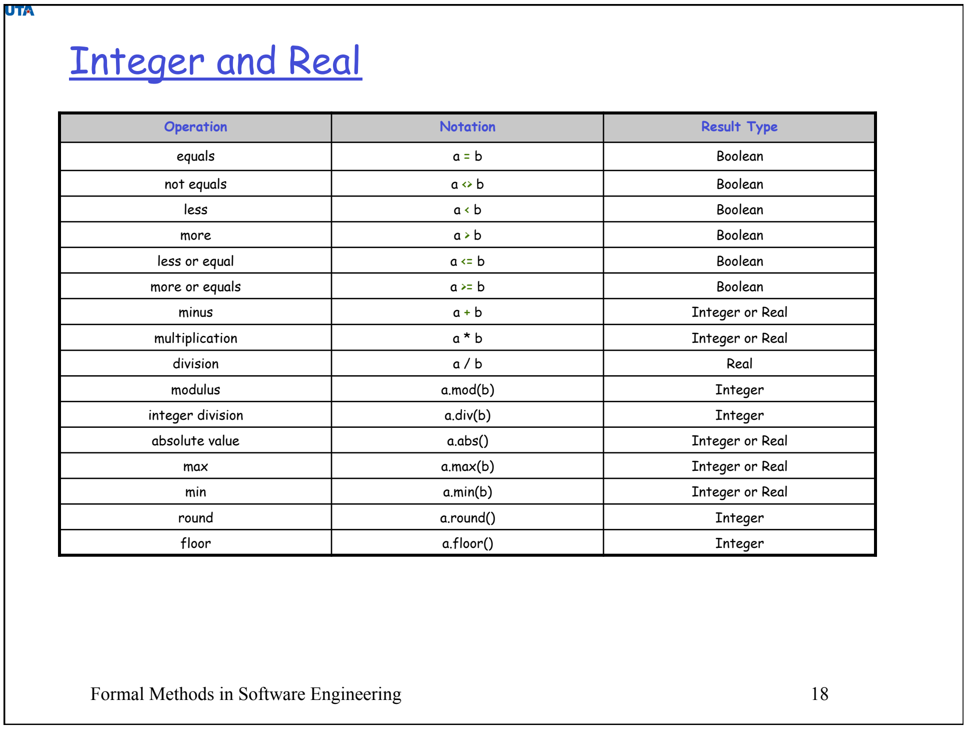
In OCL, each value has a type. Each expression is evaluated to a result value; the type of the expression is the type of the result value.

- ❑ Predefined types
  - Basic Types: Boolean, Integer, Real, and String
  - Collections Types: Collection, Set, Bag, OrderedSet, and Sequence
- ❑ User-defined types
  - Types defined in the UML diagrams, such as LoyaltyProgram, Customers, and so on.



Boolean

Operation	Notation	Result Type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean



Integer and Real

Operation	Notation	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equals	a >= b	Boolean
minus	a + b	Integer or Real
multiplication	a * b	Integer or Real
division	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs()	Integer or Real
max	a.max(b)	Integer or Real
min	a.min(b)	Integer or Real
round	a.round()	Integer
floor	a.floor()	Integer

String

Operation	Notation	Result Type
concatenation	s1.concat(s2)	String
size	s.size ()	Integer
to lower case	s.toLower ()	String
to upper case	s.toUpper ()	String
substring	s.substring(i, j)	String
equals	s1 = s2	Boolean
not equals	s1 <> s2	Boolean

User-defined Type

A user-defined type is a classifier specified in the UML model.

An OCL expression can refer to the features of a user-defined type, including attributes, operations, class attributes, class operations, and association ends.

Attributes and Operations

Attributes and query operations of a user-defined type can be used in OCL expressions.

They are both referenced using the dot notation. In order to distinguish them, the parentheses after the name of an operation is required.

Class Attributes and Operations

A class attribute or operation is referenced by the class name followed by two colons, followed by the attribute or operation name (and parameters).

```
context CustomerCard
inv: goodThru.isAfter( Date::now)
```

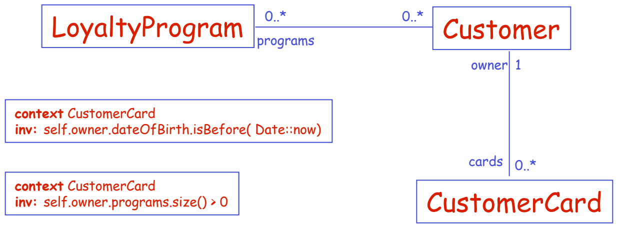
Navigation (1)

Association ends, also called navigations, can be used to navigate from one object in the model to another.

