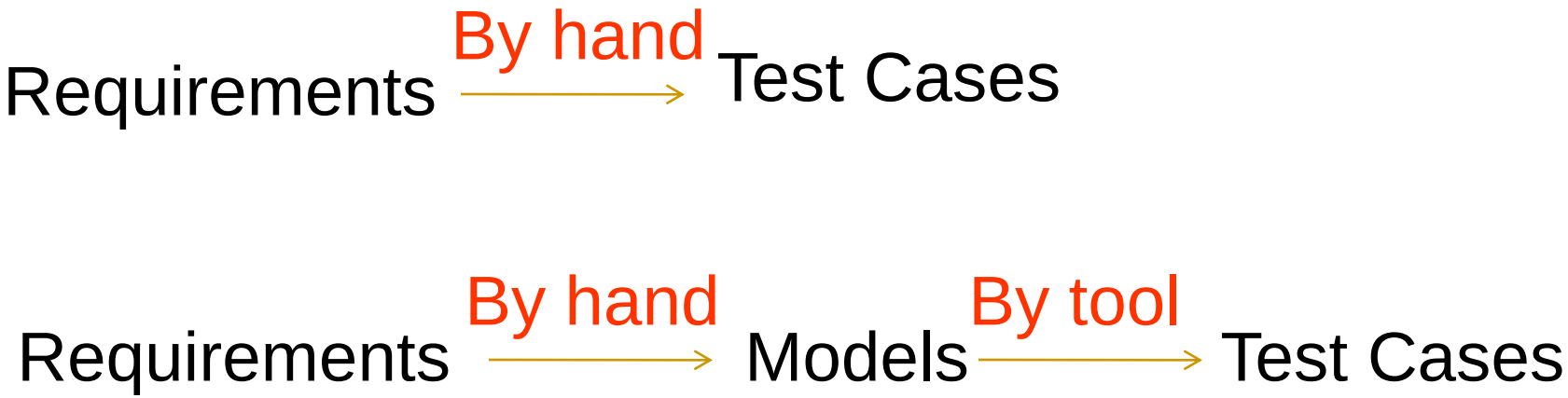
Model-Based Testing

Content

- Introduction to Model-Based Testing
- Introduction to Design by Contract
- Introduction to Universal Modeling Language (UML)
- Introduction to Object Constraint Language (OCL)

Model-Based Testing



Benefits of Model-Based Testing

- Cost saving in test design
  - Modeling time < manual test design time
- Systematic testing
  - Less ad-hoc, systematic coverage control
- Modeling benefits
  - Early detection of specification ambiguities
- Quick response to evolving requirements
  - Change model → regenerate tests
- Automated traceability
  - Requirements <-> tests

Design by Contract

- Design by Contract is a systematic approach to designing software.
- A software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations — contracts.
- It includes specifications for each method of a class and specifications for each object.

Specification of Methods

- Precondition: state the conditions that must be true before the method can work correctly.
- Impose an obligation to be guaranteed on entry by any client that calls this method.
- Provide a benefit for the supplier (the method itself), as it frees this method from having to handle cases outside of the precondition.

Specification of Methods

- Postcondition: state the conditions that will be true after execution, if the method has worked correctly.
- Impose an obligation to be guaranteed on exit by the supplier.
- Provide a benefit for the client.

Specification of Objects

- Class invariant: state the conditions that must be true for all objects of the class at any time.

Contract for a Class

- Precondition and postcondition for each (public) member method.
- Invariants for member variables.

An Example: Class Triangle

```
class Triangle
{
    public Triangle(int sa, int sb, int sc);
    public String category( );
    private int a;
    private int b;
    private int c;
};
```

An Example: Constructor Triangle( )

Triangle::Triangle(int sa, int sb, int sc)
precondition:
    sa + sb > sc and sa + sc > sb and sb + sc > sa
postcondition:
    a = sa and b = sb and c = sc

An Example: Method category( )

```
Triangle::category( ): string
precondition: none
postcondition:
result =
    if (a = b and a = c) then
       “Equilateral”
    else if (a = b or a = c or b = c)  then
       “Isosceles”
    else
       “Scalene”
    endif
```

An Example: Triangle Objects

Triangle

invarient:

a + b > c and a + c > b and b + c > a

What is UML

- The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.
- The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

UML Diagrams

- UML 2 defines thirteen basic diagram types, divided into two general sets:
- Structural Modeling Diagrams: Structure diagrams define the static architecture of a model. They are used to model the 'things' that make up a model.
- Behavioral Modeling Diagrams: Behavior diagrams capture the varieties of dynamic interaction and instantaneous state within a model as it 'executes' over time.

Structural Modeling Diagrams

- Package diagrams
- Class diagrams
- Object diagrams
- Composite diagrams
- Component diagrams
- Deployment diagrams

