

---

---

# COMP 354: INTRODUCTION TO SOFTWARE ENGINEERING

Introduction to UML

Daniel Sinnig, PhD  
d\_sinnig@cs.concordia.ca

Department for Computer Science  
and Software Engineering

28-May-14

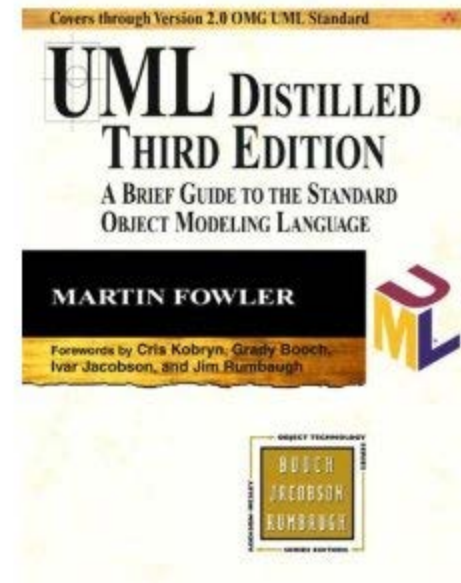


## Unified Modeling Language

- Structural Diagrams – focus on static aspects of the software system
  - **Class**, Object, Component, Deployment
- Behavioral Diagrams – focus on dynamic aspects of the software system
  - Use-case, **Interaction** (Sequence, Communication), State Chart, Activity

## UML Reference

- **Martin Fowler**, *UML distilled : a brief guide to the standard object modeling language*. 3rd edition, 2003.



## Class Diagrams

- Structural model

Show

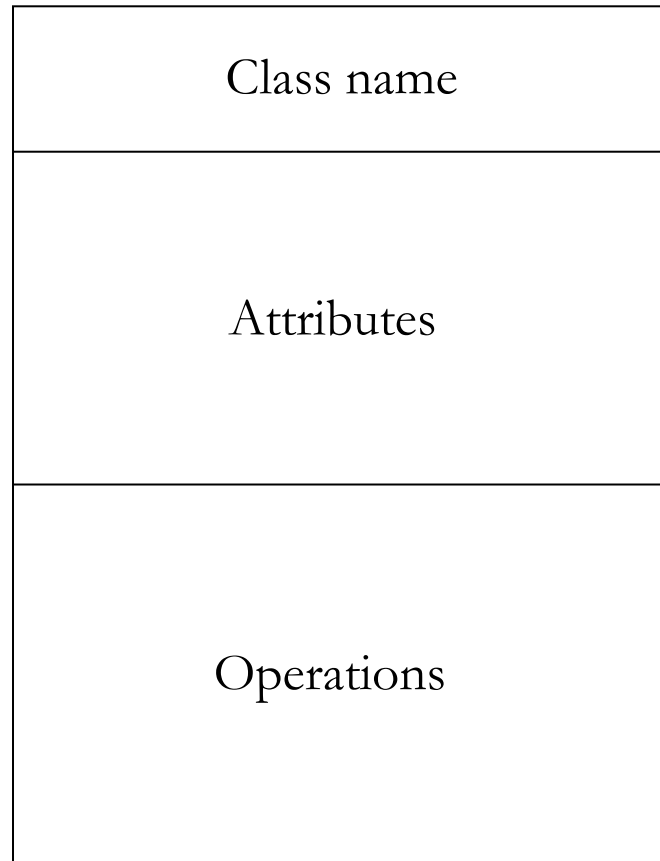
- Static view of the domain / system
- Relationship between entities
- Illustrate structural features / relationships.

Do not show

- Temporal information
  - Behavior
  - Runtime constraints
- In this course the class diagram notation is used for:
    - **(Business) Domain modeling**
    - **Static object modeling**

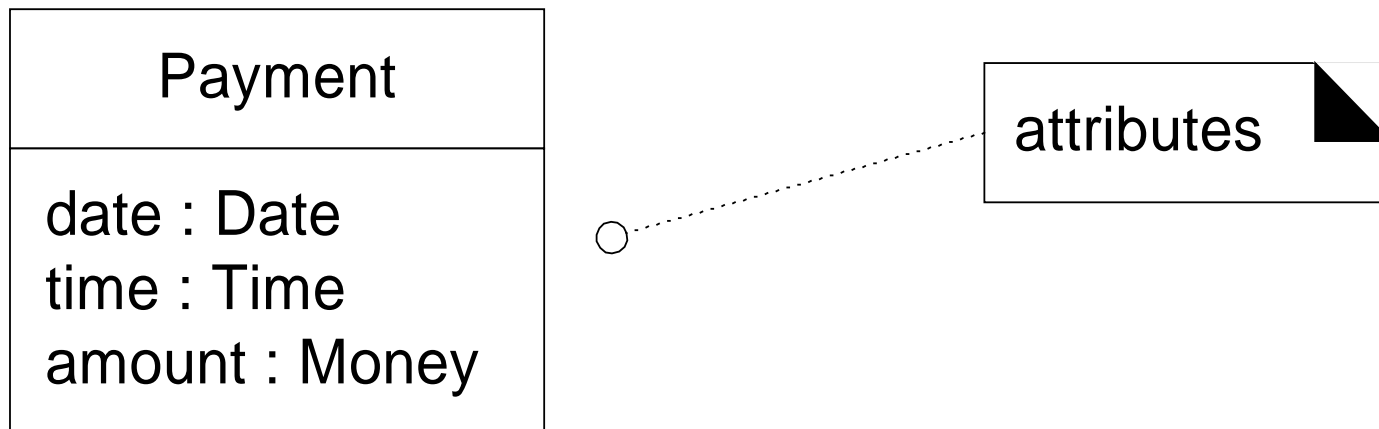
# Class Diagrams: Notation

## UML Class Diagram – Class

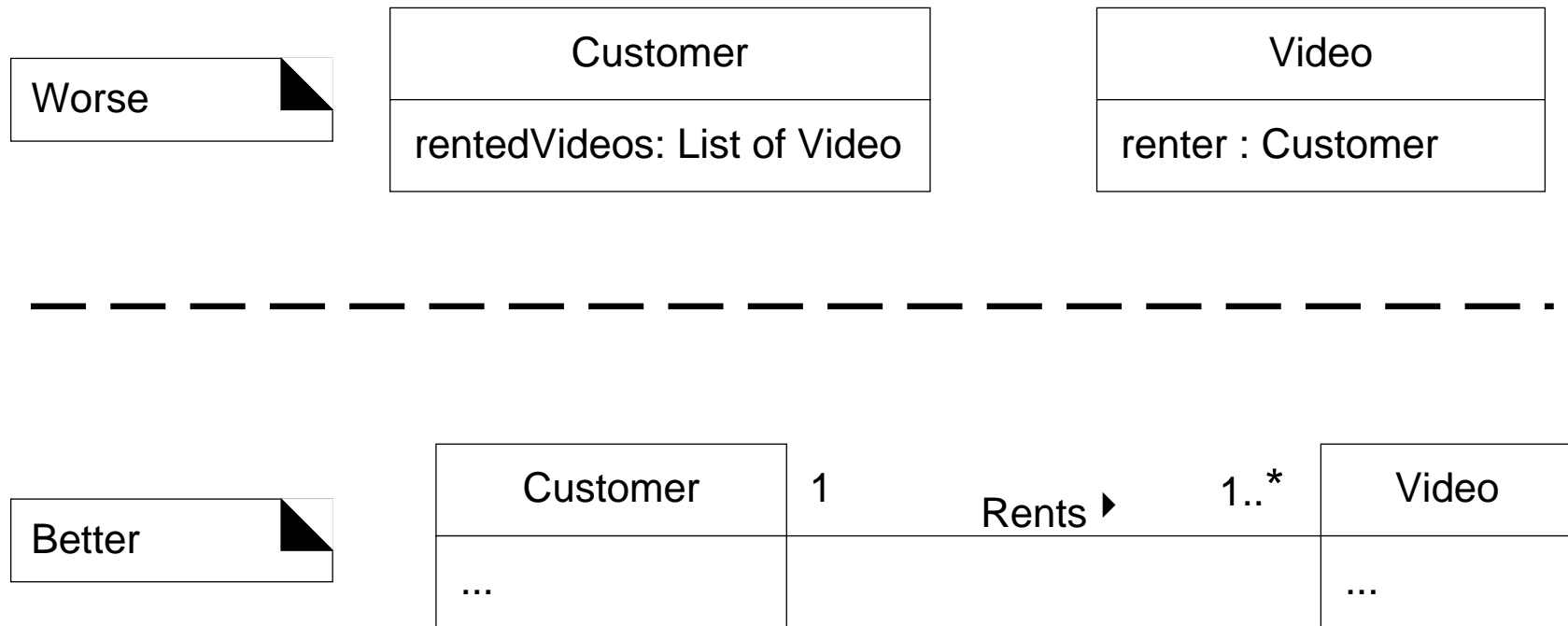


## Attributes vs. Associations

- Show only “simple” relatively primitive types as attributes.
- Connections to other concepts are to be represented as associations, not attributes.
- Syntax: *visibility name: type [multiplicity] {property}*

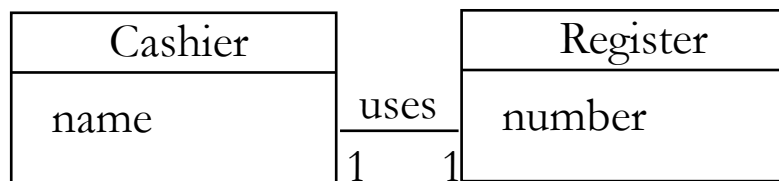
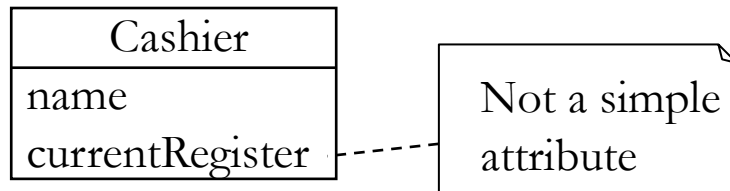


