Interaction Diagrams

- Used to illustrate how objects interact via messages
- Exhibits dynamic behavior of the system i.e., dynamic object modeling
- Interactive behavior is represented by two diagrams known as sequence and communication diagrams.
- The purpose of both the diagrams are similar
- Sequence emphasize on the time sequence of messages and communication emphasizes on the structural organization of the objects that send and receive messages.

Purpose of Interaction Diagrams

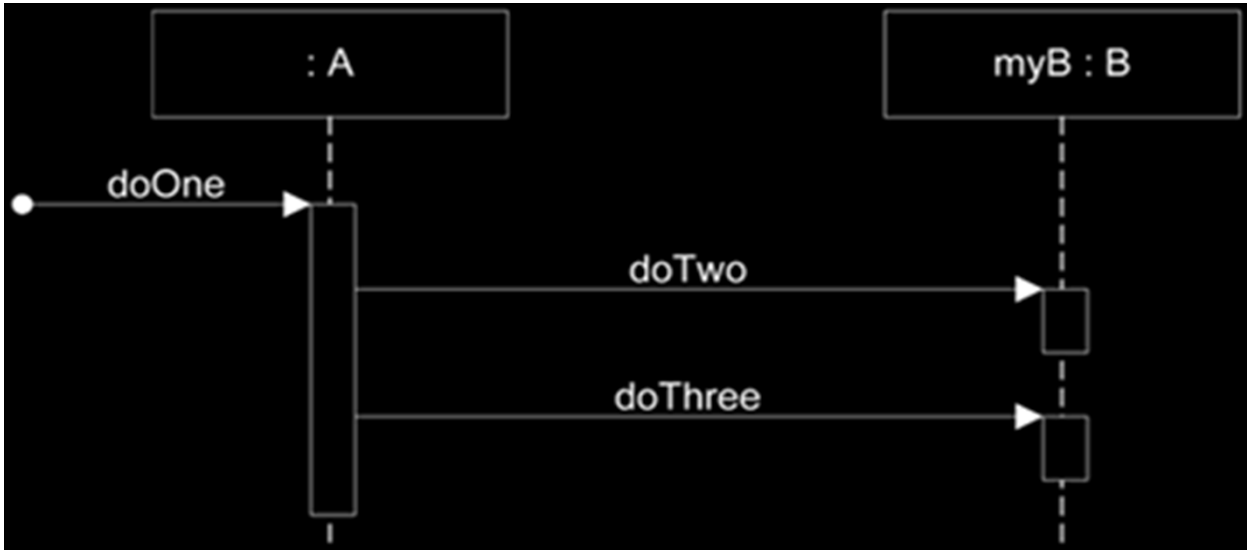
- To capture the dynamic behavior of a system
- To describe the message flow in the system
- To describe the structural organization of the objects
- To describe the interaction among objects

Sequence Diagrams

- UML sequence diagrams are used to represent or model the flow of messages, events and actions between the objects or components of a system.
- Time is represented in the vertical direction showing the sequence of interactions of the header elements, which are displayed horizontally at the top of the diagram.
- SD are models that describe how a group of objects collaborate in some behavior – typically a single use case.
- The diagrams are read left to right and descending

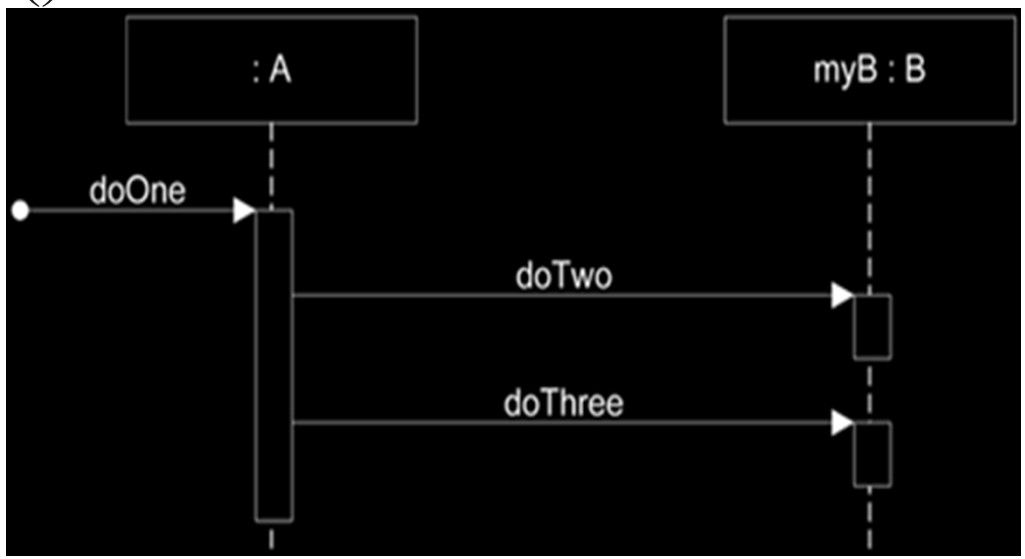
Sequence Diagrams

- SD illustrate interactions in a kind of fence format, in which each new object is added to the right