Navigations are treated as attributes, and are referenced using the dot notation. Note that the name of a navigation is the rolename or the name of the type connected at the end.

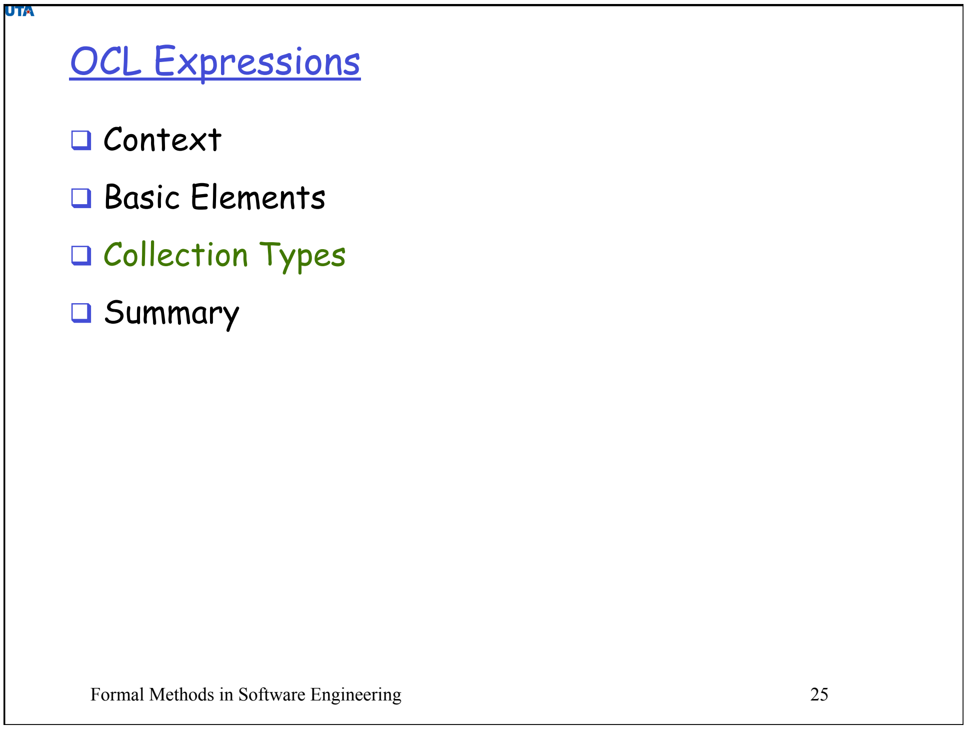
Navigation (2)

The type of a navigation is either a user-defined type or a collection of a user-defined types, depending on the multiplicity.



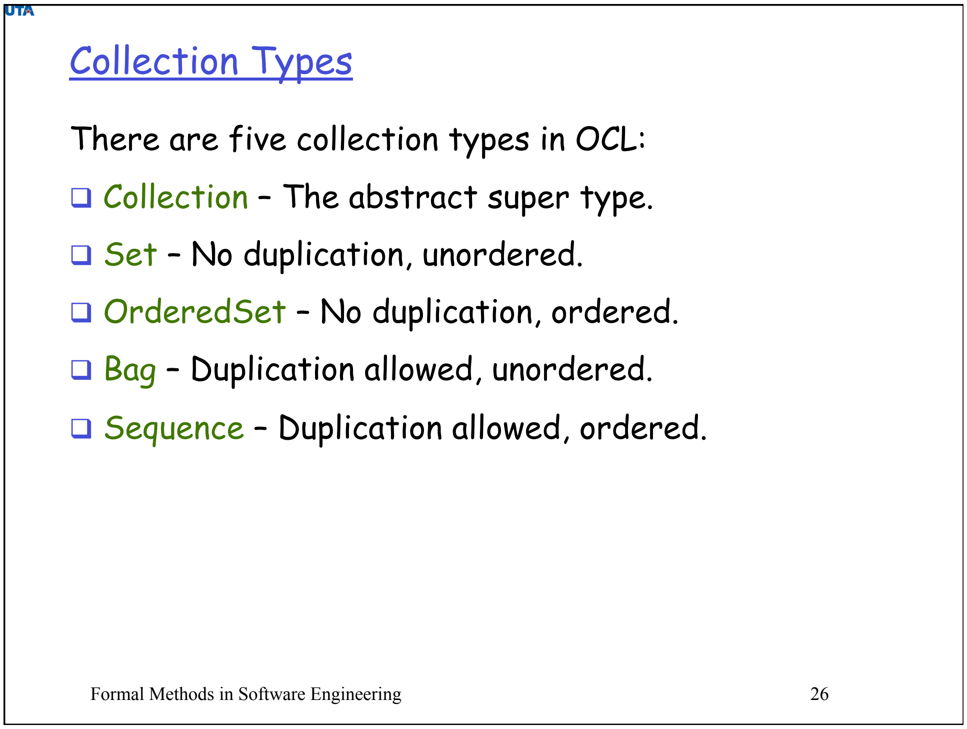
context CustomerCard
inv: self.owner.dateOfBirth.isBefore(Date::now)