## Attributes vs. Associations





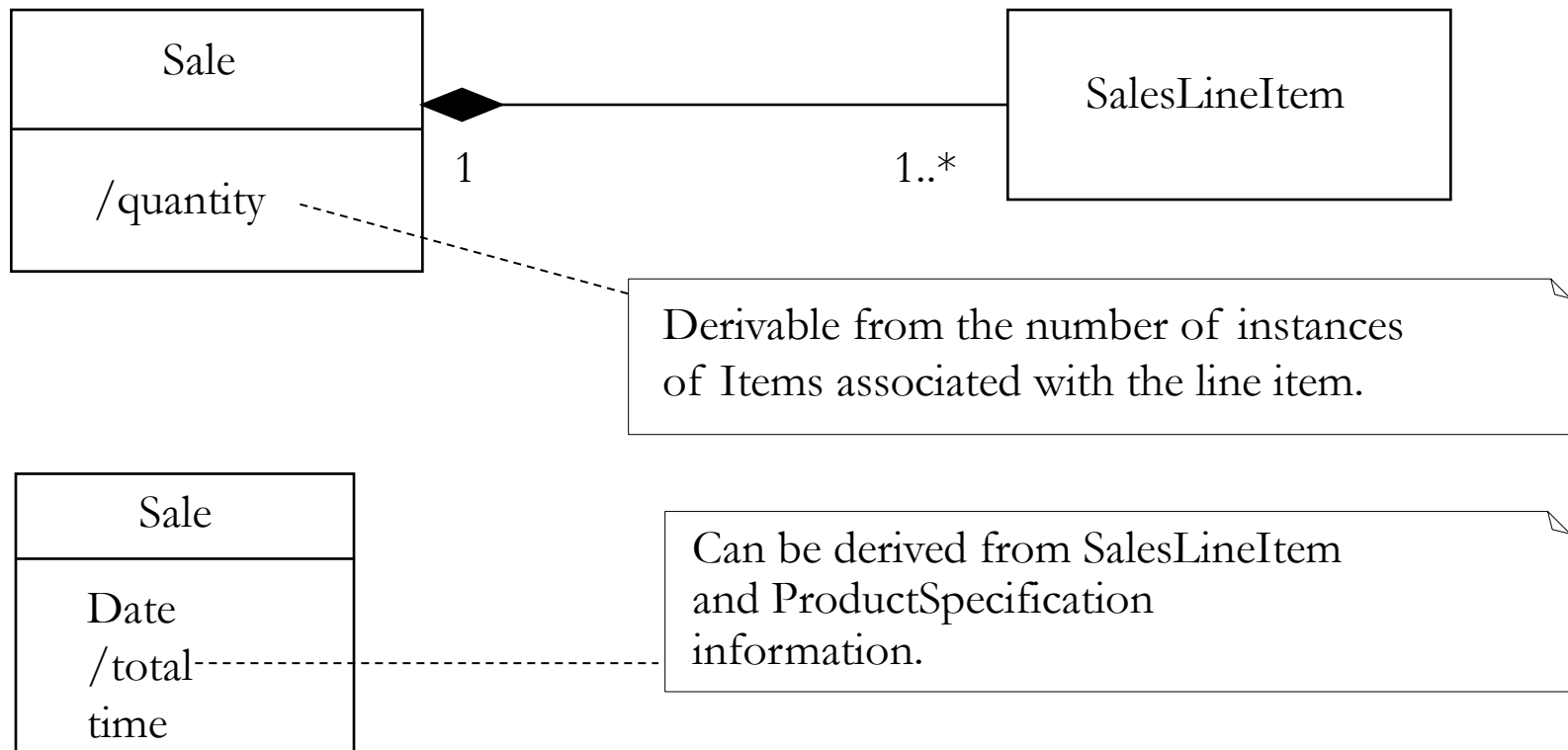
## Valid Attribute Types



- Keep attributes simple.
- The type of an attribute should not normally be a complex domain concept, such as Sale or Airport.
- Attributes in a Domain Model should preferably be
  - Pure data values: Boolean, Date, Number, String, ...
  - Simple attributes: color, phone number, zip code, universal product code (UPC), ...

## Derived Elements

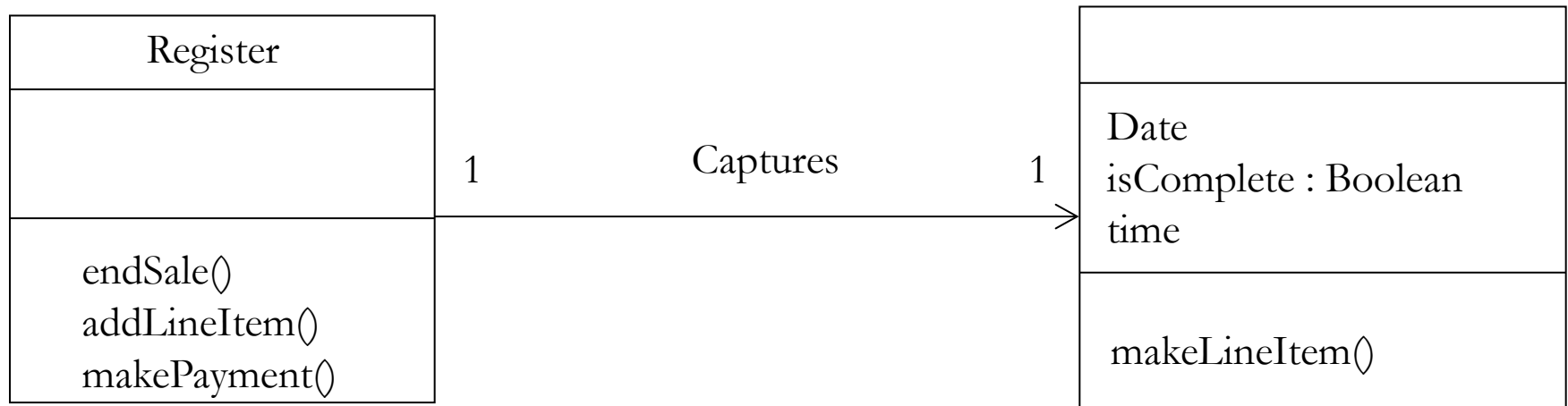
- A derived element can be determined from others.



# Operations

- **Syntax:** *visibility return type name (parameter list) {property}*
- Parameter list often omitted
- Return type is not UML 2 conform (only UML 1)
- Accessing operation typically excluded

## Operations Example



### Visibility

+ (public)

- (private)

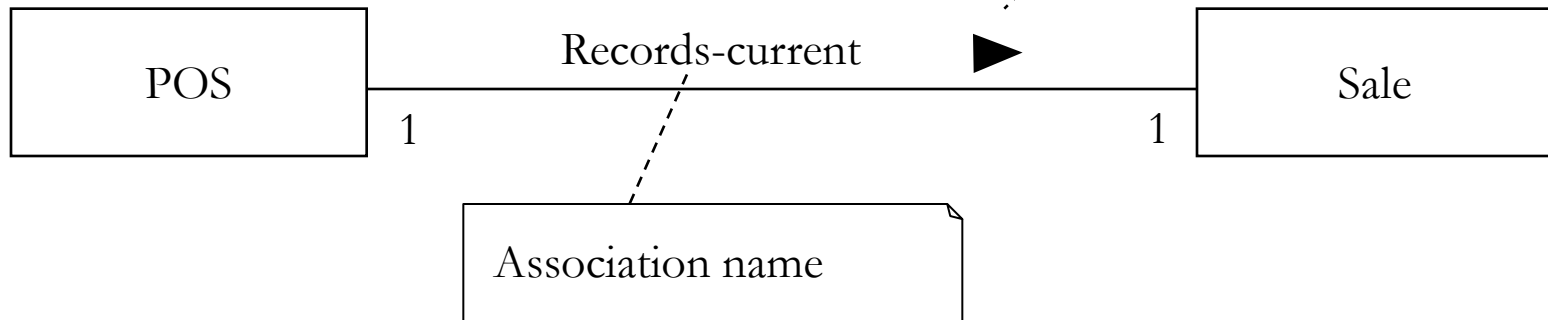
# (protected)

- Attributes are by default private
- Operations are by default public

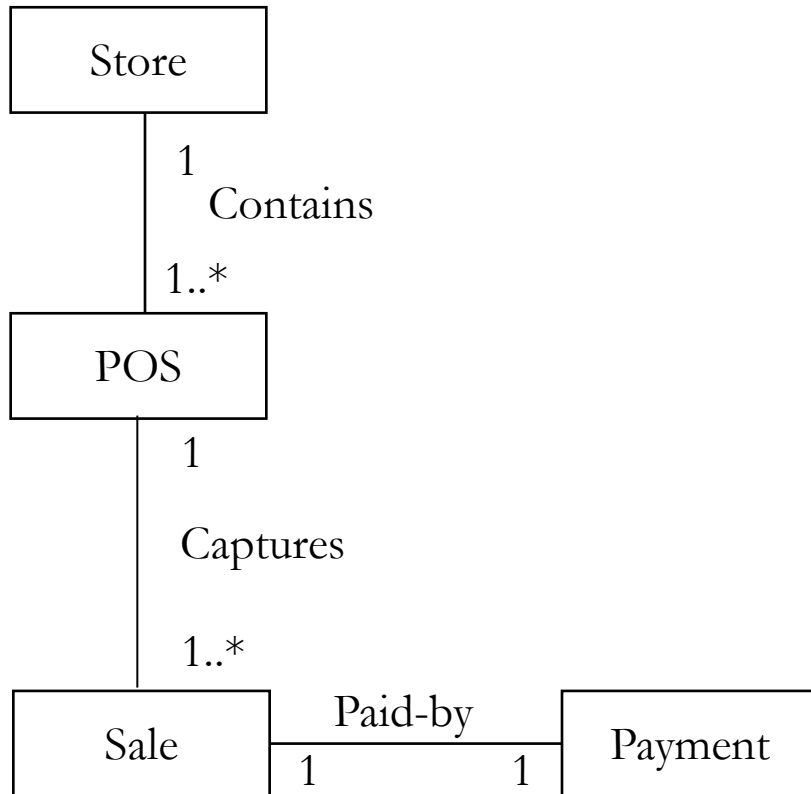
## Associations

An association is a relationship between entities that indicates some meaningful and interesting connection.

