

UML Dynamic Diagrams

Interaction Diagrams



Dynamic Diagram Types

- ❑ **Interaction Diagrams** - Set of objects or roles and the messages that can be passed among them.
 - ❑ **Sequence Diagrams** - emphasize time ordering
 - ❑ **Communication Diagrams** - emphasize structural ordering
- ❑ **State Diagrams**
 - ❑ State machine consisting of states, transitions, events and activities of an object
- ❑ **Activity & Swimlane Diagrams**
 - ❑ Emphasize and show flow of control among objects

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

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

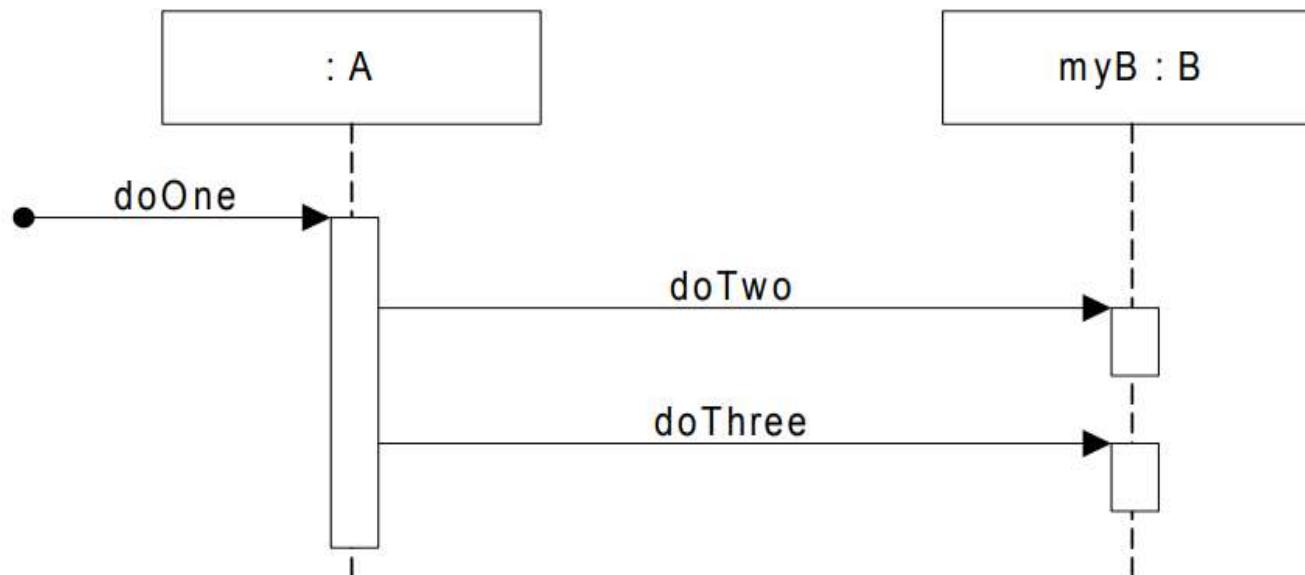
class A has a method named *doOne* and an attribute of type *B*. Also, that class *B* has methods named *doTwo* and *doThree*. Perhaps the partial definition of class A is:



Sequence Diagrams

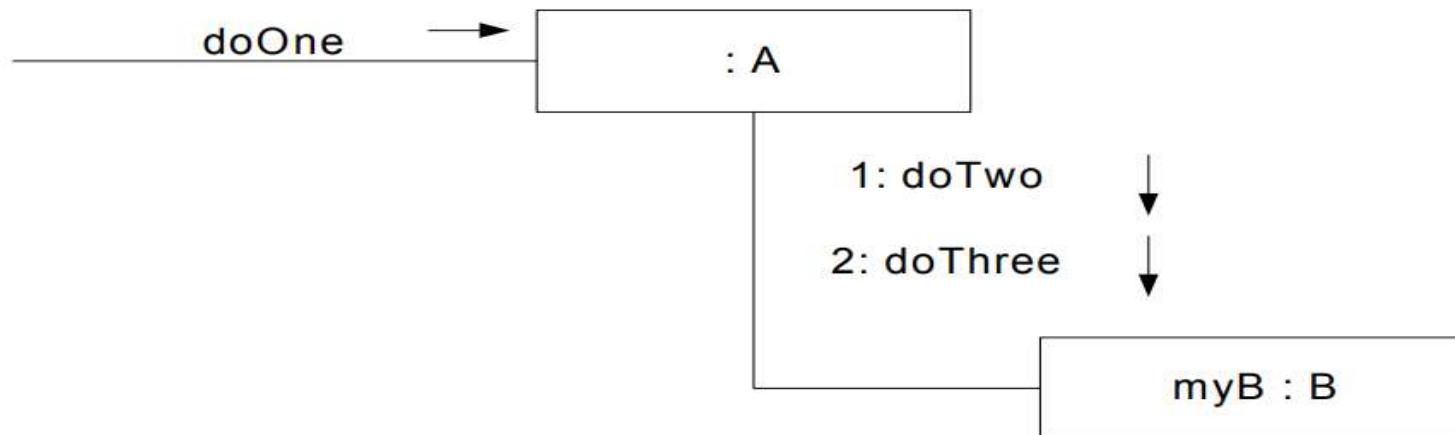
- **Sequence diagrams** illustrate interactions in a kind of fence format, in which each new object is added to the right, as shown in Figure 15.1

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.



ssn

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

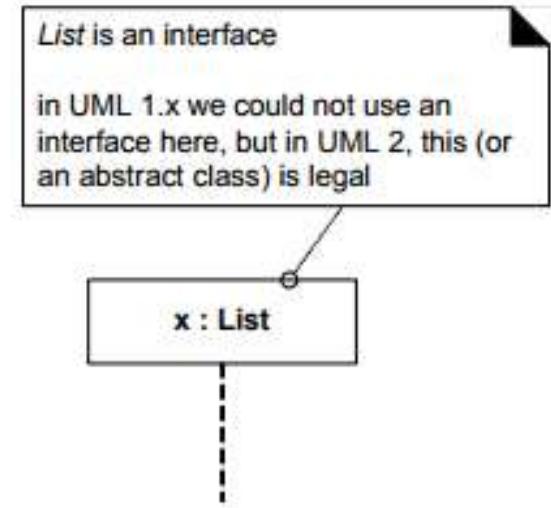
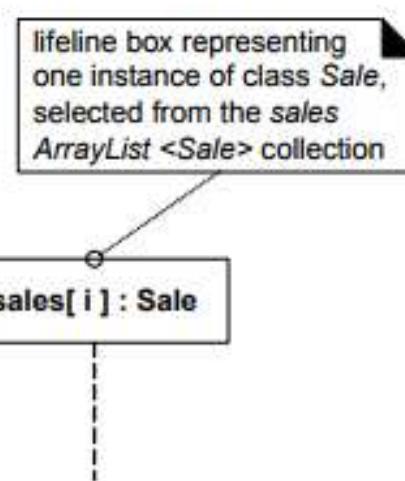
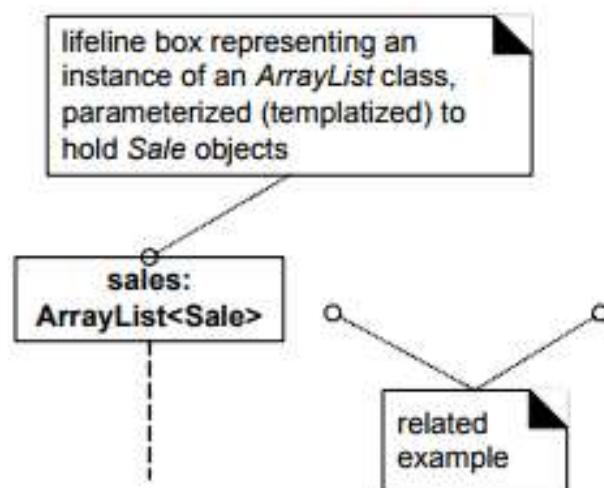
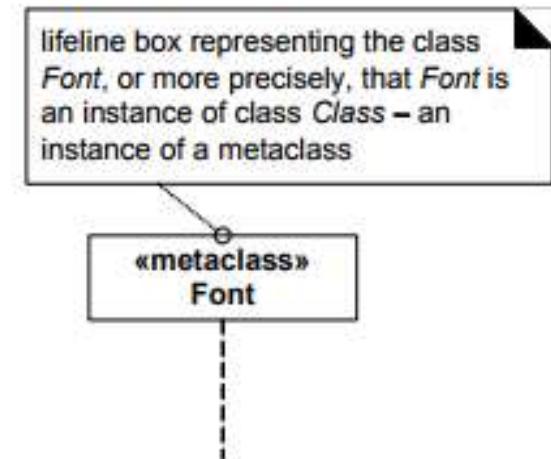
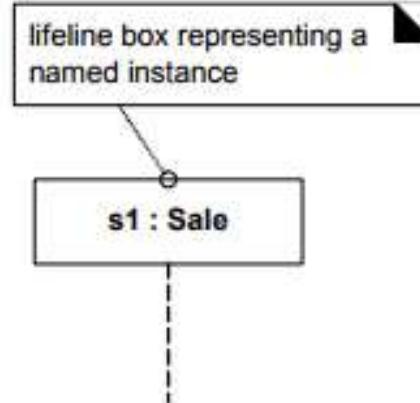
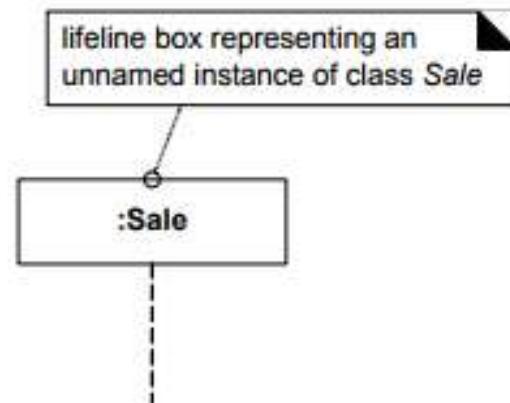


