UNIT-I

UML Interaction diagrams

MADHESWARI.K

AP/CSE

SSNCE

UML Interaction DIAGRAM

- The UML includes interaction diagrams to illustrate how objects interact via messages.

- They are used for dynamic object modeling.

- There are two common types:
    1. sequence and
    2. communication interaction diagrams.

Sequence and Communication Diagrams

- The term **interaction diagram** is a generalization of two more specialized *UML* diagram types:

1. sequence diagrams
2. communication diagrams

- Both can express similar interactions.

Sequence Diagrams

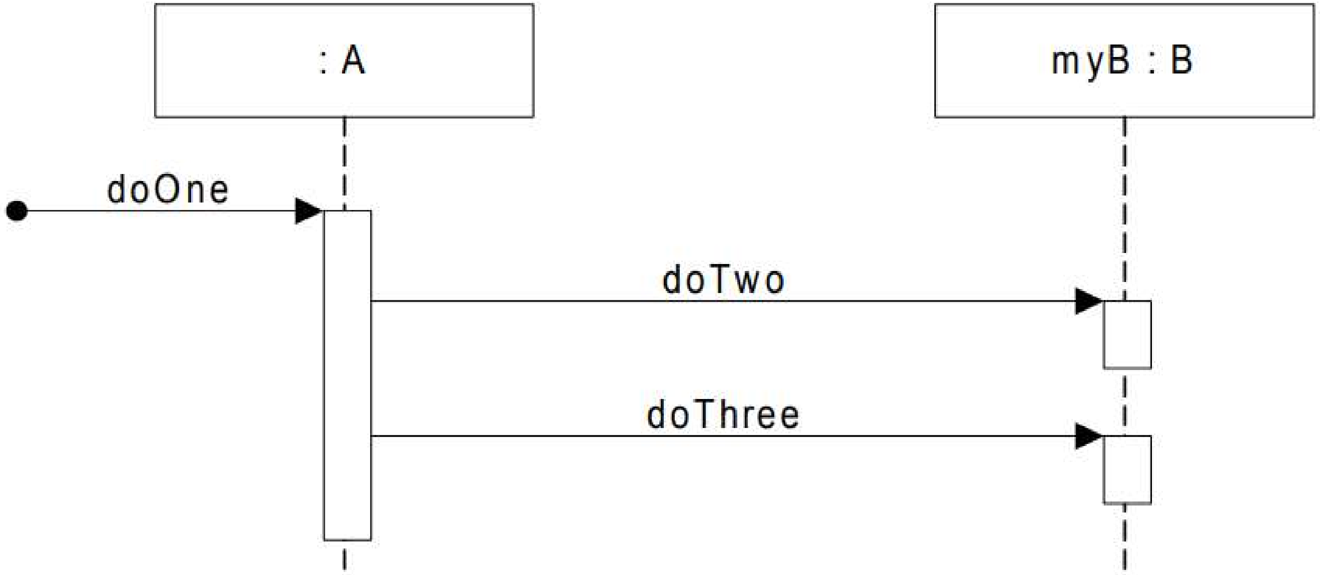
class *A* has a method named *doOne* and an attribute of type *B*. Also, that class *B* has methods named *doTwo* and *doThree*. Perhaps the partial definition of class *A* is:

```
public class A {
 private B myB = new B();
       public void doOne()
             {
                         myB.doTwo();
                         myB.doThree();
                         } // . .  }
```

Sequence Diagrams

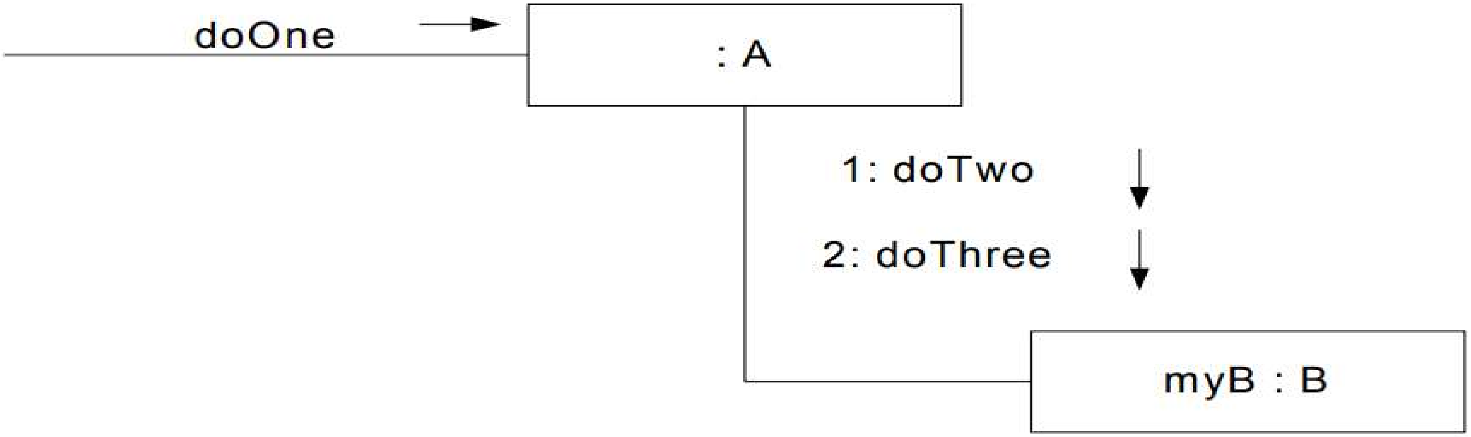
- **Sequence diagrams** illustrate interactions in a kind of fence format, in which each new object is added to the right, as shown in Figure 15.1

Fig. 15.1



Communication Diagrams

- **Communication diagrams** illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram (the essence of their wall sketching advantage), as shown in Figure 15.2.



What are the Strengths and Weaknesses of Sequence vs. Communication Diagrams?

Advantages of sequence diagram

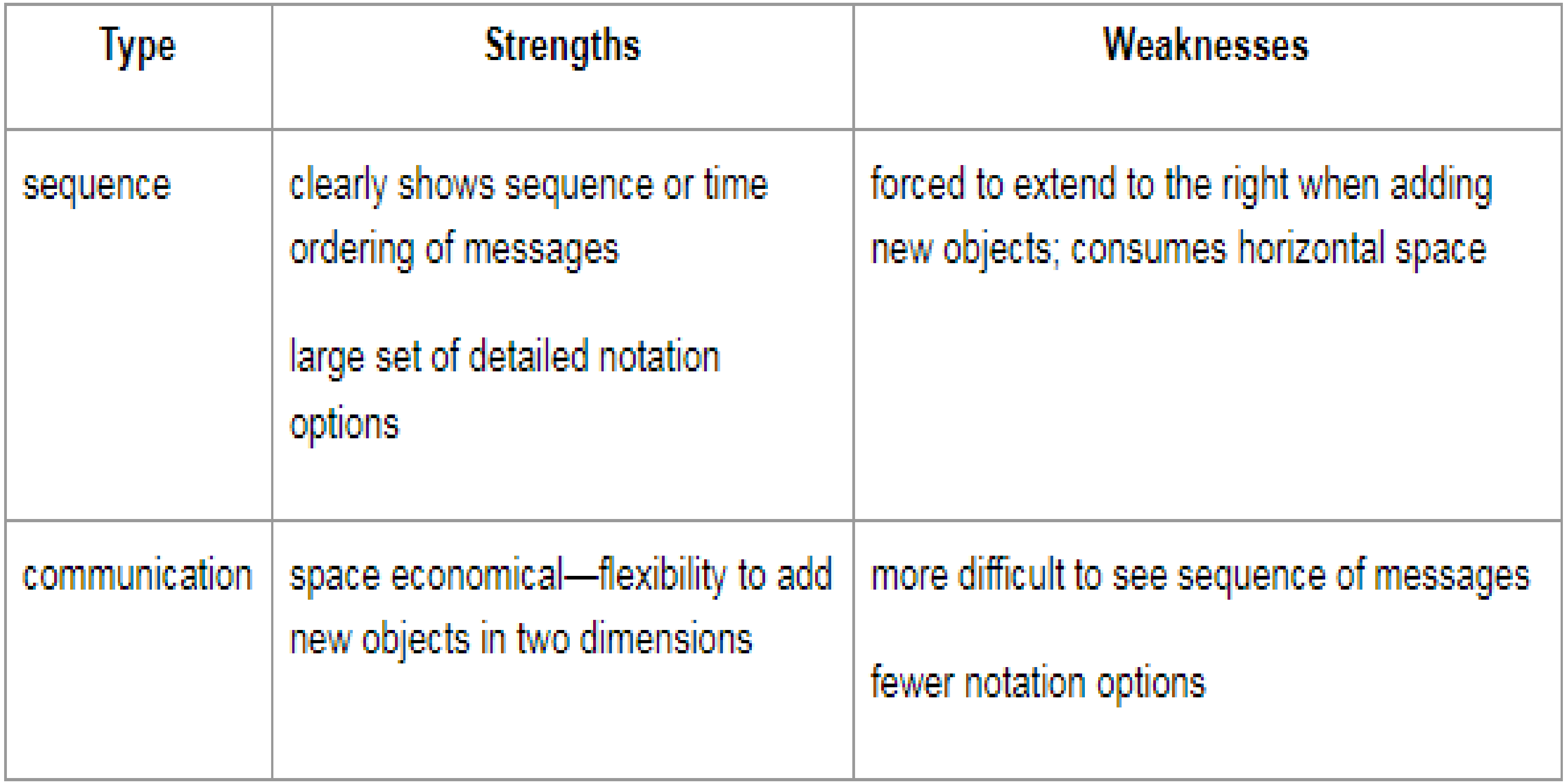
- Sequence diagrams have some advantages over communication diagrams. Perhaps first and foremost, the *UML* specification is more sequence diagram centric—more thought and effort has been put into the notation and semantics.
- Thus, tool support is better and more notation options are available. Also, it is easier to see the call-flow sequence with sequence diagrams—simply read top to bottom.
- With communication diagrams we must read the sequence numbers, such as "1:" and "2:".
- Hence, sequence diagrams are excellent for documentation or to easily read a reverse-engineered call-flow sequence, generated from source code with a *UML* tool.