“Direction reading arrow” has no meaning other than to indicate direction of reading the association name.  
Optional (often excluded)

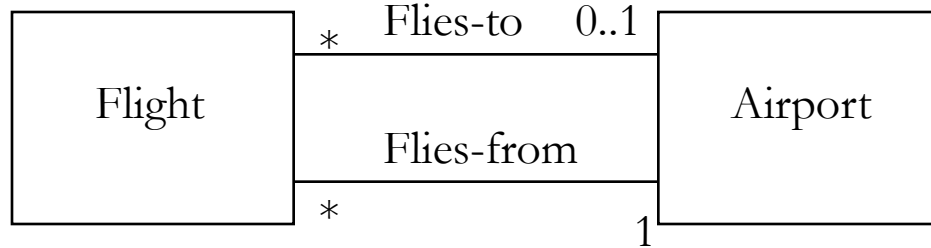


## Naming Associations



- Name an association based on a TypeName-VerbPhrase-TypeName format.
- Association names should start with a capital letter.
- A verb phrase should be constructed with hyphens.
- The default direction to read an association name is left to right, or top to bottom.
- Primarily used for problem domain modeling

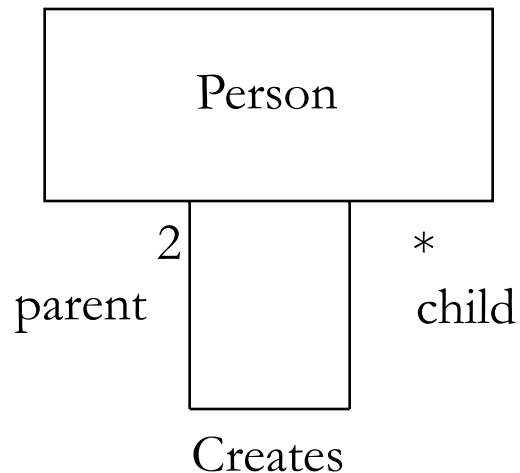
## Multiple Associations Between Two Types



- It is not uncommon to have multiple associations between two types.
- In the example, not every flight is guaranteed to land at an airport.

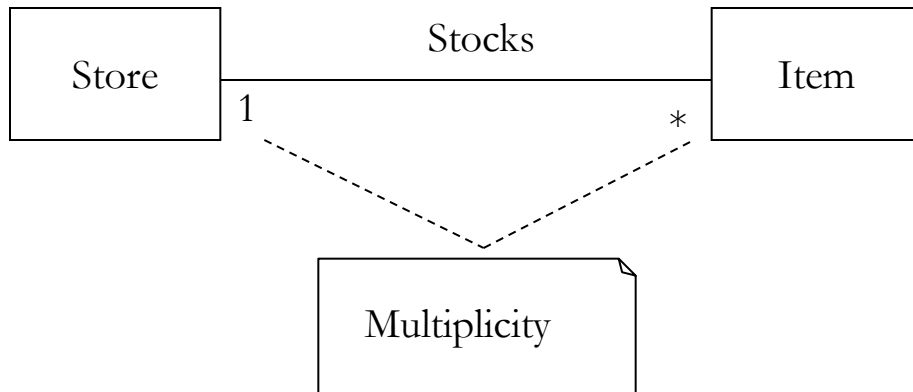


## Recursive or Reflexive Associations



- A concept may have an association to itself; this is known as a recursive association or reflective association.

## Multiplicity



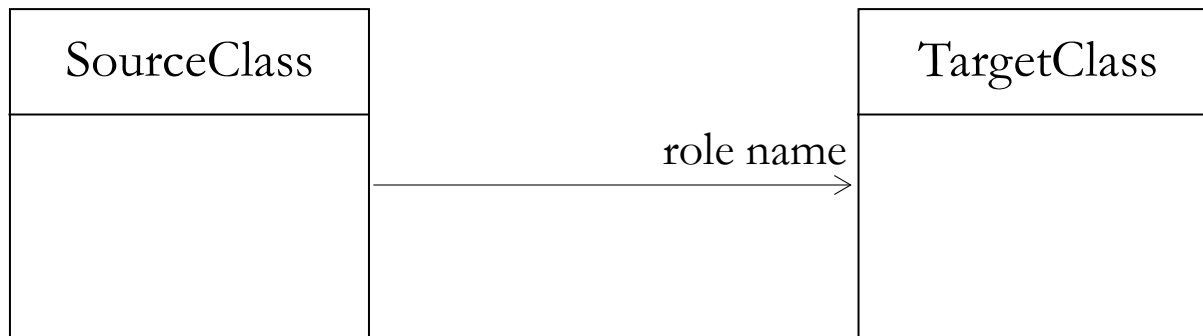
- Multiplicity defines how many instances of a type *A* can be associated with one instance of a type *B*, at a particular moment in time.
- For example, a single instance of a Store can be associated with “many” (zero or more) Item instances.

## Multiplicity

*	T	Zero or more; “many”
1..*	T	One or more
1..40	T	One to forty
5	T	Exactly five
3, 5, 8	T	Exactly three, five or eight.

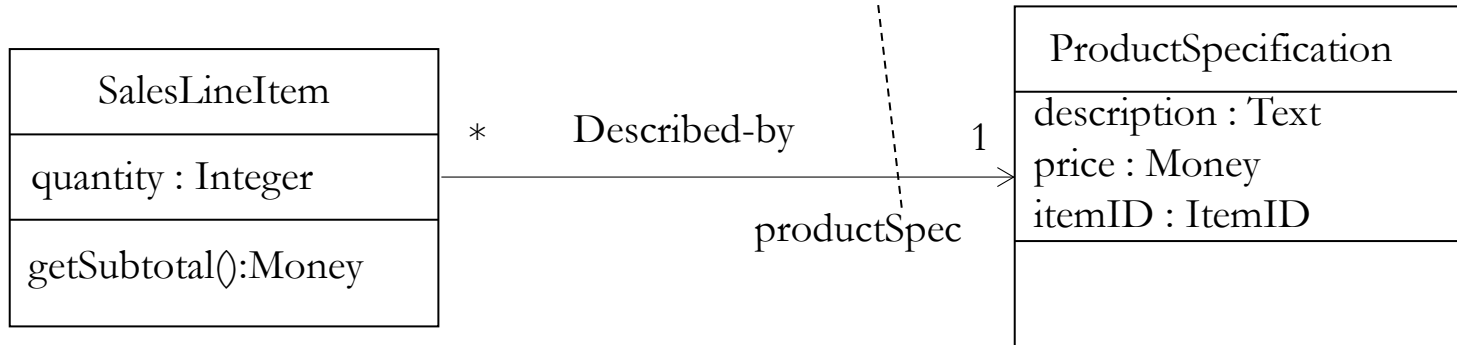
## Navigability Arrows and Role Names

- Directed
  - Source class has attribute called “role name” of type Target class (for object design models only)
- Bidirectional
  - Two directed associations.
  - Typically arrowheads are not show.



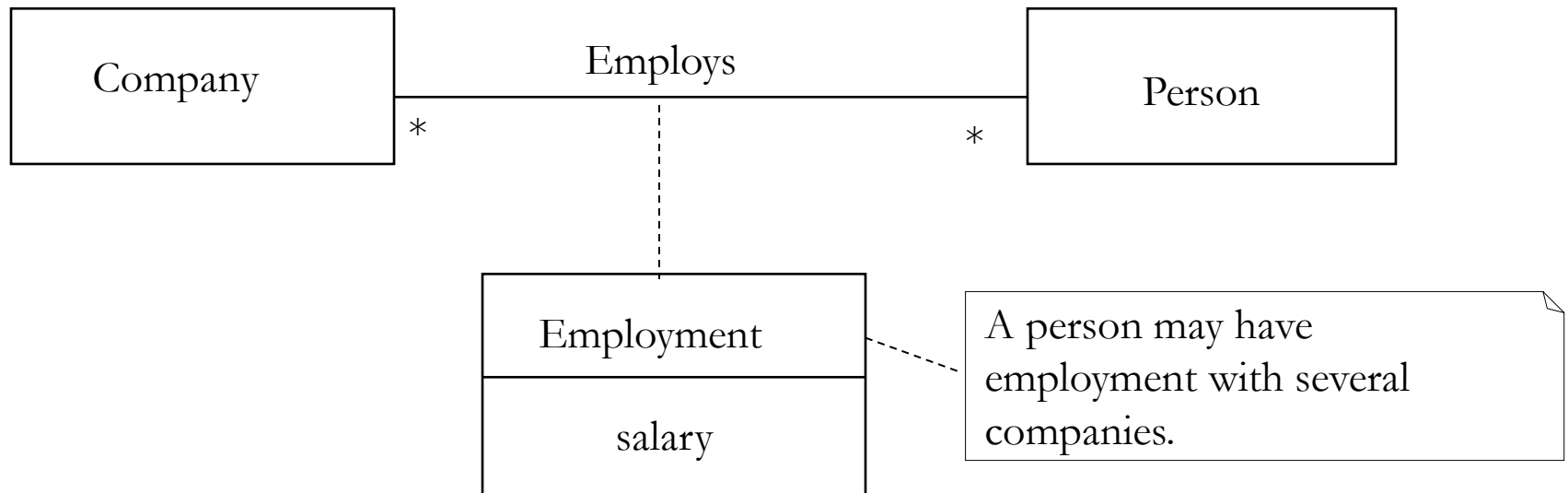
## Example

```
public class SalesLineItem {  
  
    private int quantity;  
    private ProductSpecification productSpec;  
    ...  
}
```



## Association Classes

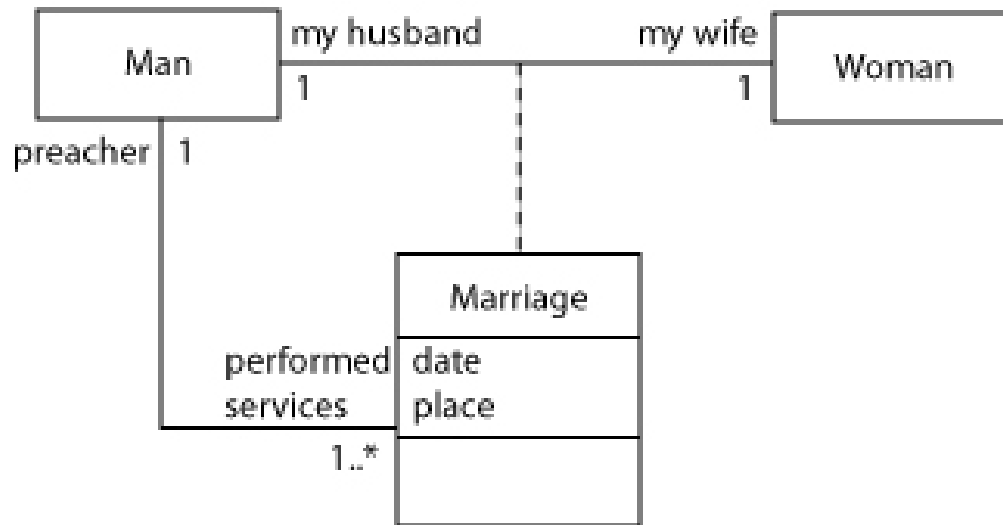
- Used to further qualify an association



## Association Classes

- Useful when the information related to the association does not fit in any of the target classes.

