Alf v1.1 Language Reference Guide

November 2018 (corrected)

Contents

I. The Standards

- Unified Modeling Language (UML)
- Foundational UML (fUML)
  - Other Precise Semantics Specifications
- Action Language for fUML (Alf)

Unified Modeling Language (UML)

The *Unified Modeling Language* (UML) is a graphical language for modeling the structure, behavior and interactions of software, hardware and business systems, standardized by the Object Management Group (OMG).

- UML Version 1.1 (first standard) 1997
- UML Version 1.5 (with action semantics) 2003
- UML Version 2.0 2005
- UML Version 2.4.1 2011
- UML Version 2.5 (spec simplification) 2015
- UML Version 2.5.1 2017

Foundational UML (fUML)

Foundational UML (fUML) is an executable subset of standard UML that can be used to define, in an operational style, the structural and behavioral semantics of systems.

- fUML Version 1.0 (based on UML 2.3) 2011
- fUML Version 1.1 (based on UML 2.4.1) 2013
- fUML Version 1.2.1 (based on UML 2.4.1) 2016
- fUML Version 1.3 (based on UML 2.4.1) 2017
- fUML Version 1.4 (based on UML 2.5.1) 2018 (current)

Key Components

- **Foundational UML Subset (fUML)** – A *computationally complete* subset of the abstract syntax of UML (Version 2.5.1)
  - **Kernel** – Basic object-oriented capabilities
  - **Common Behavior** – General behavior and asynchronous communication
  - **Activities** – Activity modeling, including structured activities (but *not* including variables, exceptions, swimlanes, streaming or other "higher level" activity modeling)
- **Execution Model** – A model of the execution semantics of user models within the fUML subset
- **Foundational Model Library**
  - **Primitive Types** – Boolean, String, Integer, Real, Unlimited Natural
  - **Primitive Behaviors** – Boolean, String and Arithmetic Functions
  - **Basic Input/Output** – Based on the concept of "Channels"

Other Precise Semantics Specifications

The following specifications build on the foundation of fUML:

- Precise Semantics of UML Composite Structure (PSCS)
  - Version 1.0 2015
  - Version 1.1 2018 (current)
  - Version 1.2 (move to UML 2.5.1) 2019 (planned)
- Precise Semantics of UML State Machines (PSSM)
  - Version 1.0 (alpha) 2016
  - Version 1.0 (finalized) 2018
- Precise Semantics of Time for Foundational UML
  - RFP issued December 2017
  - Initial submissions due August 2019
  - Revised submissions due May 2020

Action Language for fUML (Alf)

The *Action Language for Foundational UML* (Alf) is a textual surface representation for UML modeling elements with the primary purpose of acting as the surface notation for specifying executable (fUML) behaviors within an overall graphical UML model. (But which also provides an extended notation for structural modeling within the fUML subset.)

- Alf Version 1.0 beta (based on fUML 1.0) 2012
- Alf Version 1.0.1 (based on fUML 1.1) 2013
- Alf Version 1.1 (based on fUML 1.3) 2017 (current)
- Alf Version 1.2 (based on fUML 1.4) 2019 (planned)

Key Components

- **Concrete Syntax** – A BNF specification of the legal textual syntax of the Alf language.
- **Abstract Syntax** – A MOF metamodel of the abstract syntax tree that is *synthesized* during parsing of an Alf text, with additional *derived* attributes and constraints that specify the static semantic analysis of that text.
- **Semantics** – The semantics of Alf are defined by *mapping* the Alf abstract syntax metamodel to the fUML abstract syntax metamodel.
- **Standard Model Library**
  - **From the fUML Foundational Model Library**
    - Primitive Types (plus Natural and Bit String)
    - Primitive Behaviors (plus Bit String Functions and Sequence Functions)
    - Basic Input/Output
  - **Collection Functions** – Similar to OCL collection operations for sequences
  - **Collection Classes** – Set, Ordered Set, Bag, List, Queue, Deque, Map

Resources

- Foundational UML (fUML)
  - *Semantics of a Foundational Subset for Executable UML Models* standard, http://www.omg.org/spec/FUML
  - fUML Reference Implementation Project, http://fuml.modeldriven.org/
- Action Language for fUML (Alf)
  - *Action Language for Foundational UML (Alf)* standard, http://www.omg.org/spec/ALF
  - Alf Reference Implementation Repository, http://alf.modeldriven.org/

II. The Alf Language

A. Basic Concepts
B. Expressions
C. Statements
D. Units
E. Active Classes

A. Basic Concepts

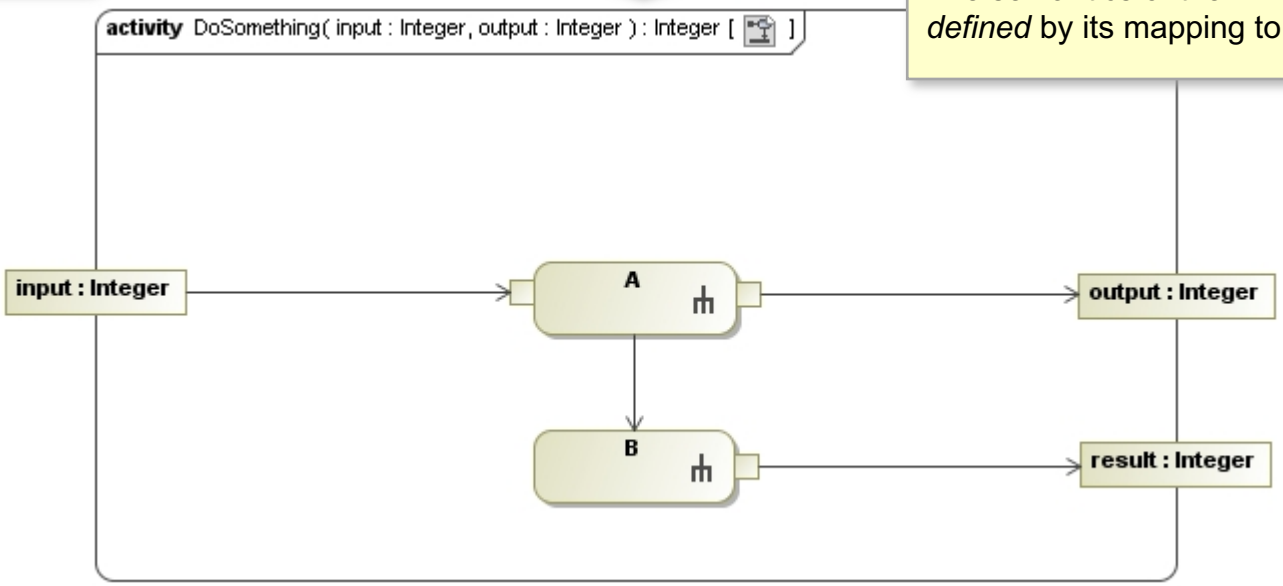
- The Basic Idea
- Assignment as Data Flow
- Sequences
- Null as the Empty Sequence

The Basic Idea: Alf maps to fUML

```
activity DoSomething(in input: Integer, out output Integer): Integer {
  output = A(input);
  return B();
}
```

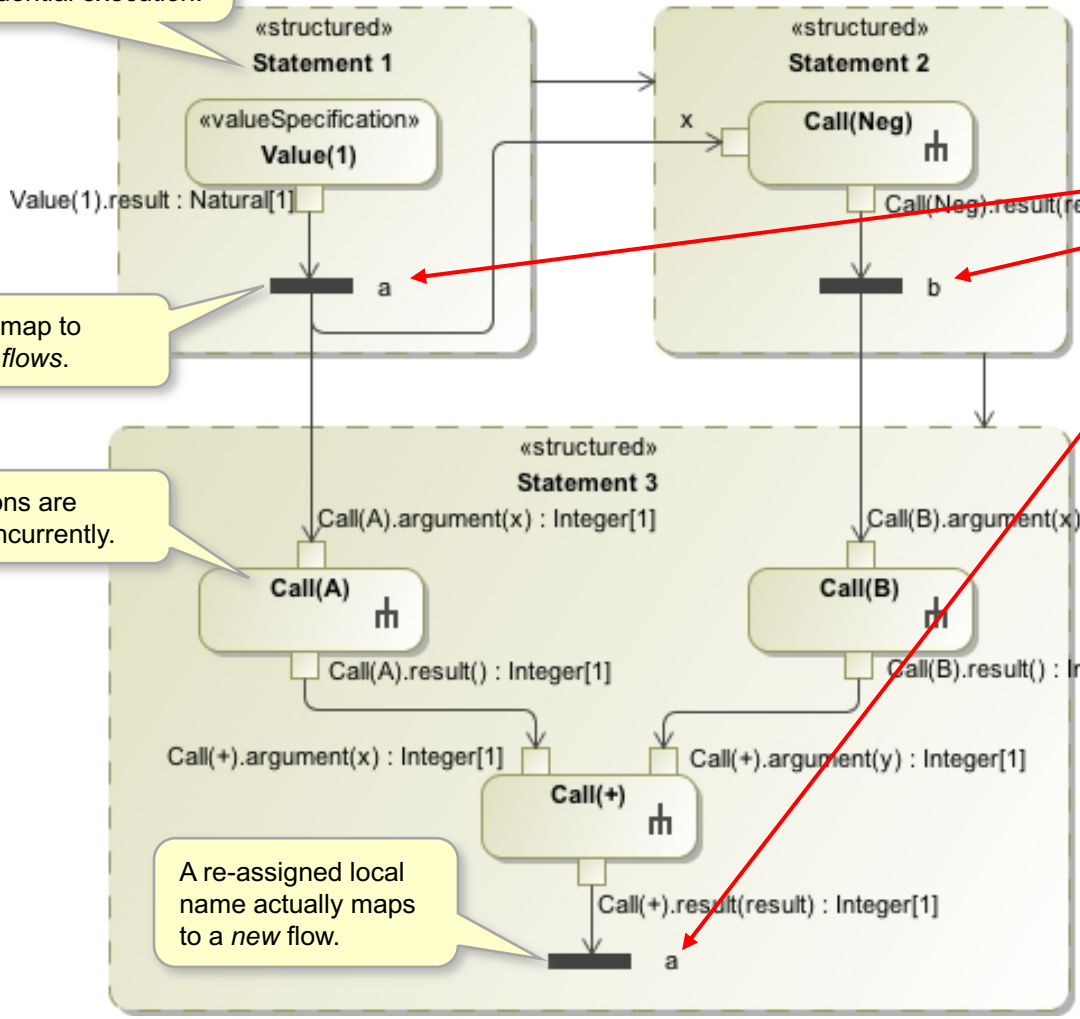
Alf behavioral notation *maps* to fUML activity models.

The semantics of the Alf notation is *defined* by its mapping to fUML



Assignment as Data Flow

Statements map to structured activity nodes with *control flows* to enforce sequential execution.