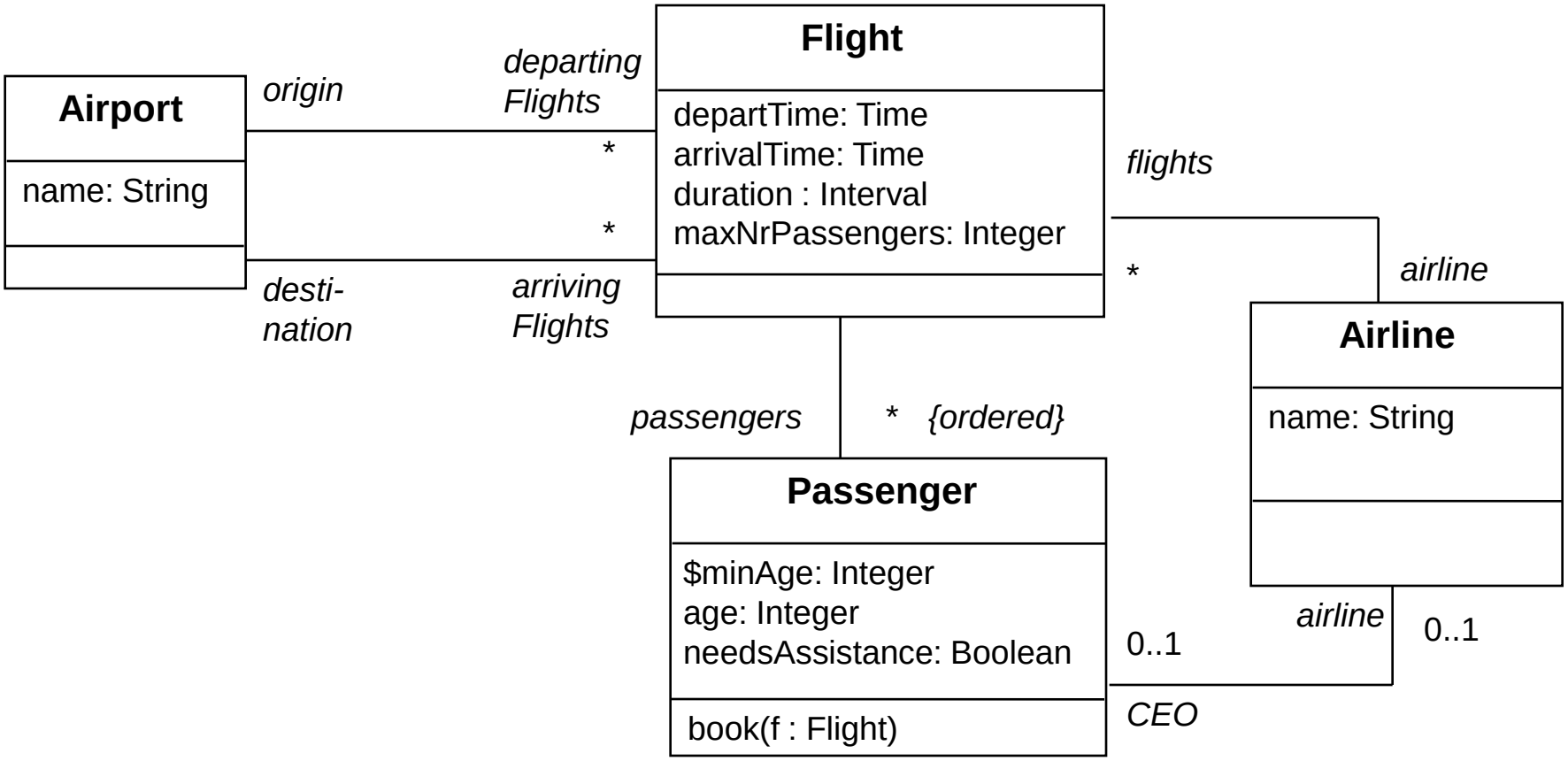
Behavioral Modeling Diagrams

- Use Case diagrams
- Activity diagrams
- State Machine diagrams
- Communication diagrams
- Sequence diagrams
- Timing diagrams
- Interaction Overview diagrams

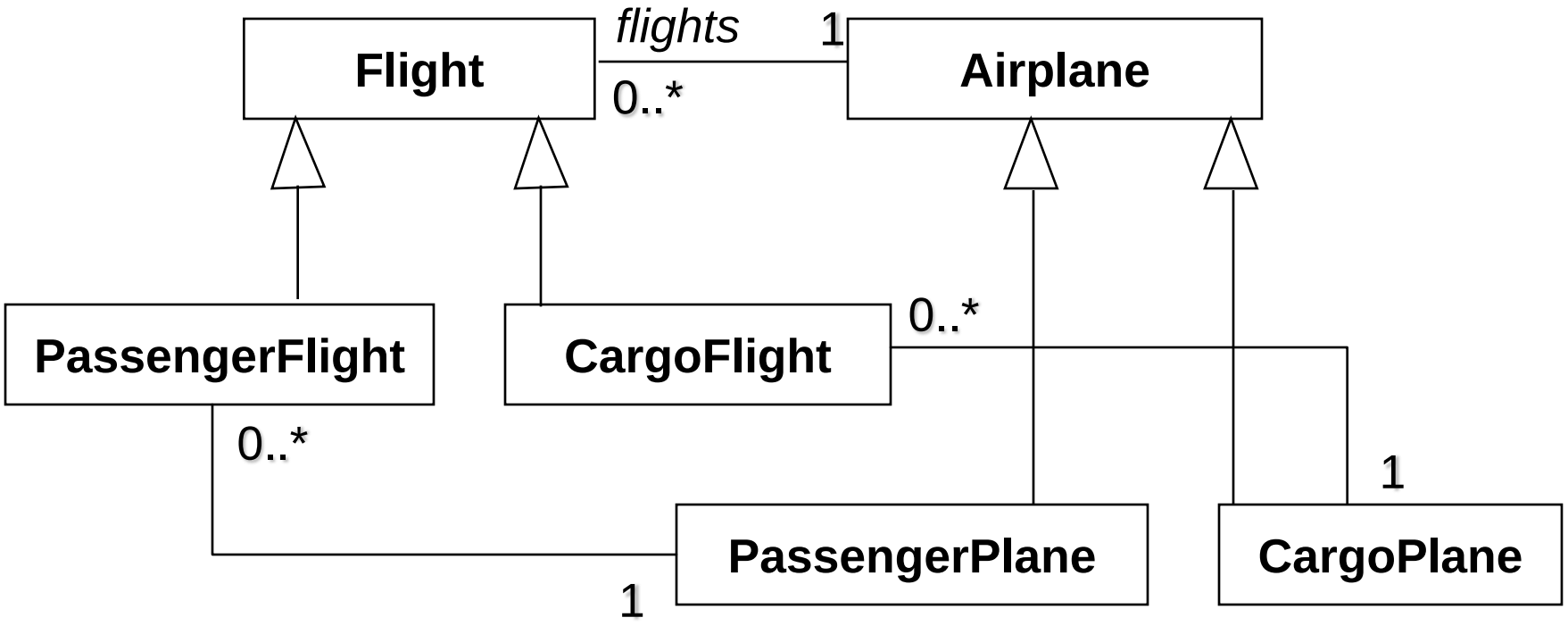
Class Diagrams

- A class diagram describes what attributes and operations a model has rather than detailing the methods for achieving operations.
- Class diagrams are most useful in illustrating relationships between classes and interfaces.
- Associations, generalizations, and aggregations are valuable in reflecting connection, inheritance and composition, respectively.