What are the Strengths and Weaknesses of Sequence vs. Communication Diagrams?

Advantages of Communication diagram

- communication diagrams have advantages when applying "*UML as sketch*" to draw on walls (an Agile Modeling practice) because they are *much* more space-efficient.
- This is because the boxes can be easily placed or erased anywhere—horizontal or vertical. Consequently as well, modifying wall sketches is easier with communication diagrams—it is simple (during creative high-change OO design work) to erase a box at one location, draw a new one elsewhere, and sketch a line to it.
- In contrast, new objects in a sequence diagrams must always be added to the right edge, which is limiting as it quickly consumes and exhausts right-edge space on a page (or wall); free space in the vertical dimension is not efficiently used.
- Developers doing sequence diagrams on walls rapidly feel the drawing pain when contrasted with communication diagrams.
- when drawing diagrams that are to be published on narrow pages communication diagrams have the advantage over sequence diagrams of allowing *vertical* expansion for new objects—much more can be packed into a small visual space.

What are the Strengths and Weaknesses of Sequence vs. Communication Diagrams?



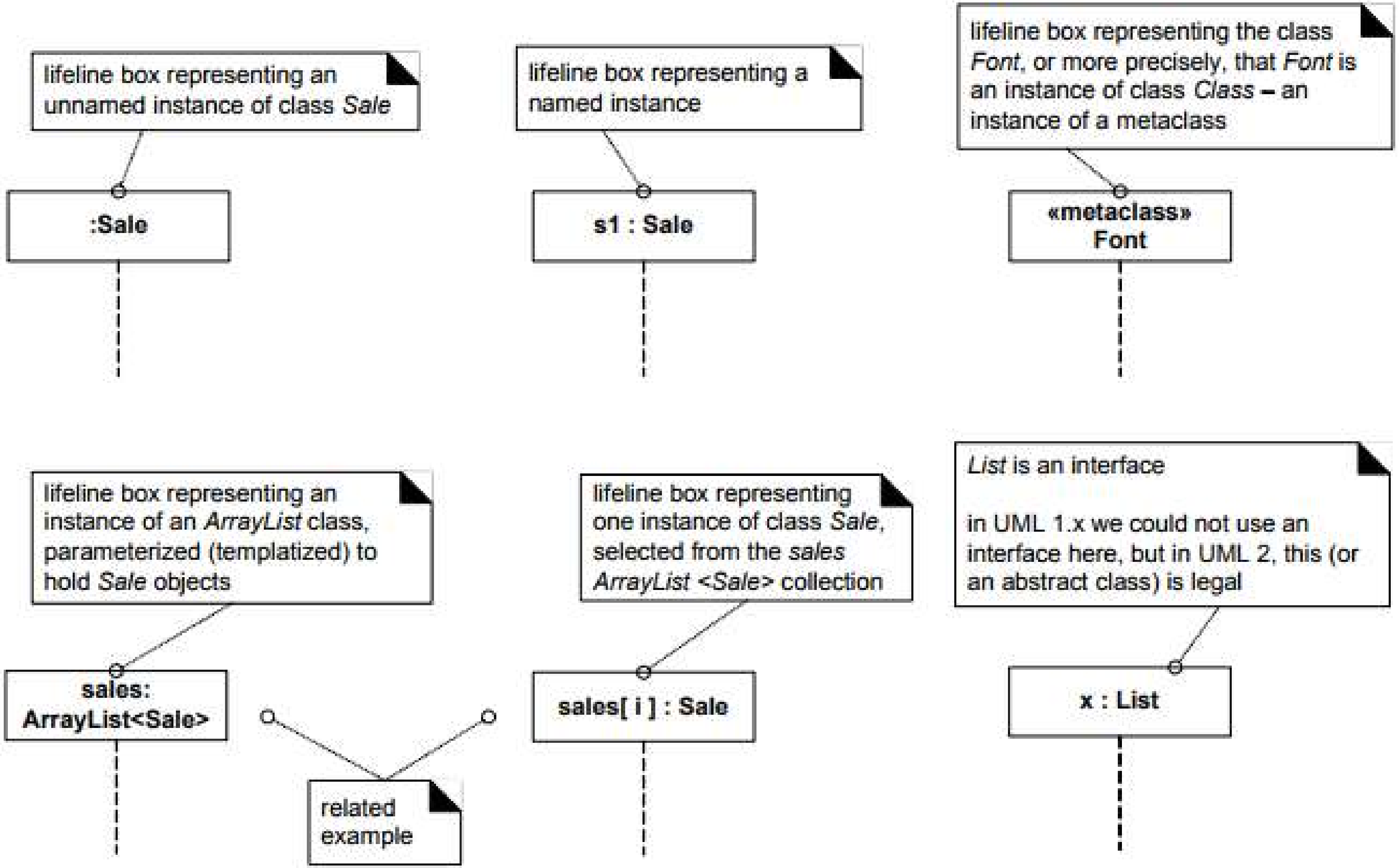
Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages large set of detailed notation options	forced to extend to the right when adding new objects; consumes horizontal space
communication	space economical—flexibility to add new objects in two dimensions	more difficult to see sequence of messages fewer notation options

Interaction Diagrams

- Essential UML models for OOAD

  1. Use cases
     - Functional requirements
  2. Class diagram
     - Objects with knowledge (attributes) and behavior (operations)
     - Static relationships between objects
  3. Interaction diagrams
     - Dynamic collaboration between objects

Common *UML* Interaction Diagram Notation



lifeline box representing an unnamed instance of class *Sale*

:Sale

lifeline box representing a named instance

s1 : Sale

lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass

«metaclass»
Font

lifeline box representing an instance of an *ArrayList* class, parameterized (templatized) to hold *Sale* objects

sales:
ArrayList<Sale>

related example

lifeline box representing one instance of class *Sale*, selected from the *sales* *ArrayList* <*Sale*> collection

sales[i] : Sale

List is an interface

in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal

x : List

Common *UML* Interaction Diagram Notation

- **Basic Message Expression Syntax**

  Interaction diagrams show messages between objects; the *UML* has a standard syntax for these message expressions:

return = message(parameter : parameterType) : returnType

Example

initialize(code)

initialize d = getProductDescription(id)

d = getProductDescription(id:ItemID)

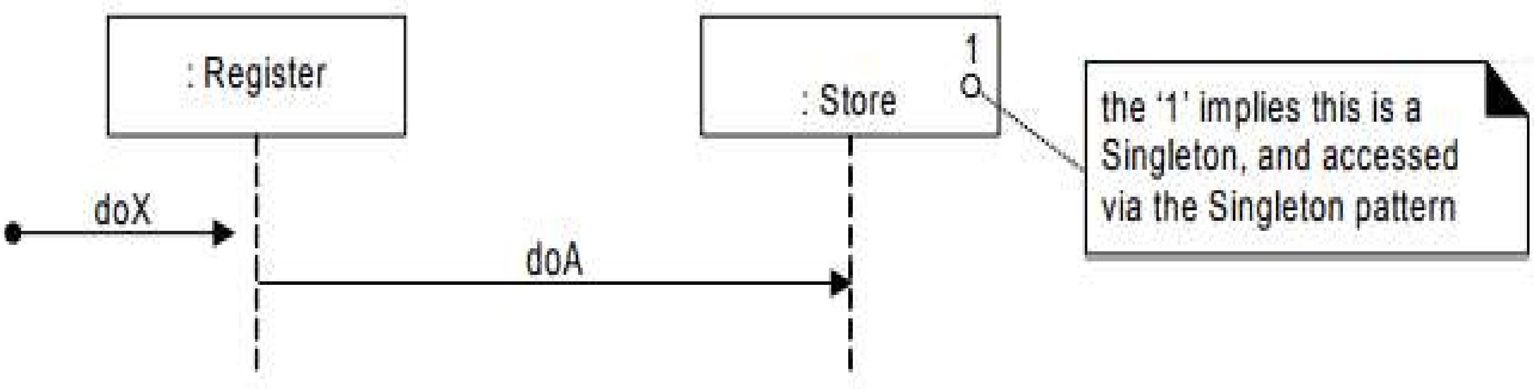
d = getProductDescription(id:ItemID) : ProductDescription

Singleton Objects

- only *one* instance of a class instantiated—never two
- such an object is marked with a '1' in the upper right corner of the lifeline box.