Interaction Diagrams

- 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



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
```

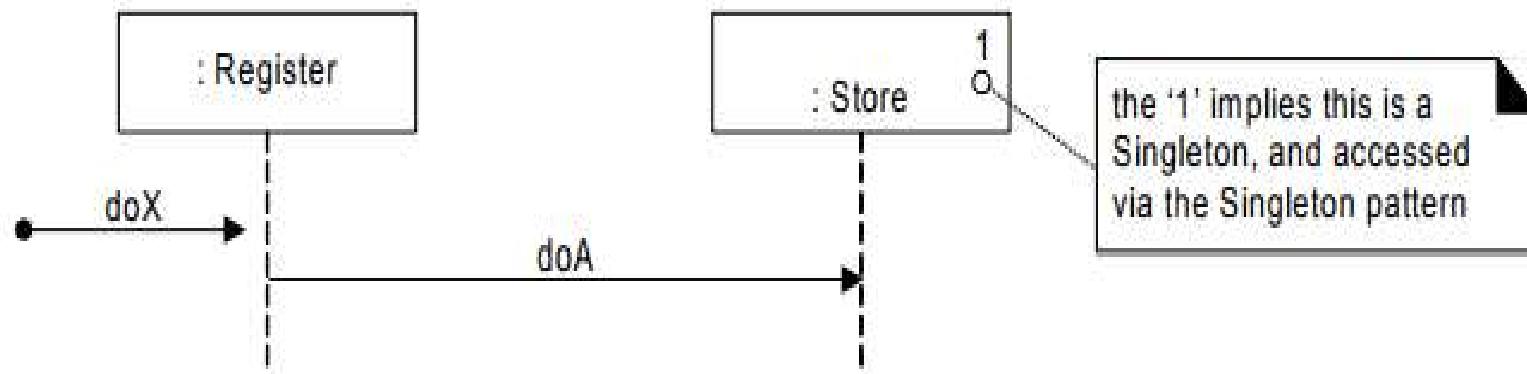
Singleton Objects

- 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**



Notation for Singleton

ssh

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

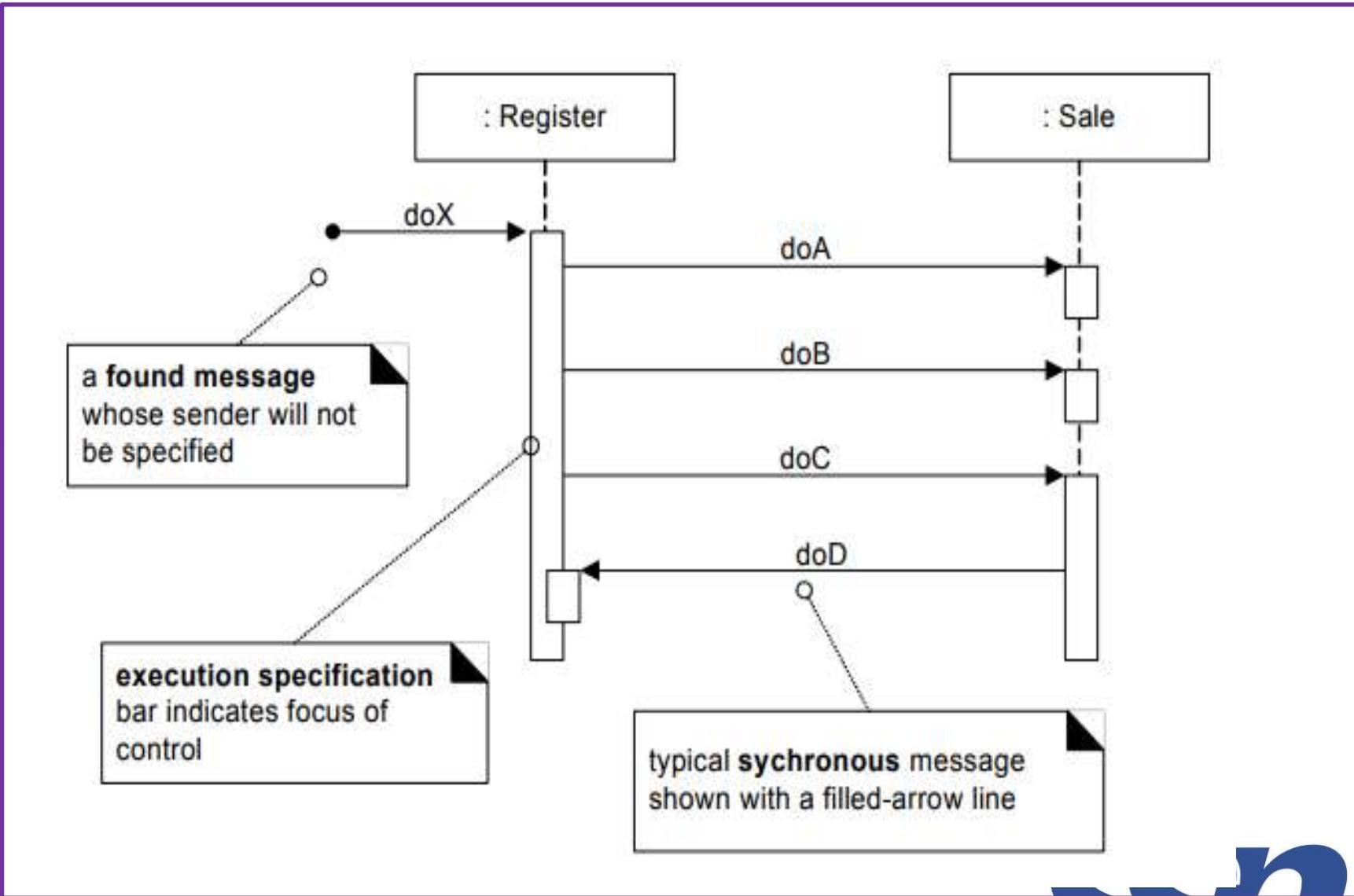
- In contrast to communication diagrams, in sequence diagrams the lifeline boxes include a **vertical line** extending below them—these are the actual lifelines.
- *UML* examples show the lifeline as dashed or solid line

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.



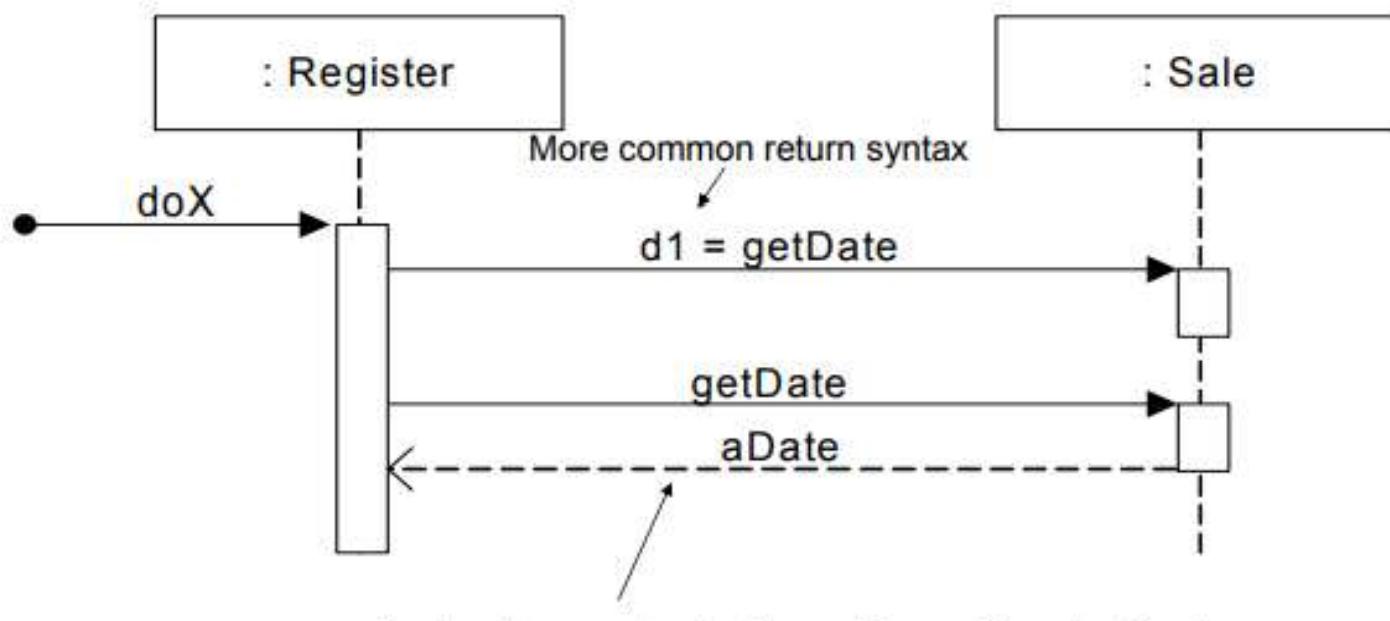
Basic Sequence Diagram Notation



Basic Sequence Diagram Notation

Illustrating Reply or Returns

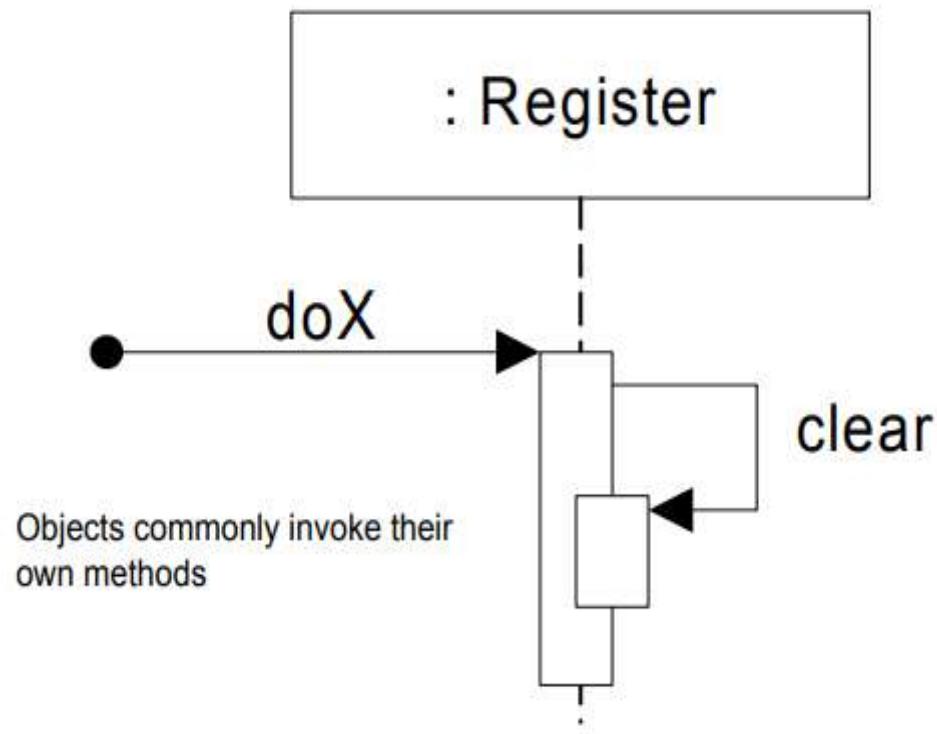
- There are two ways to show the return result from a message:
 1. Using the message syntax *returnVar = message(parameter)*.