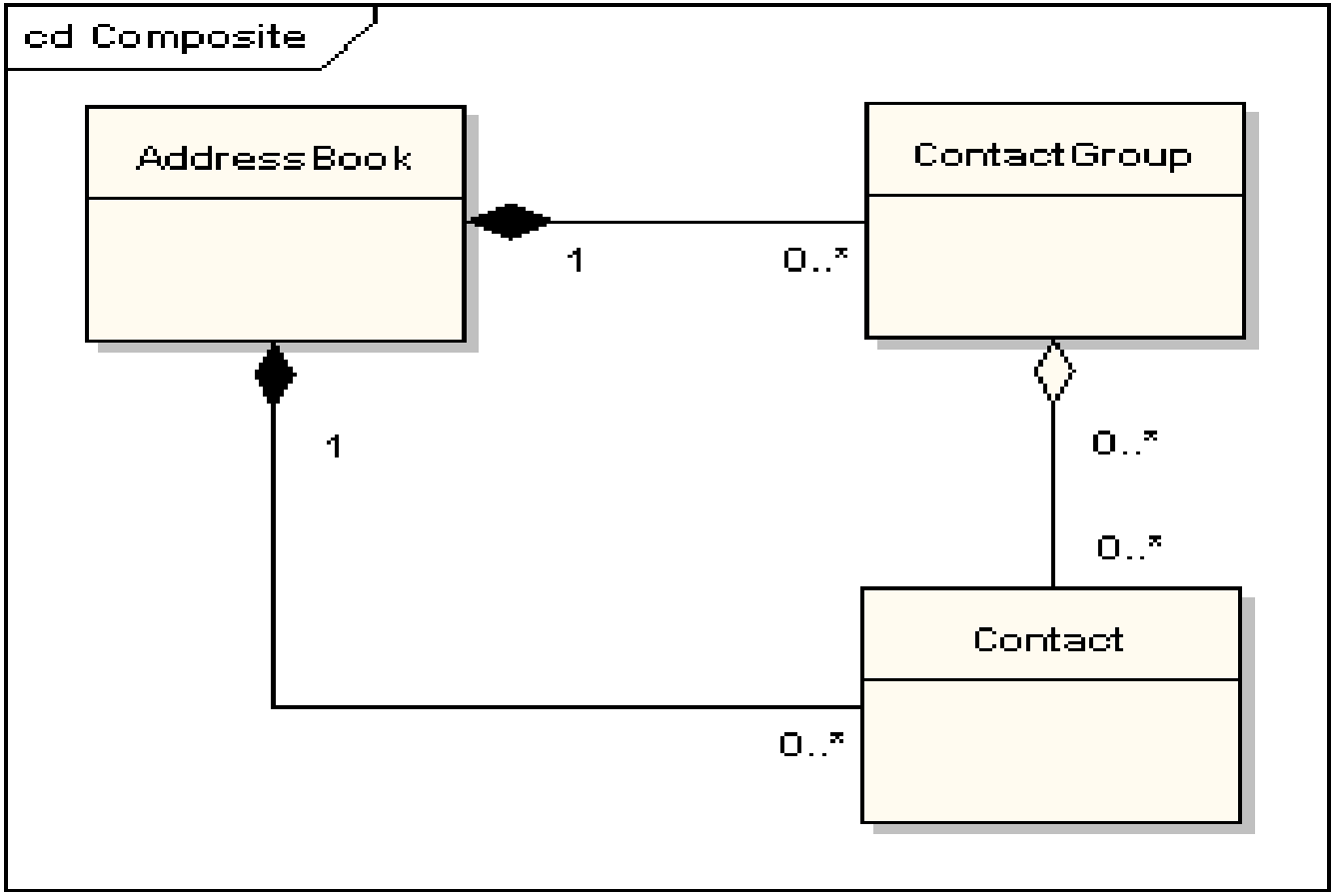
An Example: Association



An Example: Generalization



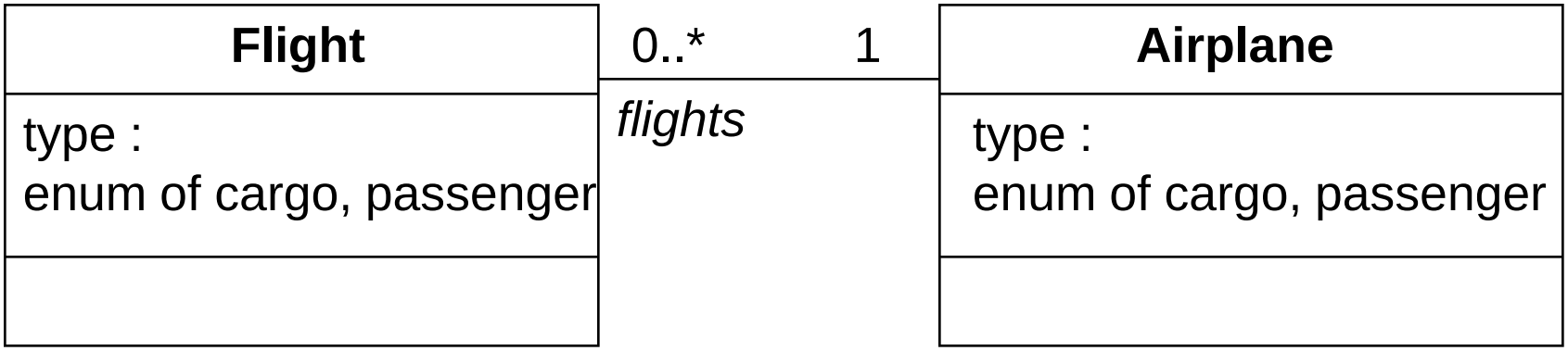
An Example: Aggregations



What is OCL

- UML diagrams alone are usually not enough to completely specify a software system.
- In general, constraints are also needed to completely specify a software system.
- A constraint is a restriction on one or more values of (part of) a software system.
- The Object Constraint Language is a textual language to describe constraints.