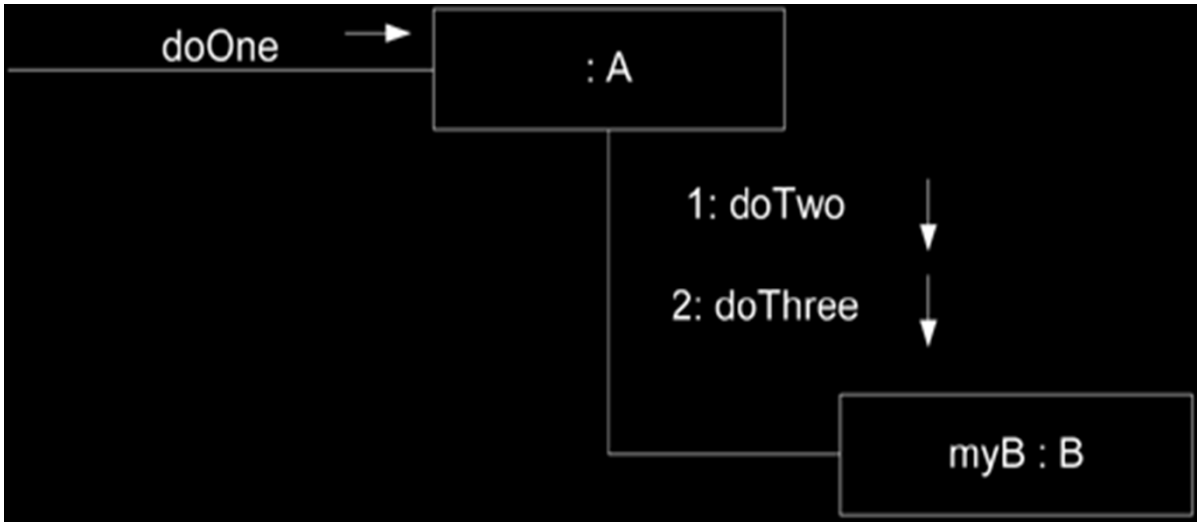
Sequence Diagrams - What might this represent in code?

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



Communication Diagrams

- illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram

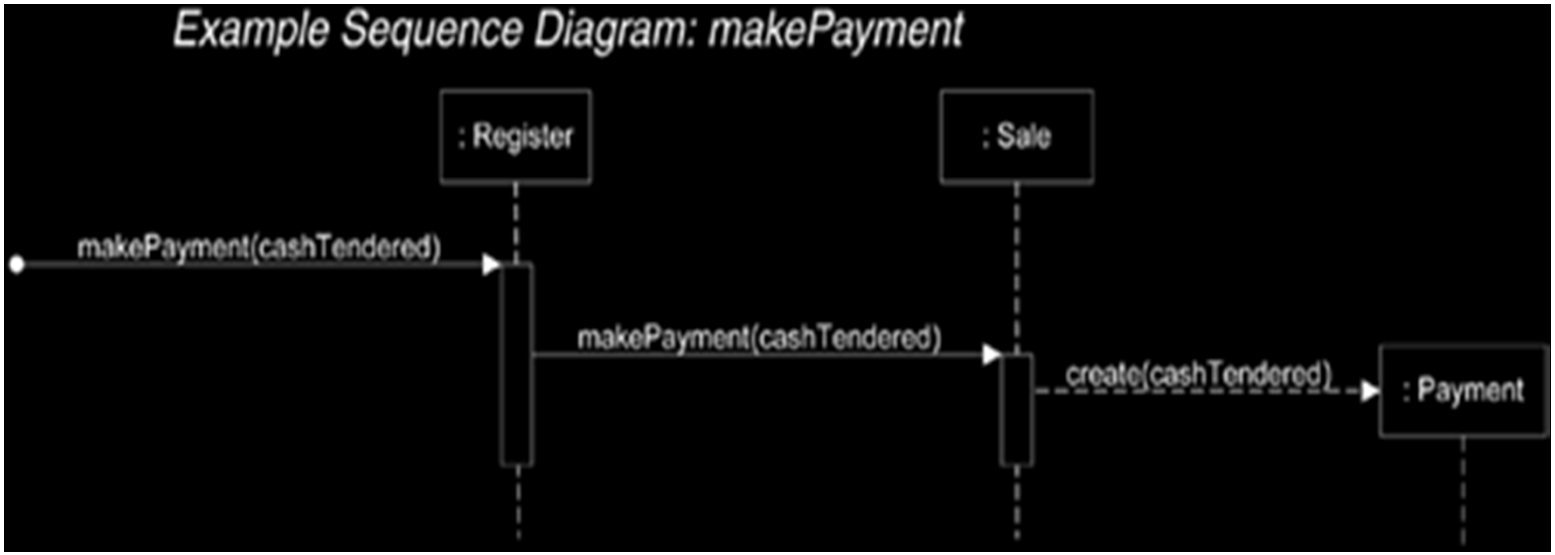


Sequence Vs. Communication Diagrams

Type	Strengths	Weakness
Sequence	Shows sequence or time ordering of messages	Force to extend to the right when adding new objects
	Large set of detailed notation options	Consumes horizontal space
Communication	Space economical – flexibility to add new objects in two dimensions	Few notation options

Example: Sequence Diagrams

- The message *makePayment* is sent to an instance of a *Register*. The sender is not identified.
- The *Register* instance sends the *makePayment* message to a *Sale* instance.
- The *Sale* instance creates an instance of a *Payment*.