 2. Using a reply (or return) message line at the end of an activation bar.



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

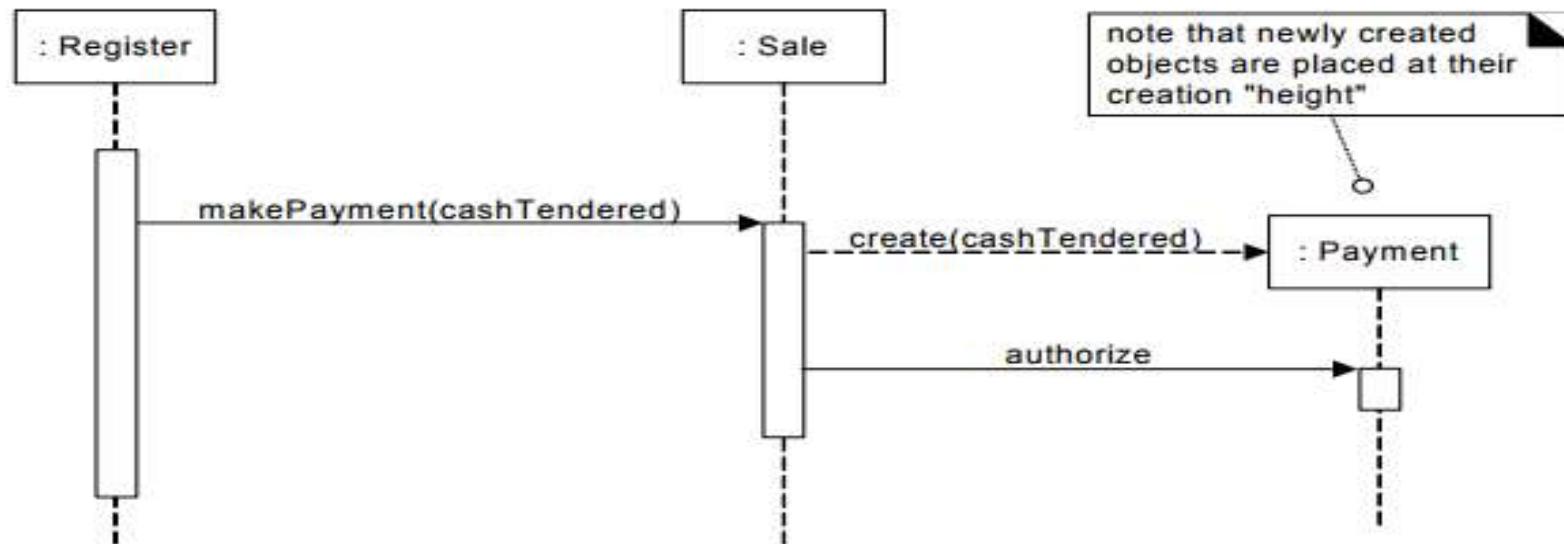


ssh

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".



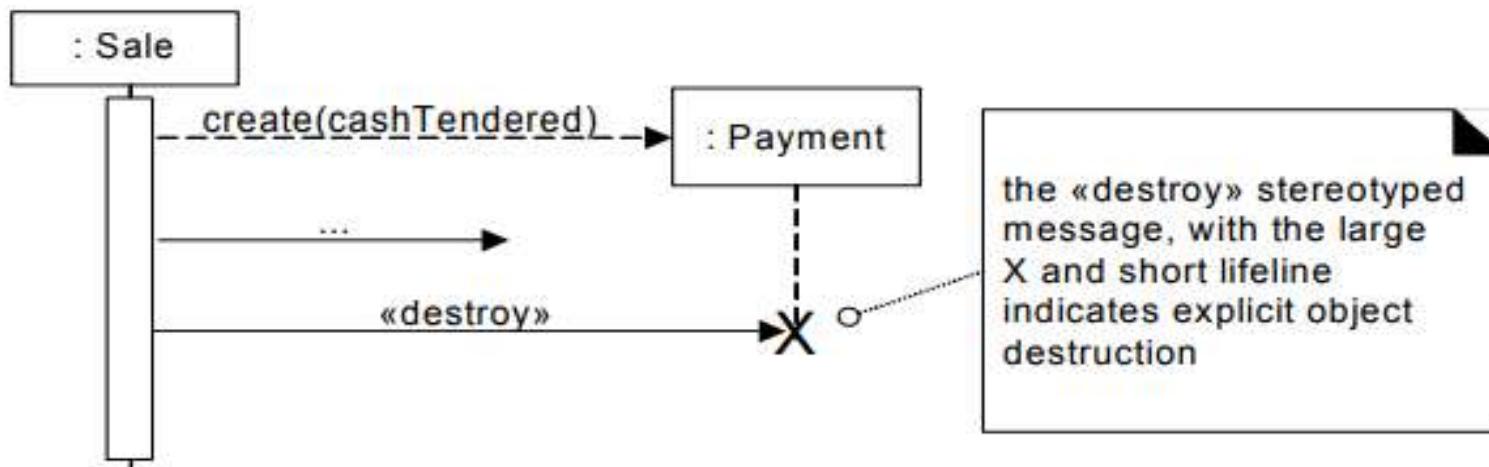
Dashed line for 'create' really not needed, though its now official UML
Use 'create' for calls to a constructor



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



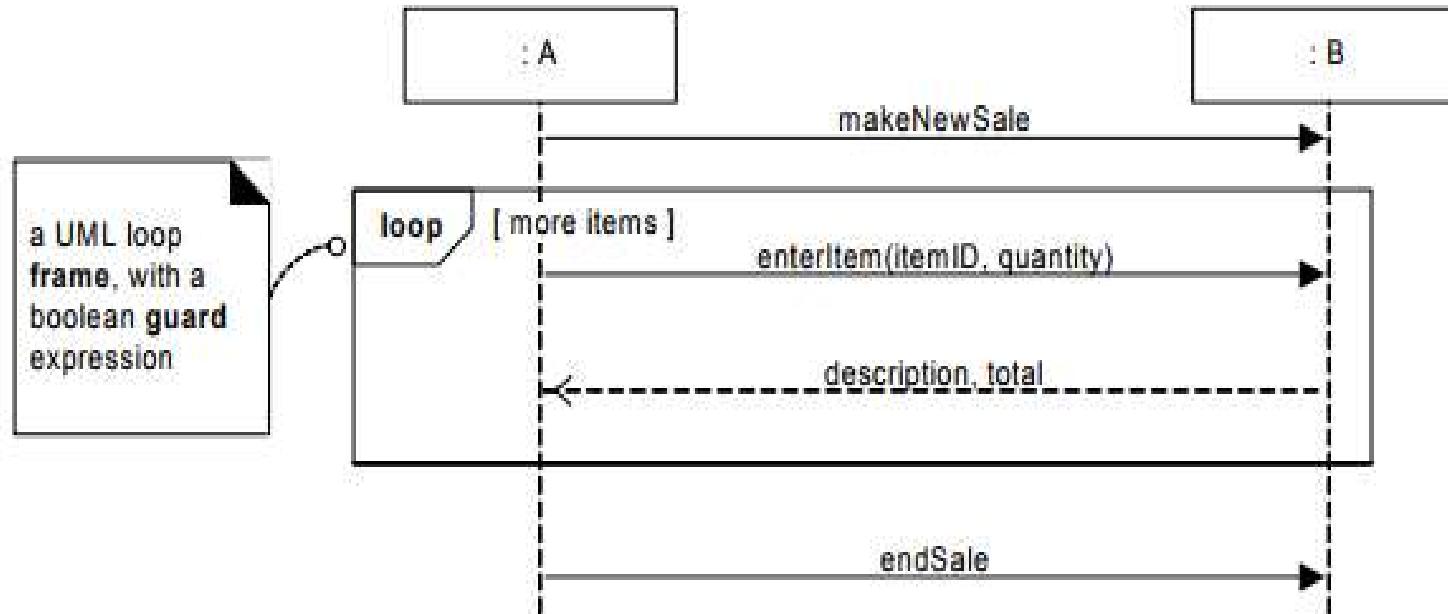
Usually not necessary for languages with automatic garbage collection (e.g., Java)



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

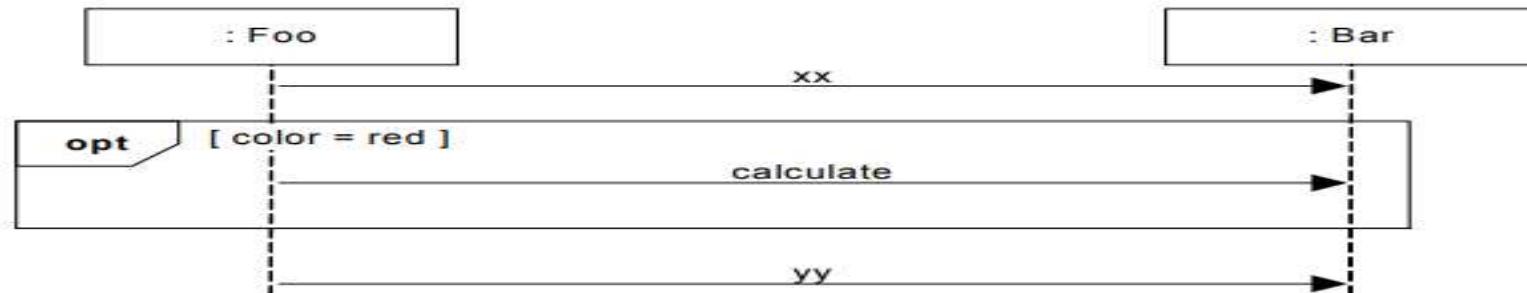


ssh

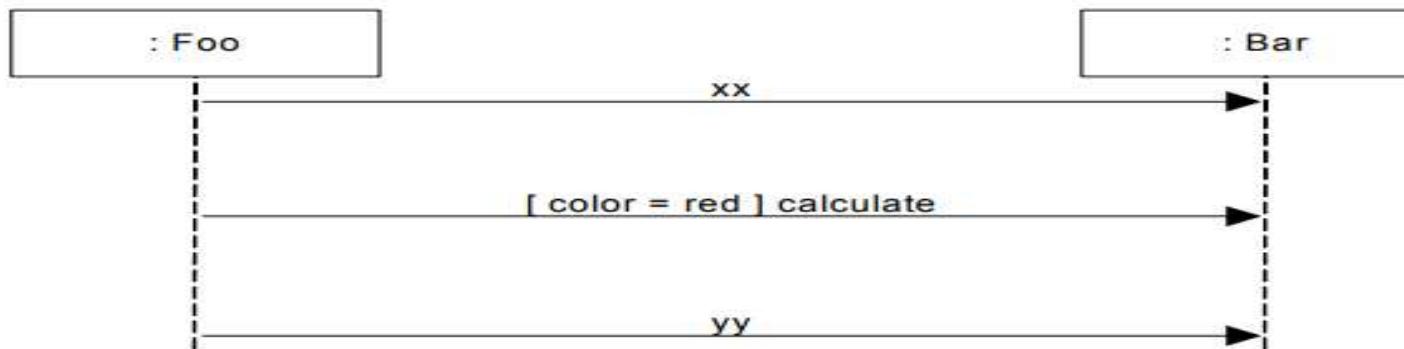
Basic Sequence Diagram Notation

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:

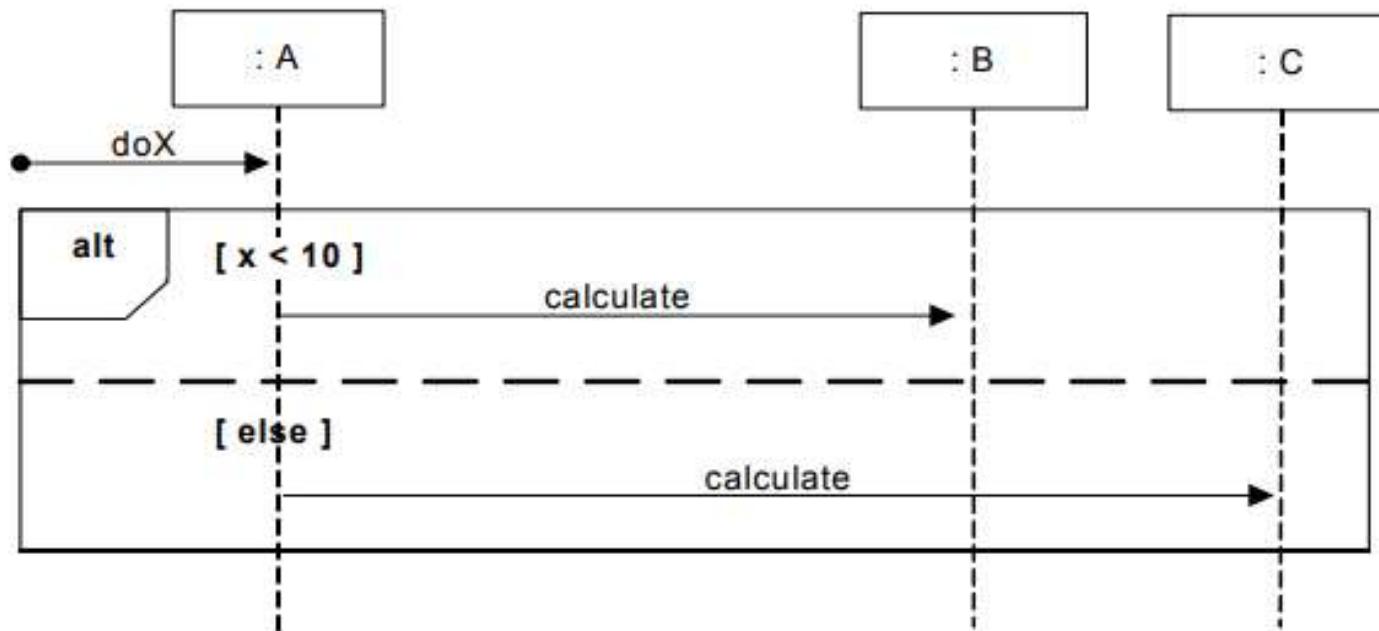
Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write <i>loop(n)</i> to indicate looping n times. There is discussion that the specification will be enhanced to define a FOR loop, such as <i>loop(i, 1, 10)</i>