Monogamous marriage as association class



<http://www.devx.com/enterprise/Article/28576/1763?supportItem=1>

## Guidelines for Association Classes

- An attribute is related to an association.
- Instances of the association class have a life-time dependency on the association.
- The presence of a many-to-many association between two concepts is often a clue that a useful associative type should exist in the background somewhere.



## Composite Aggregation - Filled diamond




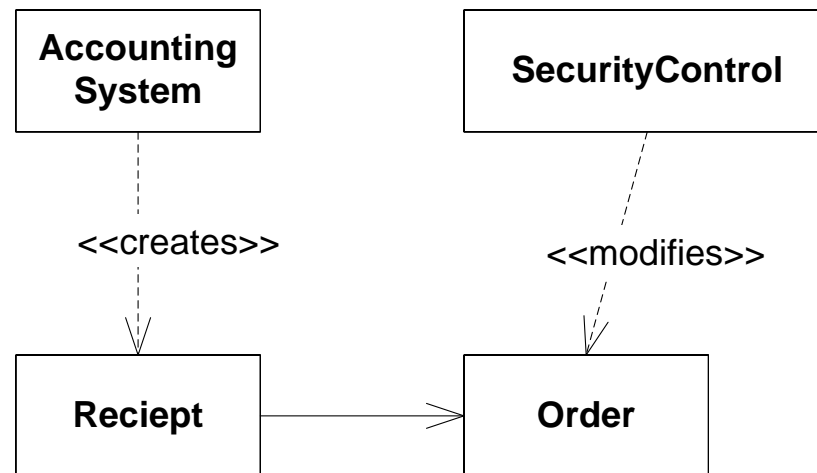
- Composite aggregation or composition means that the multiplicity at the composite end may be at most one (signified with a filled diamond).
- ProductCatalog is composed of ProductSpecification.

## How to identify Aggregation

- The lifetime of the part is bound within the lifetime of the composite.
- There is a create-delete dependency of the part on the whole.
- There is an obvious whole-part physical or logical assembly.
- Some properties of the composite propagate to the parts, such as its location.
- Operations applied to the composite propagate to the parts, such as destruction, movement, recording.

## Dependency

- Notated by a dotted line 
- Most general relation between classes
- A dependency is a *using relationship* that states that *a change ... of one thing may affect another* that uses it.

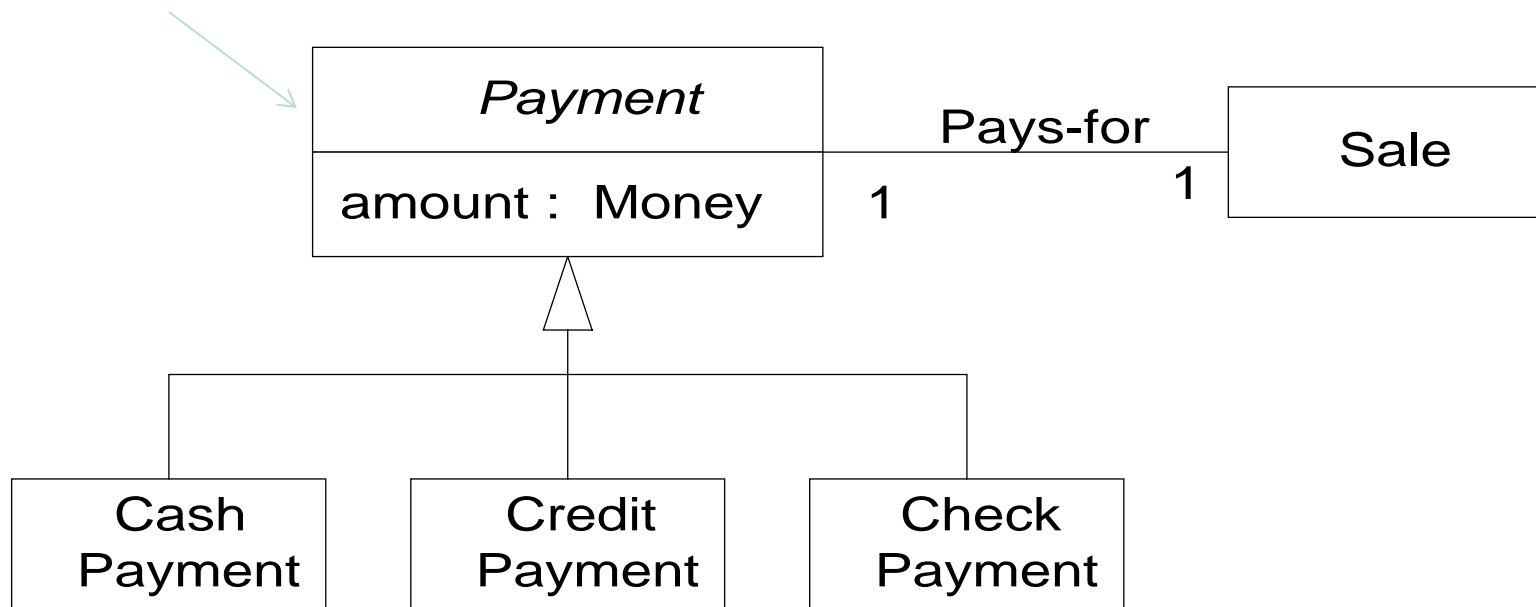


### Dependency – cont'd

- Dependencies are the most abstract type of relations.
- Properties:
  - Dependencies are always directed (If a given class depends on another, it does not mean the other way around).
  - The arrow points to the depended-on class/concept
  - Dependencies do not have cardinality.
- If instances of two classes send messages to each other, but are not tied to each other, then dependency is appropriated.

# Generalization

Superclass




Subclasses

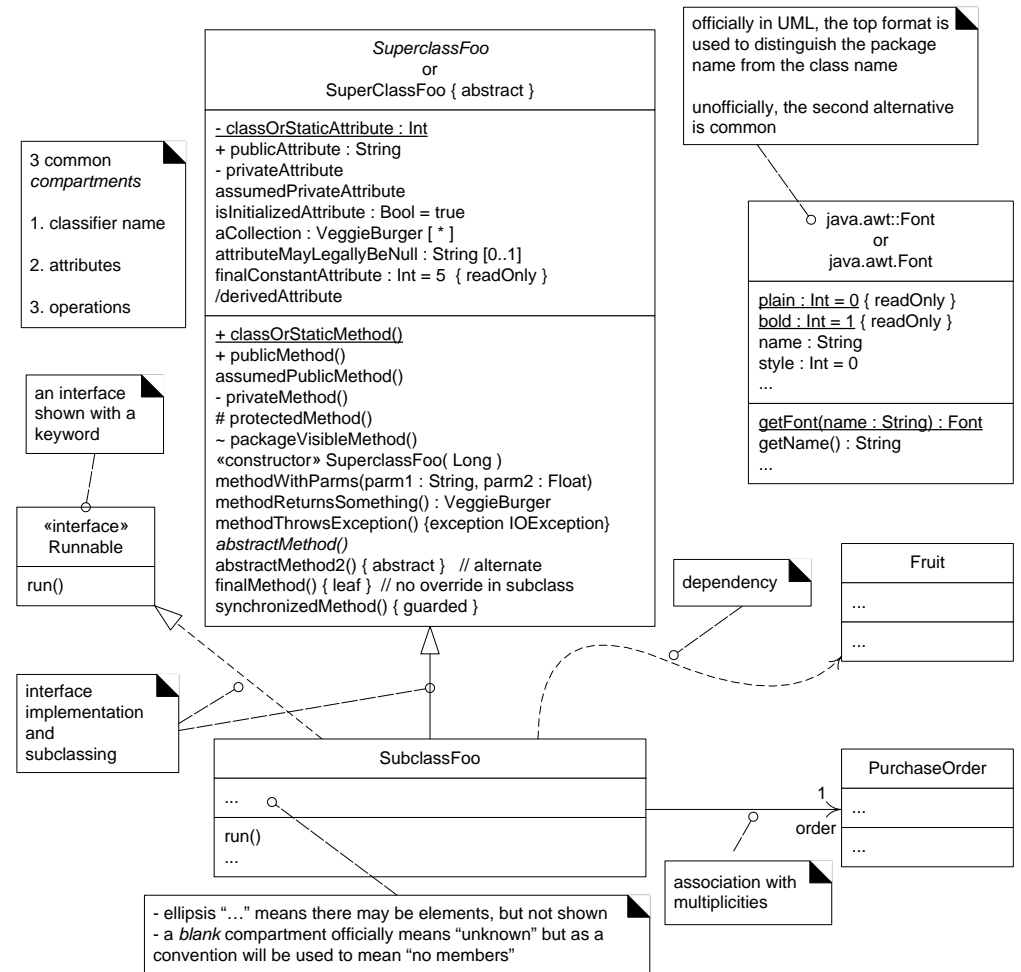
## Generalization

- Super (or parent) class – is the more general concept;
- Sub *is-a-kind-of* Super - is the more specific concept

# Properties, Stereotypes, Notations

- {ordered}
- <<actor>>
- <<singleton>> (1)
- *abstract class* (italic)
- static method, attribute (underlined)
- <<interface>>
- class implementing an interface 