Common *UML* Interaction Diagram Notation

- **Singleton Objects**



the '1' implies this is a Singleton, and accessed via the Singleton pattern

Notation for Singleton

Basic Sequence Diagram Notation

- **Lifeline Boxes and Lifelines**
- **Messages**
- **Focus of Control and Execution Specification Bars**
- **Illustrating Reply or Returns**
- **Messages to "self" or "this"**
- **Creation of Instances**
- **Object Lifelines and Object Destruction**
- **Diagram Frames in *UML* Sequence Diagrams**
- **Looping**
- **Conditional Messages**
- **Conditional Messages in *UML***
- **Mutually Exclusive Conditional Messages**
- **Iteration Over a Collection**
- **Nesting of Frames**
- **Messages to Classes to Invoke Static (or Class) Methods**
- **Polymorphic Messages and Cases**
- **Asynchronous and Synchronous Calls**

Basic Sequence Diagram Notation

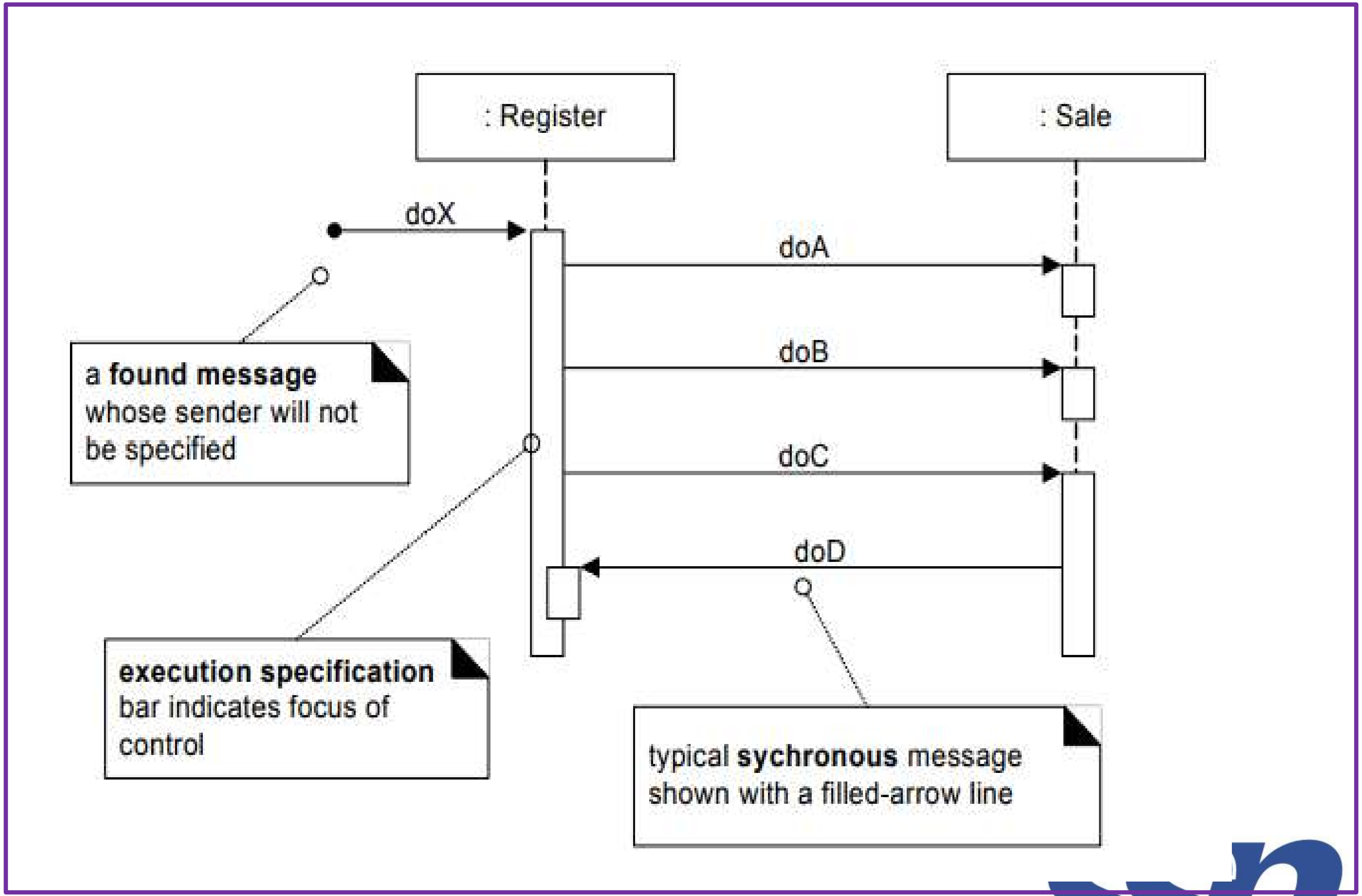
Lifeline Boxes and Lifelines

- In contrast to communication diagrams, in sequence diagrams the lifeline boxes include a vertical line extending below them—these are the actual lifelines.
- *UML* examples show the lifeline as dashed or solid line

Messages

- Each (typical synchronous) message between objects is represented with a message expression on a *filled-arrowed*
- solid line between the vertical lifelines . The time ordering is organized from top to bottom of lifelines.
- starting message is called a **found message** in the *UML*, shown with an opening solid ball; it implies the sender will not be specified, is not known, or that the message is coming from a random source.

Basic Sequence Diagram Notation



a **found message** whose sender will not be specified

execution specification bar indicates focus of control

typical **sychronous** message shown with a filled-arrow line

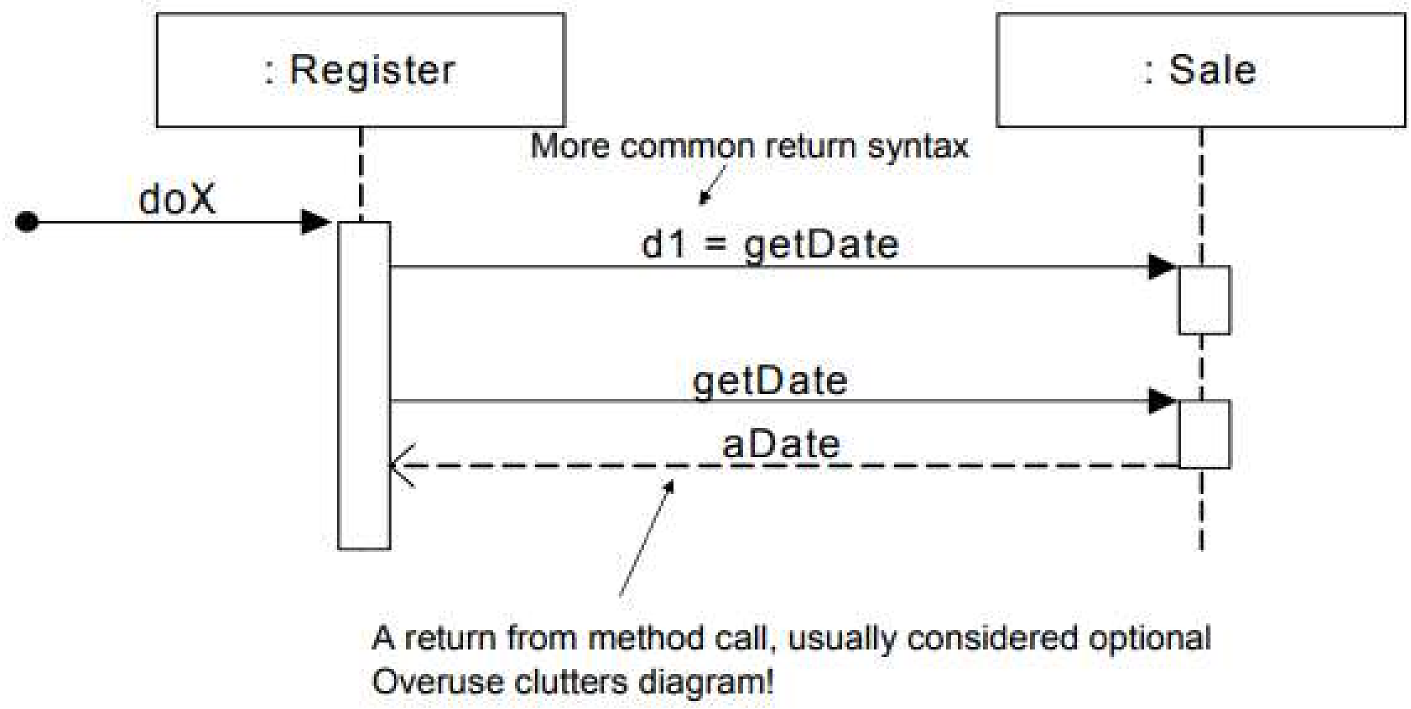
Basic Sequence Diagram Notation

Illustrating Reply or Returns

- There are two ways to show the return result from a message:

  1. Using the message syntax *returnVar = message(parameter)*.

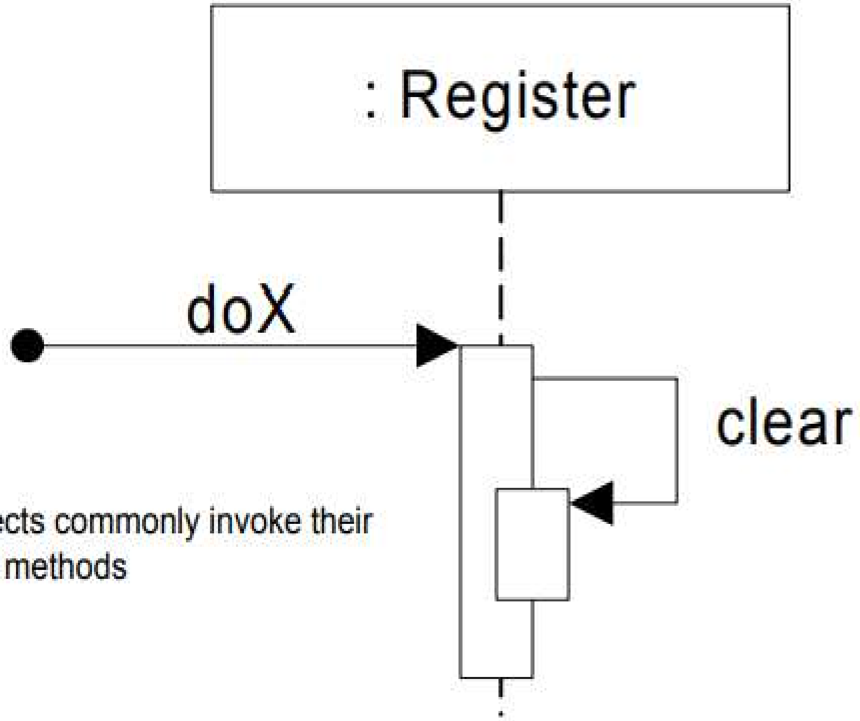
  2. Using a reply (or return) message line at the end of an activation bar.


: Register
: Sale
More common return syntax
doX
d1 = getDate
getDate
aDate
A return from method call, usually considered optional
Overuse clutters diagram!

Basic Sequence Diagram Notation

Messages to "self" or "this"

- You can show a message being sent from an object to itself by using a nested activation bar

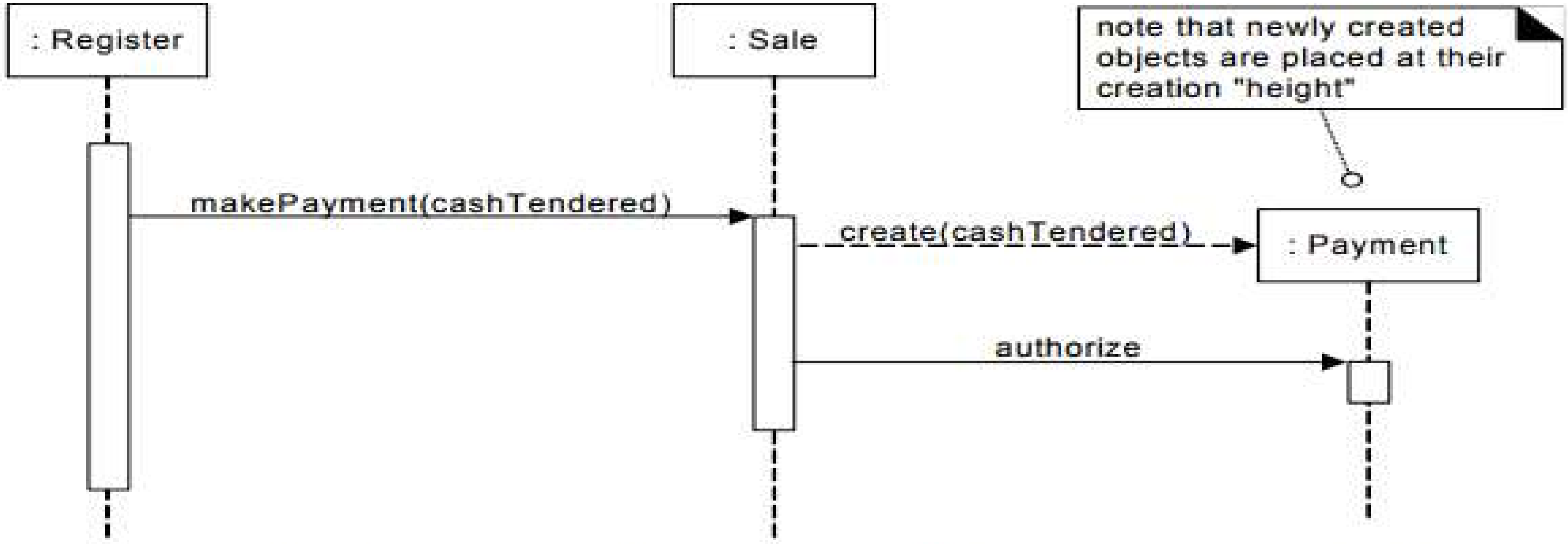


Objects commonly invoke their own methods

Basic Sequence Diagram Notation

Creation of Instances

- The arrow is filled if it's a regular synchronous message (such as implying invoking a Java constructor), or open (stick arrow) if an asynchronous call.
- The message name *create* is not required—anything is legal—but it's a *UML* idiom.
- The typical interpretation (in languages such as Java or C#) of a *create* message on a dashed line with a filled arrow is "invoke the *new* operator and call the constructor".



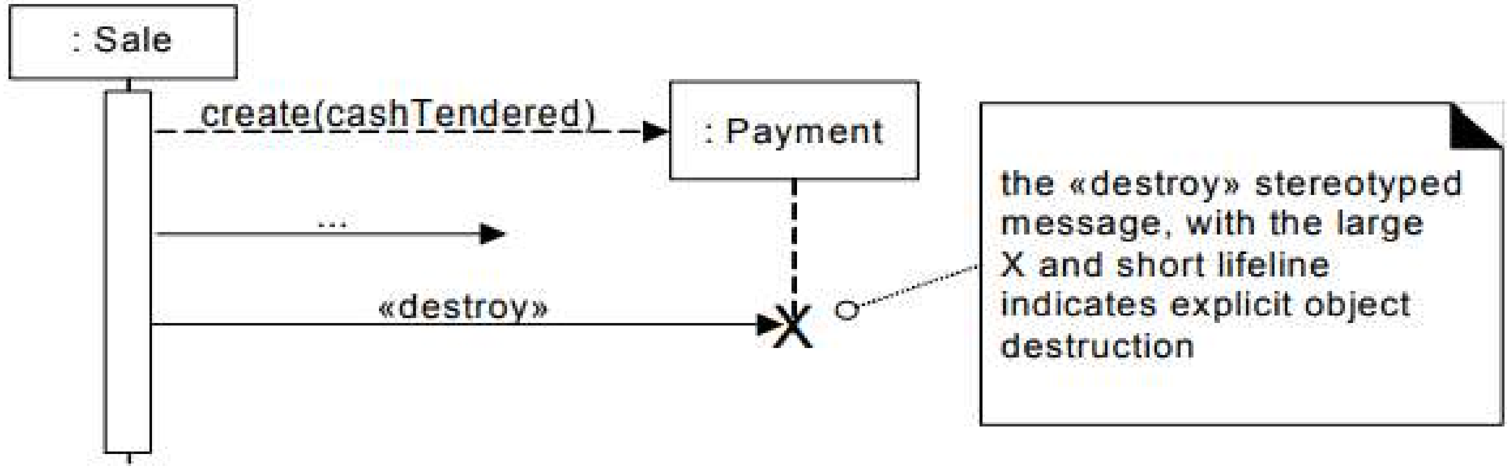
: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