context CustomerCard
inv: self.owner.programs.size() > 0



OCL Expressions

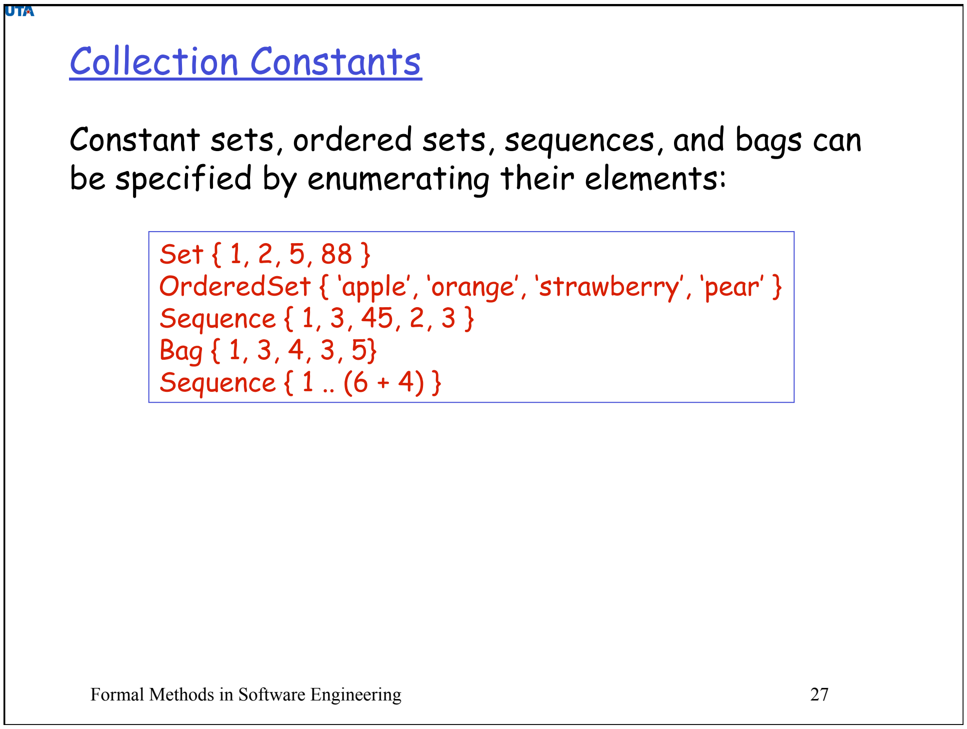
- ❑ Context
- ❑ Basic Elements
- ❑ Collection Types
- ❑ Summary



Collection Types

There are five collection types in OCL:

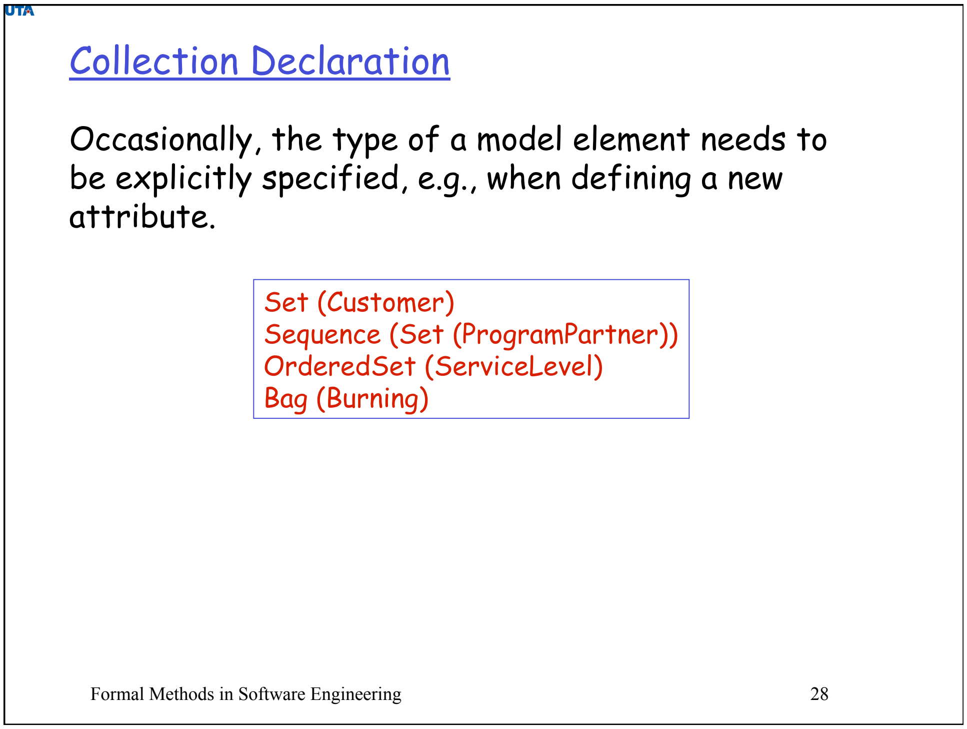
- ❑ Collection – The abstract super type.
- ❑ Set – No duplication, unordered.
- ❑ OrderedSet – No duplication, ordered.
- ❑ Bag – Duplication allowed, unordered.
- ❑ Sequence – Duplication allowed, ordered.



Collection Constants

Constant sets, ordered sets, sequences, and bags can be specified by enumerating their elements:

```
Set { 1, 2, 5, 88 }
OrderedSet { 'apple', 'orange', 'strawberry', 'pear' }
Sequence { 1, 3, 45, 2, 3 }
Bag { 1, 3, 4, 3, 5}
Sequence { 1 .. (6 + 4) }
```



Collection Declaration

Occasionally, the type of a model element needs to be explicitly specified, e.g., when defining a new attribute.

```
Set (Customer)
Sequence (Set (ProgramPartner))
OrderedSet (ServiceLevel)
Bag (Burning)
```

Collections of Collections

In most cases, collections are automatically flattened.

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

Collection Operations

Many standard collection operations are defined in OCL. They are denoted in OCL expressions using an arrow.

Important: The user-defined operations are denoted using dots; the standard operations are denoted using arrows.

```
context LoyaltyProgram
inv: self.participants -> size () < 10000
```

Standard Operations

Operation	Description
count (object)	The number of occurrences of the object in the collection
excludes (object)	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 one or more element
size ()	The number of elements in the collection
sum ()	The addition of all elements in the collection. The elements must be of a type supporting addition (such as Real or Integer)

Other Operations (1)

- equals (=) and notEquals (<>)
  - Two sets equal iff all elements are the same.
  - Two ordered sets equal iff all elements are the same, and appear in the same order.
  - Two bags equal iff all elements must appear in both with the same number of times.
  - Two sequences equal iff all elements must appear in both with the same number of times, and in the same order

Other Operations (2)

❑ including(object)

- including results in a new collection with object being added;
- For a set or ordered set, object is added only if it is not already present.
- For a sequence or an ordered set, the element is added in the end.

❑ excluding(object)

- excluding results in a new collect with object being removed.
- For a bag or sequence, it removes all the occurrences of the given object.

Other Operations (3)

❑ flatten - changes a collection of collections into a collection of single objects.

- The result of flattening a bag or set is a bag or set, respectively.
- The result of flattening a sequence or ordered set is a sequence or ordered set, respectively.
- However, if the subcollections are bags or sets, the order of the elements cannot be determined precisely.