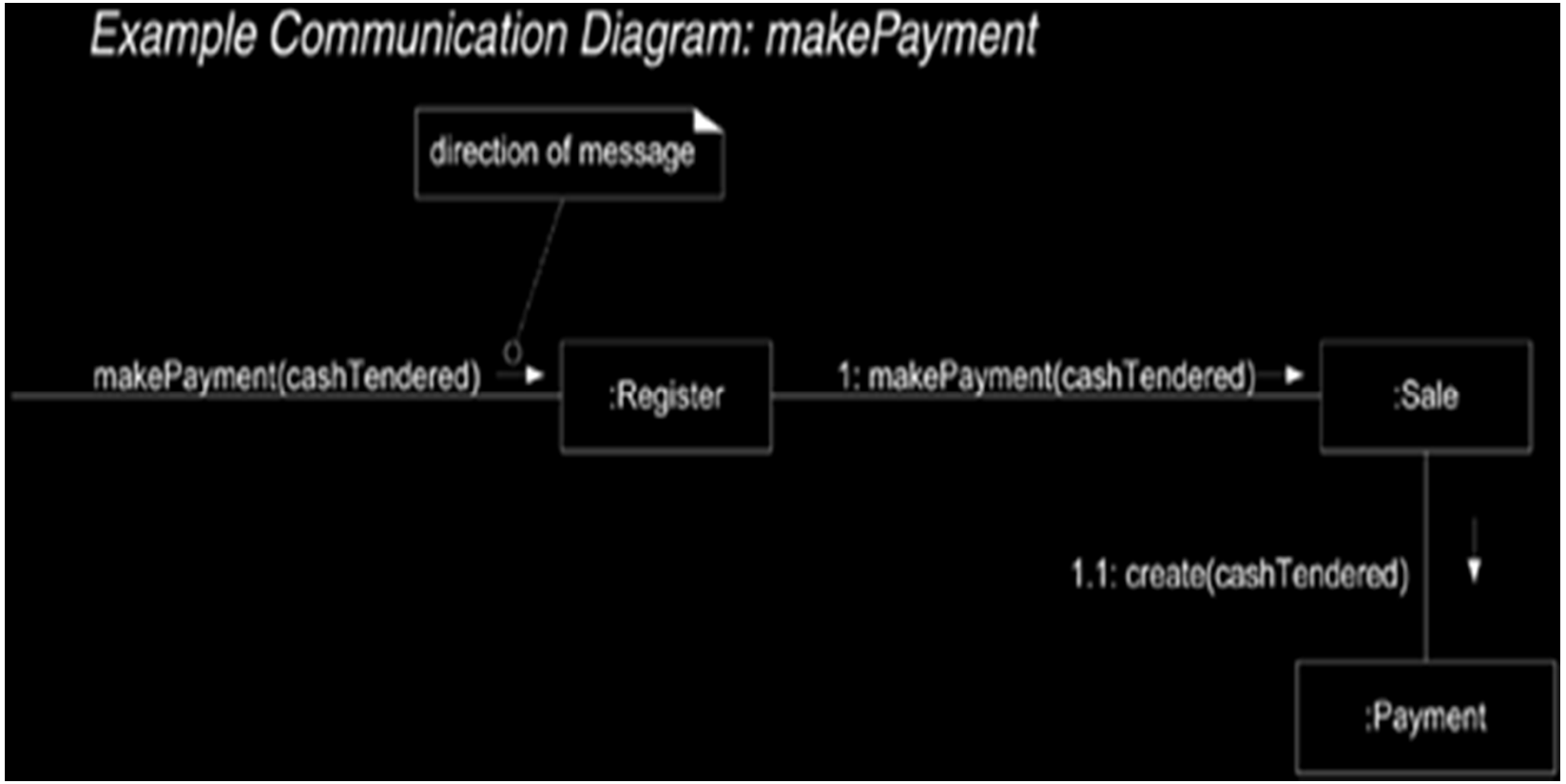


Example: Sequence Diagrams - Code

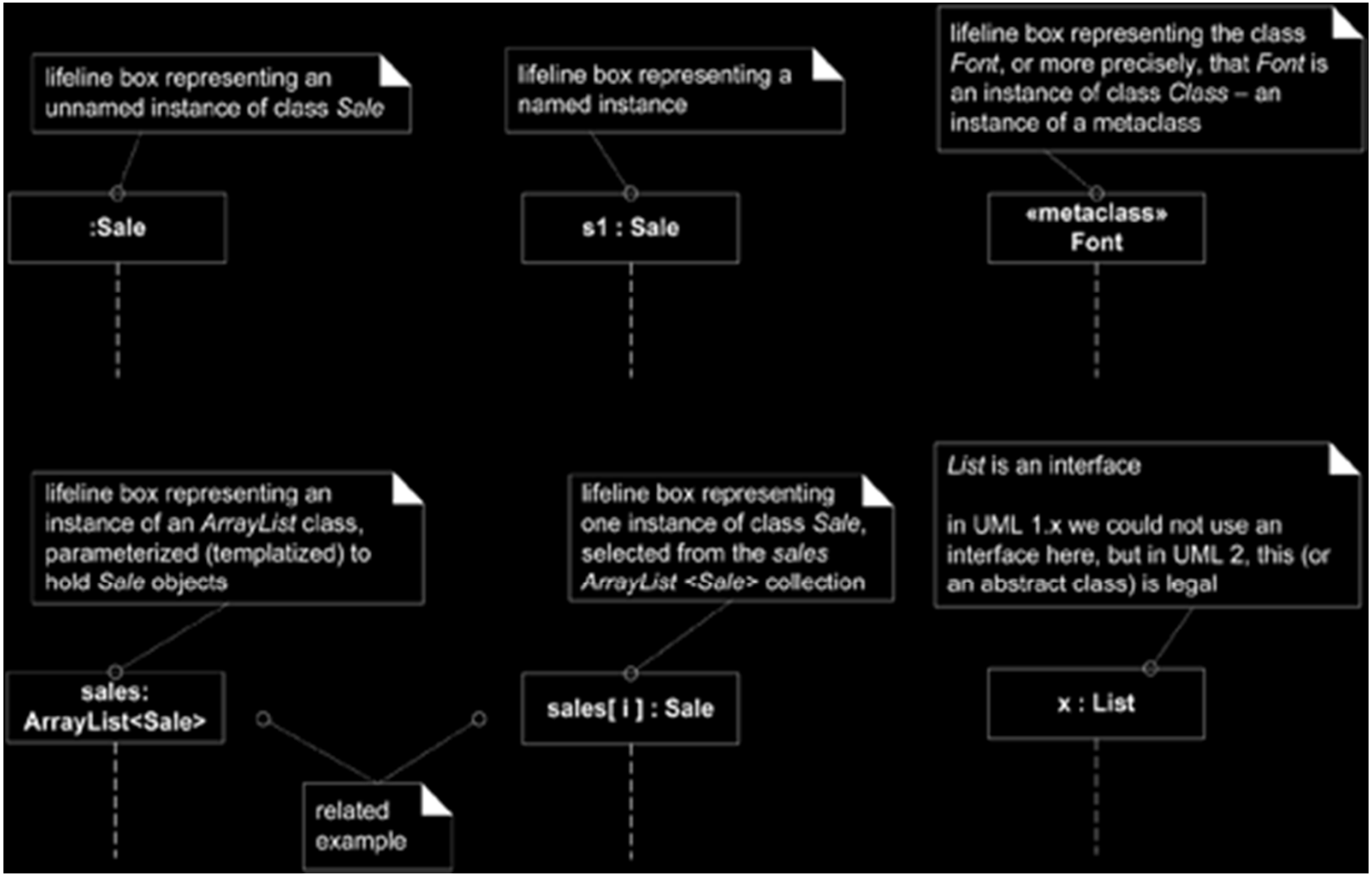
```
public class Sale
   {
      private Payment payment;

     public void makePayment( Money cashTendered )