Dashed line for 'create' really not needed, though its now official UML
Use 'create' for calls to a constructor

Basic Sequence Diagram Notation

Object Lifelines and Object Destruction

- In some circumstances it is desirable to show explicit destruction of an object.
- For example, when using C++ which does not have automatic garbage collection, or when you want to especially indicate an object is no longer usable (such as a closed database connection).
- The *UML* lifeline notation provides a way to express this destruction



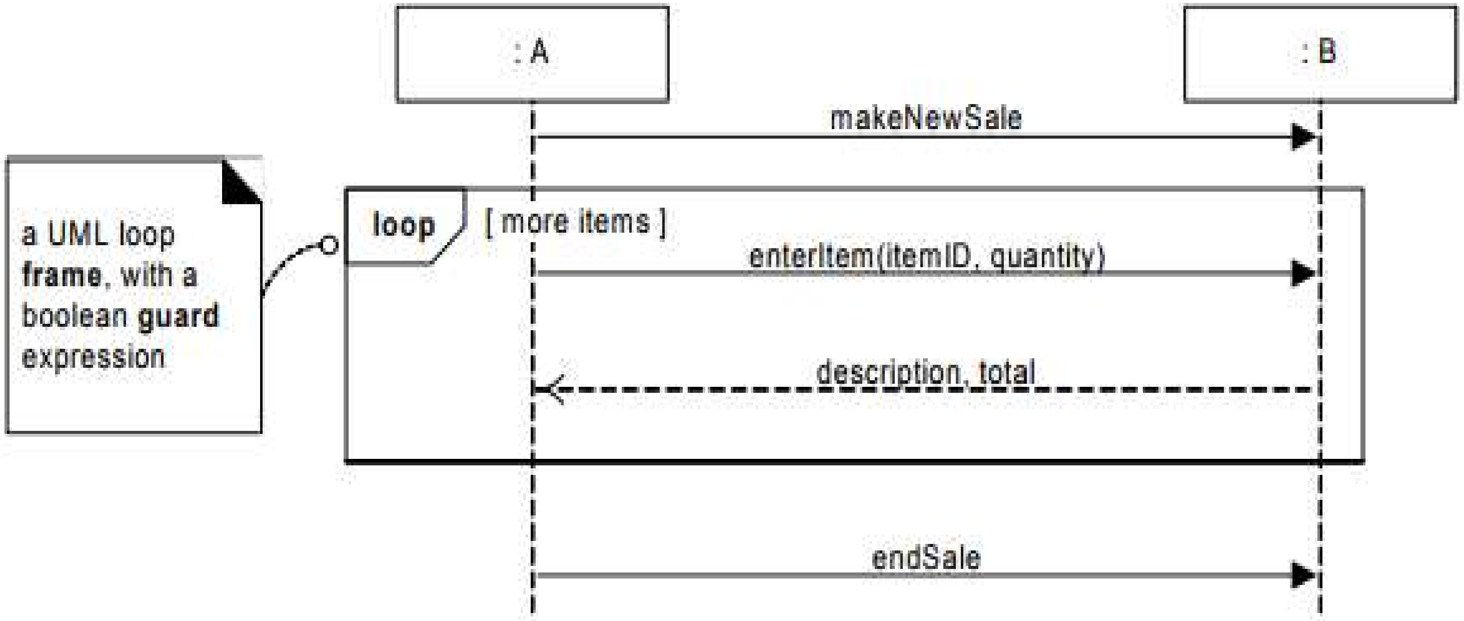
the «destroy» stereotyped message, with the large X and short lifeline indicates explicit object destruction

Usually not necessary for languages with automatic garbage collection (e.g., Java)

Basic Sequence Diagram Notation

Diagram Frames in *UML* Sequence Diagrams

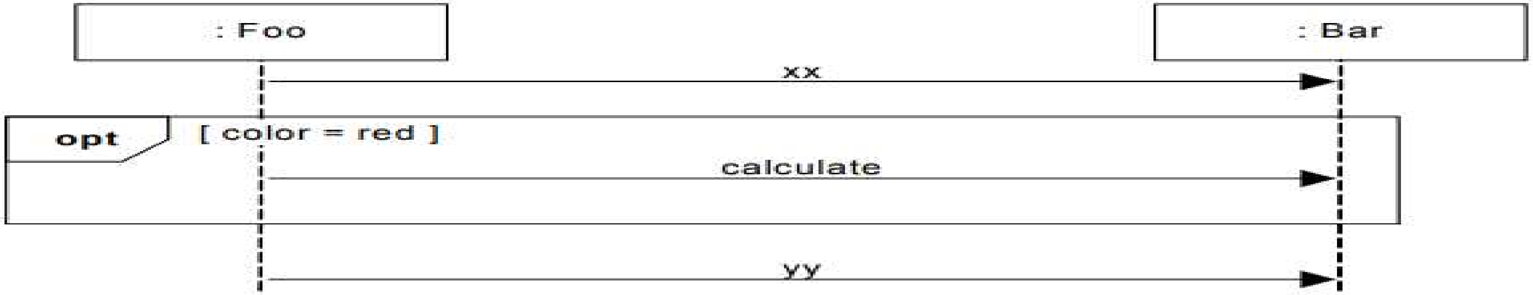
- To support conditional and looping constructs (among many other things), the *UML* uses **frames**.
- Frames are regions or fragments of the diagrams; they have an operator or label (such as *loop*) and a guard