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

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

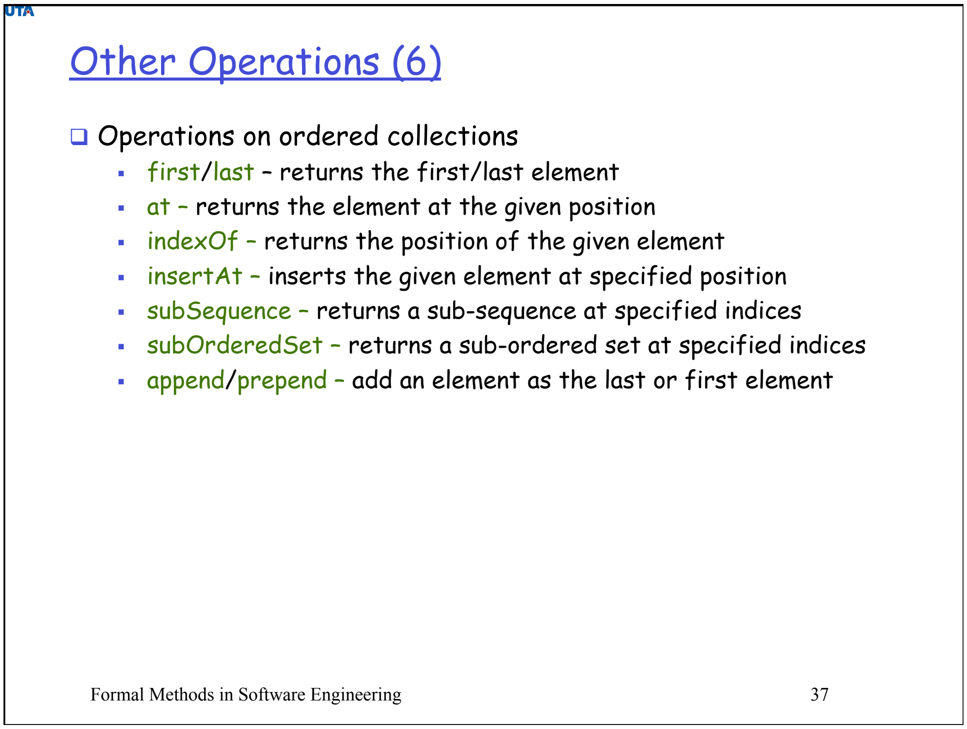
```
Sequence { Set { 1, 2 }, Set { 2, 3}, Set { 4, 5, 6 } }
Sequence { 2, 1, 2, 3, 4, 5, 6 }
```

Other Operations (4)

- asSet, asSequence, asBag, and asOrderedSet
  - These operations transform instances of one collection type to another.
  - Applying asSet on a bag or asOrderedSet on a sequence will remove duplicate elements.
  - Appying asSet on an OrderedSet or asBag on a sequence will lose the ordering info..
  - Applying asOrderedSet or asSequence on a set or bag will place elements randomly.

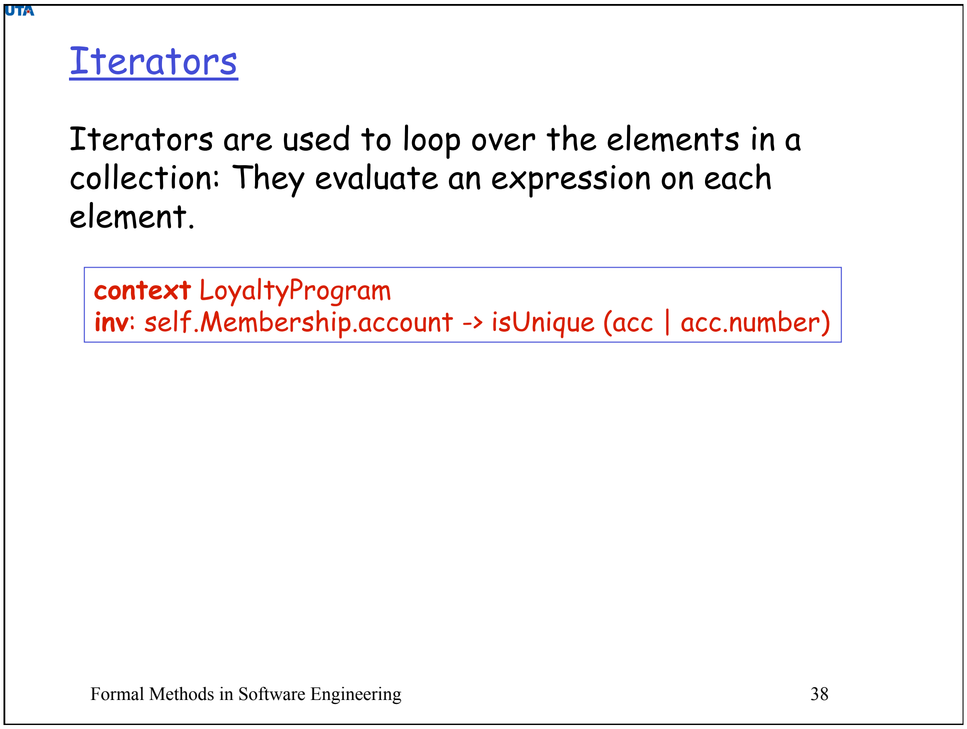
Other Operations (5)

- union – combines two collections into one
  - Ordered collections can only combined with order collections.
  - Combining a set with a bag will result in a bag.
- intersection – intersects two collections
  - cannot be applied to ordered collection
- minus – the difference between two collections
  - When applied to an ordered set, the ordering remains.