## UML Class Diagrams





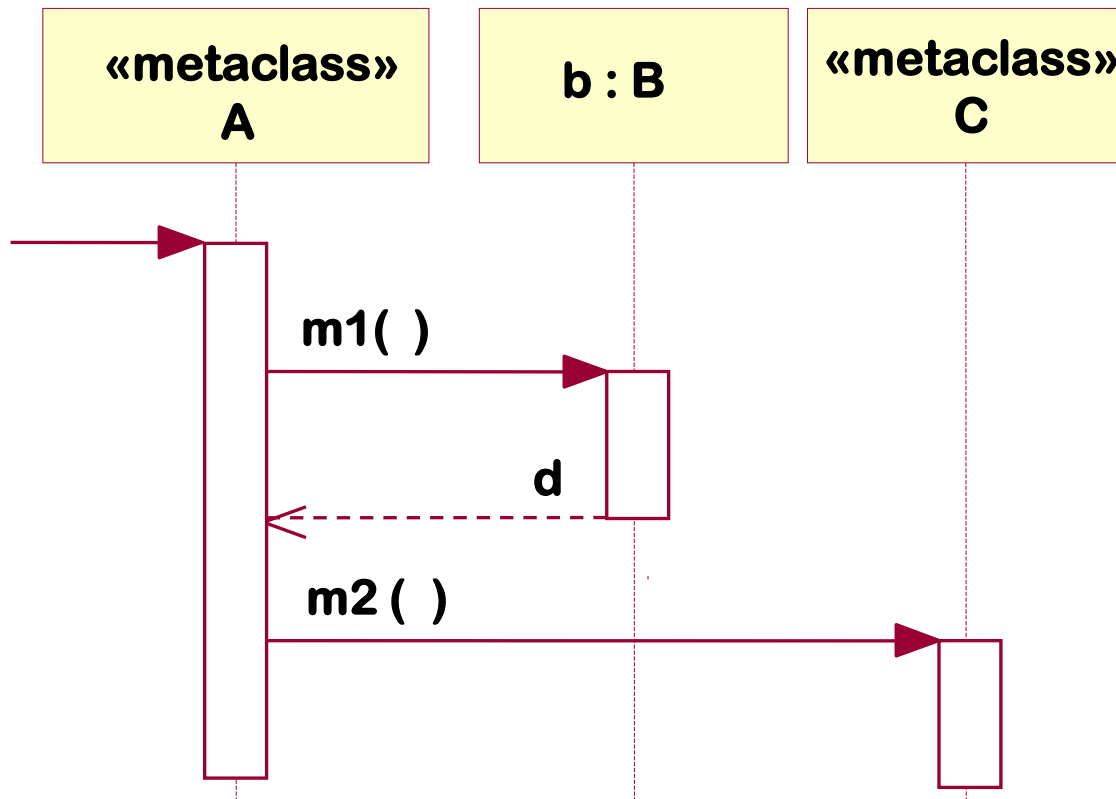
# UML Interaction Diagrams

- Behavioral model
  - Shows
    - Dynamic view of system’s internal and external interactions
    - Message exchange between instances
    - Temporal information
- UML provides two different notations:
  - **UML Sequence Diagrams**
  - UML Communication Diagrams
- In this course the interaction diagrams are used for:
  - **Black box modeling** (System Sequence Diagram)
  - **White box modeling** (Sequence and Communication Diagram)

# Sequence Diagram Notation

**“*Sequence diagrams* illustrate interactions in a kind of fence format.”**

# Sequence Diagram



# UML: Sequence Diagrams, Basic Notation

- Participants:

c : Customer,

r : Rental,

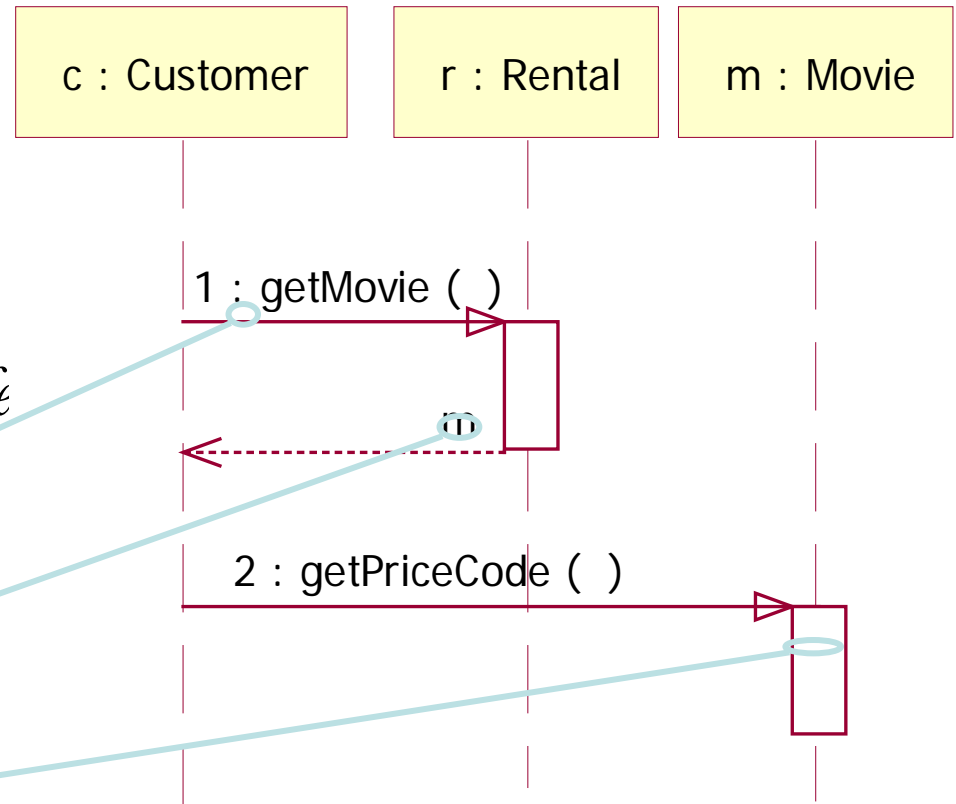
m : Movie

- Each participant has a *life line* (vertical dashed line)

- Synchronous message

- Return result

- Activation bar



# Sequence Diagrams: Participants

- Participants:

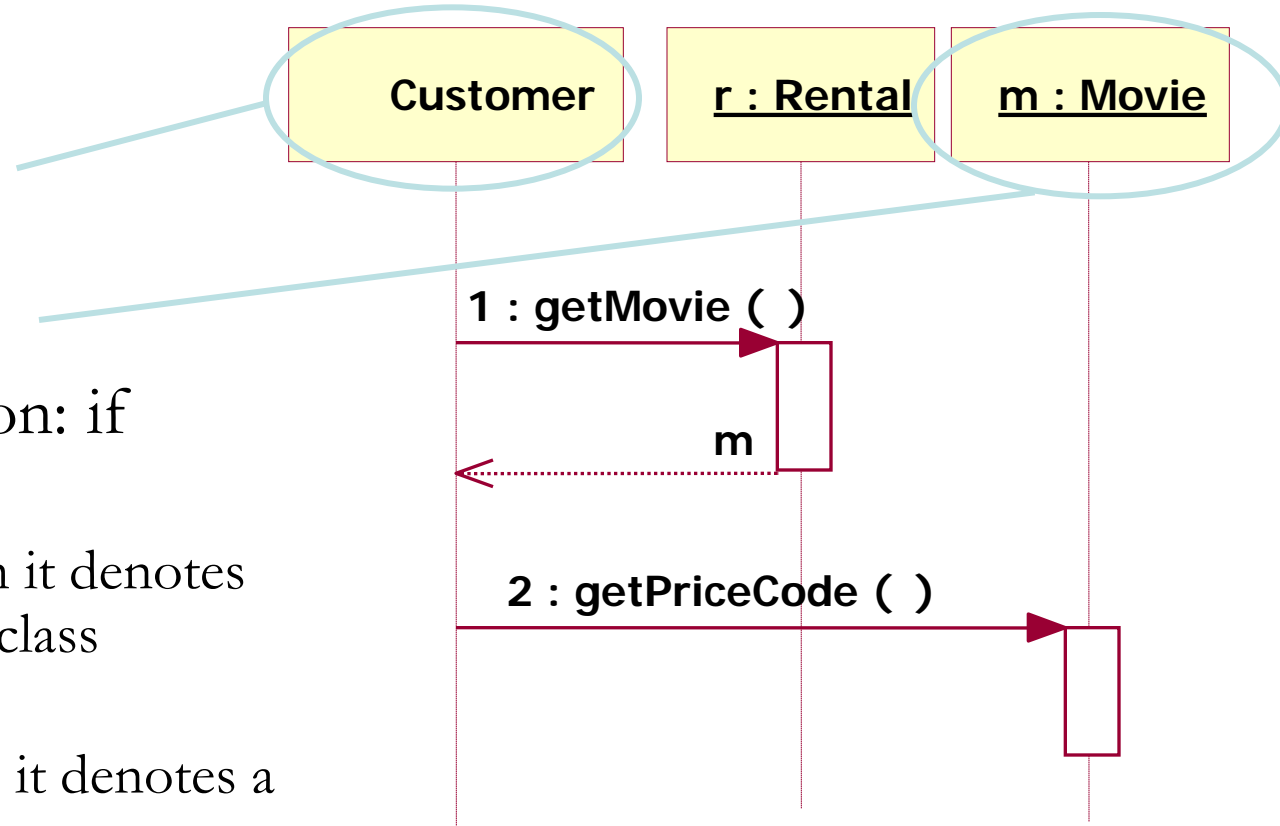
Customer *class*

r : Rental (*object*)

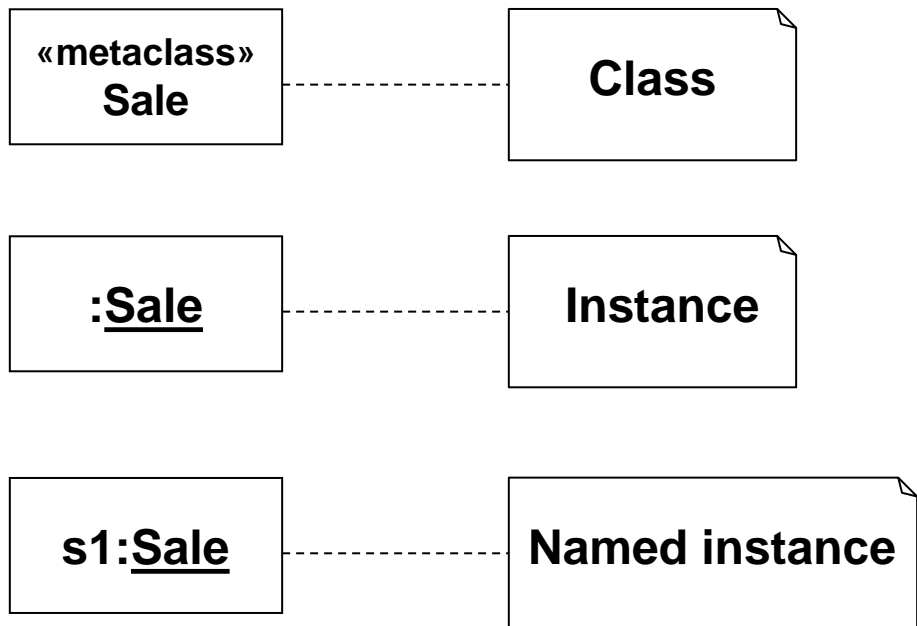
m : Movie (*object*).

- UML1 convention: if participant is ...

- Underlined then it denotes an object (i.e. a class instance).
- Not underlined: it denotes a class.