Local names map to forked *object flows*.

Subexpressions are evaluated concurrently.

A re-assigned local name actually maps to a *new* flow.

The local name *a* implicitly gets the type *Integer.*

```
a = +1;
b = -a;
a = A(a) + B(b);
```

The literal “1” has type *Natural*. The expression “+1” has type *Integer*. The expression “A(a) + B(b)” has type *Integer*, which is not compatible with *Natural*.

```
a = 1;
a = A(a);
```

type conformance

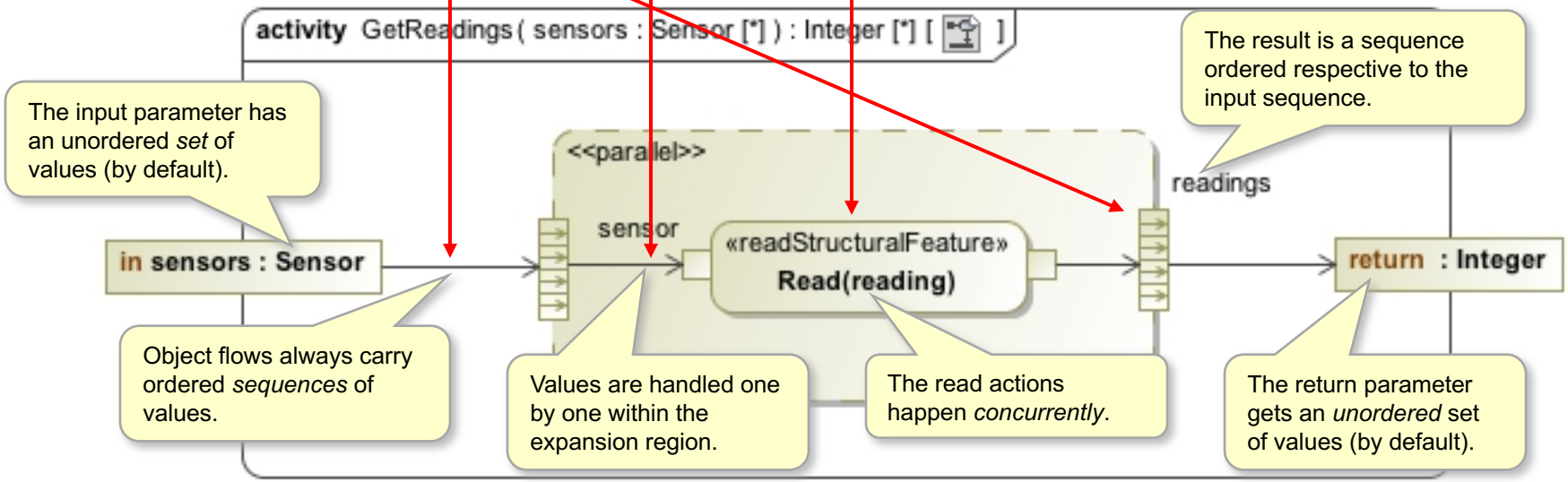
Sequences

The local name *reading* implicitly gets the type *Integer* and the multiplicity [0..*].

```
activity GetReadings(in sensors: Sensor[*]): Integer[*] {
  readings = sensors->collect sensor (sensor.reading);
  return readings;
}
```

ⓘ The Alf on the left could be written more simply as:

```
return sensors.reading;
```



⚠ Arbitrary sequences cannot be assigned to local names with multiplicity upper bound of 1.

Implicitly gets the multiplicity [1..1]

```
reading = -1;
readings = reading;
reading = readings;
```

A single value is really just a sequence of length 1.

multiplicity conformance

Null as the empty sequence

From the UML 2.5.1 specification (subclause 8.2.3):

> *A LiteralNull is intended to be used to explicitly model the lack of a value. In the context of a MultiplicityElement with a multiplicity lower bound of 0, this corresponds to the empty set (i.e., a set of no values). It is equivalent to specifying no values for the Element.*

The Alf interpretation: null is the (untyped) empty sequence

```
null = any [ ] { }
```

"null" can be assigned to any target with a multiplicity lower bound of 0.

```
sensors = null;
sensors.readings;
```

This is *not an error.* It is equivalent to sensors->collect sensor (sensor.reading); which evaluates to "null".

⚠ An argument for a parameter of multiplicity 1..1 cannot be null.

multiplicity conformance

```
WriteLine(null);
WriteLine(name ?? "no name");
```

A *null-coalescing expression* can be used to provide a non-null "default value".

B. Expressions

- Literals
- Names
- Behavior Invocation
- Arithmetic Expressions
- Boolean Expressions
- Sequence Expressions
- Sequence Expansion and Reduction
- Instance Creation
- Object Destruction
- Property Access
- Feature Invocations
- Link Expressions
- Local Name Declarations
- Other Expressions

Literals

- **Numeric literals**
  - Natural numbers — 0, 1, 2, 3, …, 1_234_567
    - Octal — 0777
    - Hexidecimal — 0xFA
    - Binary — 0b1111_1010
  - Integers — …, -2, -1, -0, +0, +1, +2, …
  - Reals — 3.14, -2.0, 1e23, 1.234E-3
  - Unbounded — *
- **Boolean literals** — false, true
- **String literals** — "xyzzy", "\b\t\n"

Underscores may be used in numeric literals.

Names

- **Identifiers** ClimateControl addItem sensor_2
- **Unrestricted names** 'Hello World' '%filled' '+' 'activity'
- **Qualified names** Alf::Library::CollectionFunctions::at
  or
  Alf.Library.CollectionFunctions.at
- **Reserved Words:**

  abstract accept active activity allInstances any as assoc break case class classify clearAssoc compose createLink datatype default destroyLink do else enum for from hastype if import in inout instanceof let namespace new nonunique null or ordered out package private protected public receive redefines reduce return sequence specializes super signal switch this to while

Behavior Invocation

A behavior invocation expression is used to call activities (in general) and primitive behaviors (from the library)

```
activity A(inout s : String, in x : Integer[0..1], out y : Integer) : Boolean
```

```
text = "hello"
b = A(text, 3, output);
b = A(text)
b = A(x=>3, s=>text);
```

The arguments for out an inout parameters must be assignable. The target for an inout parameter must have been previously defined.

Arguments may be omitted for out parameters and optional in parameters.

Arguments may also be associated with parameters by name, in any order.

Arithmetic Expressions

- **Integer / Real**
  - Unary plus `+ x`
  - Unary minus (negation) `- x`
  - Addition `x + y`
  - Subtraction `x - y`
  - Multiplication `x * y`
  - Division `x / y`
  - Remainder (Integer only) `x % y`
- **String**
  - Concatenation `a + b`

This is equivalent to the library primitive behavior call IntegerFunctions::'+'(x, y)

ⓘ For binary operators, if one argument is Integer and one is Real, the Integer value is first converted to a Real value.

Integer division truncates

⚠ Division by zero results in *null*.

Boolean Expressions

- **Comparison**
  - Equality — x == y x != y
  - Relational — x < y x > y x <= y x >= y
- **Logical / Bit String**
  - Not / Complement — !x ~x
  - And — x & y
  - Or — x | y
  - Exclusive or — x ^ y
- **Conditional Logical**
  - And — x && y
  - Or — x || y
- **Conditional Test** — test? x: y
- **Null Coalescing** — x ?? y

ⓘ String is a primitive type, so strings can be compared using the equality operators.

⚠ Relational operators are only for numeric types and cannot be used on strings.

Evaluation of the second operand depends on the result of the first.

If *test* is true, the result is *x,* else the result is *y.*

The result is *x* if that is non-null, else the result is *y.*

Sequence Expressions

- **Sequence construction**
  - List — String[ ] {"red", "yellow", "green"}
  - Range (Integer only) — Integer[ ] {1..n}
- **Sequence access** — seq[i]
seq->at(i)
- **Sequence operation** — seq->size()
seq->including(x)
seq->remove(y)

⚠ Indexing is from 1. So, if seq = Integer[ ] {1, 2, 3} then seq[1] evaluates to 1. seq[0] evaluates to null.

ⓘ Sequence operation expressions are particularly useful for "pipelining" sequence expressions.

This is equivalent to including(seq, x)

children->including(parent)->union(siblings)

Sequence Expansion and Reduction

- **Sequence expansion**
  - select on true — seq->select x (x > 0)
  - select on false — seq->reject x (x > 0)
  - test if true for all — seq->forAll x (x > 0)
  - test if true for at least one — seq->exists x (x > 0)
  - test if true for exactly one — seq->one x (x > 0)
  - test if different for each — seq->isUnique x (Identify(x))
  - concurrently compute — seq->collect x (Compute(x))
  - iteratively compute — seq->iterate x (Handle(x))

"iterate" is the only kind of sequence expansion expression that is iterative rather than concurrent.

- **Sequence reduction**

Reduces a sequence to a single value by combining values using a binary function.

```
seq->reduce '+'
seq->reduce ordered MatrixMult
seq->reduce '*' ?? 1
```

⚠ If the input sequence is empty, the result of a reduction is null.

A null-coalescing expression can be used to give a "default value" for a reduction.

Cast Expressions

The type of a cast expression is always the explicitly given type.

- **Cast expression**

(DiscountedOrder)order

(Integer)3

(Real)4

(Integer)-3.14

Remember that, before the cast, the literal is of type Natural.

ⓘ Integers can also be cast to unlimited naturals and bit strings (and vice versa). Out of bounds casts result in null.

⚠ Casting a real number to an integer is by *truncating.* Therefore, the value of (Integer)-3.14 is -3.

⚠ A cast expression *filters out* values that do not conform to the given type.

(Integer)any[]{1, "this", -2, "that", true}
evaluates to the sequence
{1, -2}

Assignment

ⓘ Assignment is an expression that evaluates to the assigned value, allowing multiple assignments such as

result = previousValue = computedValue;

- **Simple assignment**
  - To local name/parameter — totalAmount = 0
  - To property — entry.address = myAddress

⚠ Multiplicities must match:
- If lower bound of LHS > 0, then lower bound of RHS > 0.
- If upper bound of the LHS = 1, then upper bound of RHS ≤ 1.

- **Indexed assignment** — result[i] = scale * input[i]

Updates the value at the index position of a sequence.

x[i] = null

⚠ This *removes* the indexed value, reducing the size of the sequence by one (if the index is in bounds).

- **Compound assignment** — count += quantity

Equivalent to
count = count + quantity

ⓘ Compound assignment is available for arithmetic, logical and bit string operators.