Other Operations (6)

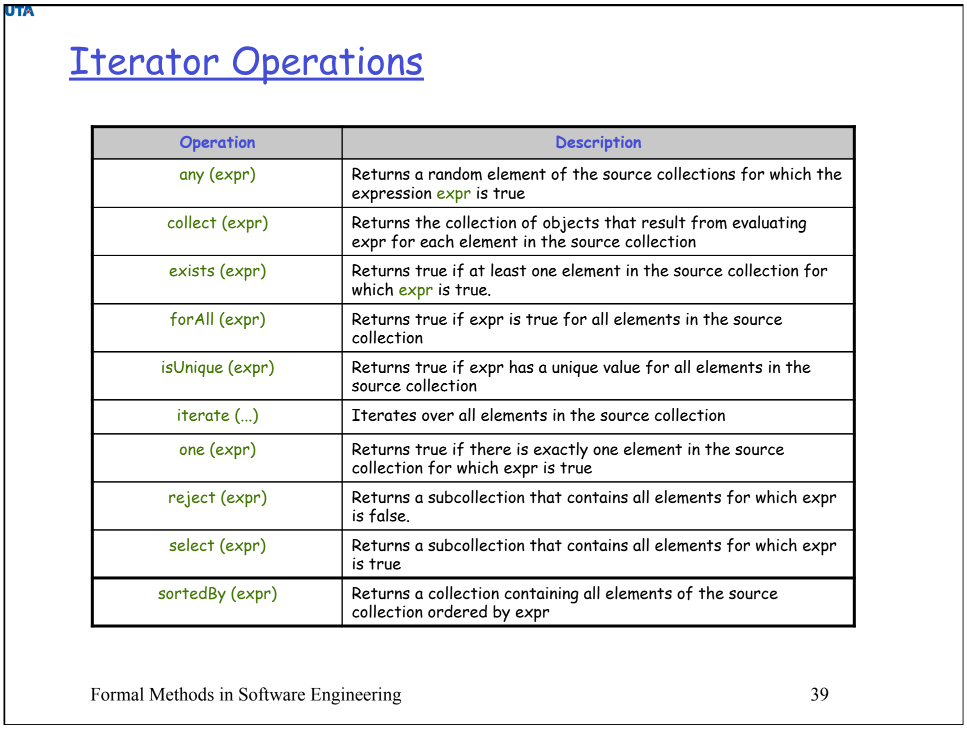
- Operations on ordered collections
  - first/last – returns the first/last element
  - at – returns the element at the given position
  - indexOf – returns the position of the given element
  - insertAt – inserts the given element at specified position
  - subSequence – returns a sub-sequence at specified indices
  - subOrderedSet – returns a sub-ordered set at specified indices
  - append/prepend – add an element as the last or first element



Iterators

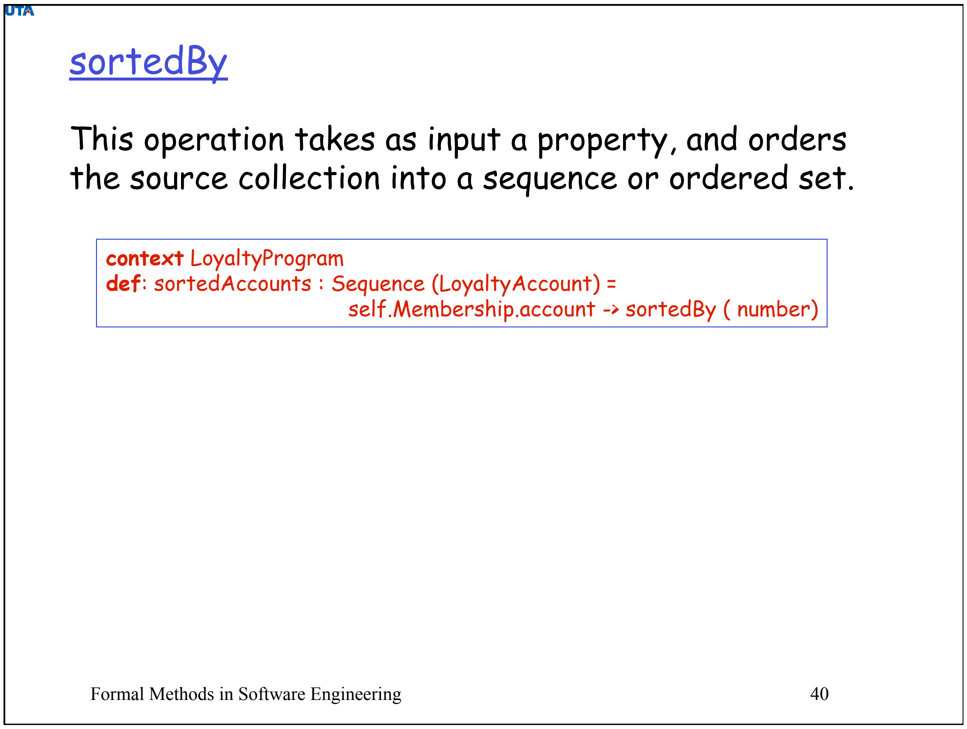
Iterators are used to loop over the elements in a collection: They evaluate an expression on each element.

```
context LoyaltyProgram
inv: self.Membership.account -> isUnique (acc | acc.number)
```