An Example



context Flight
inv: type = #cargo implies airplane.type = #cargo
inv: type = #passenger implies airplane.type = #passenger

Kinds of Constraints

- Class invariant
  - a constraint that must always be met by all instances of the class
- Precondition of an operation
  - a constraint that must always be true before the execution of the operation
- Postcondition of an operation
  - a constraint that must always be true after the execution of the operation

Characteristics of OCL

- Both query and constraint language
- Mathematical foundation, but no mathematical symbols
- Strongly typed language
- Declarative language

Constraint Context and Self

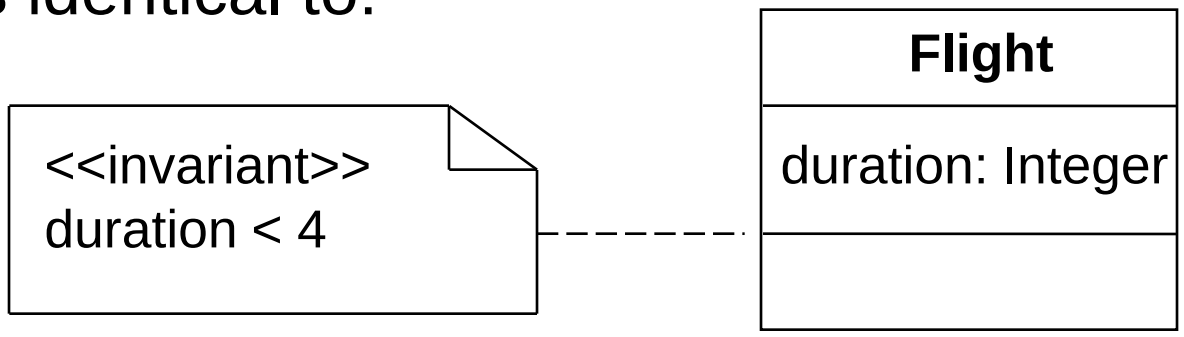
- Every OCL expression is bound to a specific context.
  - The context is often the element that the constraint is attached to
- The context may be denoted within the expression using the keyword ‘self’.
  - ‘self’ is implicit in all OCL expressions
  - Similar to`this’ in C++

Notation

- Constraints may be denoted within the UML model or in a separate document.
  - the expression:

    context Flight inv: self.duration < 4
  - is identical to:

    context Flight inv: duration < 4
  - is identical to:



Elements of an OCL Expression

- In an OCL expression these elements may be used:
  - basic types: String, Boolean, Integer, Real.
  - classifiers from the UML model and their features
    - attributes, and class attributes
    - query operations, and class query operations (i.e., those operations that do not have side effects)
  - associations from the UML model
    - Including Rolenames at either end of an association

OCL Basic Types

```
context Airline
inv: name.toLower = ‘klm’

context Passenger
inv: age >= ((9.6 - 3.5)* 3.1).floor implies
      mature = true
```

Attributes

- Object attributes

  context Flight

  inv: self.maxNrPassengers <= 1000

- Class attributes

  context Passenger

  inv: age >= Passenger.minAge

An Example: Constructor Triangle( )

```
context  Triangle::Triangle(int sa, int sb, int sc)
pre:
    sa + sb > sc and sa + sc > sb and sb + sc > sa
post:
    a = sa and b = sb and c = sc
```

An Example: Method category( )

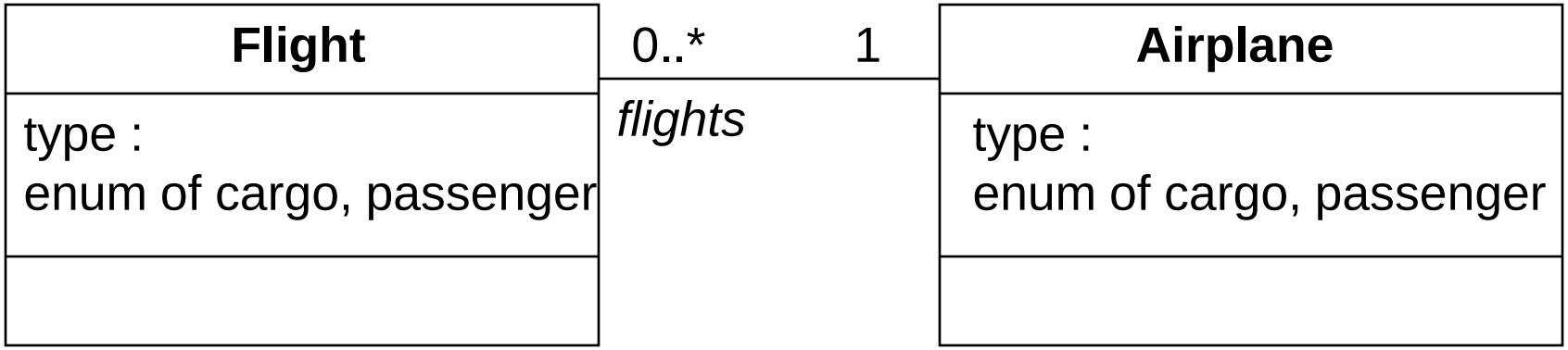
```
context  Triangle::category( ): string
pre:
post: result =
    if (a = b and a = c) then
       “Equilateral”
    else if (a = b or a = c or b = c)  then
       “Isosceles”
    else
       “Scalene”
    endif
```

An Example: Triangle Objects

```
context  Triangle
inv:
   a + b > c and a + c > b and b + c > a
```