Local Name Declarations

- **Implicit declaration on first assignment**

  name = expression

  - *name* gets the type of *expression*
  - *name* gets the multiplicity
    - lower bound of 0, if the expression has lower bound of 0, and 1 otherwise
    - upper bound of 1, if the expression has upper bound of 1, and * otherwise
- **Explicit declaration statement**
  - Single value

    let name: Type = expression; *or* Type name = expression;
    - *name* gets the given *Type*
    - *name* gets the multiplicity [0..1] or [1..1]
  - Sequence

    let name[]: Type = expression; *or* Type name[] = expression;
    - *name* gets the given *Type*
    - *name* gets the multiplicity [0..*] or [1..*]

ⓘ To declare an “untyped” local name, use any as the type.

C. Statements

- Expression Statements
- Local Name Declarations
- If Statements
- Switch Statements
- While and Do Statements
- For Statements
- Return Statements

Expression Statements

Any expression can be made into a statement by appending a semicolon. Typical uses are:

- **Assignment**

```
totalAmount = 0;
customer = customers->
    select c (c.name == name);
```

- **Invocation**

```
WriteLine("Hello World!");
order.addProduct(product, 1);
```

- **“In-place” sequence operations**

```
items->add(selectedItem);
entries->removeAll(deletedEntries);
```

Local Name Declaration

Implicit declaration on first assignment

The name gets multiplicity upper bound 1, if the expression has upper bound 1.

The name gets multiplicity upper bound *, if the expression has upper bound other than 1.

```
condition = ConditionOf(reading);

abnormalReadings =
        readings->select r (ConditionOf(r) != normal);
```

The name gets the type of the assigned expression.

Explicit local name declaration statement

The name gets multiplicity upper bound 1.

The name gets multiplicity upper bound *.

```
let condition: Condition = ConditionOf(reading);

let abnormalReadings: Reading[ ] =
        readings->select r (ConditionOf(r) != normal);
```

This is an explicit *multiplicity indicator.*

The name gets the declared type. (Use use any for “untyped”.)

ⓘ In all cases, the name gets multiplicity lower bound of 0, if expression has lower bound of 0, and 1 otherwise.

ⓘ A local name is usable from the point of definition to the end of the unit, *even* if defined within a compound statement.

⚠ Alf does *not* have nested scopes for local names.

If Statements

```
if (reading <= safeLimit) {
    condition = normal;
} else if (reading <= criticalLimit) {
     condition = alert;
} else {
    condition = critical;
}
```

⚠ Brackets are *required* around the block of statements in an if clause.

```
if (reading <= safeLimit) {
    condition = normal;
} or if (reading > safeLimit && reading <= criticalLimit) {
    condition = alert;
} or if (reading > criticalLimit) {
    condition = critical;
}
```

ⓘ If *or* is used rather than *else,* the clause conditions are executed *concurrently*.

```
if (list->isEmpty()) {
    result = null;
} else {
    x = list[1];
    list->removeAt(1);
    result = Quicksort(list->select a (a < x))-> including(x)->
             union(Quicksort(list->select b (b >= x)));
}
```

ⓘ A new local name defined in one if clause but not others will have a multiplicity lower bound of 0 after the if statement.

Switch Statements

```
switch (month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        numDays = 31;
    case 4: case 6: case 9: case 11:
        numDays = 30;
    case 2:
        if ( ((year % 4 == 0) && !(year % 100 == 0)) ||
             (year % 400 == 0) ) {
            numDays = 29;
        } else {
            numDays = 28;
        }
    default:
        WriteLine("Invalid month.");
        numDays = 0;
}
```

"Break" statements are not required. Execution does *not* "fall through" to the next set of cases. Case expressions are evaluated *concurrently.*

```
switch (description) {
    case "red": color = Color::red;
    case "green": color = Color::green;
    case "blue": color = Color::blue;
    case "white": case "": color = Color::white;
    default: color = Color::black;
}
```

The type of the case expressions is not restriction. They are tested against the result of the switch expression using the equality operation.

While and Do Statements

```
while (file.hasMore()) {
    nextRecord = file.getNextRecord();
    if (nextRecord->isEmpty()) {
        break;
    }
    ProcessRecord(nextRecord);
}

lastRecord = nextRecord;
```

While statements normally continue as long as their condition evaluates to true, but can be exited using a *break* statement.

nextRecord will be null if the loop does not execute its body.

⚠ There is no "continue" statement.

ⓘ A local name defined in the condition or a body of a do or while statement is available after the loop. If defined in the body of a while statement, it has multiplicity lower bound of 0 after the loop.

```
while ((nextRecord = file.getNextRecord())->notEmpty() {
    ProcessRecord(nextRecord);
}

lastRecord = nextRecord;
```

```
do {
    reading = sensor.reading();
    RecordReading(reading);
} while (sensor.isActive());

lastReading = reading;
```

A do statement always executes its body at least once, before testing its condition.

For Statements

A for statement iterates over the values in a sequence.

```
for (sensor in sensors) {
    if ((reading = sensor.reading())->isEmpty()) {
        break;
    }
    if (reading > noiseLimit) {
        readings->add(reading);
    }
}
```

The loop variable name is only defined within the for statement.

A break statement can also be used to exit out of a for statement.

ⓘ A local name defined in the body of a for statement is available after the loop, with multiplicity lower bound of 0.

This is a shorthand for iterating over a range sequence.

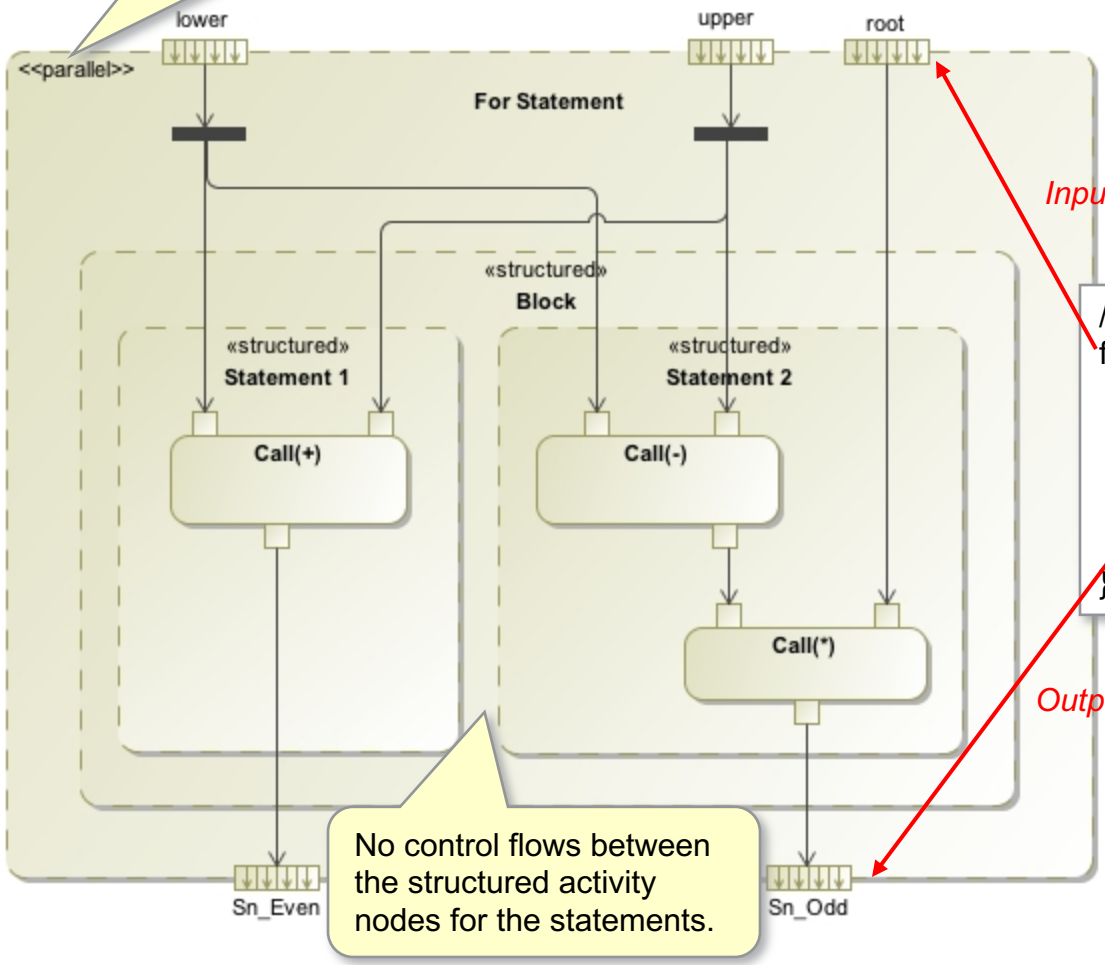
```
for (i in 1..names->size()) {
    addressBook.add(names[i], addresses[i]);
}
```

It is also possible to iterate over a range of sequences together.

```
for (name in names, address in addresses) {
    addressBook.add(name, address);
}
```

Parallel Annotation

The expansion region is parallel.



Inputs

Outputs

"//@parallel" annotations can be used to obtain concurrent iteration and statement execution.

```
//@parallel(Sn_Even,Sn_Odd)
for (lower in S_Lower, upper in S_Upper, root in V) {
    //@parallel
    {
        Sn_Even -> add(lower+upper);
        Sn_Odd -> add((lower-upper)*root);
    }
}
```

⚠ Parallel loop names can only be used to collect values. Names in parallel statements can only be assigned once.

No control flows between the structured activity nodes for the statements.

ⓘ If the parallel annotations are removed, the computations is the same (barring side effects), just not carried out concurrently.