Iterator Operations

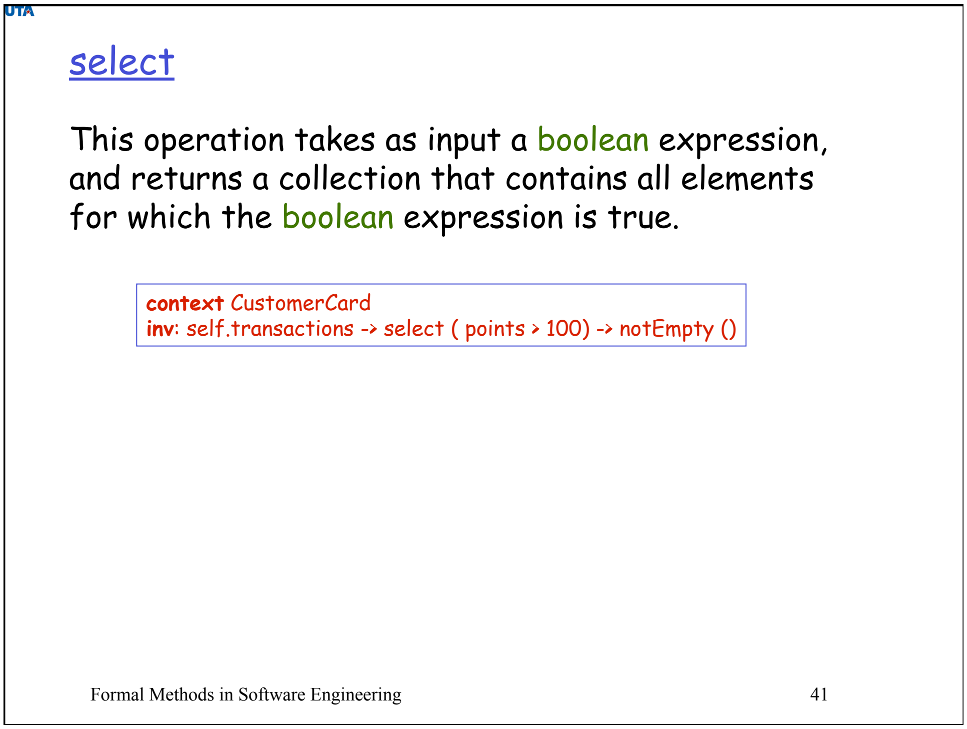
Operation	Description
any (expr)	Returns a random element of the source collections for which the expression expr is true
collect (expr)	Returns the collection of objects that result from evaluating expr for each element in the source collection
exists (expr)	Returns true if at least one element in the source collection for which expr is true.
forAll (expr)	Returns true if expr is true for all elements in the source collection
isUnique (expr)	Returns true if expr has a unique value for all elements in the source collection
iterate (…)	Iterates over all elements in the source collection
one (expr)	Returns true if there is exactly one element in the source collection for which expr is true
reject (expr)	Returns a subcollection that contains all elements for which expr is false.
select (expr)	Returns a subcollection that contains all elements for which expr is true
sortedBy (expr)	Returns a collection containing all elements of the source collection ordered by expr



sortedBy

This operation takes as input a property, and orders the source collection into a sequence or ordered set.