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.

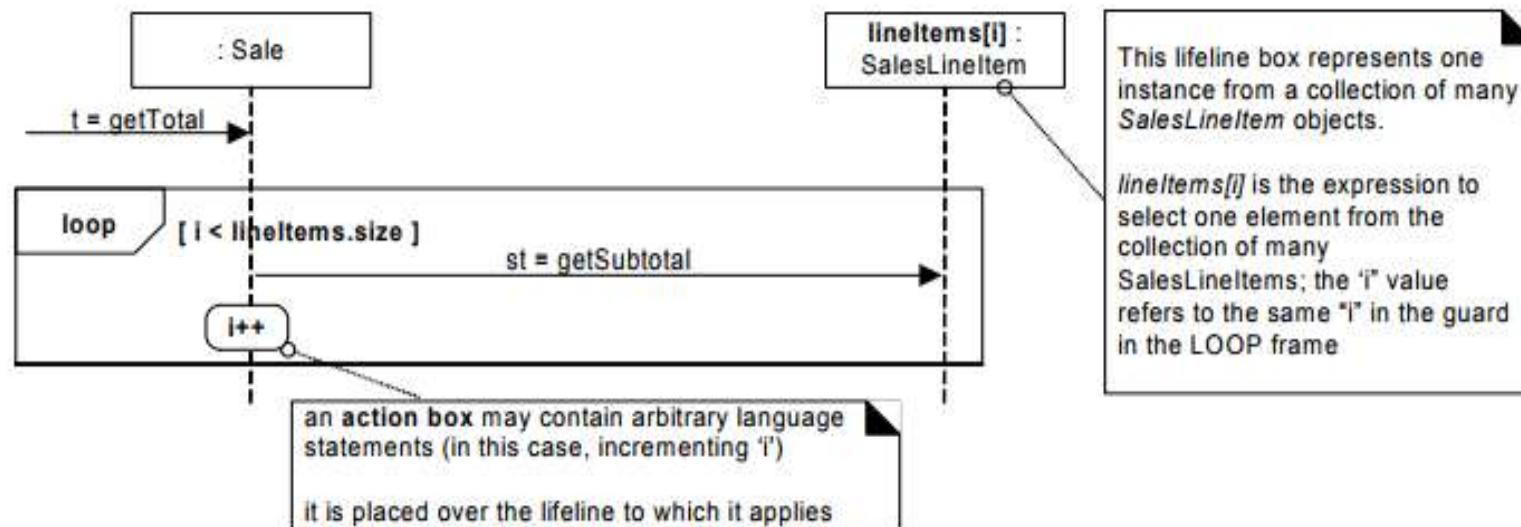


Alt frame show mutually exclusive interactions

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.



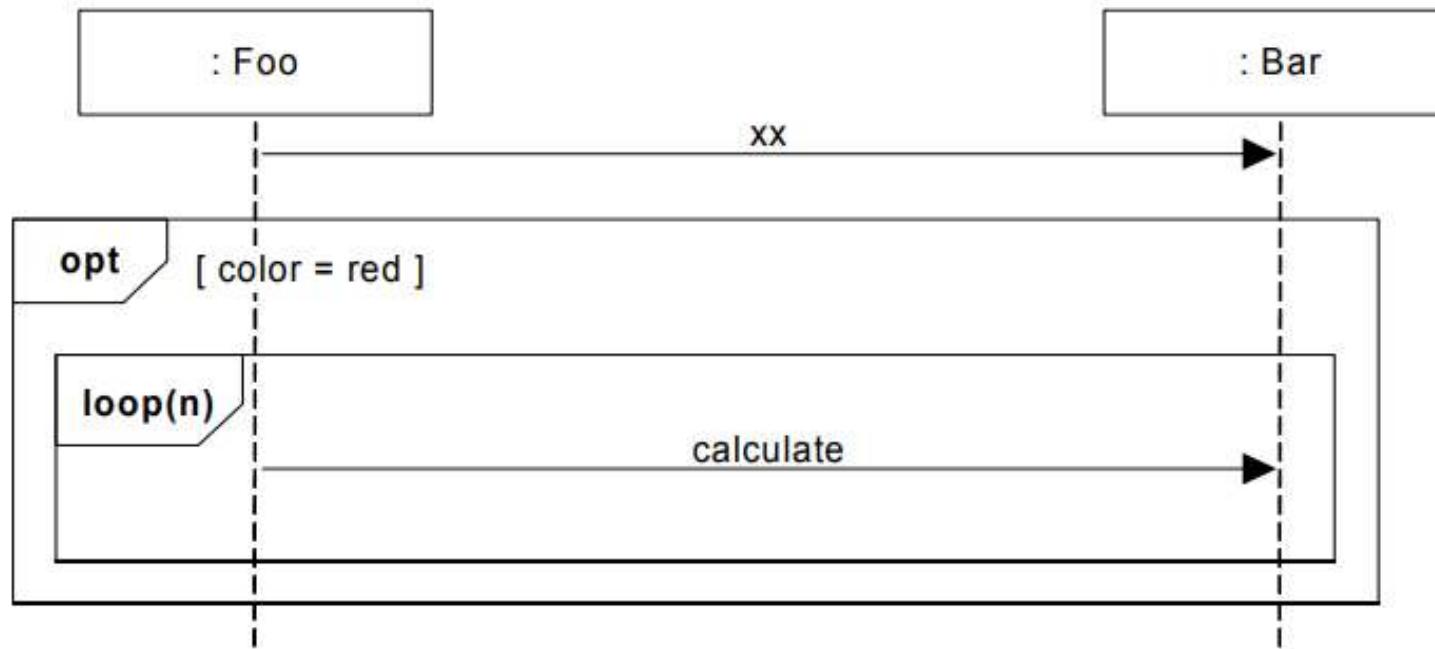
Technique for looping over a collection
Loop details are explicit; diagram more cluttered
Note Java code on p. 234 showing new for loop syntax



Basic Sequence Diagram Notation

Nesting of Frames

Frames can be nested.



A loop frame nested within an optional frame

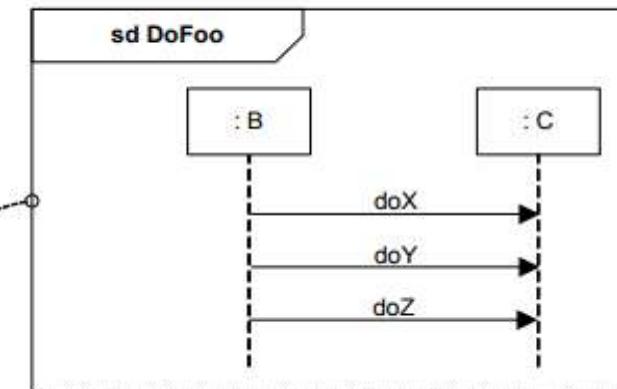
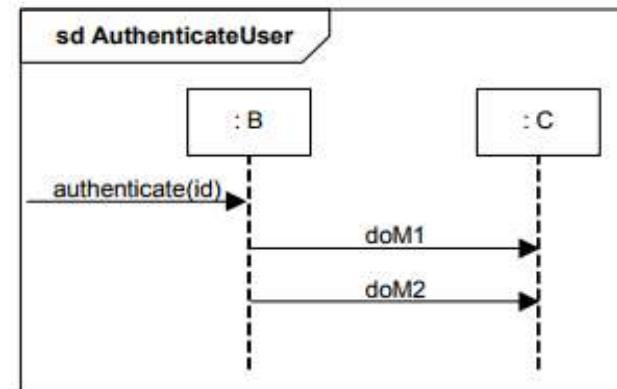
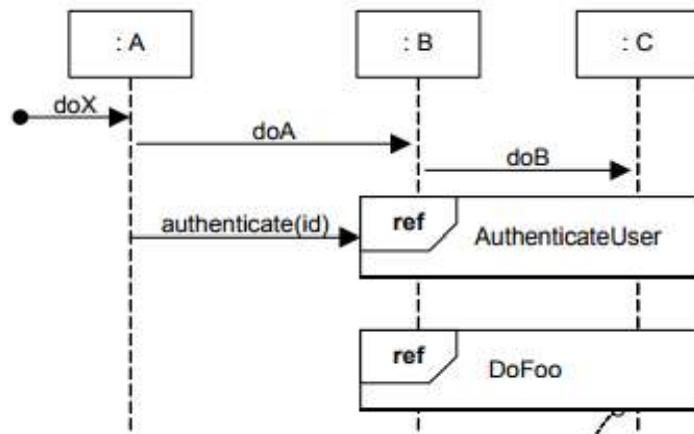


Basic Sequence Diagram Notation

Nesting of Frames

Frames can be nested.

SD can contain frames that
Reference other SDs

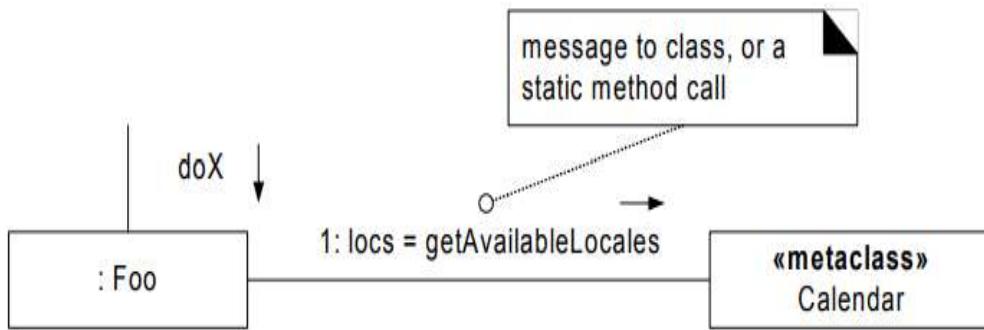


ssh

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**



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(); // ...
    }
}
```