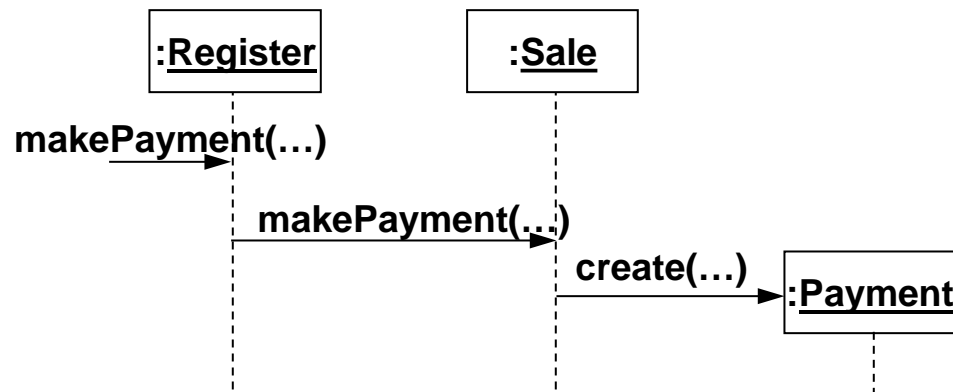


## Illustrating Classes and Instances



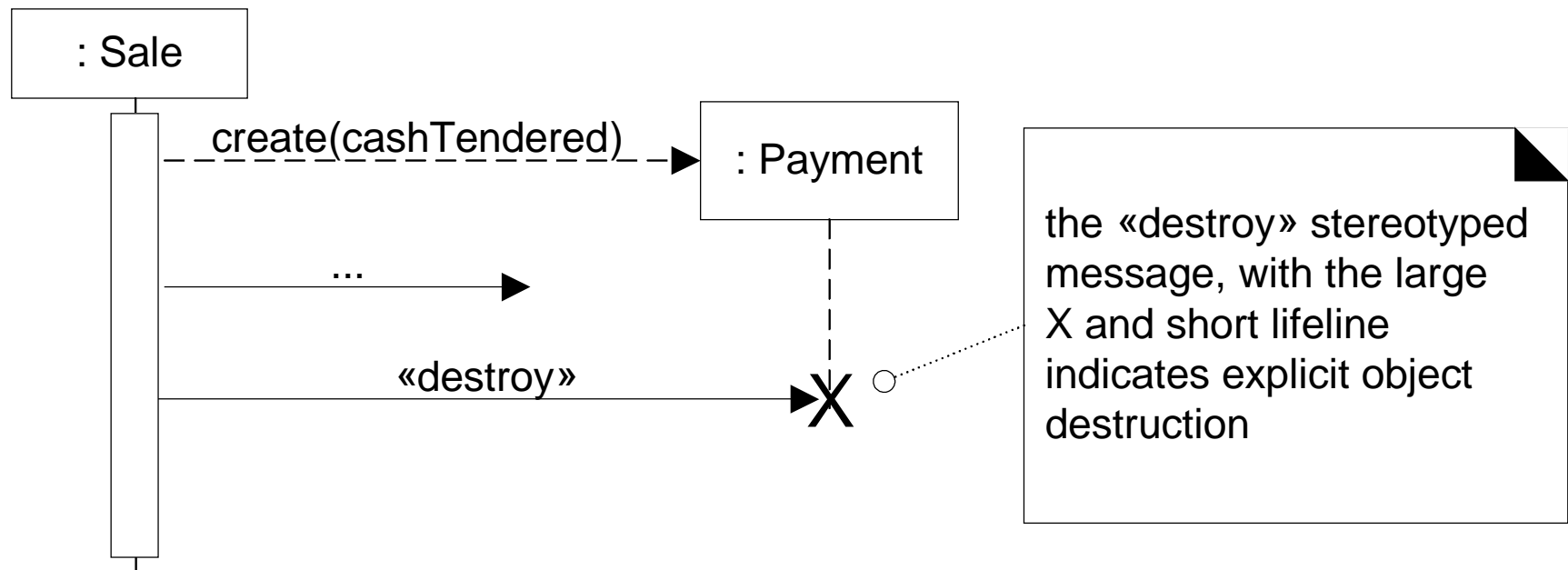
- To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined. Additionally a class name should be preceded by a colon.
- An instance name can be used to uniquely identify the instance.

## Creation of Instances



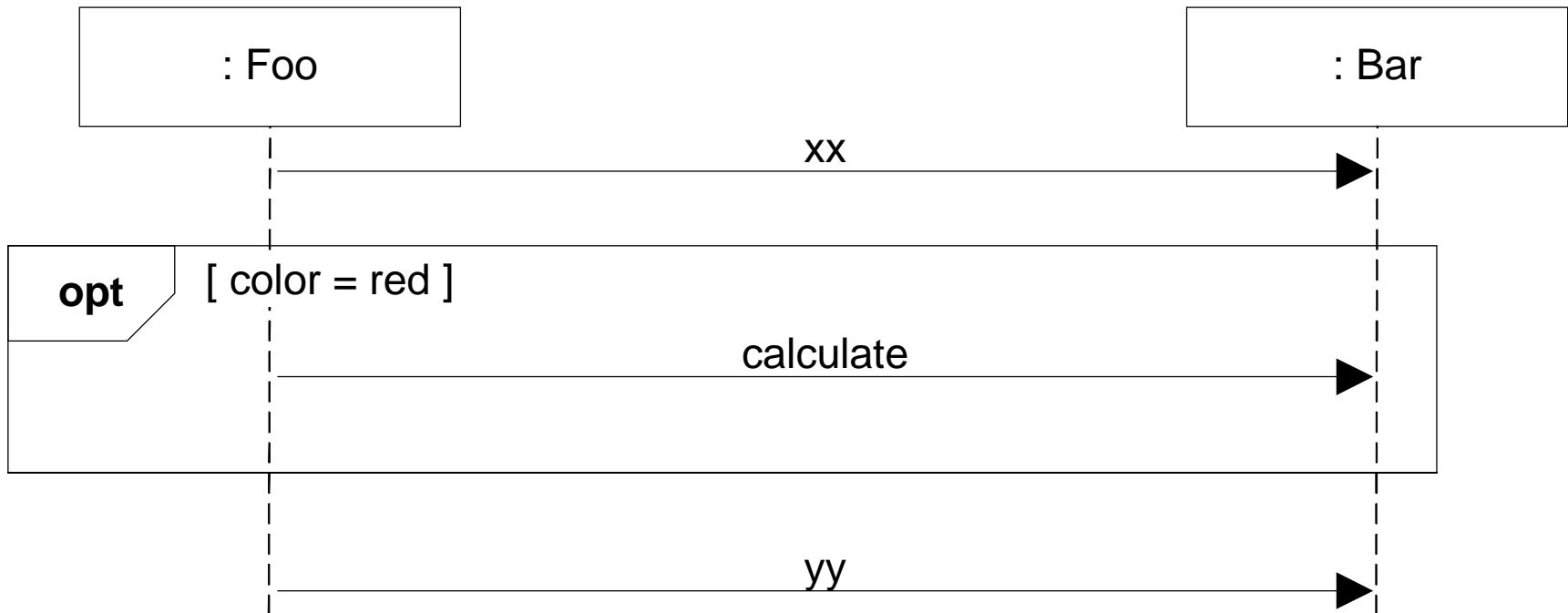
- An object lifeline shows the extend of the life of an object in the diagram.
- Note that newly created objects are placed at their creation height.