Significance of Collections in OCL

- Most navigations return collections rather than single elements



Four Subtypes of Collection

- Set:
  - arrivingFlights(from the context Airport)
  - Non-ordered, unique
- OrderedSet:
  - passengers (from the context Flight)
  - Ordered, unique
- Bag:
  - arrivingFlights.duration (from the context Airport)
  - Non-ordered, non-unique
- Sequence:
  - passengers.age (from the context Flight)
  - Ordered, non-unique

Collection Operations

- OCL has a great number of predefined operations on the collection types.
- Syntax: collection**->**operation

Use of the "->" (arrow) operator instead of the "." (dot) operator for predefined operations

Basic Collection Operations

- isEmpty: True if the collection contains no elements
- notEmpty: True if the collection contains one or more elements
- size: The number of elements in the collection
- count(object): The number of occurrences of the object in the collection
- sum( ): the addition of all elements in the collection

Basic Collection Operations

- includes(object): True if the object is an element of the collection
- excludes(object): True if the object is not an element of the collection
- includesAll(collection): True if all elements of the parameter collection are present in the current collection
- excludesAll(collection): True if all elements of the parameter collection are not present in the current collection

Basic Collection Operations

- including(object): returns a new collection with one element added to the original collection
- excluding(object): returns a new collection with an element removed from the original collection

Basic Collection Operations

- union(collection): returns a new collection that combines the parameter collection and the current collection
- intersection(collection): returns a new collection that contains the elements in both the parameter collection and the current collection

Basic Collection Operations

- - (collection): returns a new set that contains all the elements in the current set, but not in the parameter set.
- symmetricDifference(collection): returns a new set that contains all elements in the current set, or in the parameter set, but not in both.

Operations for Ordered Collection

- first: returns the first element of the collection
- last: returns the first element of the collection
- at(index): returns the element of the collection at the given position (index starts from 1)
- indexOf(object): returns the position of the parameter element in the collection
- insertAt(index, object): results in a sequence or an orderedSet that has an extra element inserted at the given position

Operations for Ordered Collection

- append(object): adds an element to a sequence as the last element
- prepend(object): adds an element to a sequence as the first element
- subSequence(lower, upper): returns a sequence that contains the elements from the lower index to the upper index, inclusive
- subOrderedSet(lower, upper): returns an orderedSet that contains the elements from the lower index to the upper index, inclusive

Loop Collection Operations

- Operation collect
- Operation select
- Operation forAll
- Operation exists
- Operation iterate

The collect Operation

- Syntax:

  collection->collect(elem : T | expr)

  collection->collect(elem | expr)

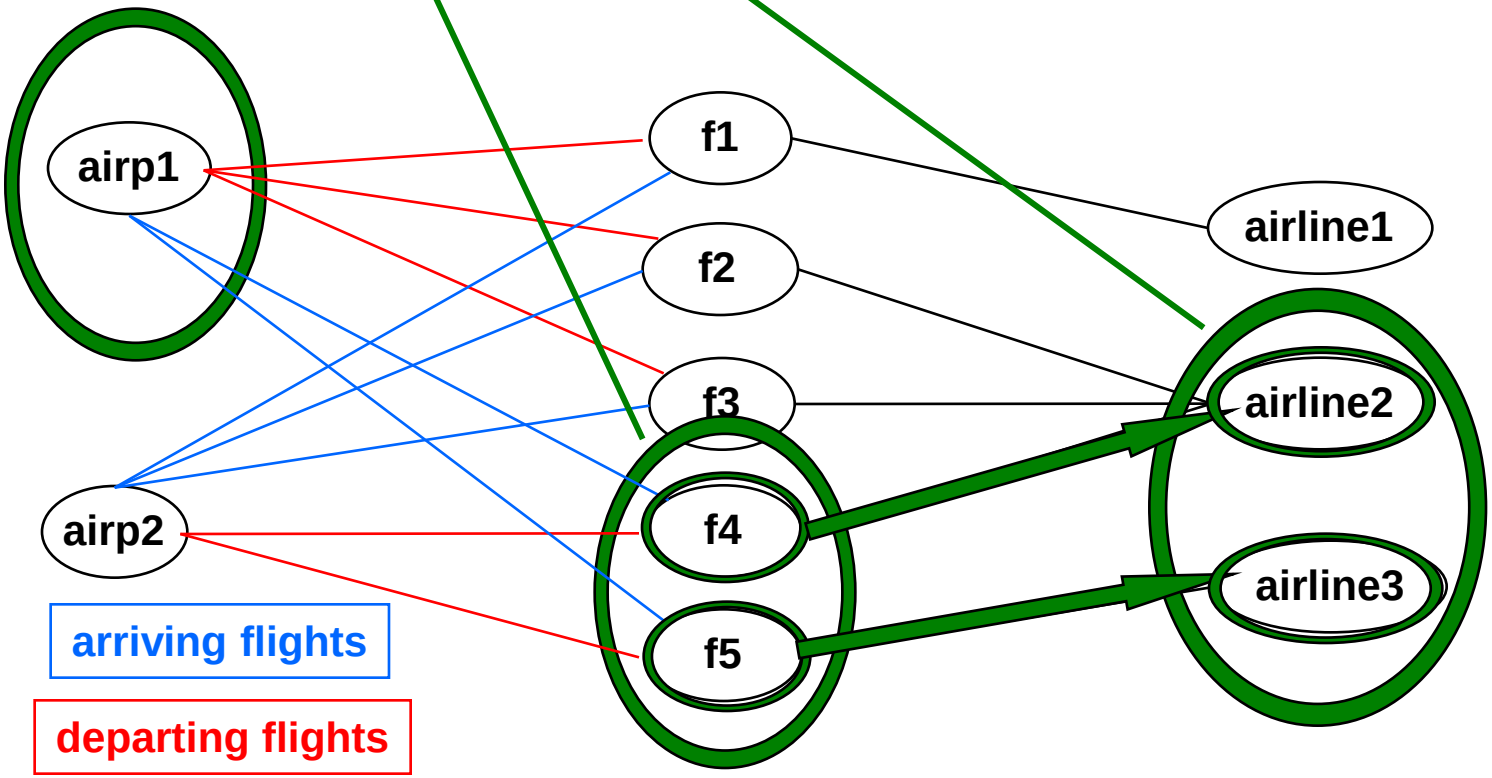
  collection->collect(expr)