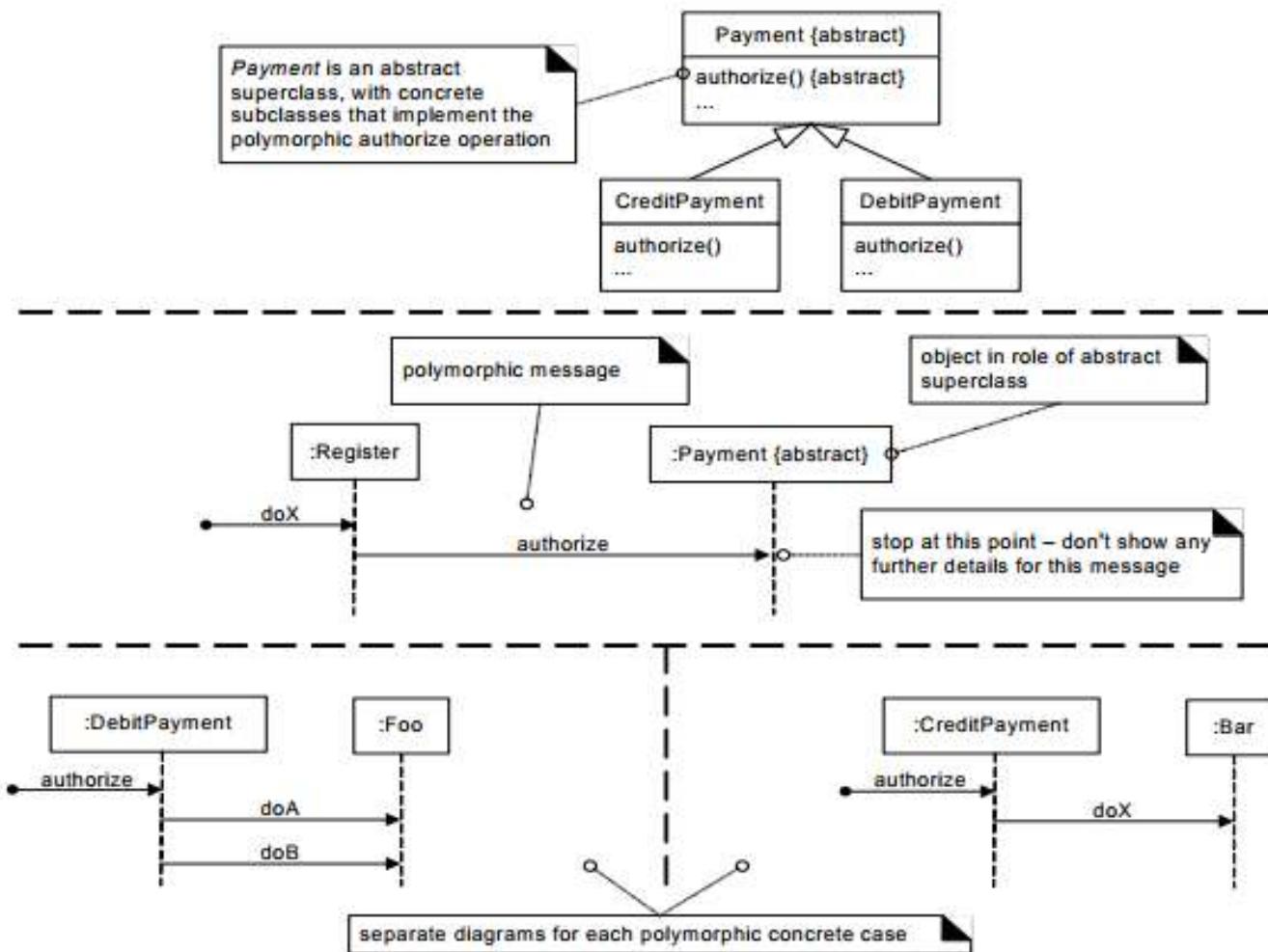
Basic Sequence Diagram Notation

- **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



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**

a stick arrow in UML implies an asynchronous call

a filled arrow is the more common synchronous call

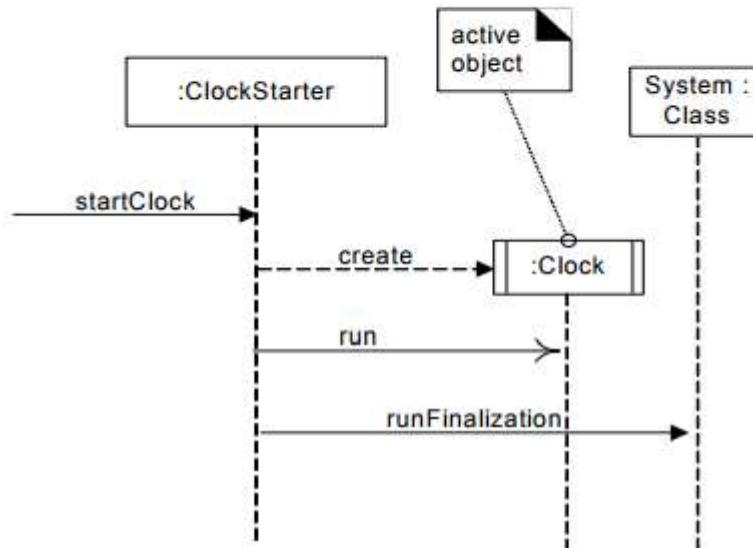
In Java, for example, an asynchronous call may occur as follows:

```
// 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

Active objects run in their own thread



Solid vs. stick arrowheads easily confused when sketching models

See Java code p. 239-240

ssh

Basic Communication diagram Notation

①

The notations are

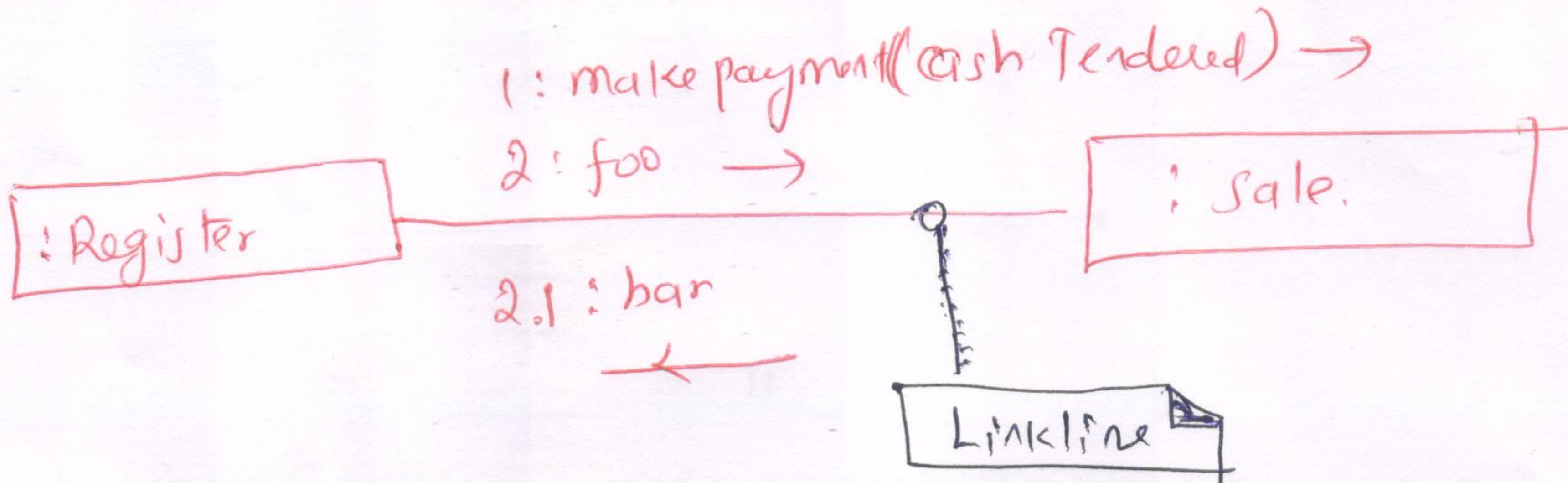
- (i) Links
- (ii) Messages
- (iii) Message to 'self' or 'this'
- (iv) Creation of instances
- (v) Message number sequencing
- (vi) conditional messages
- (vii) Mutually Exclusive Conditional paths
- (viii) Iteration or Looping
- (ix) Iteration over a collection
- (x) Messages to a classes to invoke static(class) methods
- (xi) polymorphic messages and cases
- (xii) Asynchronous and Synchronous calls.

Links:

- A link is connection path b/w two objects.
- It indicates some form of navigation and visibility b/w the objects is possible.