      {
                payment = new Payment( cashTendered );
                //. .
      }
  // . .
   }
```

Example: Communication diagram



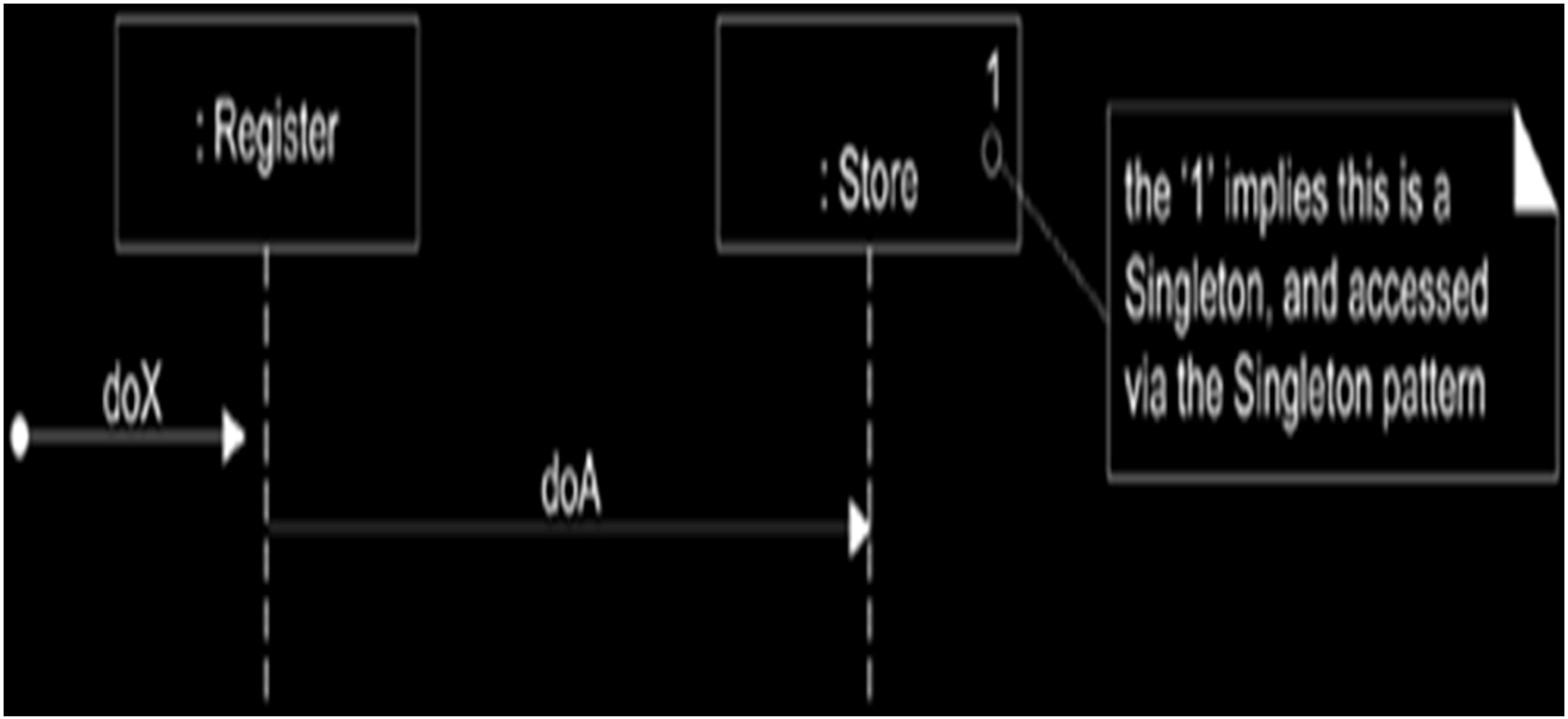
Common *UML* Interaction Diagram Notation