- The collect operation returns the collection of objects that result from evaluating expr for each element in the source collection

Example: collect Operation

context Airport inv:

self.arrivingFlights -> collect(airLine) ->notEmpty



The select Operation

- Syntax:

  collection->select(elem : T | expr)

  collection->select(elem | expr)

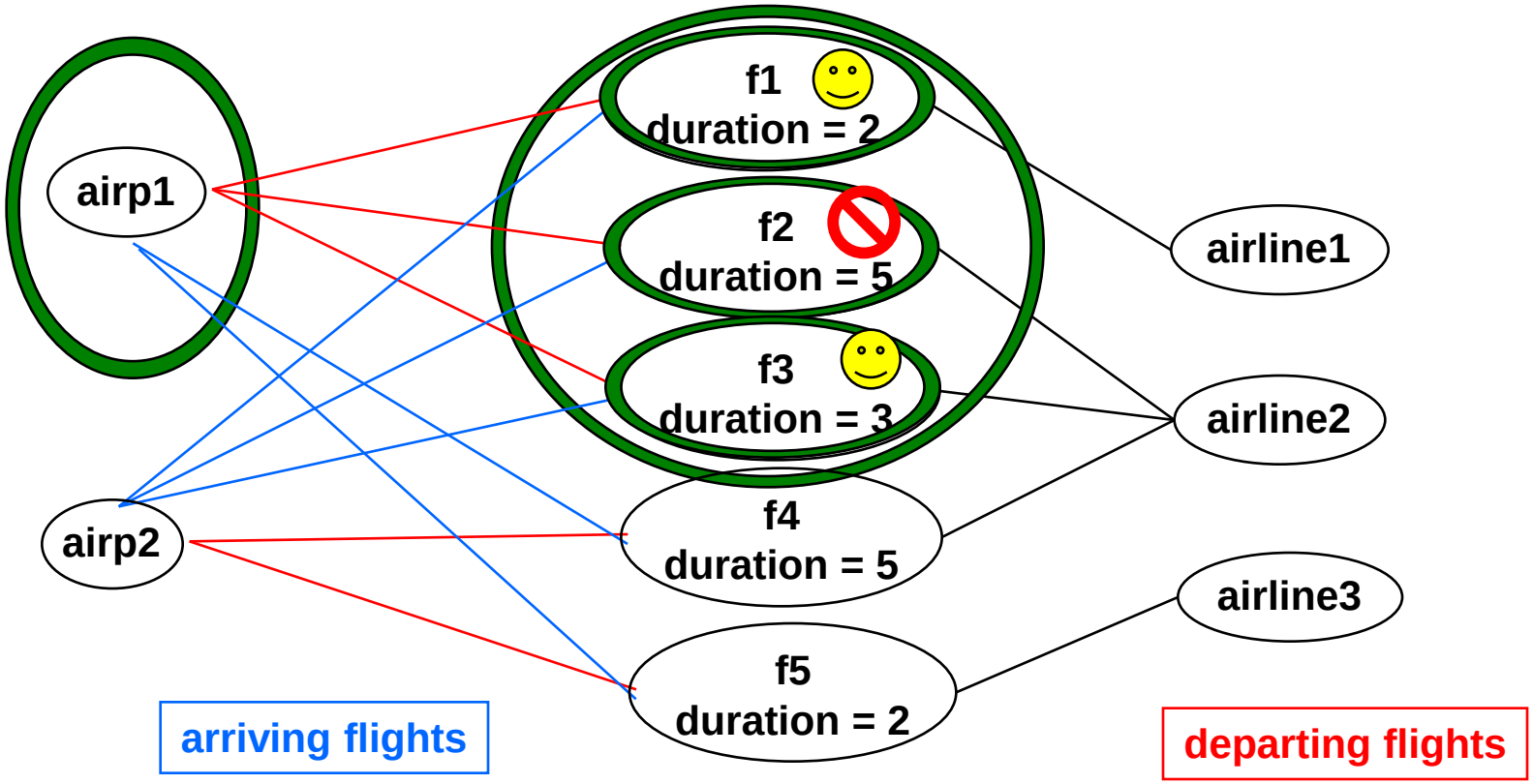
  collection->select(expr)

- The select operation returns a subcollection of the source collection containing all elements for which expr is true

Example: select Operation

context Airport inv:

self.departingFlights->select(duration<4)->notEmpty



The forAll Operation

- Syntax:

  collection->forAll(elem : T | expr)

  collection->forAll(elem | expr)