## Destruction of Instances

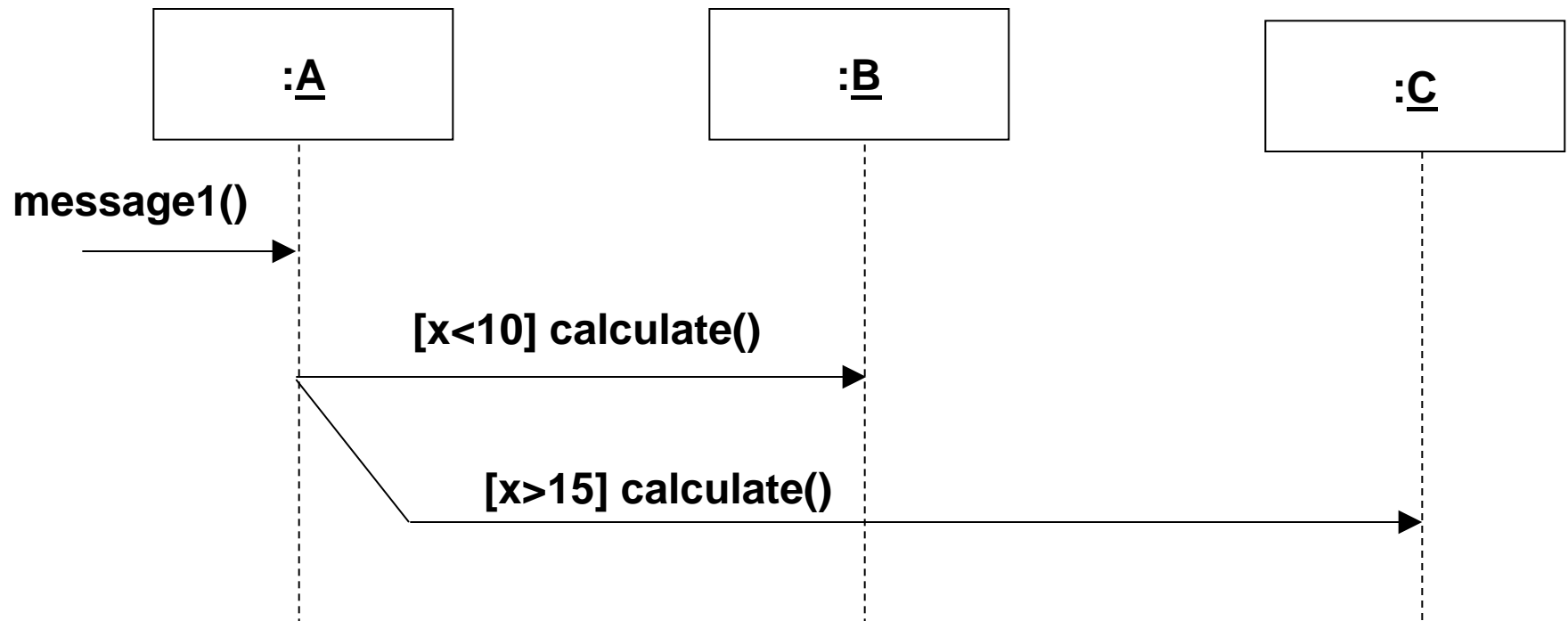




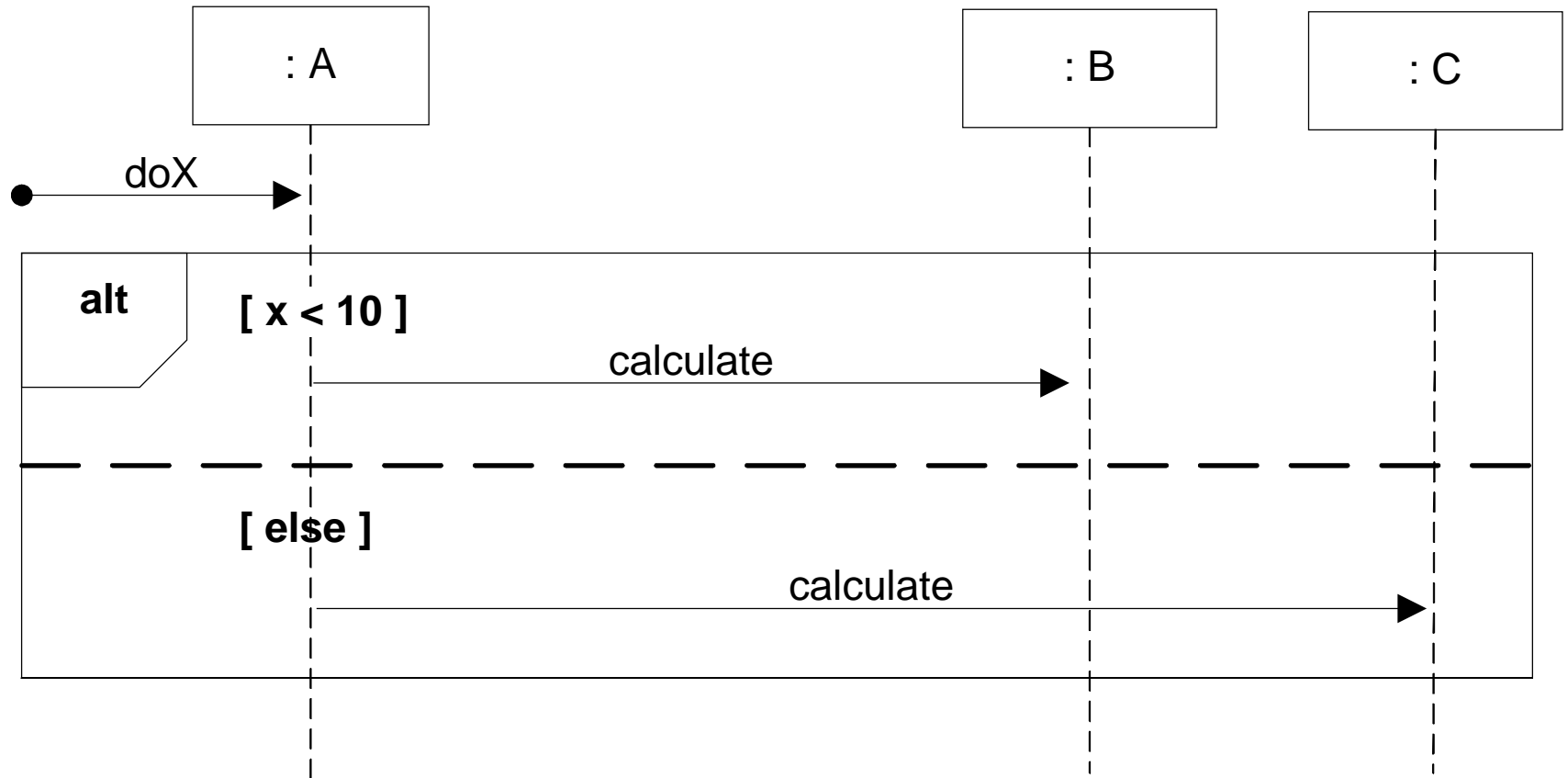
# Conditional Messages



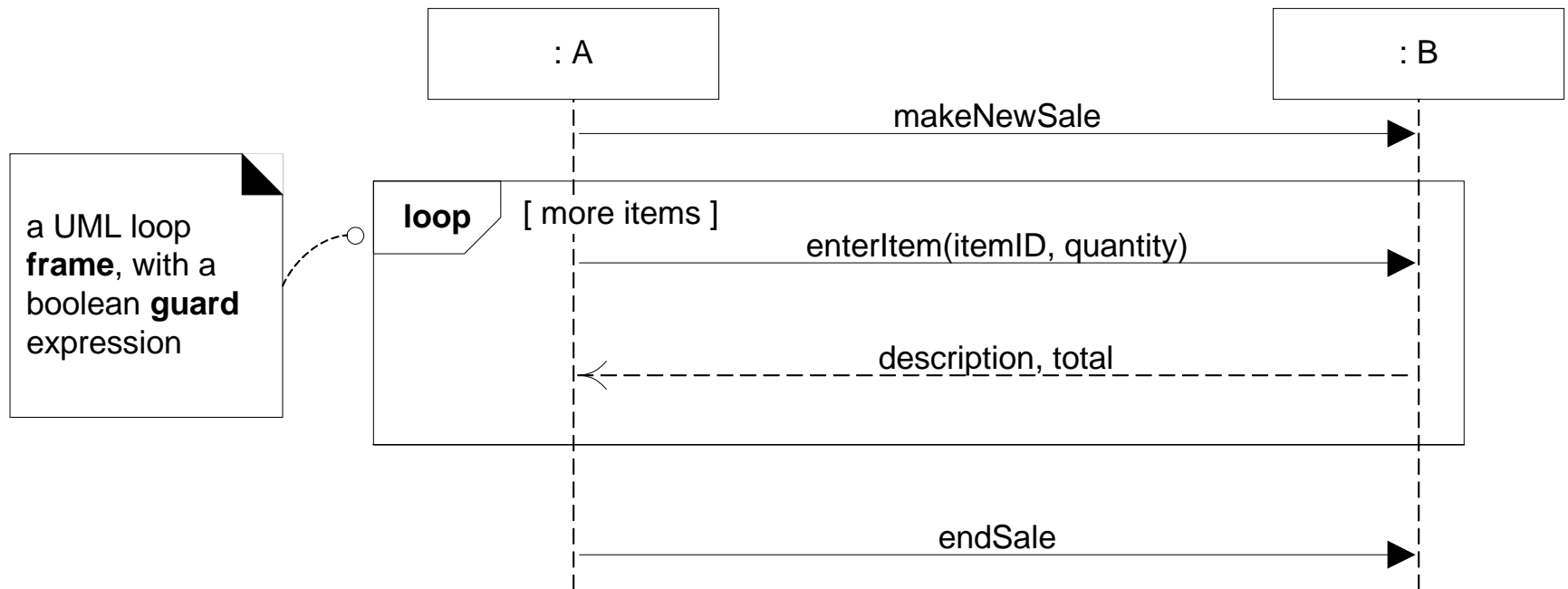
## Mutually Exclusive Conditional Messages



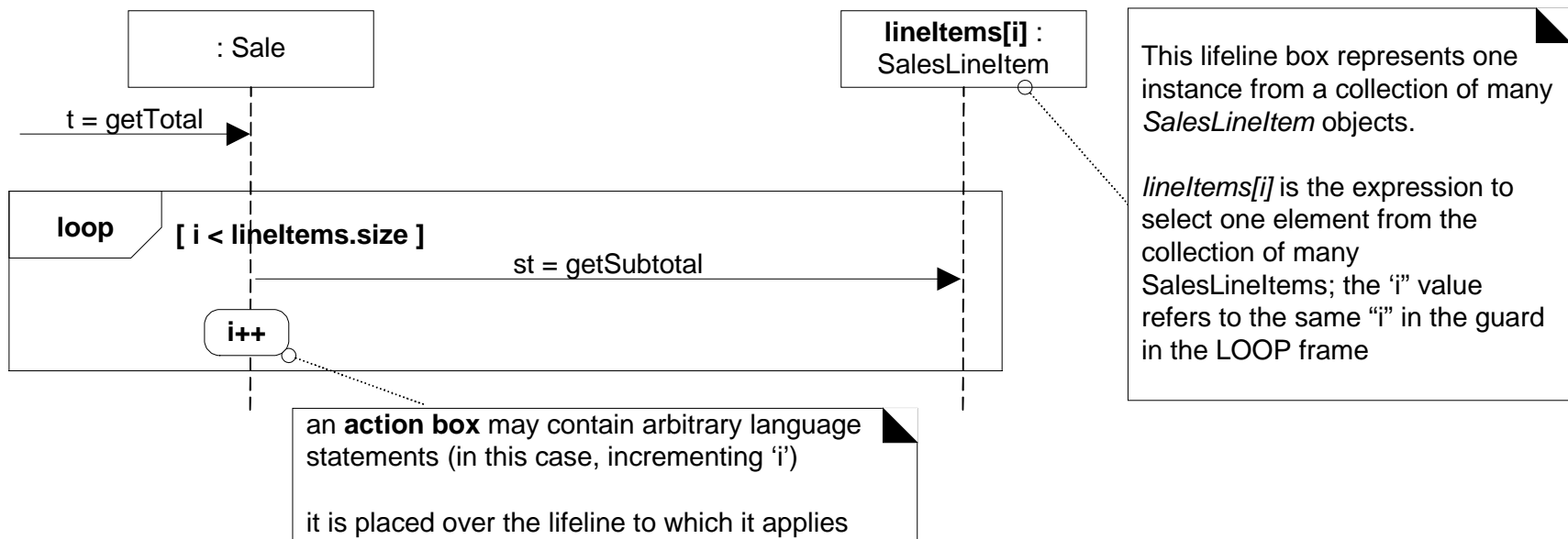
# Mutually Exclusive Conditional Messages