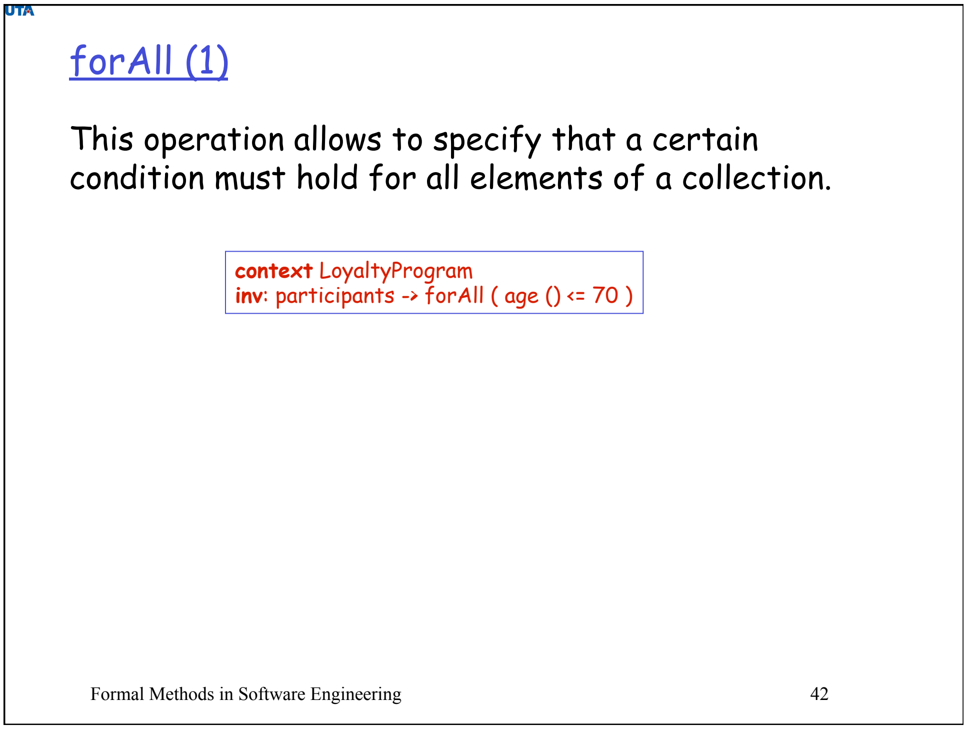
```
context LoyaltyProgram
def: sortedAccounts : Sequence (LoyaltyAccount) =
                        self.Membership.account -> sortedBy ( number)
```



select

This operation takes as input a boolean expression, and returns a collection that contains all elements for which the boolean expression is true.

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



forAll (1)

This operation allows to specify that a certain condition must hold for all elements of a collection.

```
context LoyaltyProgram
inv: participants -> forAll ( age () <= 70 )
```

forAll (2)

In this operation, multiple iterator variables can be declared. The following two operations are equivalent:

```
context LoyaltyProgram
inv: participants -> forAll ( c1, c2 |
                        c1 <> c2 implies c1.name <> c2.name)
```

```
context LoyaltyProgram
inv: participants -> forAll ( c1 |
          participants -> forAll ( c2 |
                        c1 <> c2 implies c1.name <> c2.name))
```

exists

This operation allows to specify that a certain condition must hold for at least one element of a collection.

```
context LoyaltyAccount
inv: points > 0 implies transactions -> exists ( t | t.points > 0 )
```

collect (1)

This operation iterates over a collection, computes a value for each element, and gathers these values into a new collection.

```
context LoyaltyAccount
inv: transactions -> collect ( points ) ->
                                        exists ( p : Integer | p = 500 )
```

collect (2)

Taking a property of a collection using a dot is interpreted as applying the collect operation.

```
context LoyaltyAccount
inv: transactions.points -> exists ( p : Integer | p = 500 )
```

iterate (1)

This operation is the most generic one, in the sense that all other loop operations can be described as a special case of iterate.

```
collection -> iterate ( element : Type1;
                        result : Type2 = <expression>
                       | <expression-with-element-and-result> )
```

iterate (2)

```
Set { 1, 2, 3 } -> iterate ( i : Integer, sum : Integer = 0 | sum + i)
```

```
context ProgramPartner
def: getBurningTransactions () : Set (Transaction) =
       deliveredServices.transactions -> iterate (
           t : Transaction; resultSet : Set (Transaction) = Set {}
         | if t.oclIsTypeOf ( Burning ) then
               resultSet.including ( t )
           else
               resultSet
            endif
```

OCL Expressions

- Context
- Basic Elements
- Collection Types
- Summary

Summary

- The context of an OCL expression can be a class, interface, attribute, or operation.
- An OCL type can be a predefined type (basic or collection type) or a user-defined type.
- Navigations are used to navigate from one object in the UML model to another.
- There are four types of concrete collections: Set, OrderedSet, Bag, and Sequence.
- Many operations are predefined to manipulate these collection types.