Return Statements

A return statement can be used to return a value from an activity or operation with a return parameter.

```
activity Factorial(in n: Integer): Integer {

    if (x <= 0) {
        return 0;
    } else {
        return n * Factorial(n – 1);
    }

}
```

A return statement with no expression can be used to return from an activity or operation with no return parameter.

```
activity computeTotalAmount() {

    if (this.lineItems->isEmpty()) {
        this.totalAmount = 0;
        return;
    }

    this.totalAmount = this.lineItems.amount->reduce '+';
}
```

D. Units

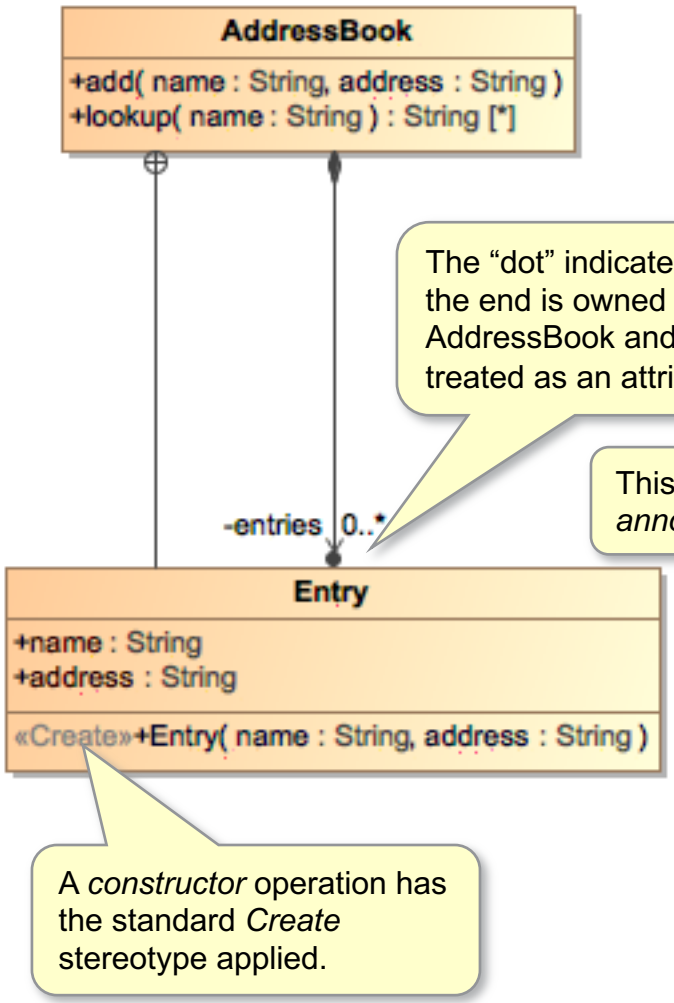
- Classes
  - Object Creation
  - Property Access
  - Operation Invocation
  - Generalization
  - Extents
  - Object Destruction
- Packages
- Data Types
- Associations
- Enumerations
- Units and Subunits
- Active Classes

Classes

⚠ A top-level *model unit* does *not* have a visibility indicator.

⚠ No visibility indicator on a member means *package* visibility.

Allowed visibility indicators are public, private and protected.



The "dot" indicates that the end is owned by AddressBook and is treated as an attribute.

This is a *stereotype annotation.*

A *constructor* operation has the standard *Create* stereotype applied.

```
class AddressBook {
```

Attributes

```
    private entries: compose Entry[0..*];
```

Operations

```
    public add(in name: String, in address: String) {
        this.entries->add(new Entry(name, address));
    }

    public lookup(in name: String): String[*] nonunique {
        return this.entries->a
            select e (e.name == name).address;
    }
```

Nested Class

```
    private class Entry {
        public name: String;
        public address: String;

        @Create
        public Entry(in name: String, in address: String) {
            this.name = name;
            this.address = address;
        }
    }
}
```

ⓘ Members may be defined in any order.

Object Creation

```
class AddressBook {

    private entries: compose Entry[0..*];

    public add(in name: String, in address: String) {
        this.entries->add(new Entry(name, address));
    }

    me: String): String[*] nonunique {
    es->a
    ame == name).address;
    }

    private class Entry {
        public name: String;
        public address: String;

        @Create
        public Entry(in name: String, in address: String) {
            this.name = name;
            this.address = address;
        }
    }
}
```

An object is created using an *instance creation expression* referencing a constructor operation.

```
book = new AddressBook();
```

A class without an explicit constructor definition gets a *default constructor* with the name of the class and no arguments.

⚠ A class with an explicit constructor definition does *not* have a default constructor.

```
entry= new Entry();
```

✗ *no matching constructor*

Object Creation

```
class Entry{

    public name: String;
    public address: String;

    @Create
    public Entry(in name: String, in address: String) {
        this.name = name;
        this.address = address;
    }

    @Create
    public copy(in entry: Entry) {
        this.name = entry.name;
        this.address = entry.address;
    }

}
```

A constructor can have a different name than the class.

```
entry1 = new Entry("Ed", "123 fUML Ln");
entry2 = new Entry::copy(entry1);
```

Property Access

Properties may be accessed and assigned using the usual dot notation.

```
entry = new Entry(aName, anAddress);

name2 = entry.name;
entry.address = address2;
```

⚠ Properties must be *visible* at the point accessed.

A *this* expression is used to access the current *context object*.

⚠ A *this* expression *must* be used to access a context attribute. Implicit access is not allowed.

```
class Entry{

    public name: String;
    public address: String;

    @Create
    public Entry(in name: String, in address: String) {
        this.name = name;
        this.address = address;
    }

}
```

Operation Invocation

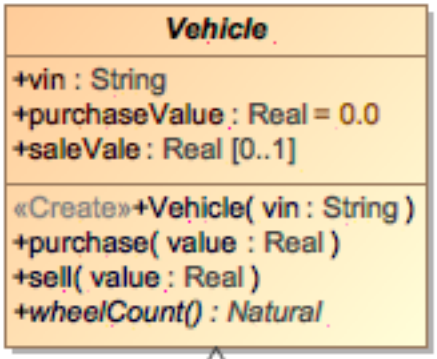
An *operation invocation* is used to call an operation on an object.

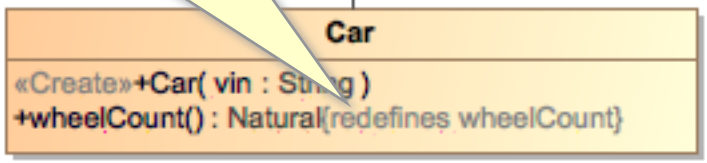
ⓘ The same notations and rules for arguments apply as for behavior invocations.

⚠ Features must be *visible* at the point accessed.

```
activity TestAddressBook() {
    book = new AddressBook();
    book.add("Ed", "123 fUML Lane");
    book.add("Ed", "456 Alf Street");
    book.add(“Sandy”, “SysML Place”);

    for (String address: book.lookup("Ed")) {
        WriteLine(address);
    }
}
```

Generalization






```
abstract class Vehicle {
    public vin: String;
    public purchaseValue: Real = 0.0;
    public saleValue: Real[0..1];
    @Create public Vehicle(in vin: String) {
        this.vin = vin;
    }
    public purchase(in value: Real) {
        this.purchaseValue = value;
    }
    public sell(in value: Real) {
        this.saleValue = value;
    }
    public abstract wheelCount(): Natural;
}
```

```
class Car specializes Vehicle {
    @Create public Car(in vin: String) {
        super(vin);
    }
    public wheelCount(): Natural {
        return 4;
    }
}
```

Attributes can be given default vaues.

Public and protected features are *inherited* by subclasses.

An *abstract class* can have abstract operations that must be defined in subclasses.

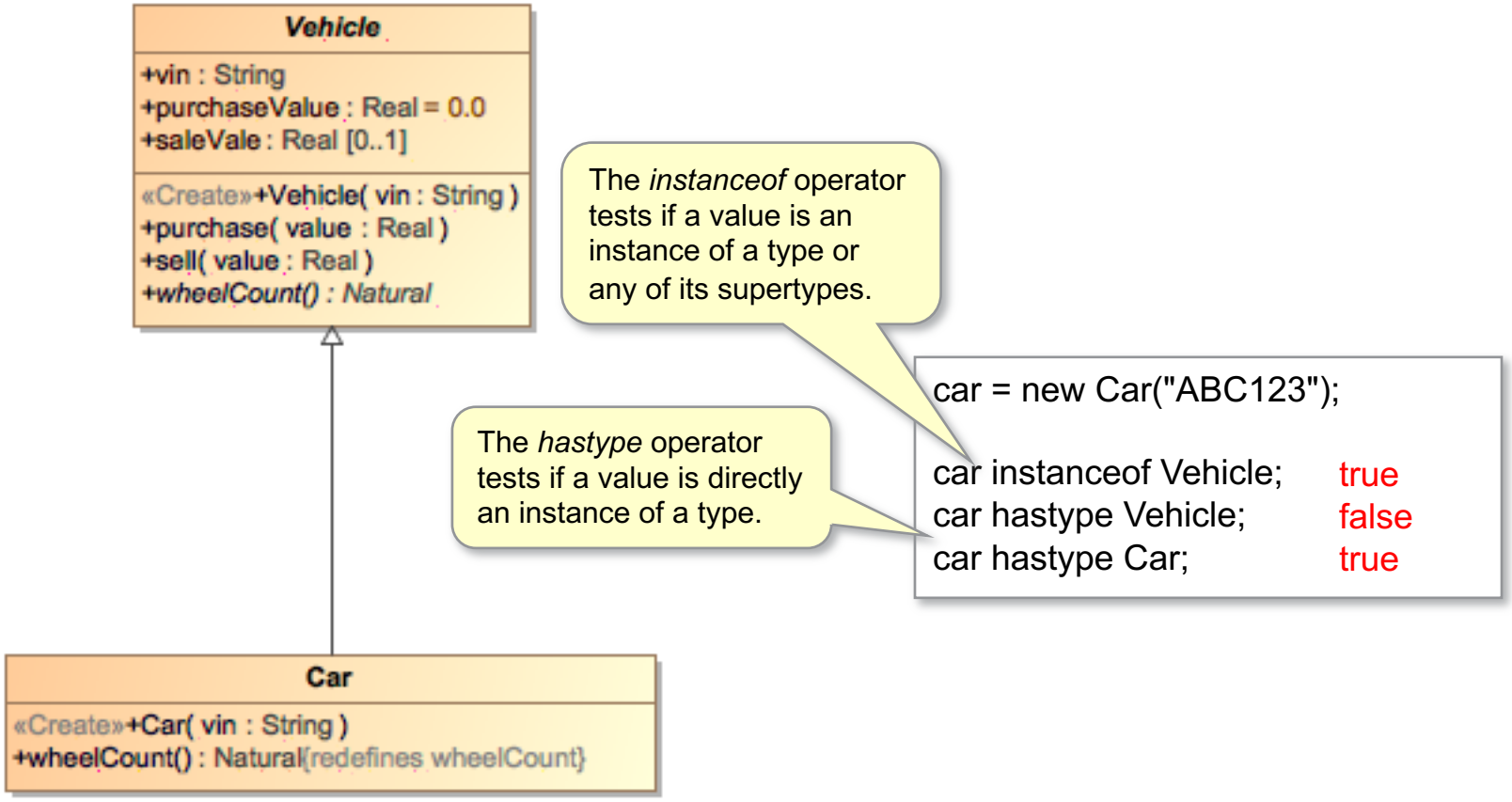
In fUML, an overriding operations must *redefine* the overridden operation.

A subclass constructor may call a superclass constructor.