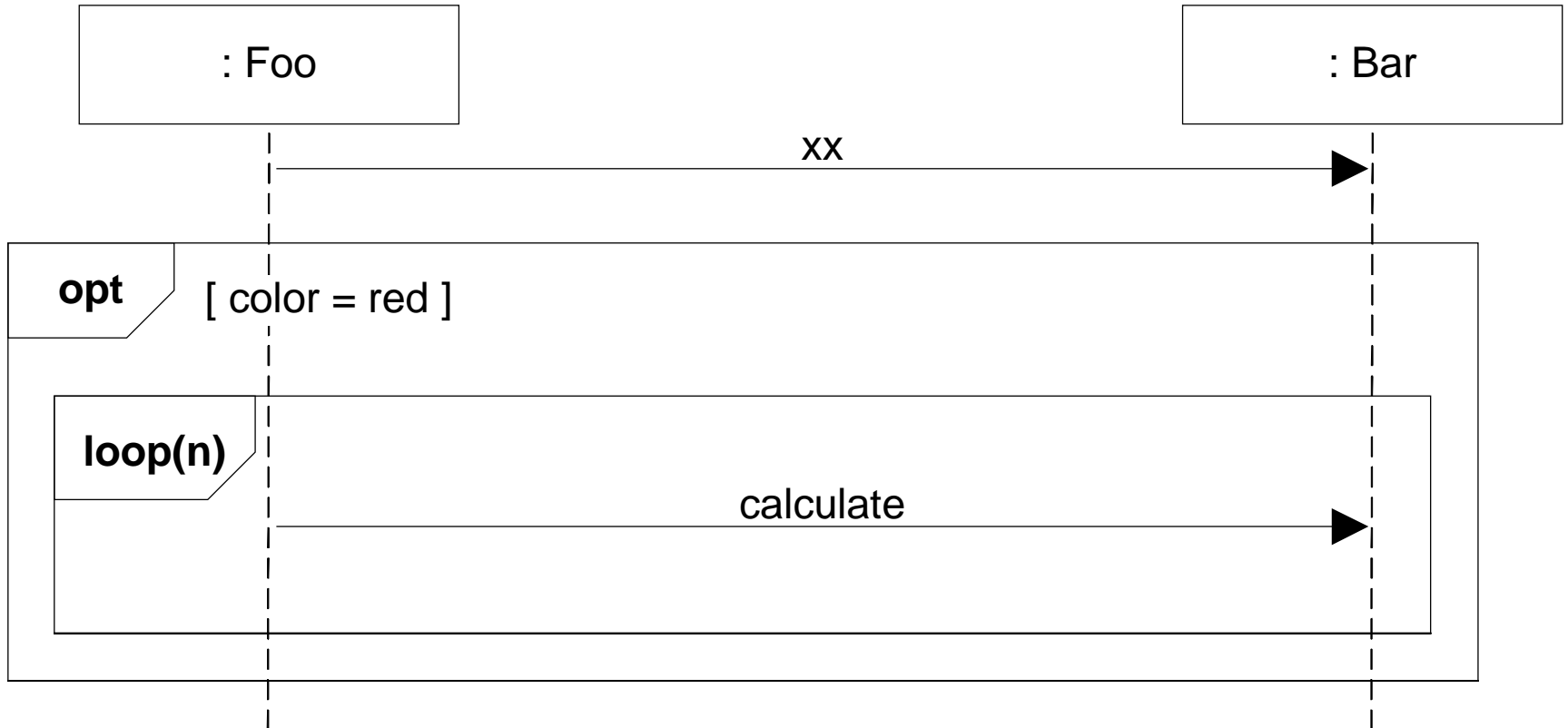
## Loops



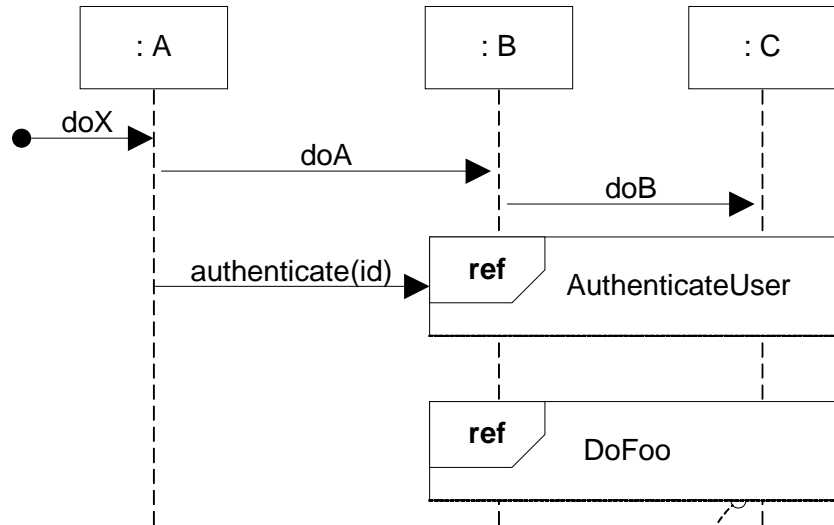
## Iterating through a List



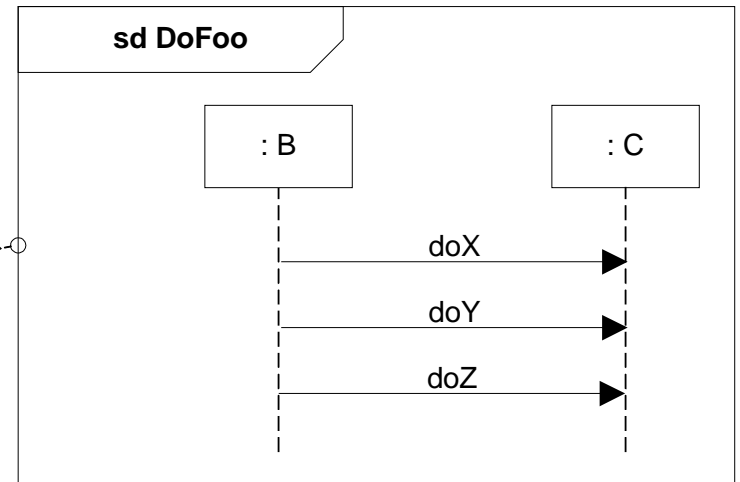
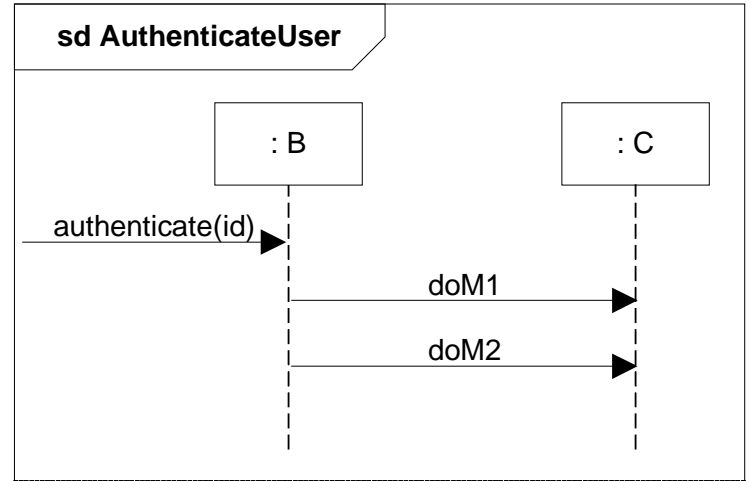
# Nesting



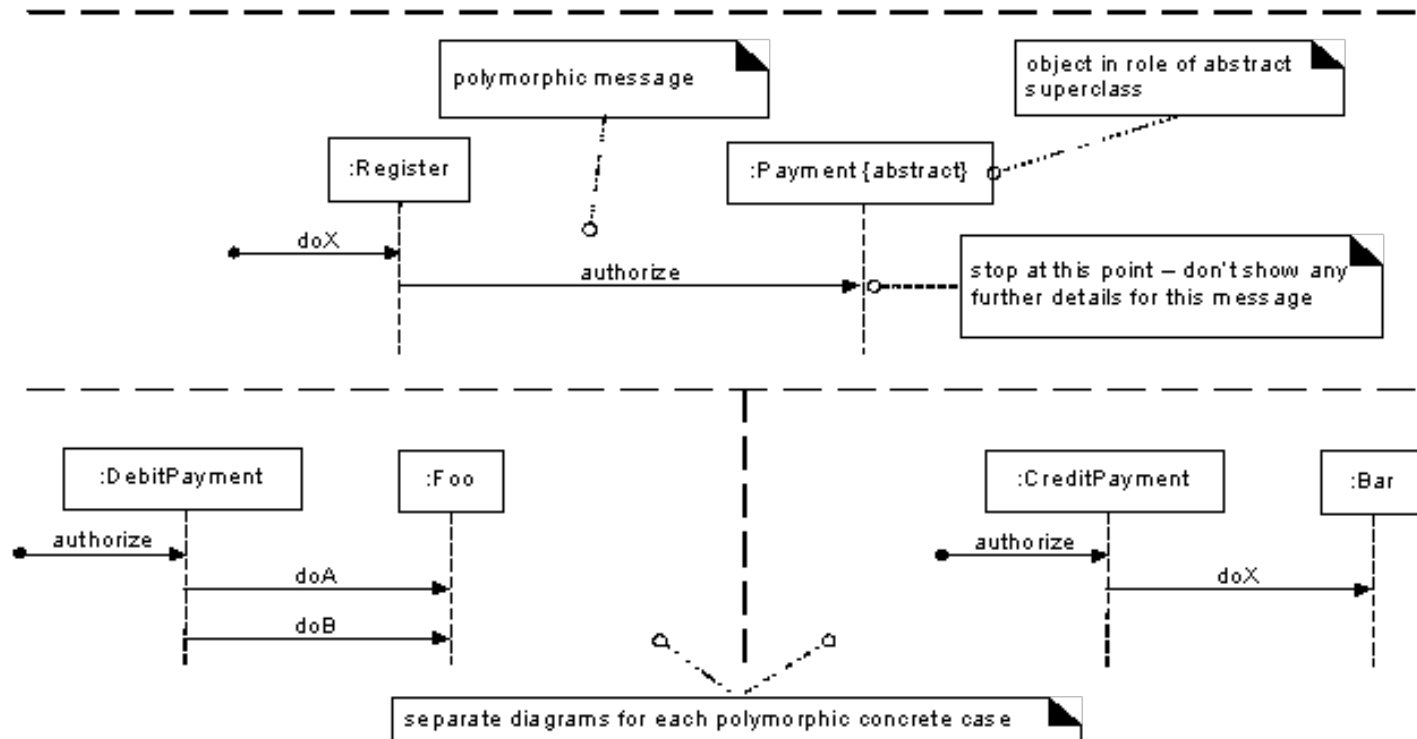
# References



interaction occurrence  
 note it covers a set of lifelines  
 note that the sd frame it relates to  
 has the same lifelines: B and C



# Polymorphic Messages





## Asynchronous (active object) calls

a stick arrow in UML implies an asynchronous call

a filled arrow is the more common synchronous call

In Java, for example, an asynchronous call may occur as follows:

```
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

to simplify the UML diagram, the *Thread* object and the *start* message may be avoided (they are standard “overhead”); instead, the essential detail of the *Clock* creation and the *run* message imply the asynchronous call