Message Expression Syntax

- Interaction diagrams show messages between objects; the *UML* has a standard syntax for these message expressions:
- return = message(parameter : parameterType) : returnType
- Don’t have to use full syntax in all cases

Notation for Singleton Object



Basic Sequence diagram notation

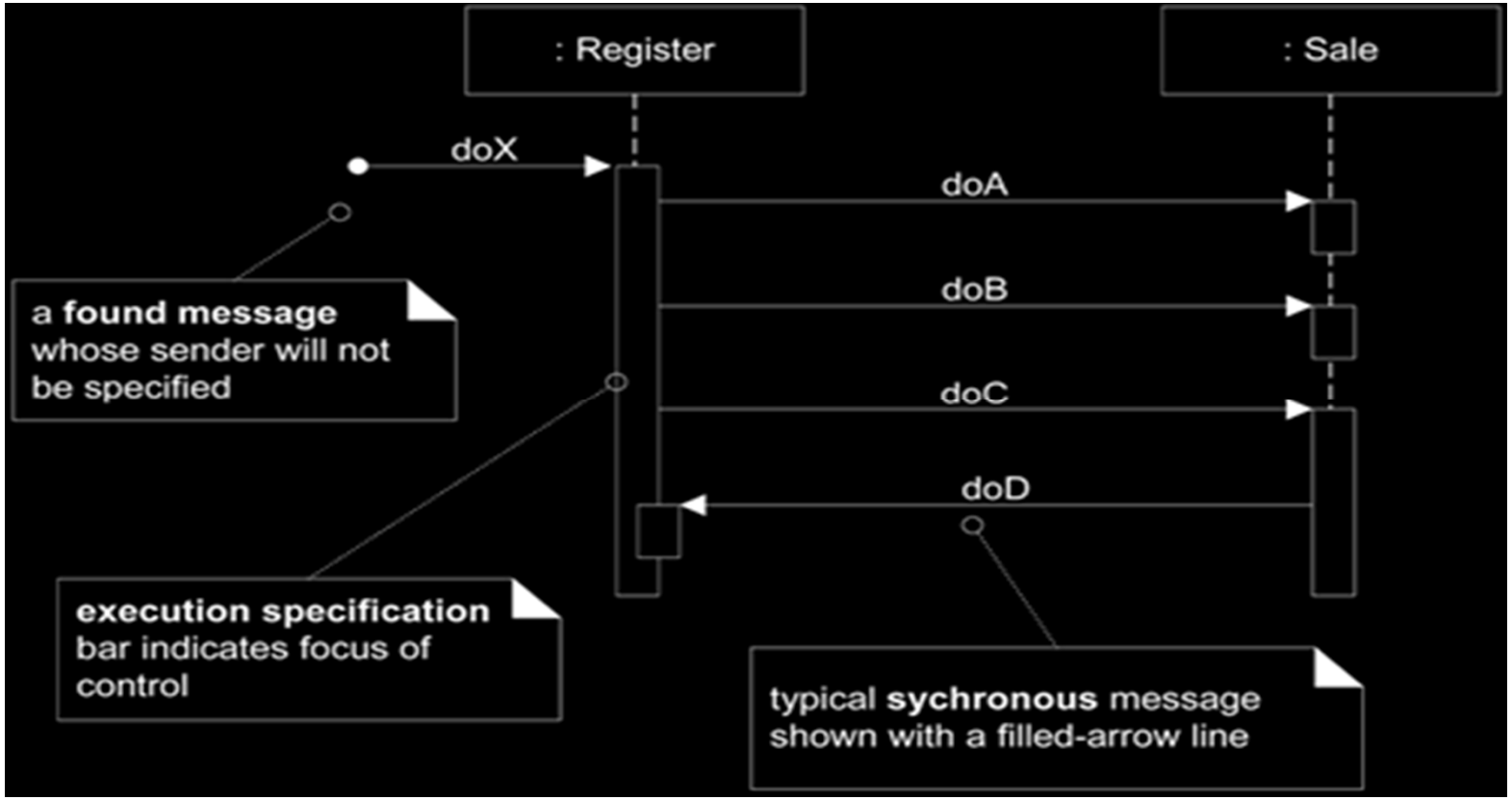
Lifeline Boxes and Lifelines

- In sequence diagrams the lifeline boxes include a vertical line extending below them—these are the actual lifelines.

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.