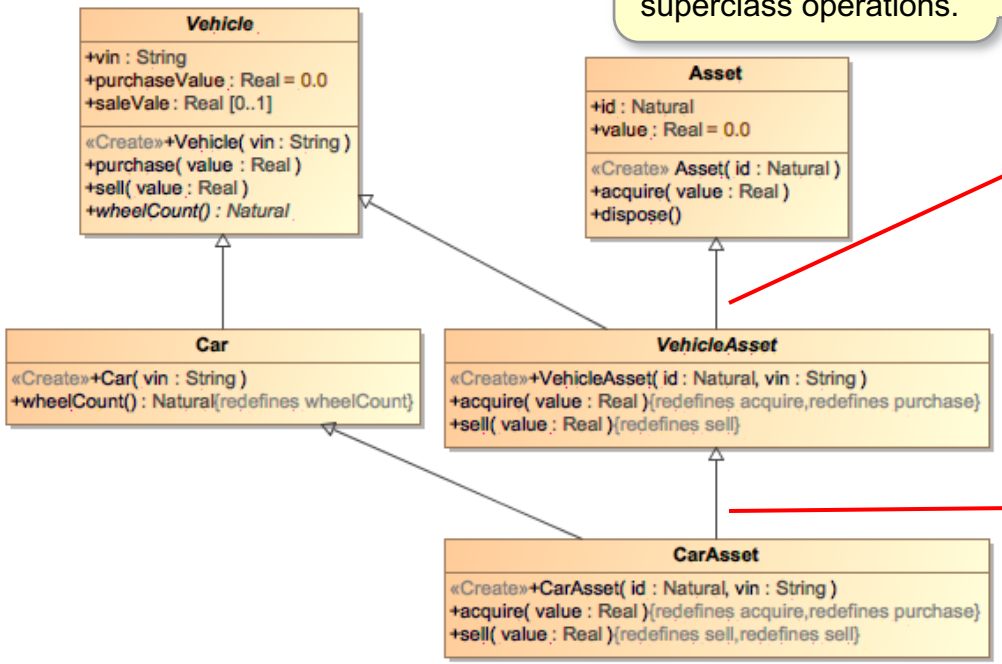
In Alf, an redefines is automatic if an operation is *indistinguishable* from (otherwise) inherited operation.

⚠ fUML does not include redefinition of attributes.

Classification Expressions



Multiple Generalization



The prefix “super” is used to invoke superclass operations.

```
abstract class VehicleAsset specializes Vehicle, Asset {
    @Create VehicleAsset (in id: Natural, in vin: String) {
        super.Vehicle(vin);
        super.Asset(id);
    }
    public acquire(in value: Real) redefines
        Vehicle::purchase, Asset::acquire {
        super.acquire(value);
        super.purchase(value);
    }
    public sell(in value: Real) {
        super.dispose();
        super.sell(value);
    }
}
```

Redefinition can be declared explicitly.

ⓘ If an operation is redefined, it is no longer inherited.

```
class CarAsset specializes VehicleAsset, Car {
    @Create public CarAsset(in id: Natural, in vin: String) {
        super.VehicleAsset(id, vin);
    }
    public acquire(in value: Real) redefines
        VehicleAsset::acquire, Car::purchase {
        super.VehicleAsset::acquire(value);
    }
    public sell(in value: Real) {
        super.VehicleAsset::sell(value);
    }
    public wheelCount(): Natural {
        return super.Car::wheelCount();
    }
}
```

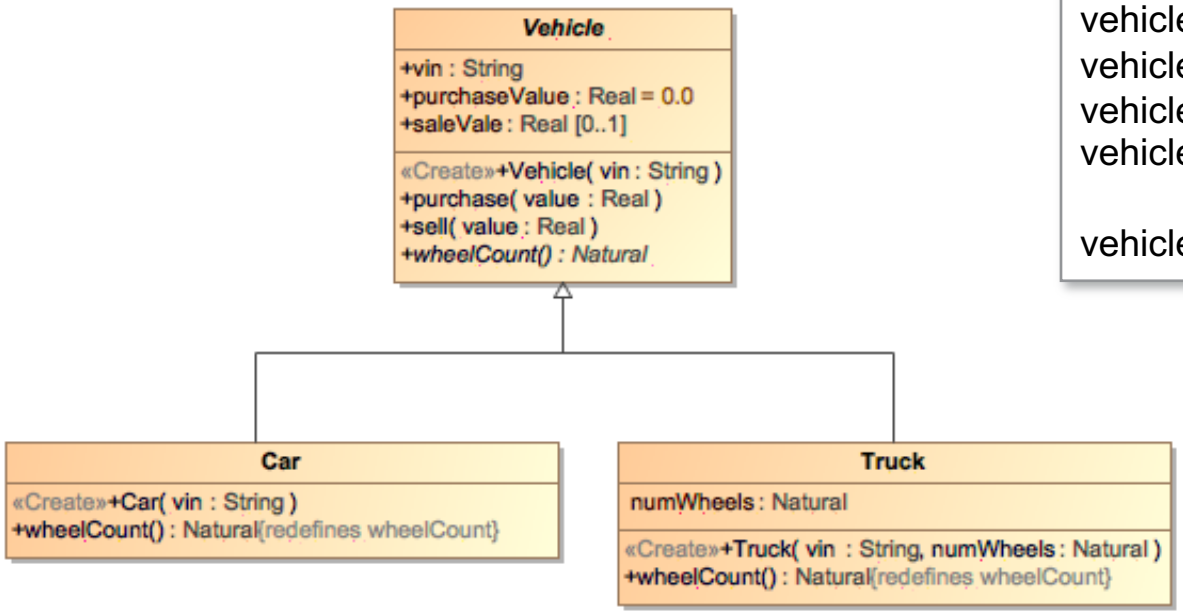
Superclass operation invocations can be qualified to resolve ambiguity.

Abstract superclass operations still need to be explicity defined.

Dynamic Reclassification

Dynamic reclassification is allowed between (disjoint) subclasses of a common superclass. (This is a restriction of fUML-allowed reclassification.)

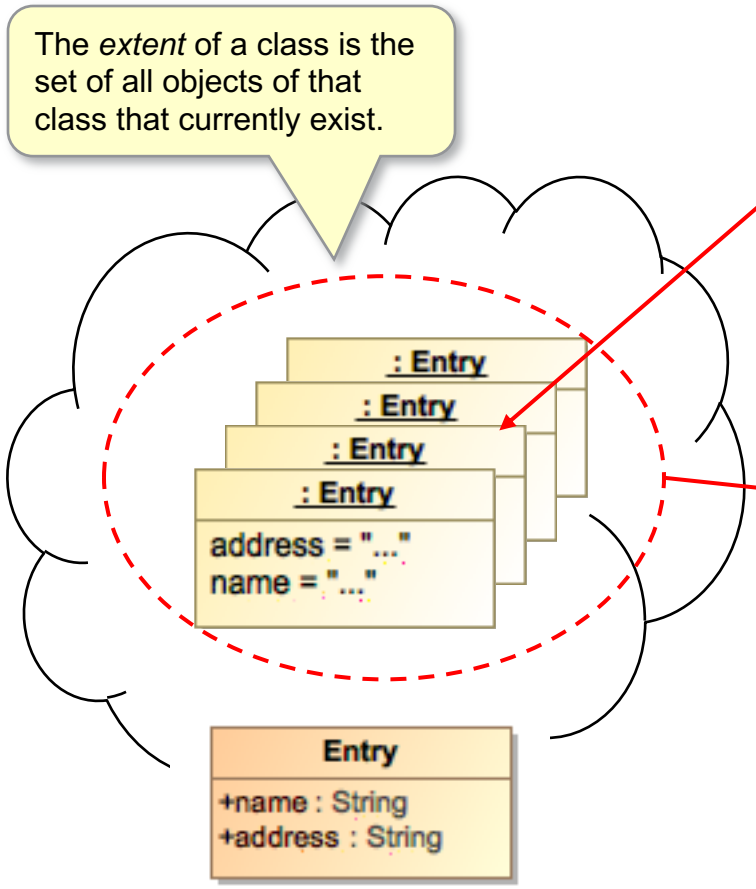
```
let vehicle: Vehicle = car;
classify vehicle from Car to Truck;

vehicle instanceof Vehicle;      true
vehicle hastype Vehicle;         false
vehicle hastype Car;             false
vehicle hastype Truck;           true

vehicle.wheelCount() == null;    true
```



⚠ Constructors of target classes are *not* invoked.

⚠ Reclassification is not type safe in general.

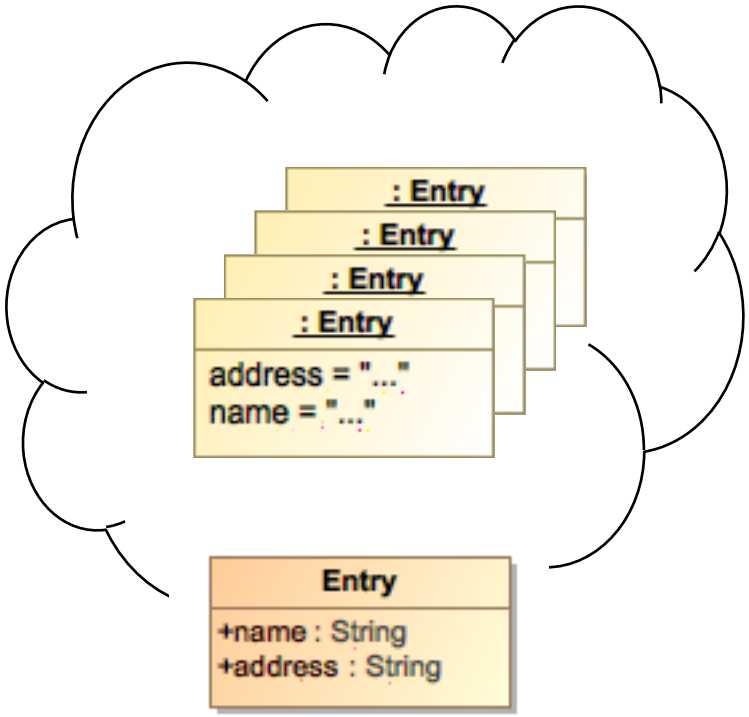
Extents




```
activity addEntry(in name: String, in address: String) {
    new Entry(name, address);
}

activity removeEntries(in name: String) {
    Entry->select entry (entry.name == name).destroy();
}

activity lookupAddresses(in name: String):
    String[*] nonunique {
    return Entry->select entry (entry.name == name).address;
}

class Entry {
    public name: String;
    public address: String;

    @Create public Entry(in name: String, in address: String) {
        this.name = name;
        this.address = address;
    }
}
```

Using the class name here is a shorthand for Entry.allInstances(), usable in sequence expressions.

Object Destruction




```
entry = new Entry();
entry.destroy();
```

An object is destroyed by invoking a *destructor operation* on it. A class without an explicit destructor gets a *default destructor* called “destroy”.

ⓘ *Destruction* means removing an object from all extents, so it becomes effectively “untyped.”

ⓘ Objects related by composition are automatically destroyed when the composite object is destroyed.

⚠ References can still exist to a destroyed object. However, property access and operation invocation on such an object is *undefined.*

Object Destruction

```
class Entry{

    public name: String;
    public location: Location;

    @Create
    public Entry(in name: String, in address: String) {
        this.name = name;
        location = Location->
            select loc (loc.address == address)[1];
        this.location = location ?? new Location(address);
    }

    @Destroy
    public destroy() {
        if (!(Entry-> exists entry (entry != this &&
                entry.location == this.location)) {
            this.location.destroy();
        }
    }
}
```

This is a common idiom for creating a singleton object.