- a link is an instance of association

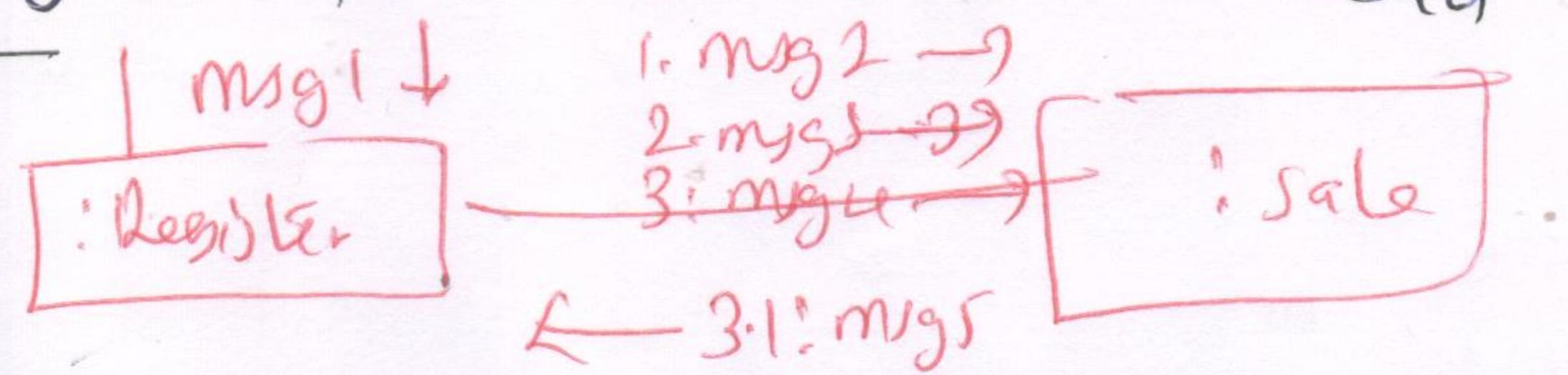
Example: Linklines

(2)



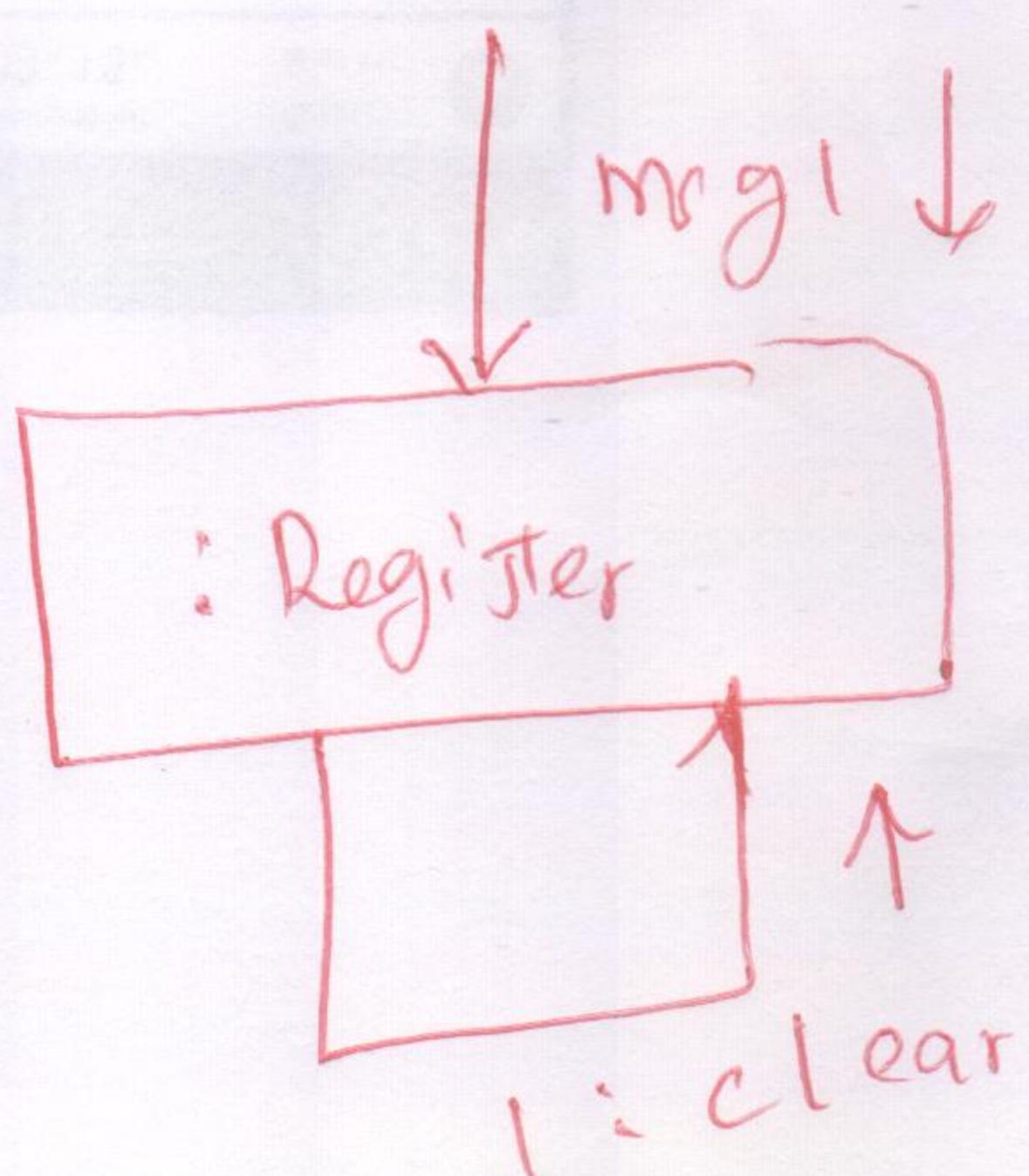
Messages

- * Each message between objects is represented with a message expression and small arrow indicating the direction of the message,
- * A sequence number is added to show the sequential order of messages in the current thread of control.



Message to "self" or "This"

- * A message can be sent from an object to itself.
- * This is illustrated by a link to itself, with messages flowing along the link.



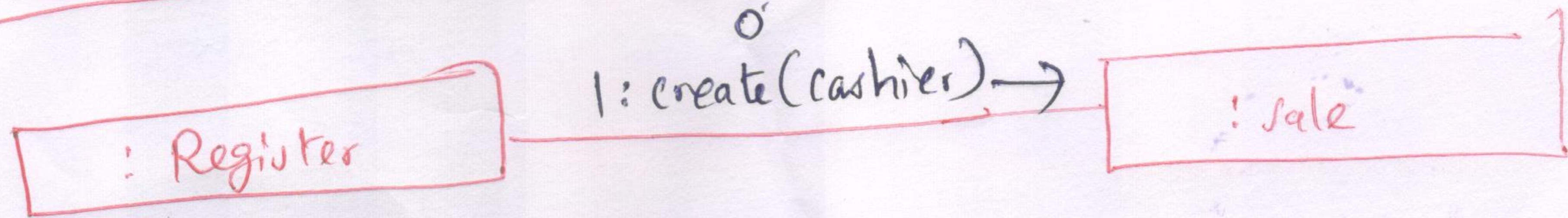
(3)

Creation of instances

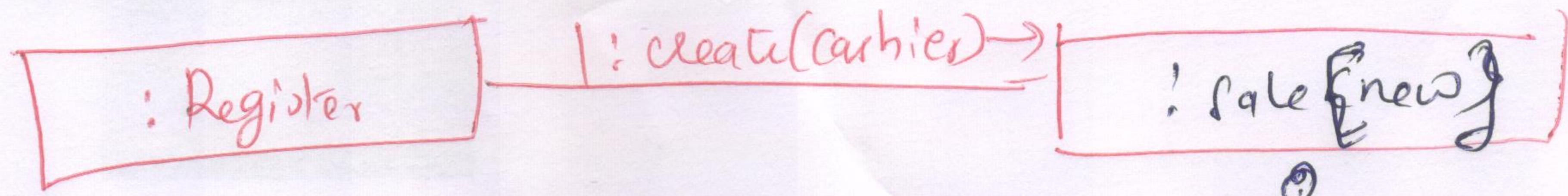
- Any message can be used to create an instance, but in UML, a message named **create** is used for this purpose.
- If another message name is used, the message may be annotated with a **UML stereotype**, **LLCreate**.
- The create message may include parameters indicating the passing of initial values.
- Furthermore, the **UML Tagged Value {new}** may be optionally added to the lifeline box to highlight the creation.

Three ways to show creation in communication

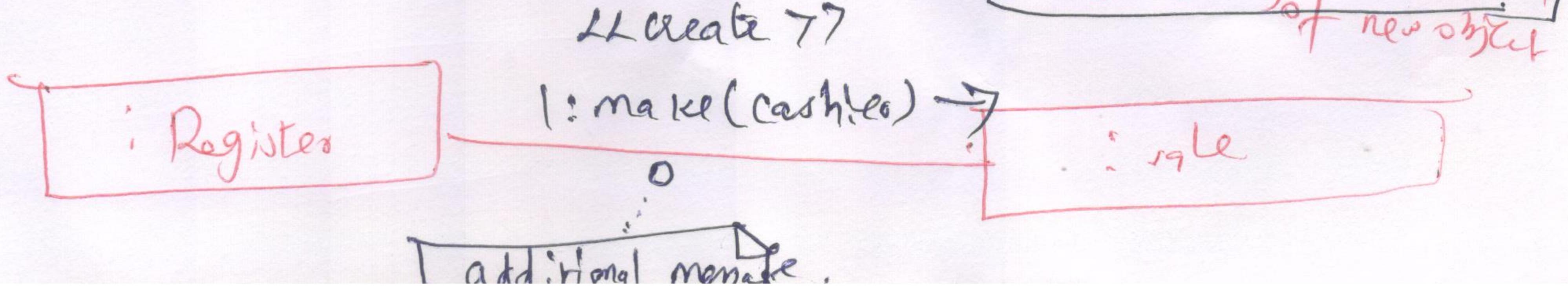