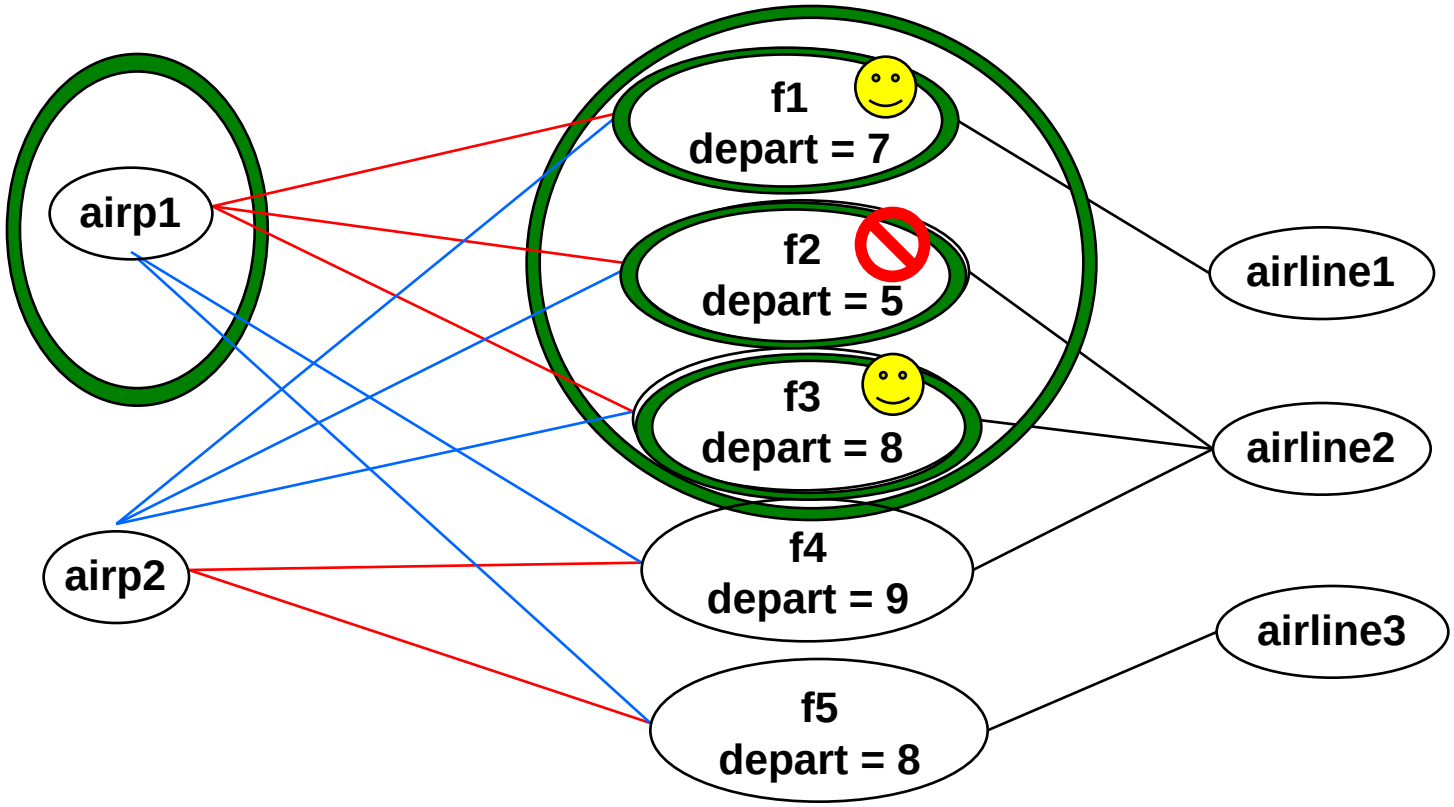
  collection->forAll(expr)

- The forAll operation returns true if expr is true for all elements of the collection

Example: forAll Operation

context Airport inv:

self.departingFlights->forAll(departTime.hour>6)



departing flights

arriving flights

The exists Operation

- Syntax:

  collection->exists(elem : T | expr)

  collection->exists(elem | expr)

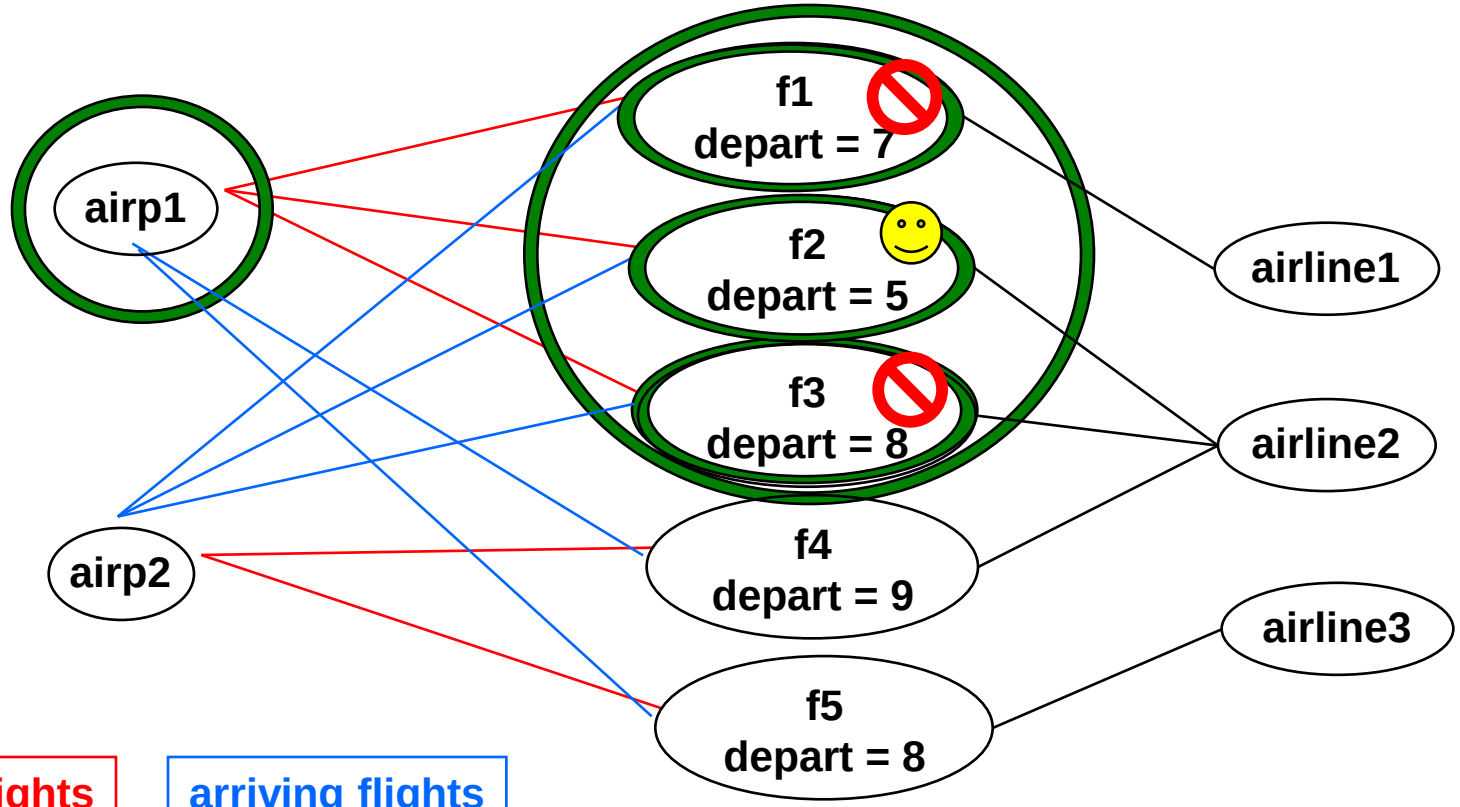
  collection->exists(expr)

- The exists operation returns true if there is at least one element in the collection for which the expression expr is true.