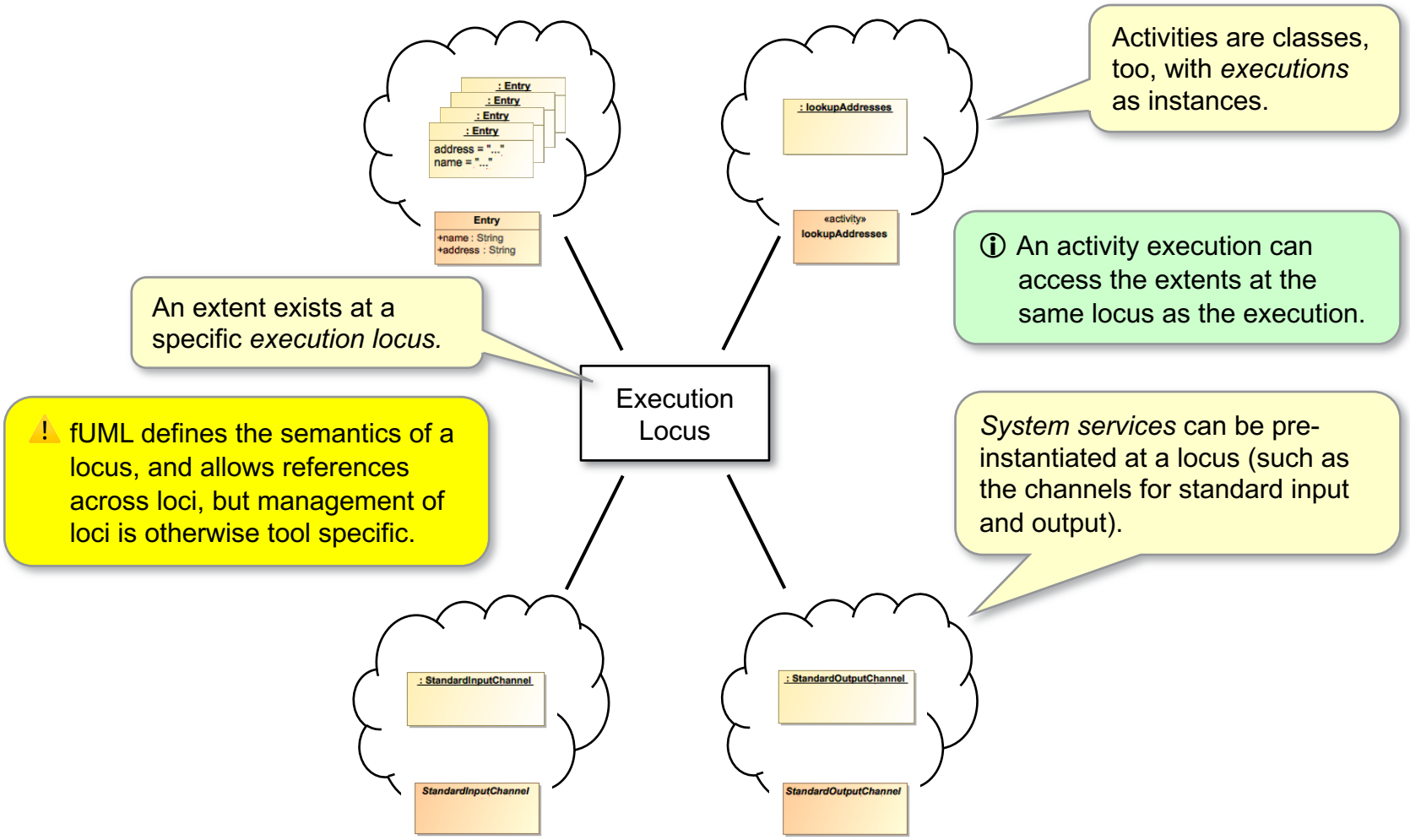
A class my also explicitly define destructor operations with the standard *Destroy* stereotype.

```
entry = new Entry();
entry.destroy();
```

ⓘ The name “destroy” is conventional, but not required for an explicit destructor.

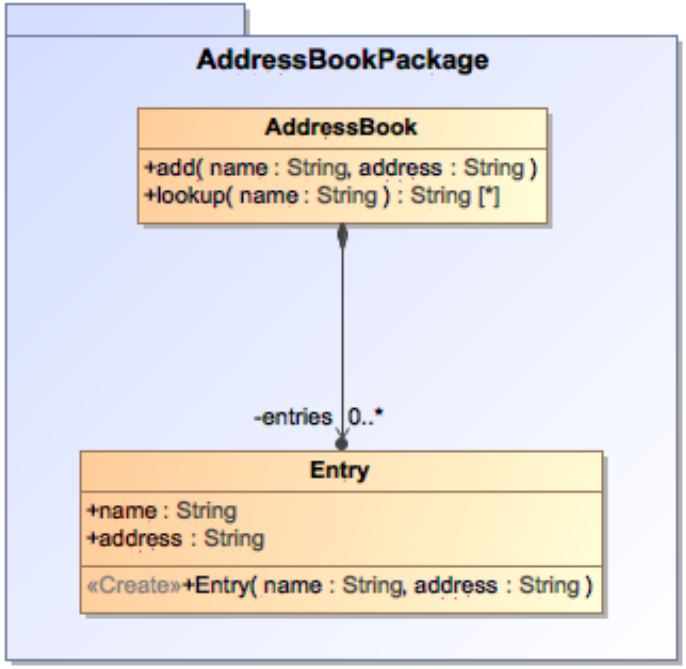
⚠ A class with an explicit destructor definition does *not* have a default destructor, even if the destructor name is different than “destroy”.

Execution Loci



Packages

Package members *must* have a visibility indicator of public or private.




```
package AddressBookPackage {
    public class AddressBook {
        private entries: compose Entry[*];

        public add(in name: String, in address: String) {
            this.entries->add(new Entry(name, address));
        }

        public lookup(in name: String): String[*] nonunique {
            return this.entries->
                select entry (entry.name == name).address;
        }
    }

    private class Entry {
        public name: String;
        public address: String;

        @Create public Entry(in name: String, in address: String) {
            this.name = name;
            this.address = address;
        }
    }
}
```

Import Declarations

An *import declaration* can be used to import members of a package, so they can be referenced without qualified notation.

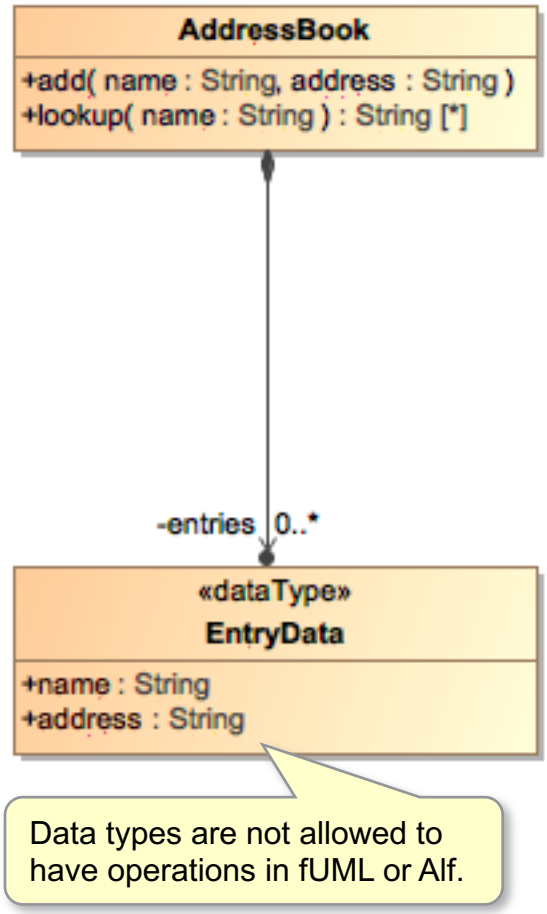
The name in an import declaration must always be *fully* qualified.

ⓘ A *private* import means that an element is private to the importing namespace. A *public* import means that an element is re-exported as a public member of the importing namespace.

⚠ The imported element must be *visible* to the importing namespace.

```
private import AddressBookPackage::AddressBook;

activity TestAddressBookPackage() {
    book = new AddressBook();
    book.add("Ed", "123 fUML Lane");
    book.add("Ed", "456 Alf Street");
    book.add("Sandy", "SysML Place");

    for (String address: book.lookup("Ed")) {
        WriteLine(address);
    }
}
```

Data Types




```
package AddressBookPackage {

    public class AddressBook {
        private entries : compose EntryData[*];

        public add(in name: String, in address: String) {
            this.entries->add(new EntryData(name, address));
        }

        public lookup(in name: String): String[*] nonunique {
            return this.entries->
                select entry (entry.name == name).address;
        }
    }

    private datatype EntryData {
        public name: String;
        public address: String;
    }

}
```

A *data type* is a structured value type, with equality based on attribute values, not identity.

Data Value Creation

```
package AddressBookPackage {

    public class AddressBook {
        private entries: compose EntryData[*];

        public add(in name: String, in address: String) {
            this.entries->add(new EntryData(name, address));
        }

        public lookup(in name: String): String[*] nonunique {
            return this.entries->
                select entry (entry.name == name).address;
        }
    }

    private datatype EntryData {
        public name: String;
        public address: String;
    }

}
```

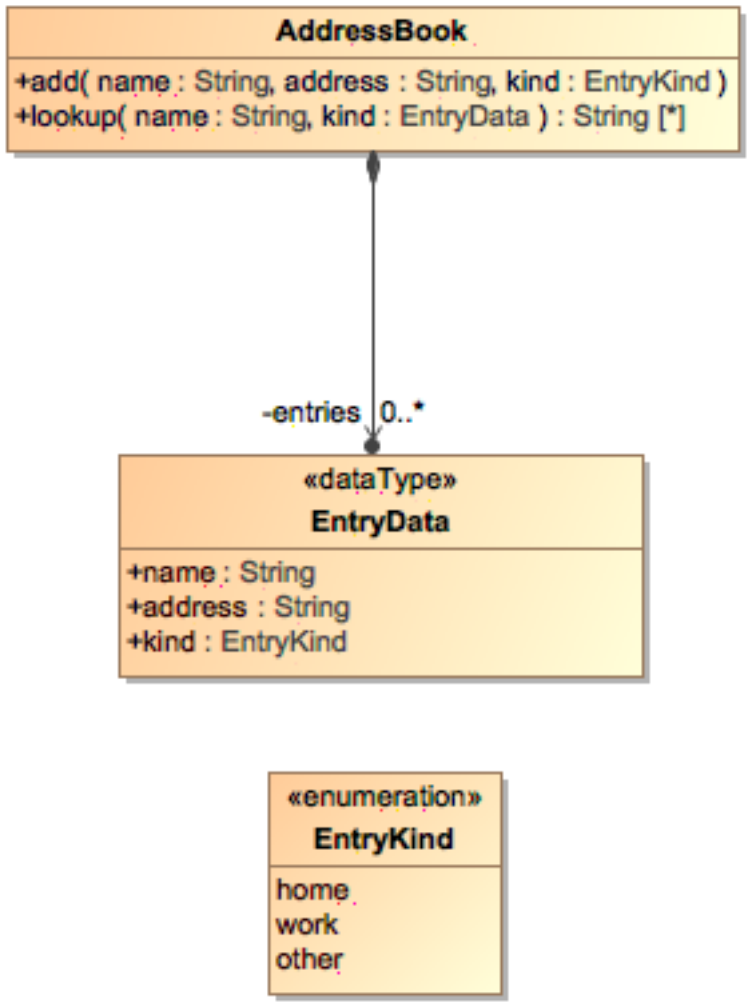
A data value is created using a *data value creation expression,* giving values must be given for at least all mandatory attributes.