Basic Sequence diagram notation



Basic Sequence diagram notation

Focus of Control and Execution Specification Bar

- Sequence diagrams may also show the focus of control (informally, in a regular blocking call, the operation is on the call stack) using an **execution specification** bar (previously called an **activation bar** or simply an **activation** in *UML* 1)

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

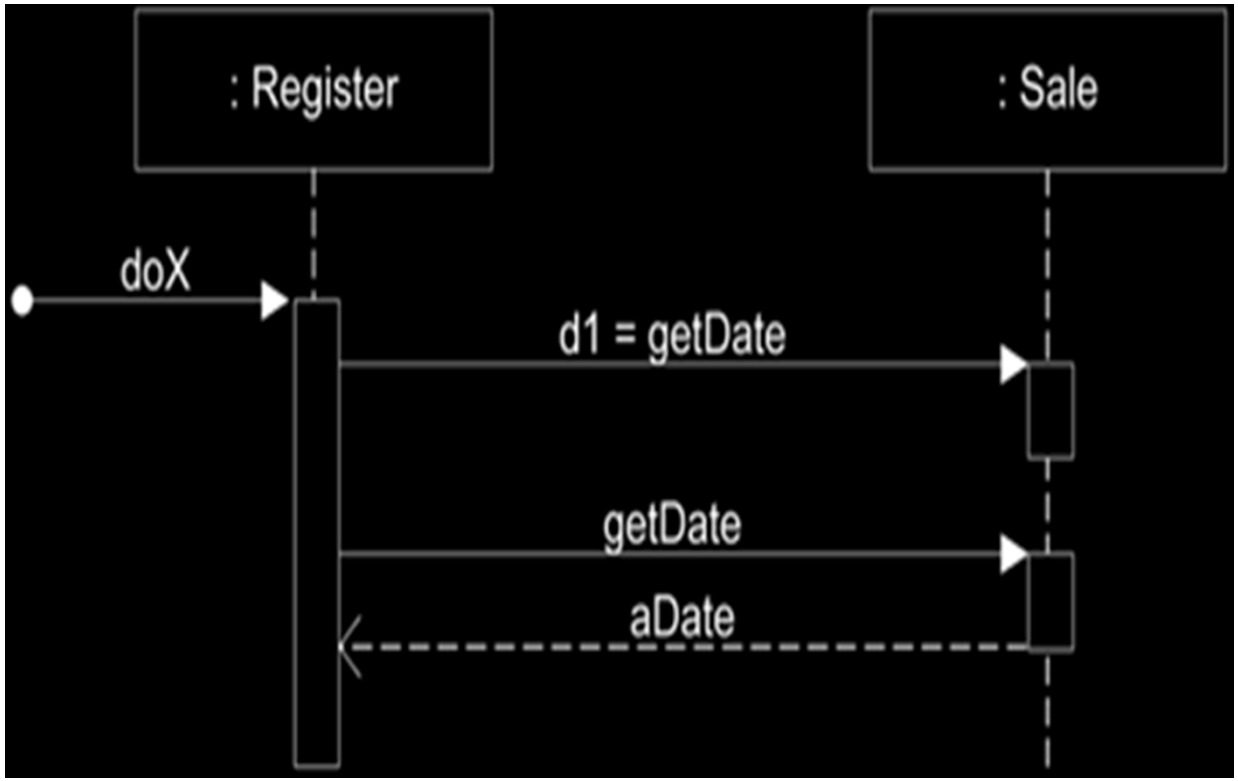
   returnVar = message(parameter).

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

- Both are common in practice.

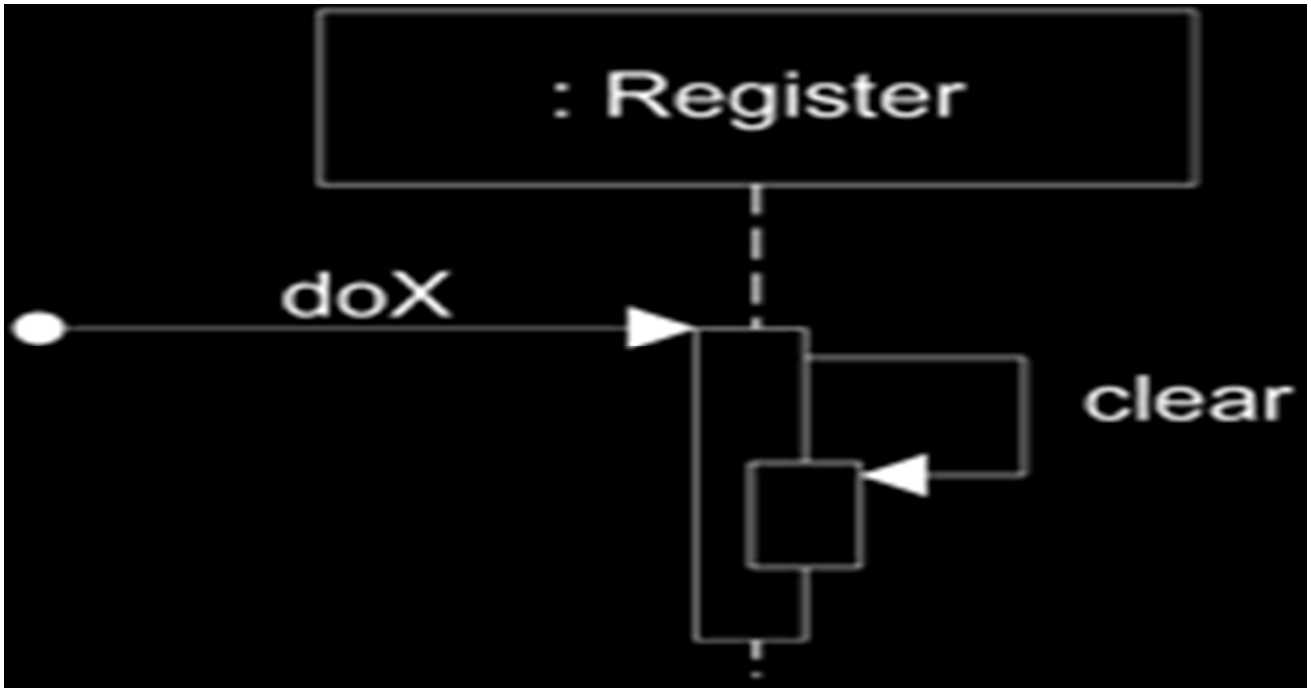
Basic Sequence diagram notation

Illustrating Reply or Returns



Basic Sequence diagram notation

Messages to "self" or "this"