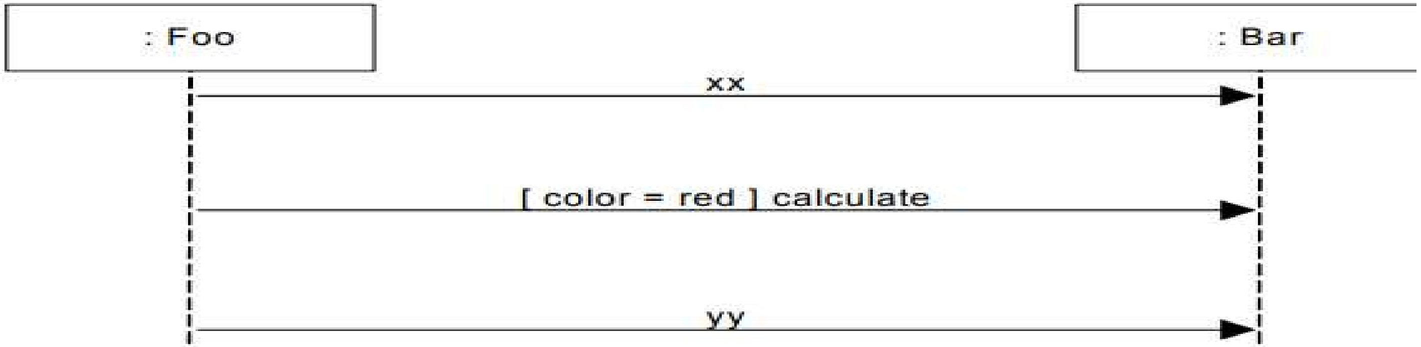
Basic Sequence Diagram Notation

Conditional Messages

An OPT frame is placed around one or more messages



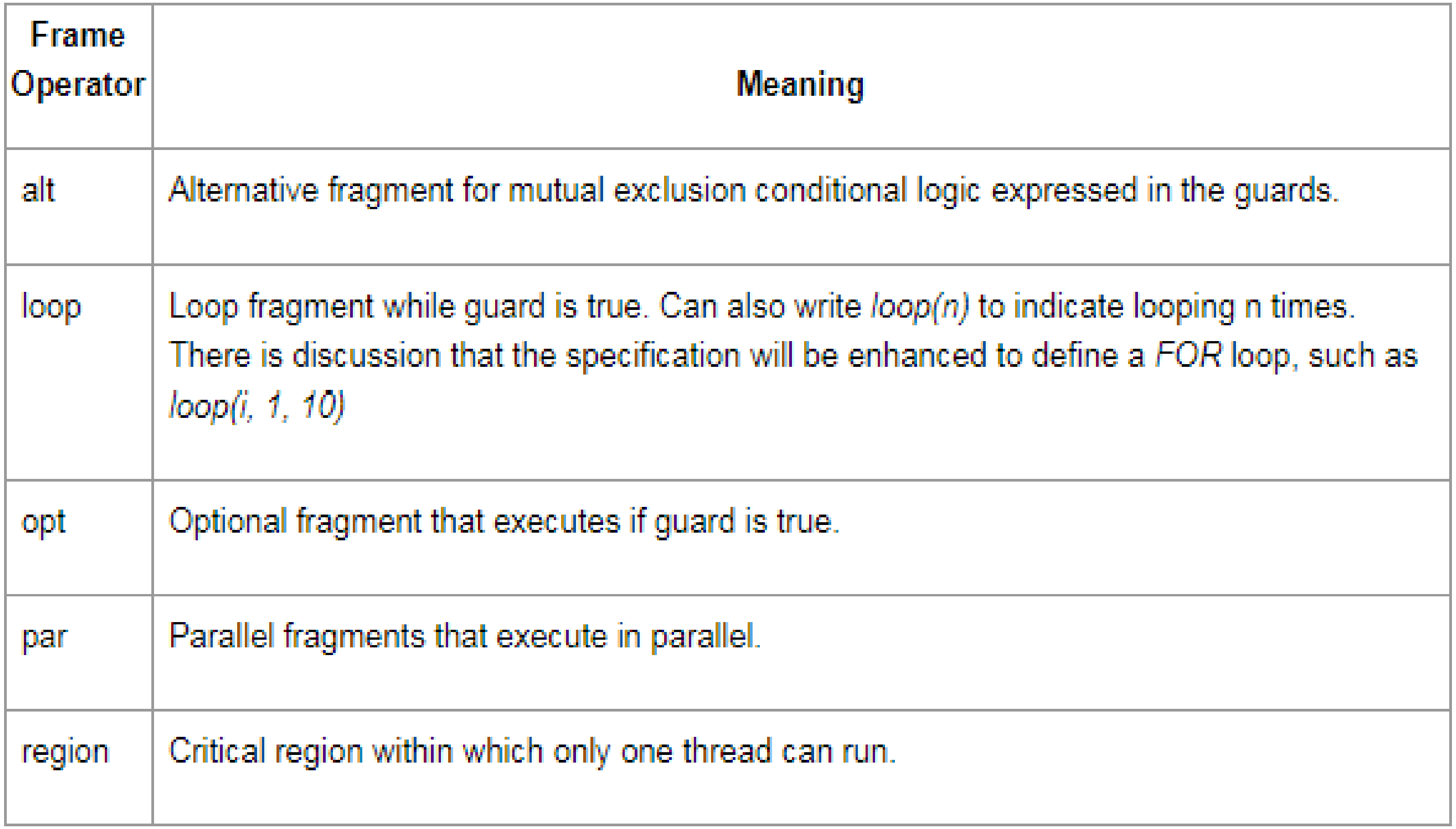
UML 2 frame showing an optional message



The same ID using pre-UML 2 notation
Guards must evaluate to true for the message to be sent

Basic Sequence Diagram Notation

common frame operators:

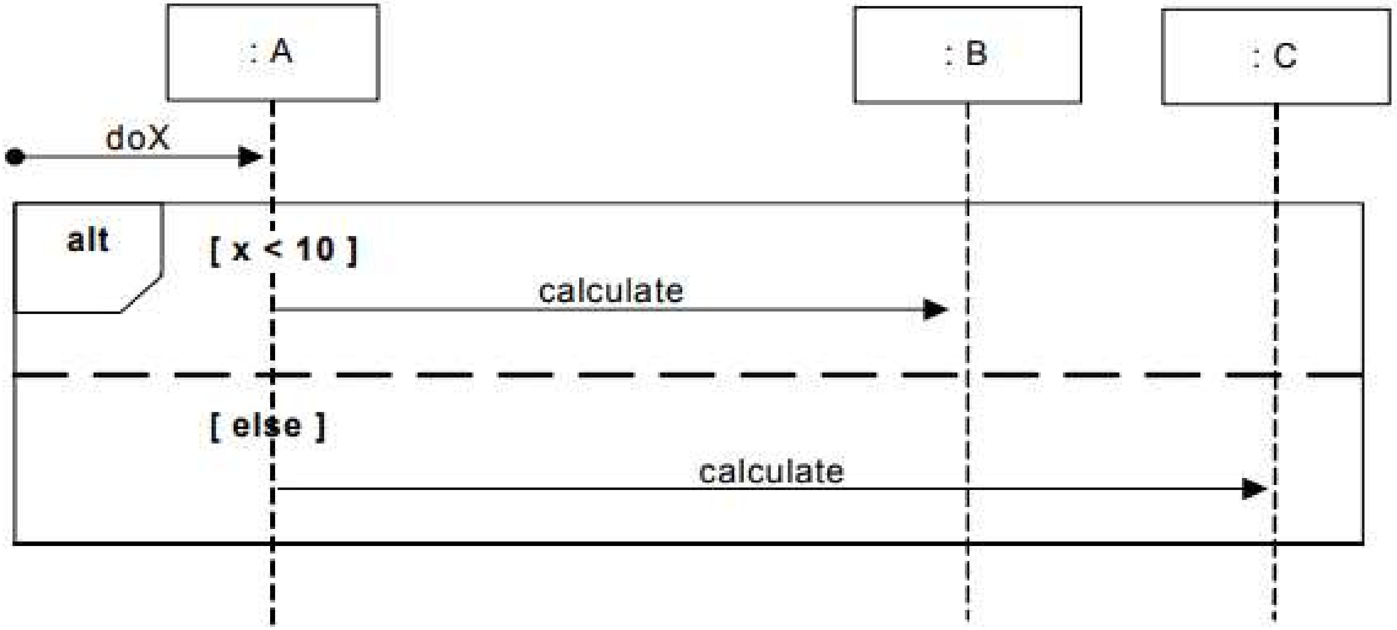


Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i, 1, 10)*
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

Basic Sequence Diagram Notation

- **Mutually Exclusive Conditional Messages**

An ALT frame is placed around the mutually exclusive alternatives.

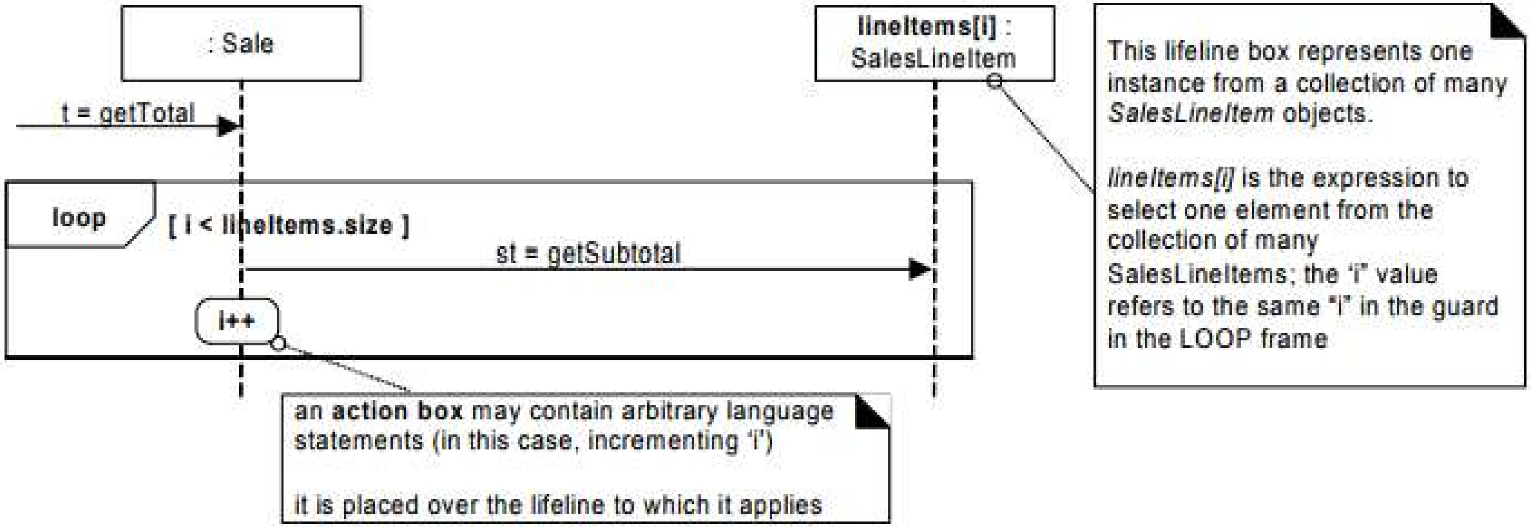


Alt frame show mutually exclusive interactions

Basic Sequence Diagram Notation

- **Iteration Over a Collection**

  A common algorithm is to iterate over all members of a collection (such as a list or map), sending the same message to each.