ⓘ Data values are immutable, so they cannot be destroyed.

ⓘ Named parameter notation can also be used in data value creation expressions.

```
entryData = new EntryData(
    name => “Ed”,
    address => “Somewhere”
);
```

Enumerations

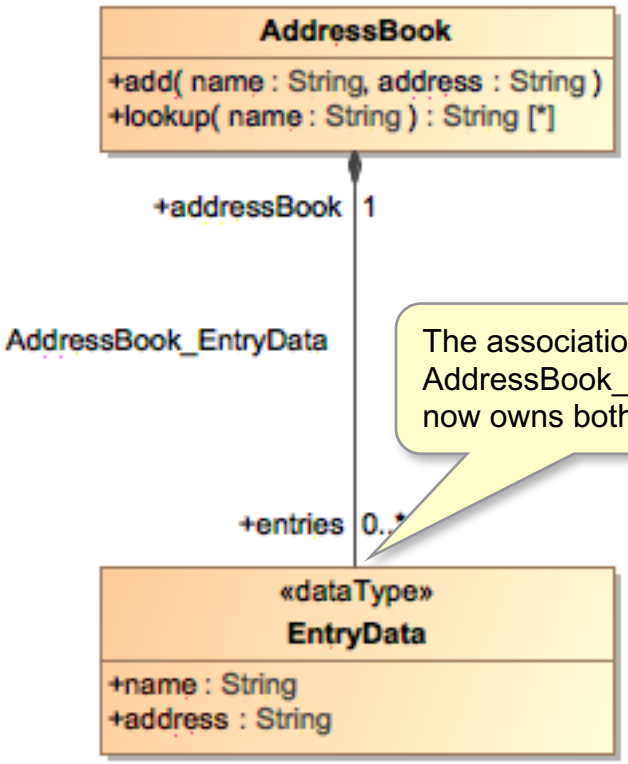



```
package AddressBookPackage {
    public class AddressBook {
        private entries: compose EntryData[*];

        public add(in name: String, in address: String) {
            this.entries->add(new EntryData(name, address,
                kind ?? EntryKind::home));
        }
        public lookup(in name: String): String[*] nonunique {
            return this.entries->
                select entry (entry.name == name &&
                entry.kind == (kind ?? EntryKind::home)).address;
        }
    }

    private datatype EntryData {
        public name: String;
        public address: String;
        public kind: EntryKind;
    }

    public enum EntryKind {home, work, other}
}
```

An *enumeration literal* is typically written qualified by its enumeration's name.

An *enumeration* is a data type defined to have a specific set of enumerated literal values.

Associations



The association AddressBook_EntryData now owns both its ends.

If a *binary* association is visible, its association ends may be used as if they were attributes of the opposite end classes.

```
package AddressBoo

    public class AddressBo
        public add(in na   : String, in address: String) {
            this.entries->add(new EntryData(name, address));
        }

        public lookup(in name: String): String[*] nonunique {
            return this.entries->
                select entry (entry.name == name).address;
        }

    private assoc AddressBook_EntryData {
        public addressBook: AddressBook;
        public entries: compose EntryData[*];
    }

    private datatype EntryData {
        public name : String;
        public address : String;
    }
}
```

⚠ Association ends *must* be public in Alf.

Link Expressions

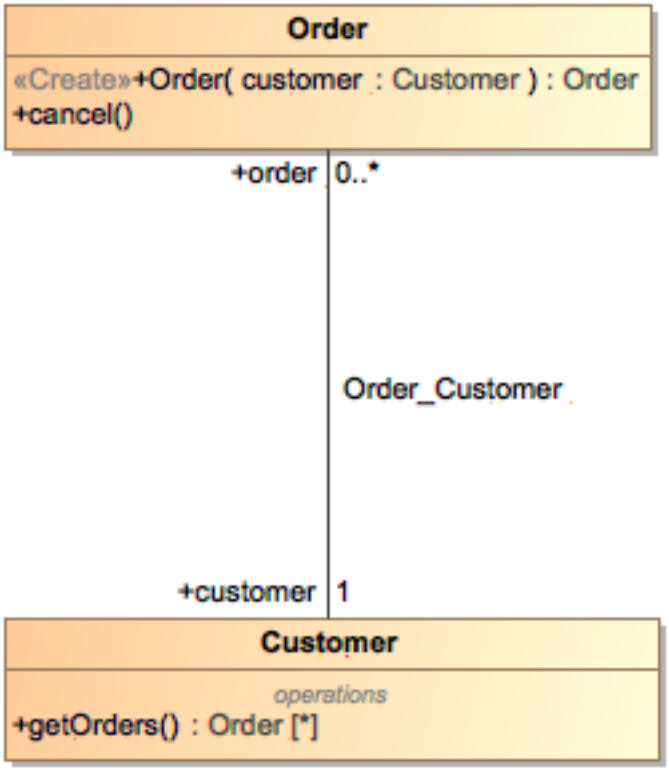
```
package AddressBookPackage {

    public class AddressBook {
        public add(in name: String, in address: String) {
            AddressBook_Entry.createLink(this,
                new EntryData(name, address));
        }

        public lookup(in name: String): String[*] nonunique {
            return AddressBook_Entry.entries(addressBook=>this).
                    address;
        }
    }

    private assoc AddressBook_EntryData {
        public addressBook: AddressBook;
        public entries: compose EntryData[*];
    }

    private datatype EntryData {
        public name: String;
        public address: String;
    }
}
```

Association instances (*links*) can also be created using the *link operation* createLink. (And destroyed using destroyLink.)

An association end can also be read using a functional notation.

ⓘ These kind of link expressions also work for association with more than two ends.

Link Semantics




```
public class Order {
    @Create public Order(in customer: Customer) {
        Order_Customer.createLink(this, customer);
    }

    public cancel() {
        Order_Customer.clearAssoc(this);
    }
}
```

Equivalent to:
this.customer = customer;

Equivalent to:
this.customer = null;

ⓘ Instances of (binary) associations are *bidirectional* links. It is unnecessary to explicitly set both sides.

```
public class Customer {
    public getOrders() : Order[0..*] {
        return this.orders;
    }
}
```

Units and Subunits

```
package AddressBookModel {

    public class AddressBook;

    private assoc AddressBook_EntryData {
        public addressBook: AddressBook;
        public entries: compose EntryData[*];
    }

    private datatype EntryData {
        public name : String;
        public address : String;
    }
}
```

A *stub* is a declaration of a namespace member without its definition.

The definition for a stub is provided in a separate *subunit.*

```
namespace AddressBookModel;

class AddressBook {
    public add(in name: String, in address: String);
    public lookup(in name: String): String[*] nonunique;
}
```

⚠ Visibility is declared on the stub, not the subunit.

A *namespace declaration* links a subunit to the namespace containing its stub.

```
namespace AddressBookModel::AddressBook;

activity lookup(in name: String): String[*] nonunique {
    return this.entries->
        select entry (entry.name == name).address;
}
```

⚠ The namespace name must be *fully* qualified.

⚠ The subunit for an operation is an *activity* definition. (Technically, this is the *method* for the operation.)

E. Active Classes

- Signals and Receptions
- Classifier Behaviors
- Asynchronous Behavior

Signals and Receptions

```
signal SubmitCharge {
    public card: CreditCard;
}
signal ChargeApproved {
    public charge: CreditCardCharge;
}
signal ChargeDenied {
    public charge: CreditCardCharge;
}
```

A *signal* is a classifier whose instances may be communicated *asynchronously.*

A signal may have *attributes* that represent transmittable data.

```
active class Order {
    …
    receive SubmitCharge;
    receive ChargeApproved;
    receive ChargeDenied;

} do OrderBehavior
```

A *reception* is a declaration of the ability to receive a signal. Only *active classes* can have receptions.

```
active class Customer {
    …
    receive ChargeApproved;
    receive ChargeDenied;

} do CustomerBehavior
```

More than one class can receive the same signal.

Classifier Behaviors

```
active class Order {
    …
    receive SubmitCharge;
    receive ChargeApproved;
    receive ChargeDenied;

} do {
     accept (submission: SubmitCharge);
     card = submission.card;

     do {
        new CreditCardCharge(card, this);

        accept (response: ChargeApproved) {
           this.customer.ChargeApproved(response.charge);
           break;

        } or accept (response: ChargeDeclined) {
           this.customer.ChargeDeclined(response.charge);

        }
     while (true);
}
```