Way 1



Way 2



Way 3



(4)

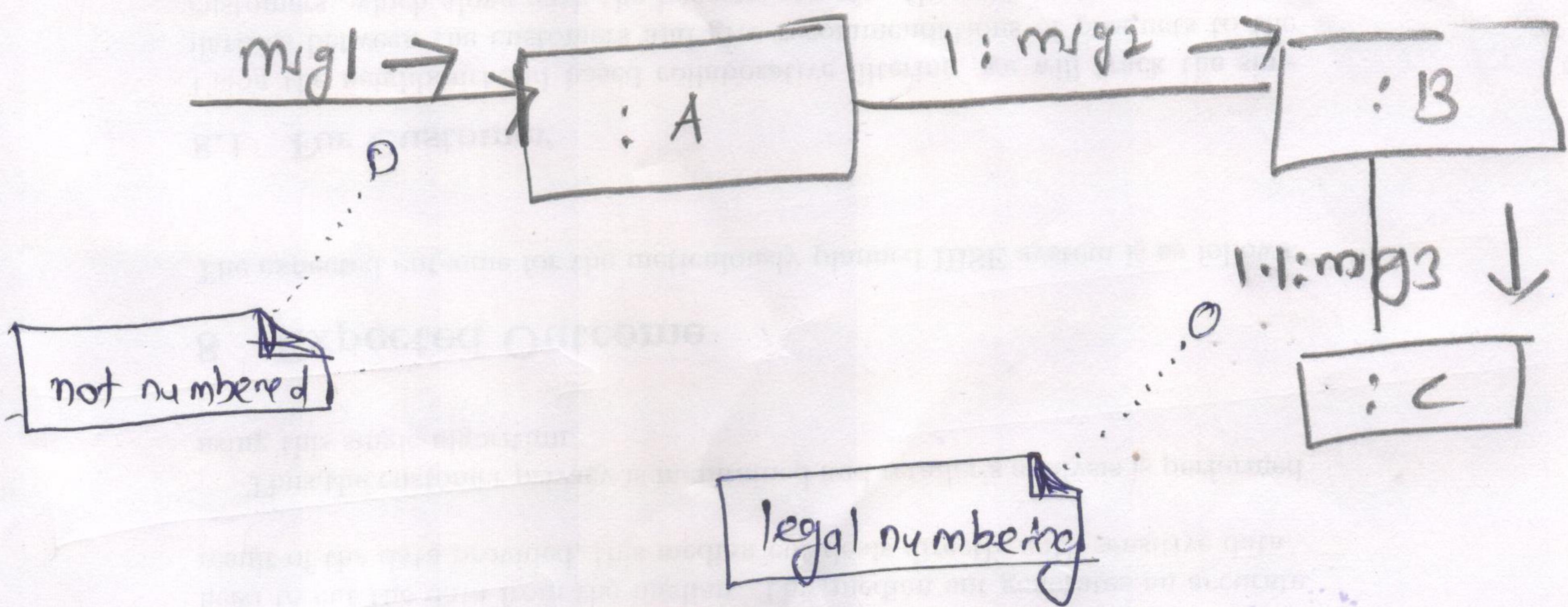
Message Number Sequencing

The order of messages is illustrated with sequence numbers

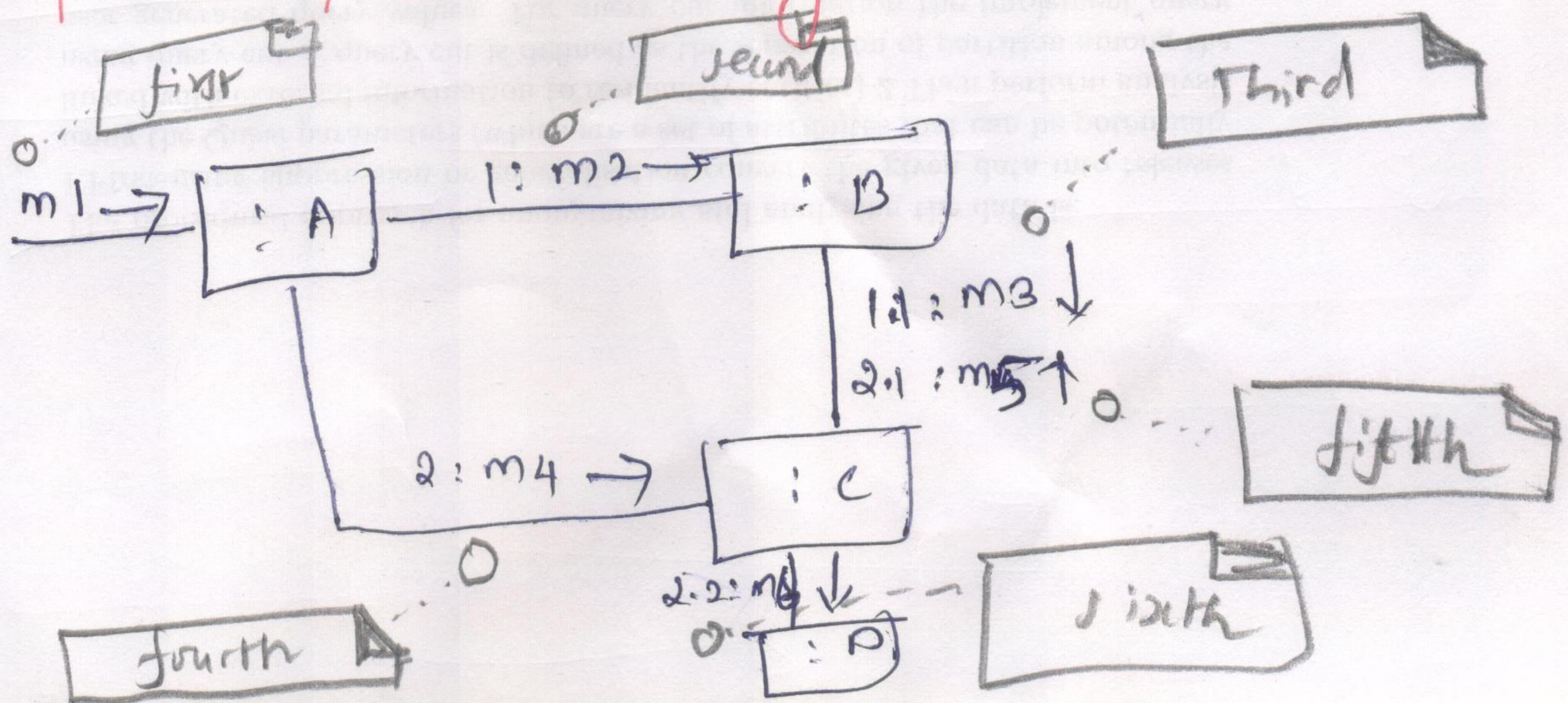
① The first message is not numbered

② The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them.

Fig: sequence numbering



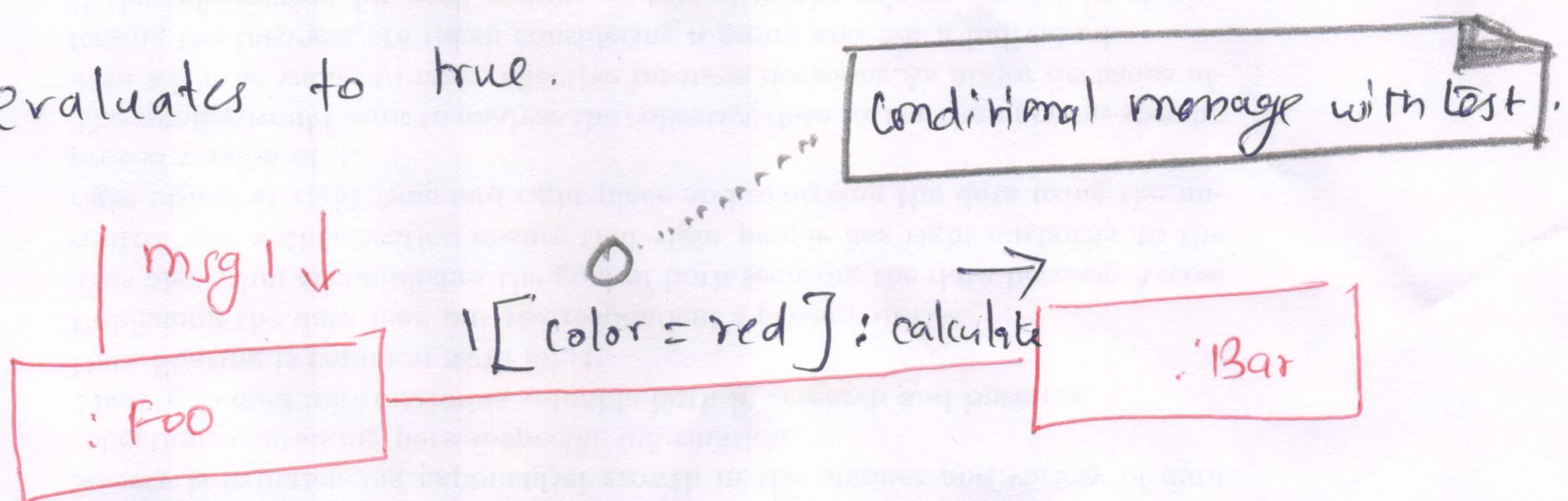
Complex Sequence numbering



⑤

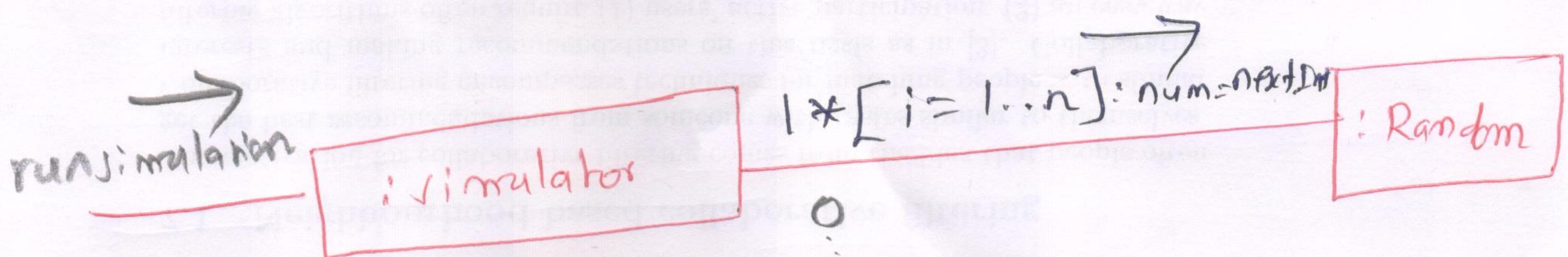
Conditional messages

- Conditional message is shown by sequence number with a conditional message clause in square brackets, similar to an iteration clause.
- The message is only sent if the clause evaluates to true.



Mutually Exclusive Conditional Paths

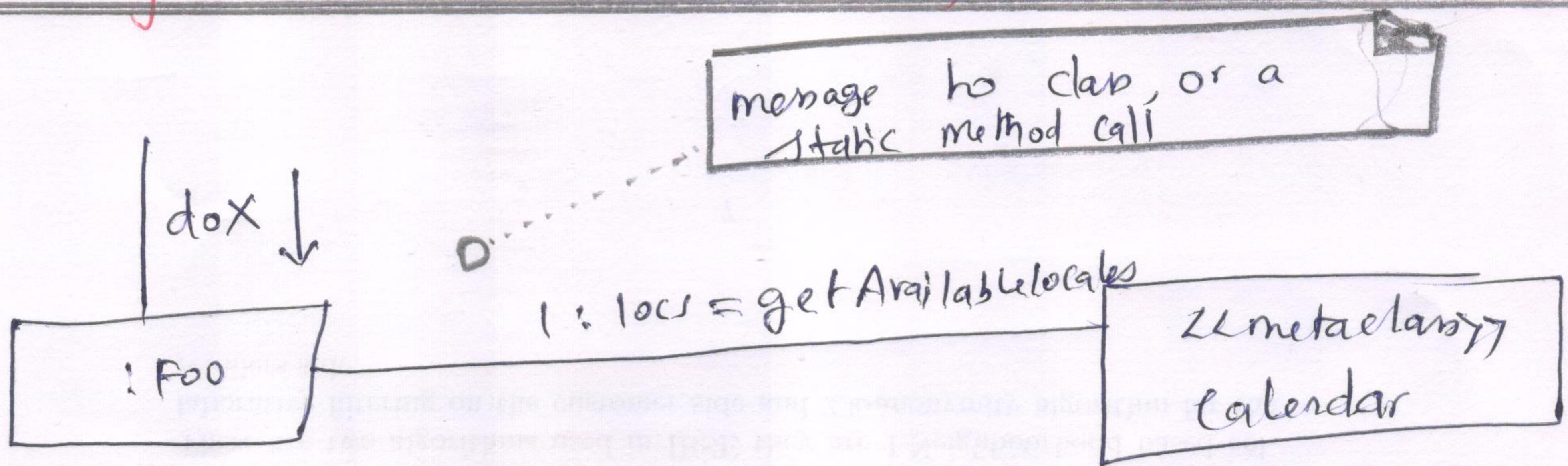
- Fig states that either 1a or 1b could execute after msg1. Both are sequence number 1 since either could be the first internal message.
- 1b.1 is nested message within 1b.



Iteration is indicated with a * and an optional iteration clause following the sequence number.

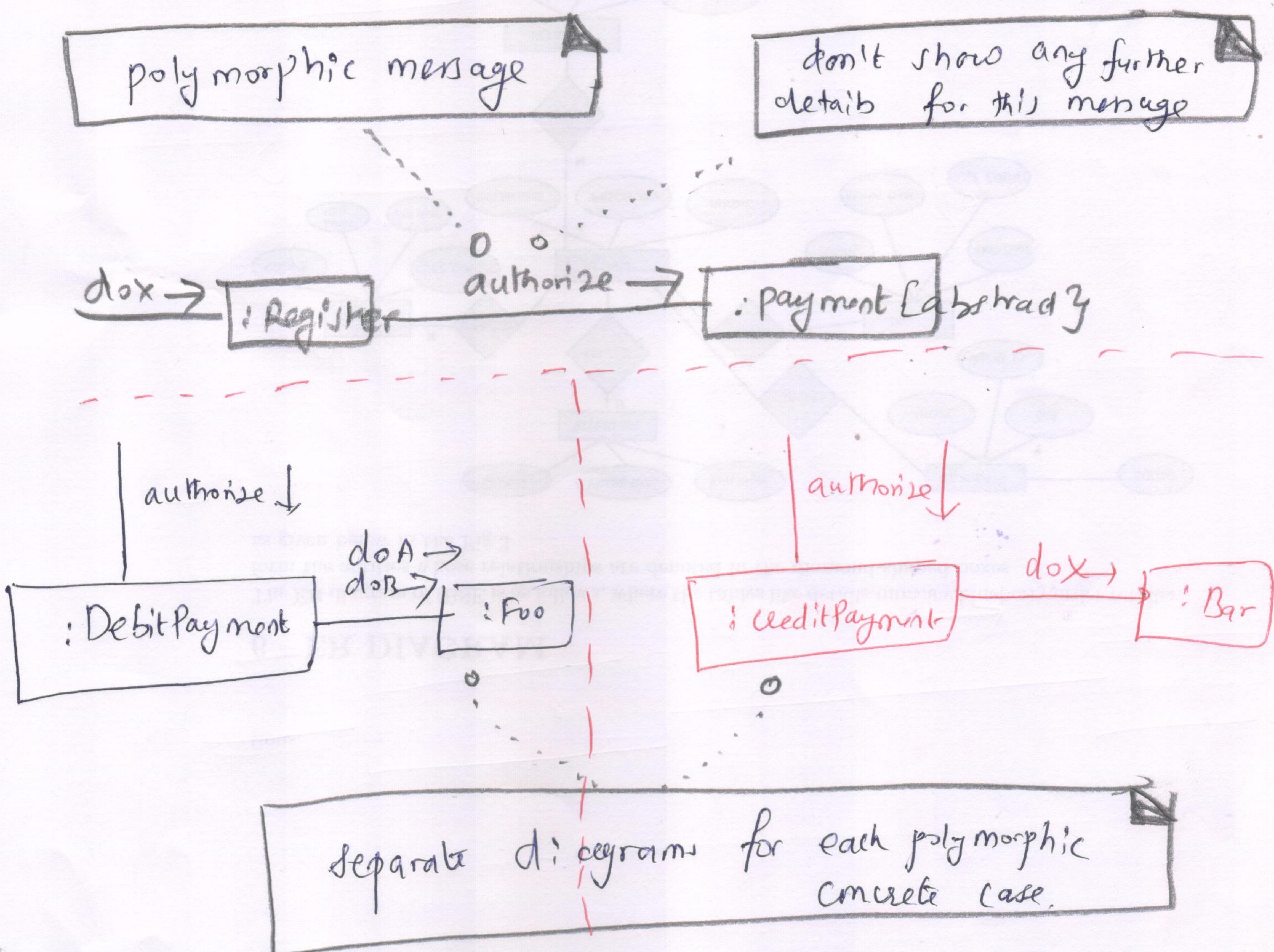
(6)

Messages to a classes to invoke static (class) methods



Poly morphic Messages and cases

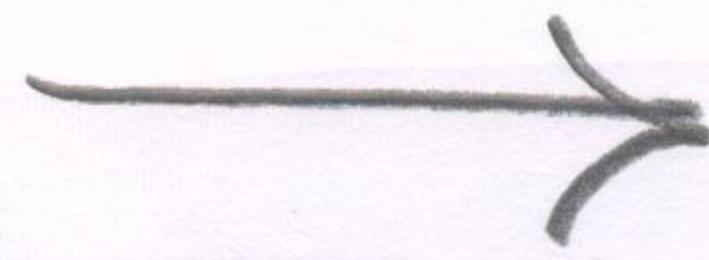