This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the 'i" value refers to the same "i" in the guard in the LOOP frame

an **action box** may contain arbitrary language statements (in this case, incrementing 'i')

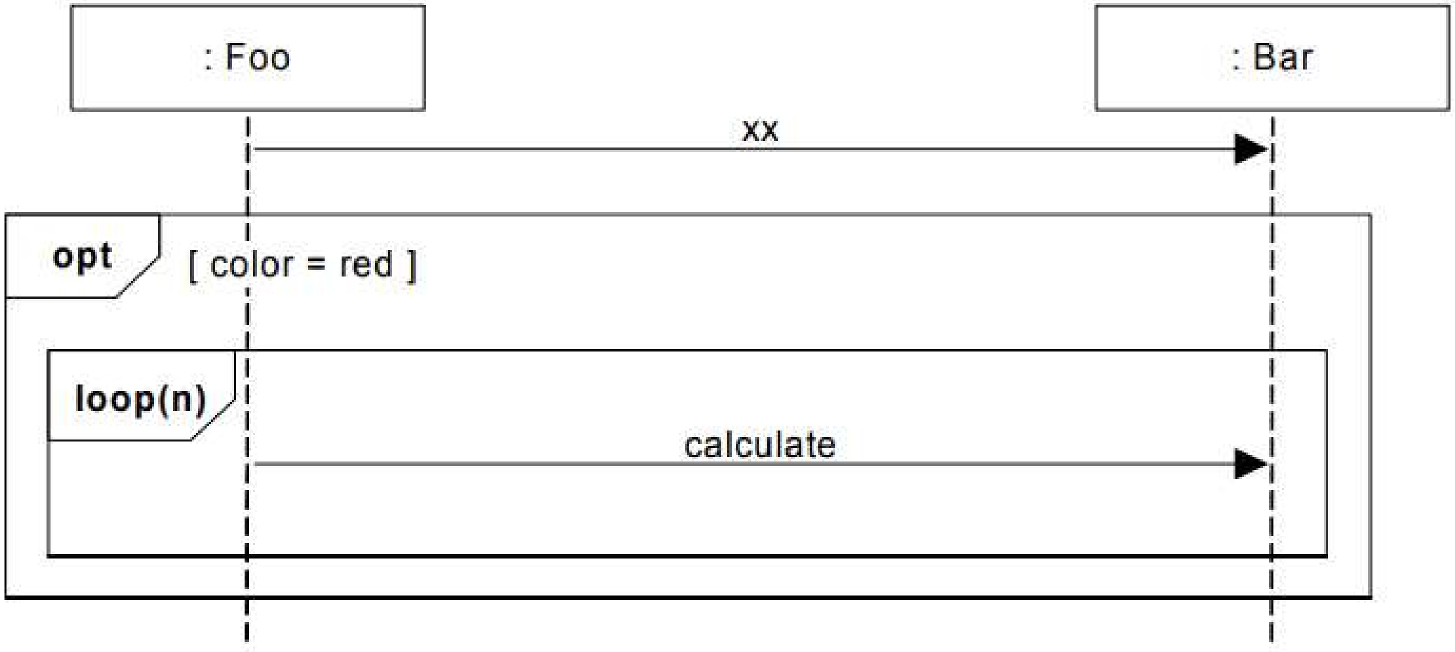
it is placed over the lifeline to which it applies

Technique for looping over a collection
Loop details are explicit; diagram more cluttered
Note Java code on p. 234 showing new for loop syntax

Basic Sequence Diagram Notation

Nesting of Frames

Frames can be nested.

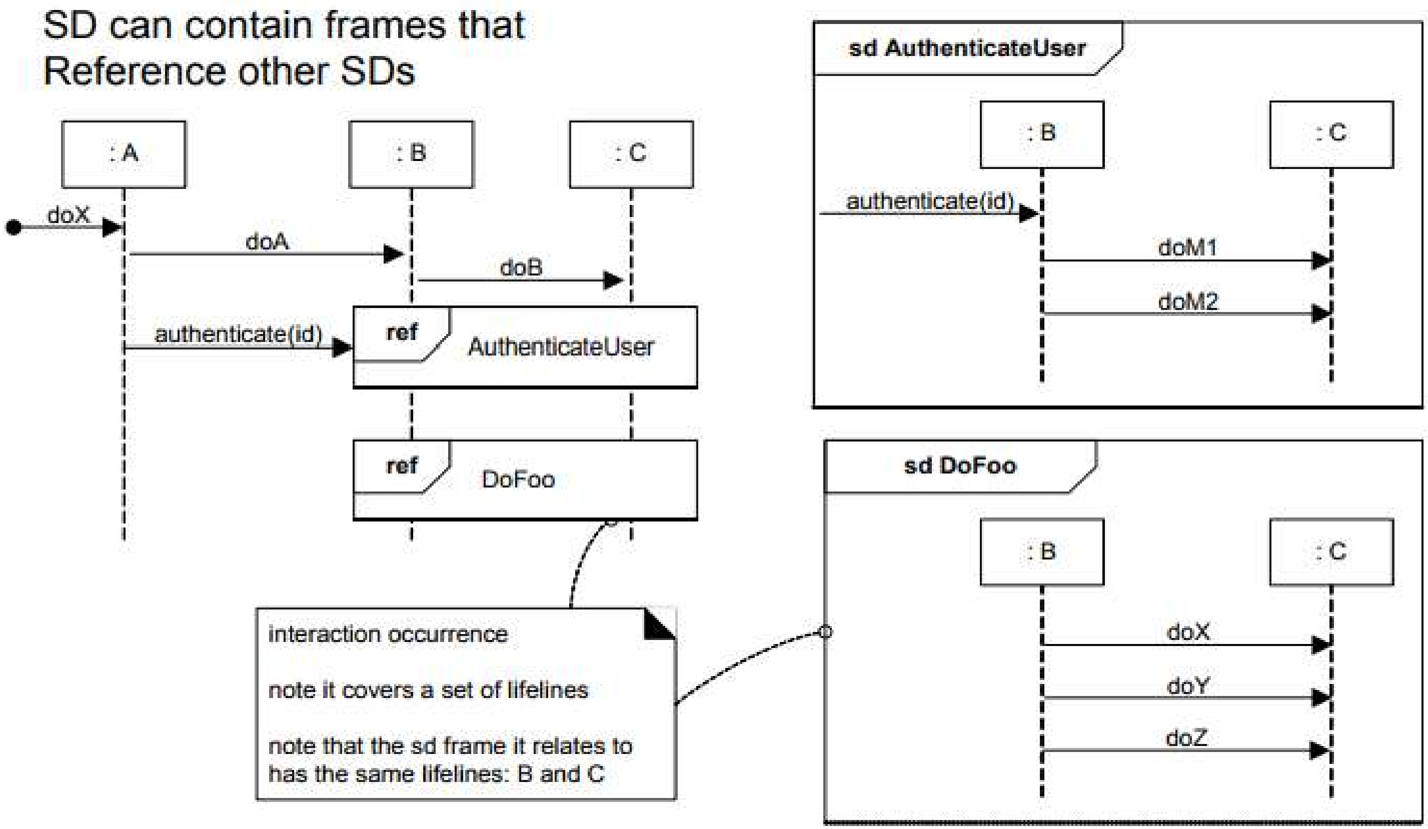


A loop frame nested within an optional frame

Basic Sequence Diagram Notation

Nesting of Frames

Frames can be nested.



SD can contain frames that Reference other SDs

interaction occurrence

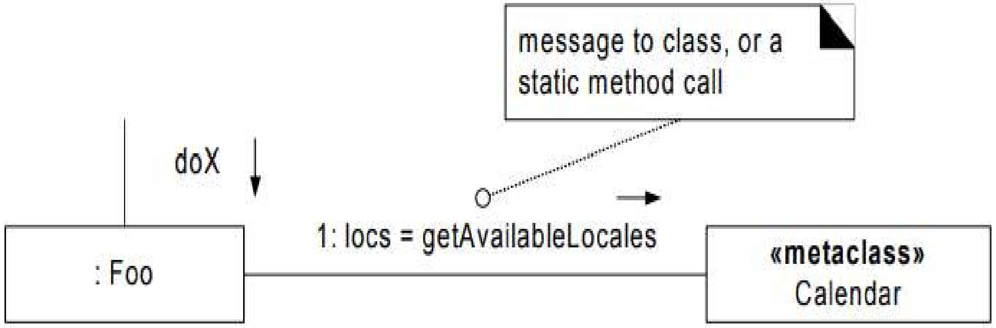
note it covers a set of lifelines

note that the sd frame it relates to has the same lifelines: B and C

Basic Sequence Diagram Notation

Messages to Classes to Invoke Static (or Class) Methods

- You can show class or static method calls by using a lifeline box label that indicates the receiving object is a class, or more precisely, an *instance* of a **metaclass**


message to class, or a static method call
doX
1: locs = getAvailableLocales
: Foo
«metaclass»
Calendar