Example: exists Operation

context Airport inv:

self.departingFlights->exists(departTime.hour<6)



The iterate Operation

- The iterate operation for collections is the most fundamental and generic building block.
- All other loop operations are a special case of iterate operation.

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

iterator variable

initial value

accumulator

body expression

Example: iterate Operation

```
context Airline inv:
flights->select(maxNrPassengers > 150)->notEmpty

context Airline inv:
   flights->iterate (f : Flight;
         answer : Set(Flight) = Set{ } |
         if f.maxNrPassengers > 150 then
               answer->including(f)
         else
               answer
         endif )->notEmpty
```

Local Variables

- The let construct defines variables local to one constraint:

  Let var : Type = <expression1> in
  <expression2>

- Example:

  context Airport inv:

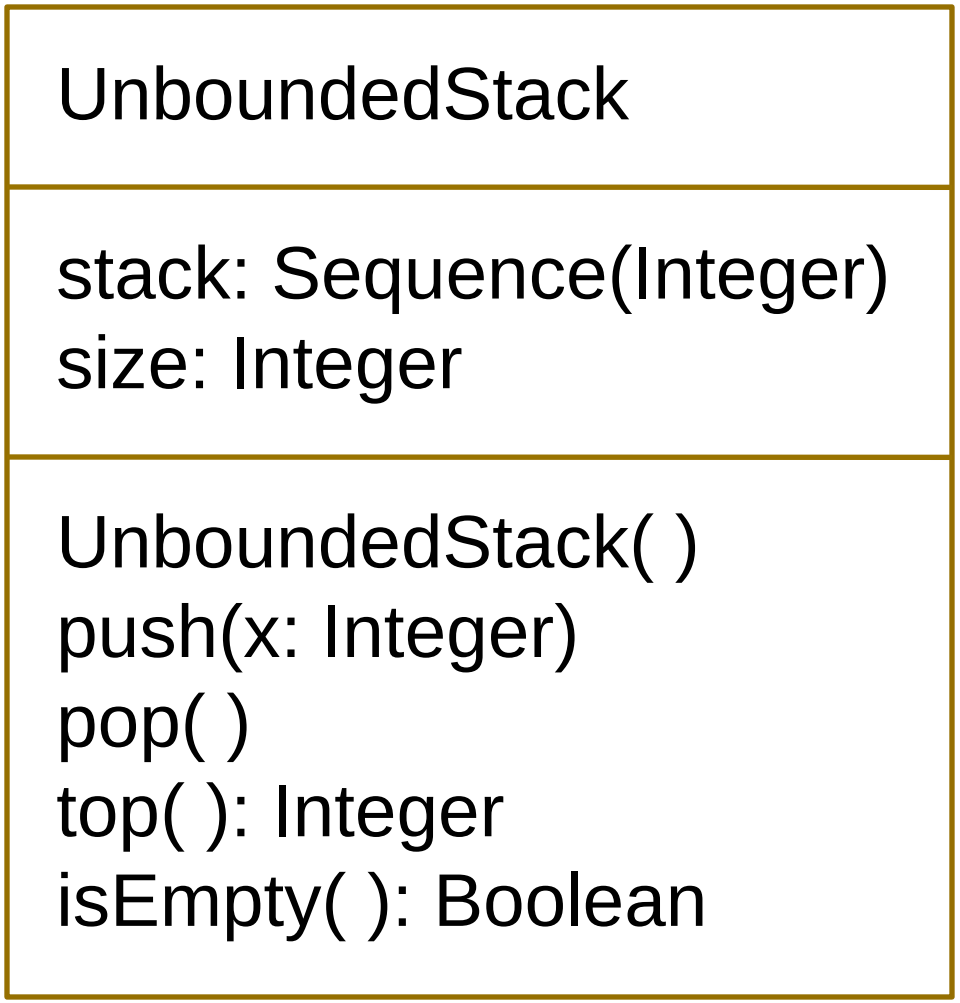
  Let supportedAirlines : Set (Airline) =
  self.arrivingFlights -> collect(airLine) in
  (supportedAirlines ->notEmpty) and
  (supportedAirlines ->size < 500)

The @pre Keyword

- The @pre keyword indicates the value of an attribute at the start of the execution of the operation
- The keyword must be postfixed to the name of the item concerned

```
answer = answer@pre->including(x)
```

An Example: Class UnboundedStack



An Example: Constructor UnboundedStack

context UnboundedStack:: UnboundedStack()
pre:
post:
    stack = Sequence{ } and size = 0

An Example: Method push( )

```
context UnboundedStack::push(x: Integer)
pre:
post:
    stack = stack@pre->prepend(x) and
    size = size@pre + 1
```

An Example: Method pop( )

context UnboundedStack::pop( )
pre:
    size > 0
post:
    stack = stack@pre->subSequence(2,size@pre) and
    size = size@pre - 1

An Example: Method top( )

```
context UnboundedStack::top() : Integer
pre:
    size > 0
post:
    result = stack->first()
```

Inheritance of Constraints

- Liskov's Substitution Principle (LSP):
  "Whenever an instance of a class is expected, one can always substitute an instance of any of its subclasses."

Inheritance of Constraints

- Consequences of LSP for invariants:
  - An invariant is always inherited by each subclass.
  - Subclasses may strengthen the invariant.
- Consequences of LSP for preconditions and postconditions:
  - A precondition may be weakened (contravariance)
  - A postcondition may be strengthened (covariance)