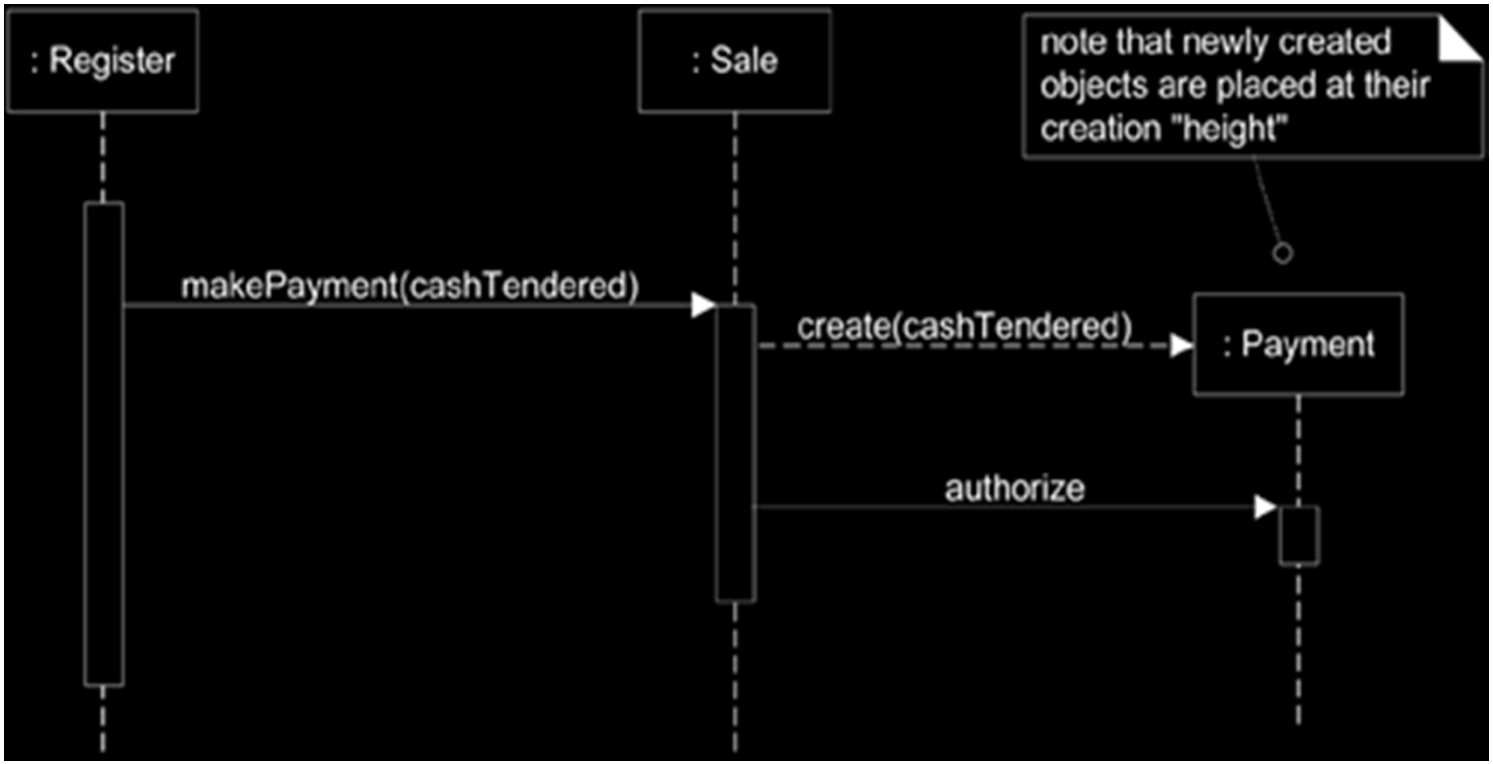
Basic Sequence diagram notation

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".