- As in sequence diagram Case multiple communication diagrams can be used to show each ^{concrete} poly morphic case



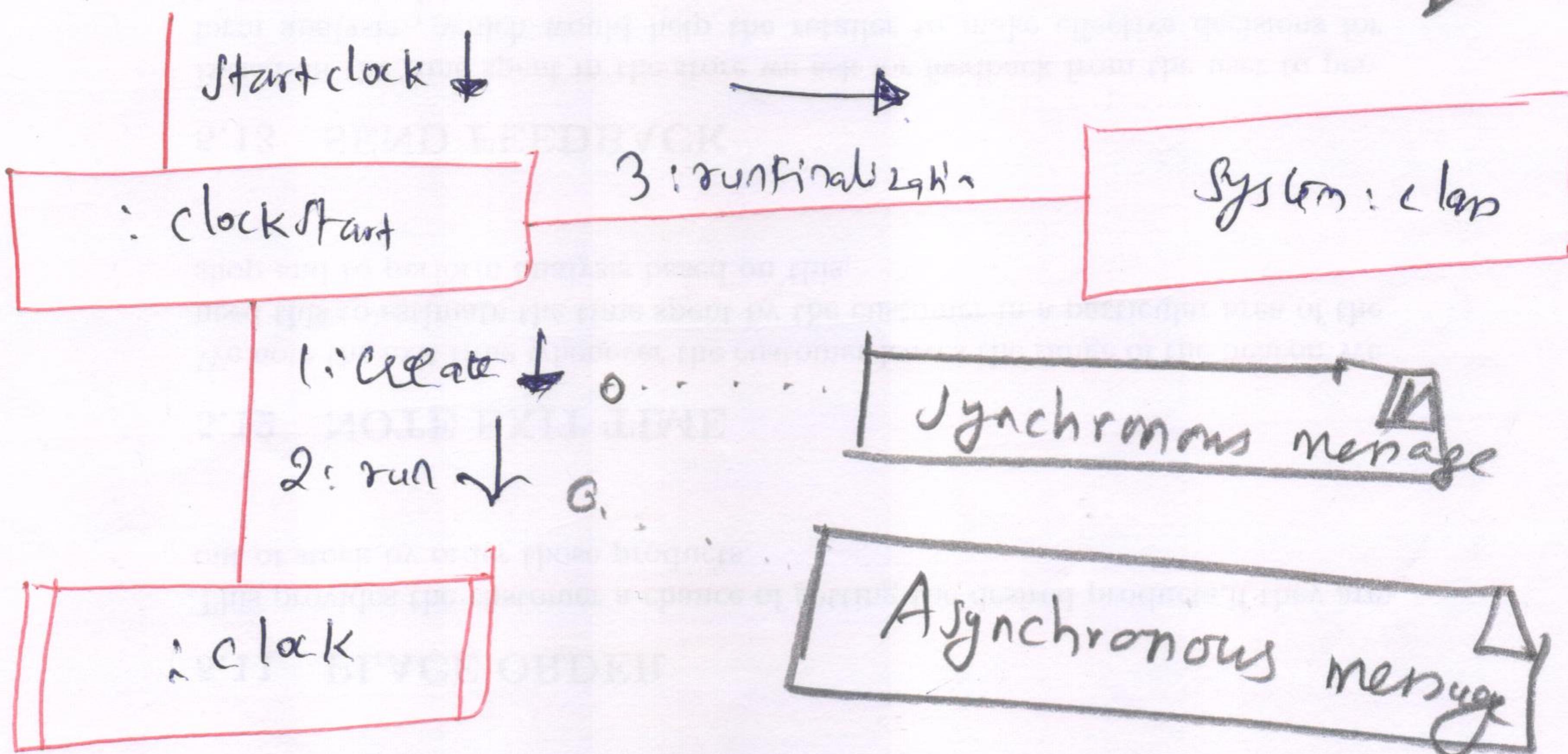
⑦

Asynchronous and Synchronous Calls

→ Asynchronous calls are shown with stick arrow



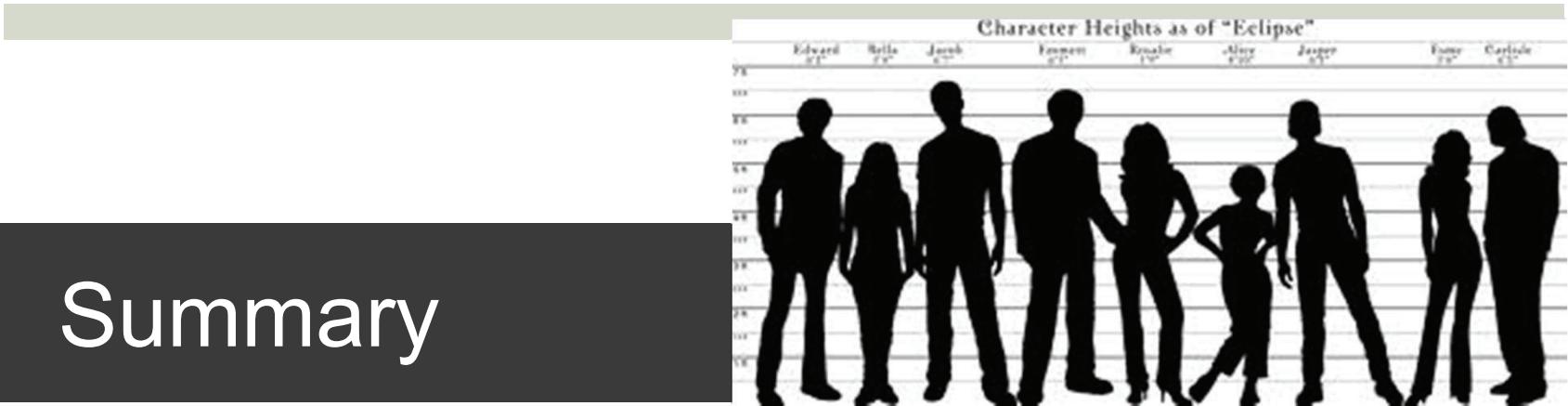
→ Synchronous calls are shown with filled arrow



2.2. DIGITAL RECORDING ADVICE

Rules of thumb

- ❑ Rarely use options, loops, alt/else
 - ❑ These constructs complicate a diagram and make them hard to read/interpret.
 - ❑ Frequently it is better to create multiple simple diagrams
- ❑ Create sequence diagrams for use cases when it helps clarify and visualize a complex flow
- ❑ Remember: the goal of UML is communication and understanding



Summary

- ❑ Sequence diagrams model object interactions with an emphasis on time ordering
- ❑ Method call lines
 - ❑ Must be horizontal!
 - ❑ Vertical height matters!
“Lower equals Later”
 - ❑ Label the lines
- ❑ Lifeline – dotted vertical line
- ❑ Execution bar – bar around lifeline when code is running
- ❑ Arrows
 - ❑ Synchronous call (you’re waiting for a return value) – triangle arrow-head
 - ❑ Asynchronous call (not waiting for a return) – open arrow-head
 - ❑ Return call – dashed line