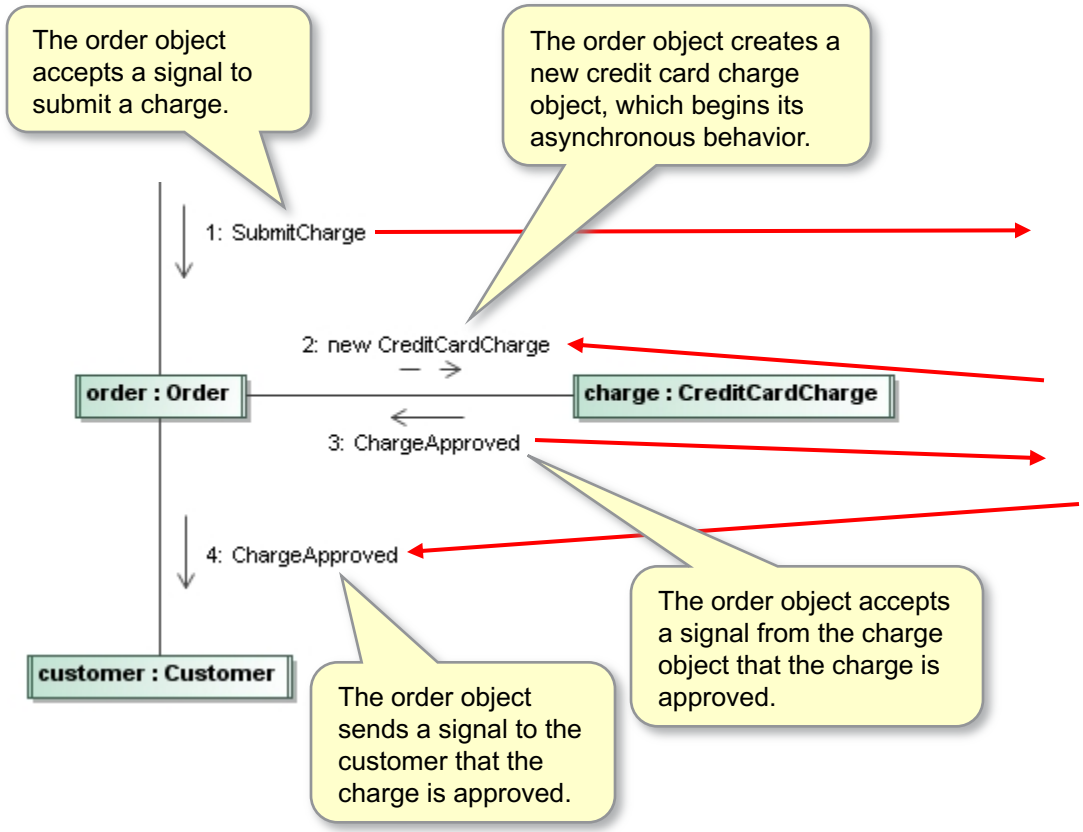
An *active* class is one that has a *classifier behavior,* which is an autonomous behavior started when the class is instantiated.

The classifier behavior may either be defined in-line or as a subunit.

An *accept statement* is used to receive signals within a classifier behavior.

The context class of the activity containing the accept statements must have receptions for the accepted signals.

Asynchronous Behavior



Note: UML semantics require a separate action to start the behavior of a new object. However, Alf notation for creating an active class maps to *both* create and start object behavior actions.

```
accept (submission: SubmitCha
card = submission.card;

do {
  new CreditCardCharge(card, this);

  accept (response: ChargeApproved) {
    this.customer.ChargeApproved(response.charge);
    break;

  } or accept (response: ChargeDeclined) {
    this.customer.ChargeDeclined(response.charge);

  }

while (true);
```

III. The Standard Model Library

- Primitive Behaviors
- Collection Functions
- Collection Classes
- Basic Input/Output

Primitive Behaviors

- **IntegerFunctions** – Arithmetic, Comparison, Conversion
- **RealFunctions** – Arithmetic, Comparison, Conversion
- **UnlimitedNaturalFunctions** – Comparison, Conversion
- **BooleanFunctions** – Logical Operations, Conversion
- **BitStringFunctions** – Bit-wise operations, Conversion
- **StringFunctions** – Concatenation, Size, Substring
- **SequenceFunctions** – *See CollectionFunctions*

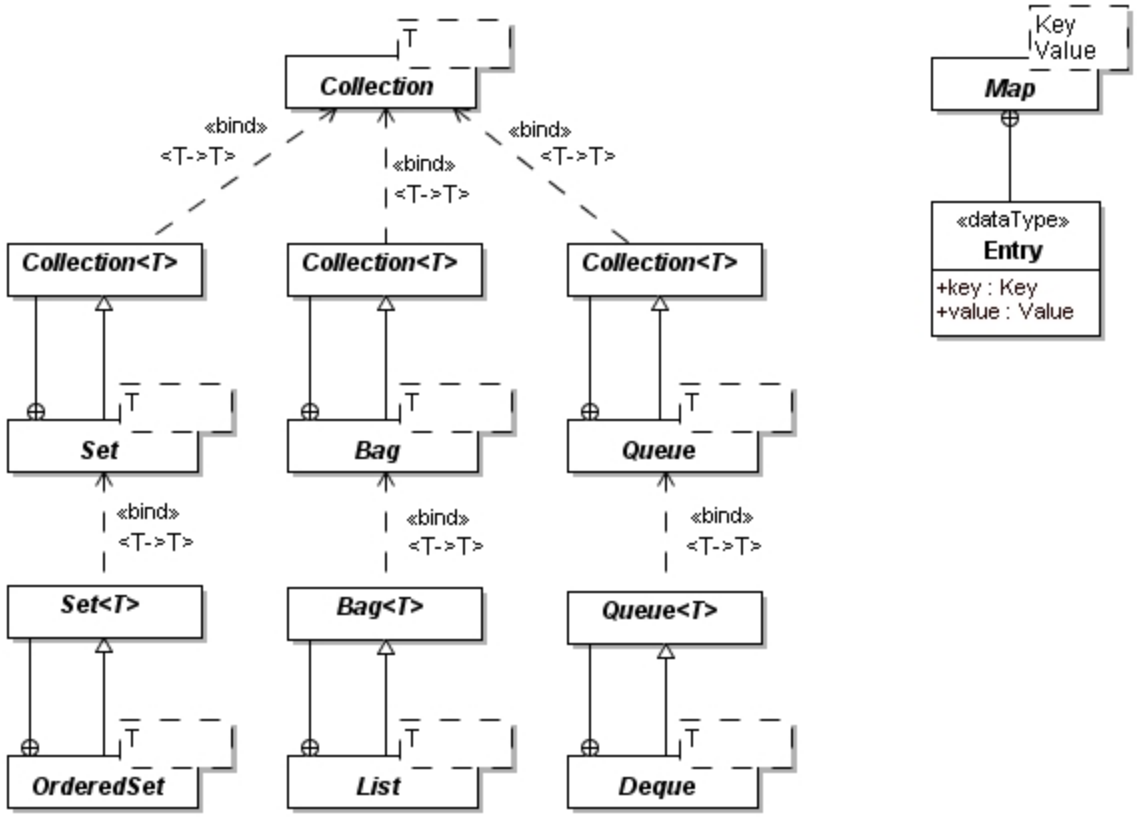
Collection Functions

Collection functions operate on sequences of values of any type.

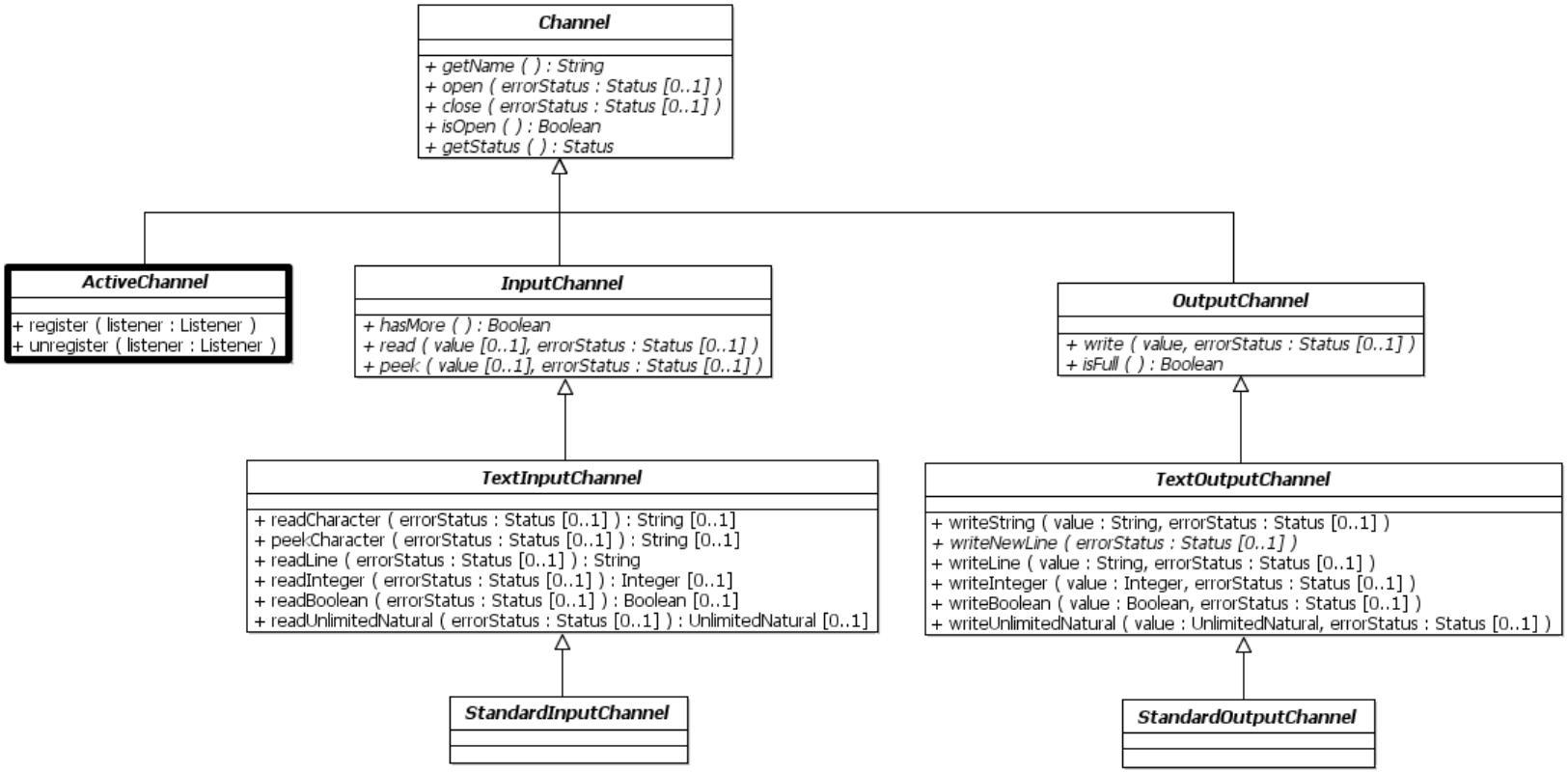
- **Testing and accessing functions**
  - size, count, isEmpty, notEmpty, includes, excludes, includesAll, excludesAll, equals, at, indexOf, first, last
  - *Examples:* seq->isEmpty(), seq1->equals(seq2), seq->at(n)
- **Non-mutating functions**
  - including, includeAt, includeAllAt, excluding, excludingOne, excludeAt, replacing, replacingOne, replacingAt, union, intersection, difference, subsequence, toOrderedSet
  - *Examples:* seq2 = seq->including(x), seq3 = seq1->union(seq2)
- **Mutating "in place" functions**
  - add, addAll, addAt, addAllAt, remove, removeAll, removeOne, removeAt, replace, replaceOne, replaceAt, clear
  - *Examples:* seq->add(x), seq1->addAll(seq2), seq2->remove(x)

Collection Classes

Collection classes define objects that represent collections of values of a given type.



Basic Input Output: Channels



Basic Input Output: Reading and Writing Lines