Call to a static class method
Notice there is no implied instance to the Calendar class (':' is omitted)

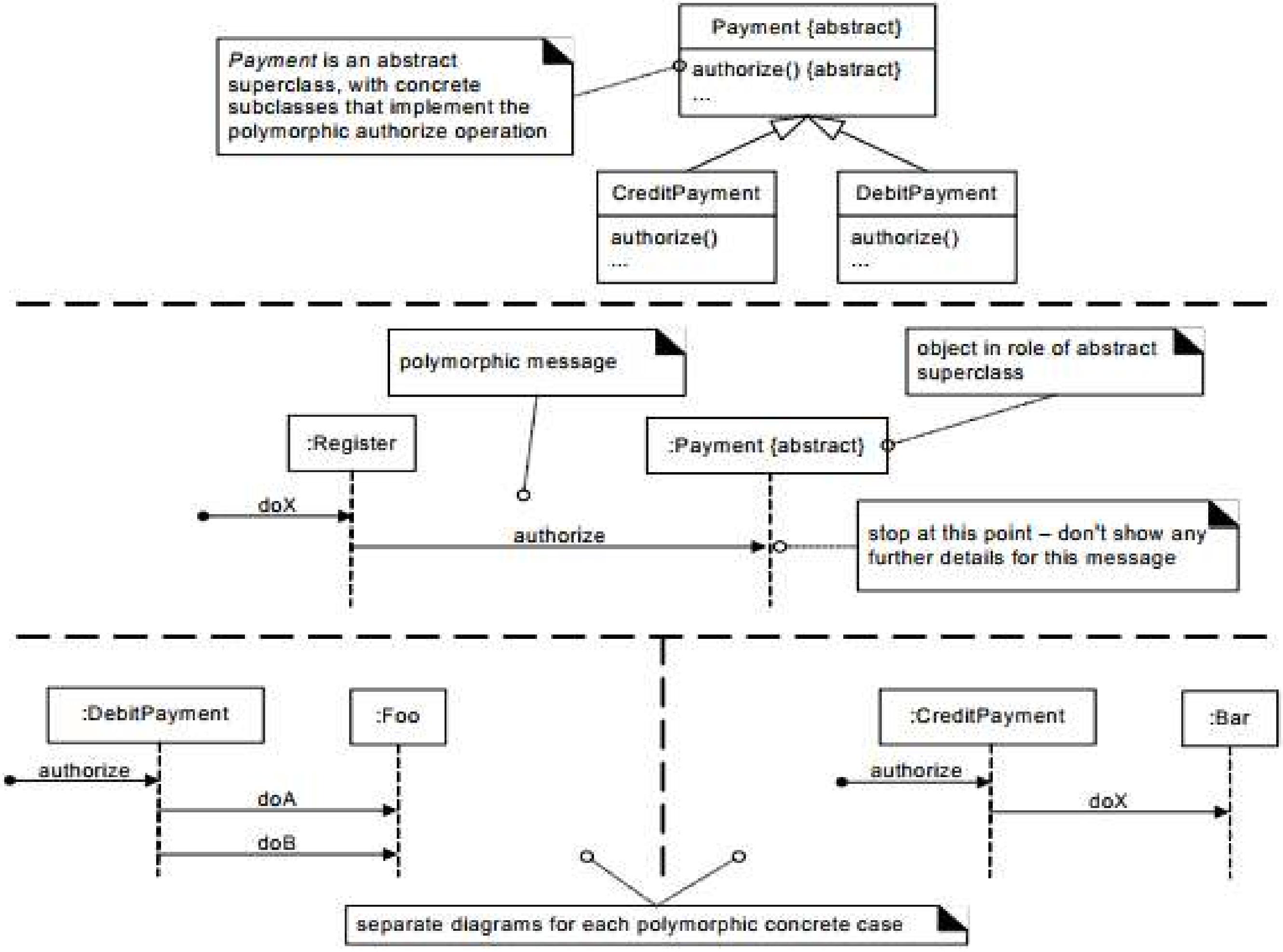
```
public class Foo
{
public void doX()
{
 // static method call on class
Calendar Locale[] locales =
Calendar.getAvailableLocales(); //
. .  }

 // . .  }
```

Basic Sequence Diagram Notation

- **Polymorphic Messages and Cases**

Polymorphism is fundamental to *OO* design. How to show it in a sequence diagram? That's a common *UML* question. One approach is to use multiple sequence diagrams—one that shows the polymorphic message to the abstract superclass or interface object, and then separate sequence diagrams detailing each polymorphic case, each starting with a *found* polymorphic message. Figure 15.21 illustrates.

Basic Sequence Diagram Notation


Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation
Payment {abstract}
authorize() {abstract}
...
CreditPayment
authorize()
...
DebitPayment
authorize()
...
polymorphic message
object in role of abstract superclass
:Register
:Payment {abstract}
doX
authorize
stop at this point – don't show any further details for this message
:DebitPayment
:Foo
authorize
doA
doB
:CreditPayment
:Bar
authorize
doX
separate diagrams for each polymorphic concrete case

Polymorphic method calls

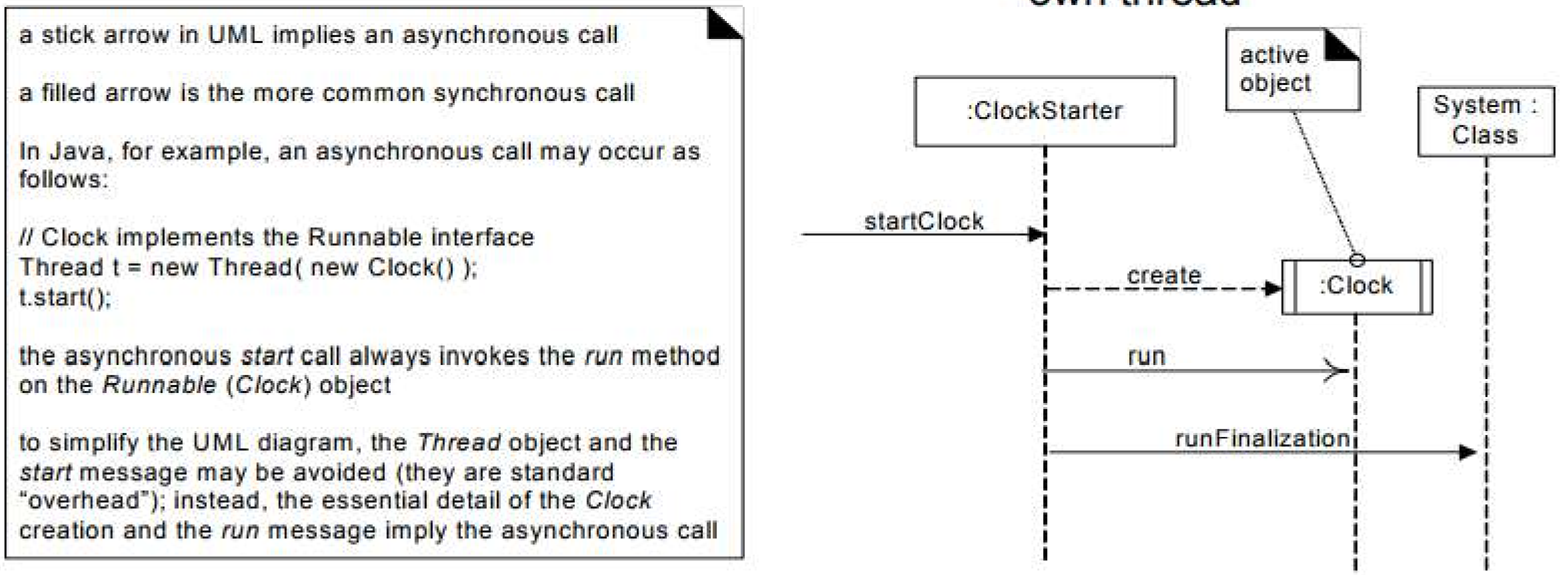
Basic Sequence Diagram Notation

- **Asynchronous and Synchronous Calls**
- An **asynchronous message** call does not wait for a response; it doesn't *block*. They are used in multi-threaded environments such as .NET and Java so that new **threads** of execution can be created and initiated. In Java, for example, you may think of the *Thread.start* or *Runnable.run* (called by *Thread.start*) message as the asynchronous starting point to initiate execution on a new thread.
- The *UML* notation for asynchronous calls is a stick arrow message; regular synchronous (blocking) calls are shown with a filled arrow (see Figure 15.22).

Basic Sequence Diagram Notation

- **Asynchronous and Synchronous Calls**

Active objects run in their own thread



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

Solid vs. stick arrowheads easily confused when sketching models
See Java code p. 239-240