Basic Sequence diagram notation

Creation of Instances



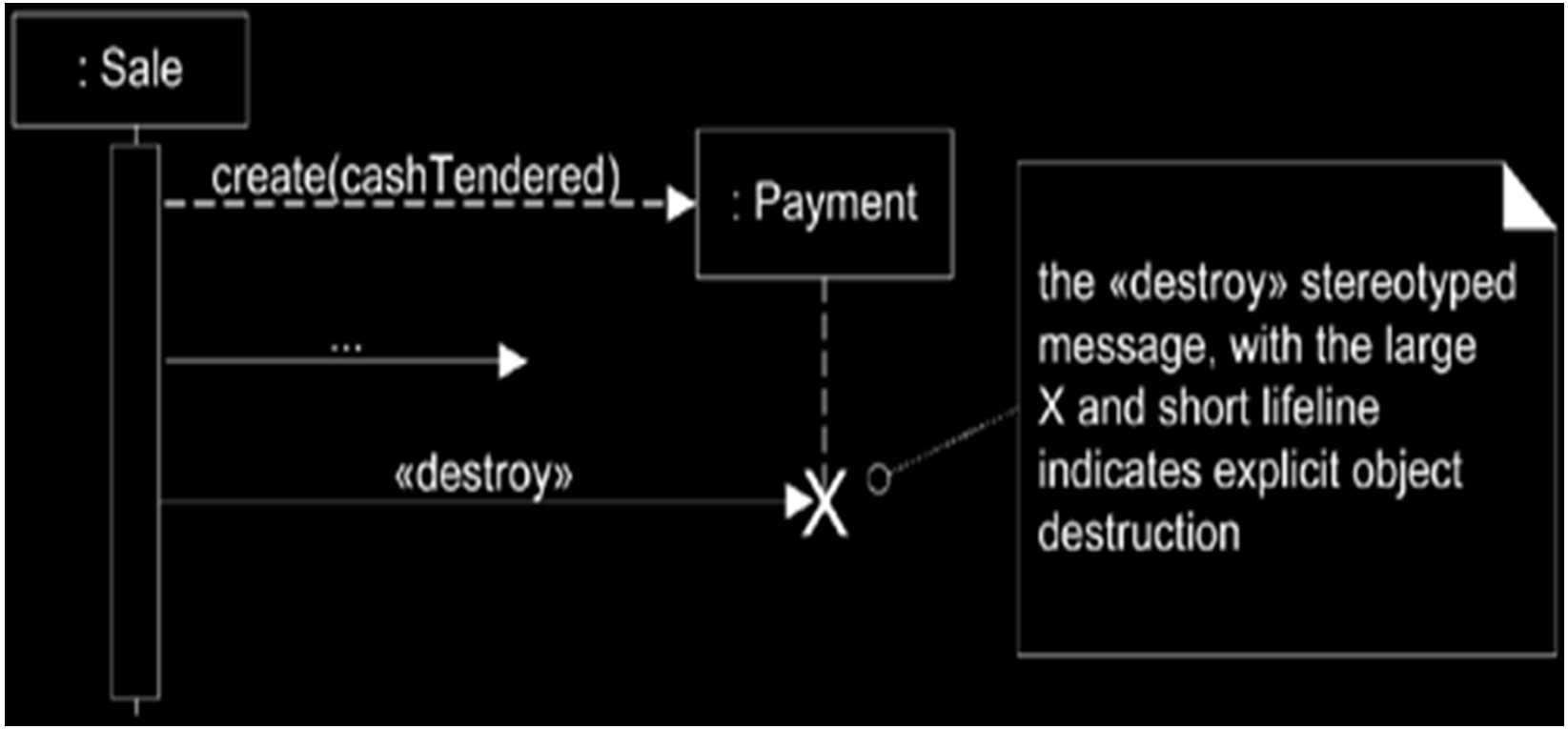
Basic Sequence diagram notation

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

Basic Sequence diagram notation

Object Lifelines and Object Destruction



Basic Sequence diagram notation

Diagram Frames in *UML* Sequence Diagrams

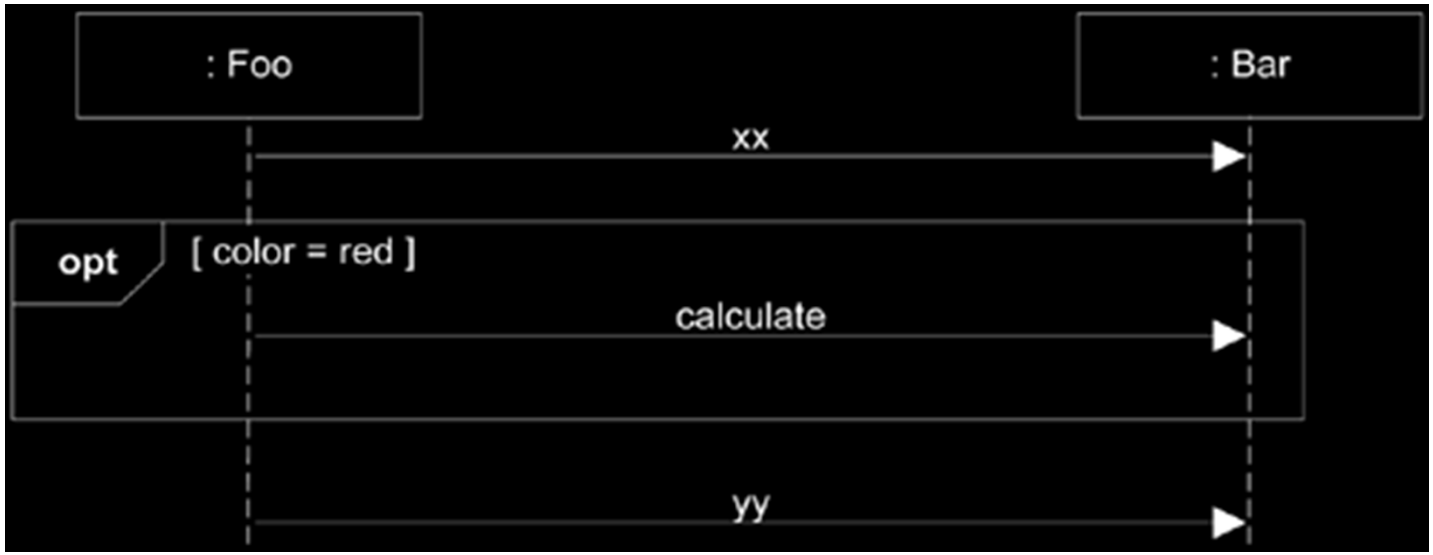
- To support conditional and looping constructs (among many other things), the *UML* uses **frames**.
- Frames are regions or fragments of the diagrams; they have an operator or label (such as *loop*) and a guard

Basic Sequence diagram notation – 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.

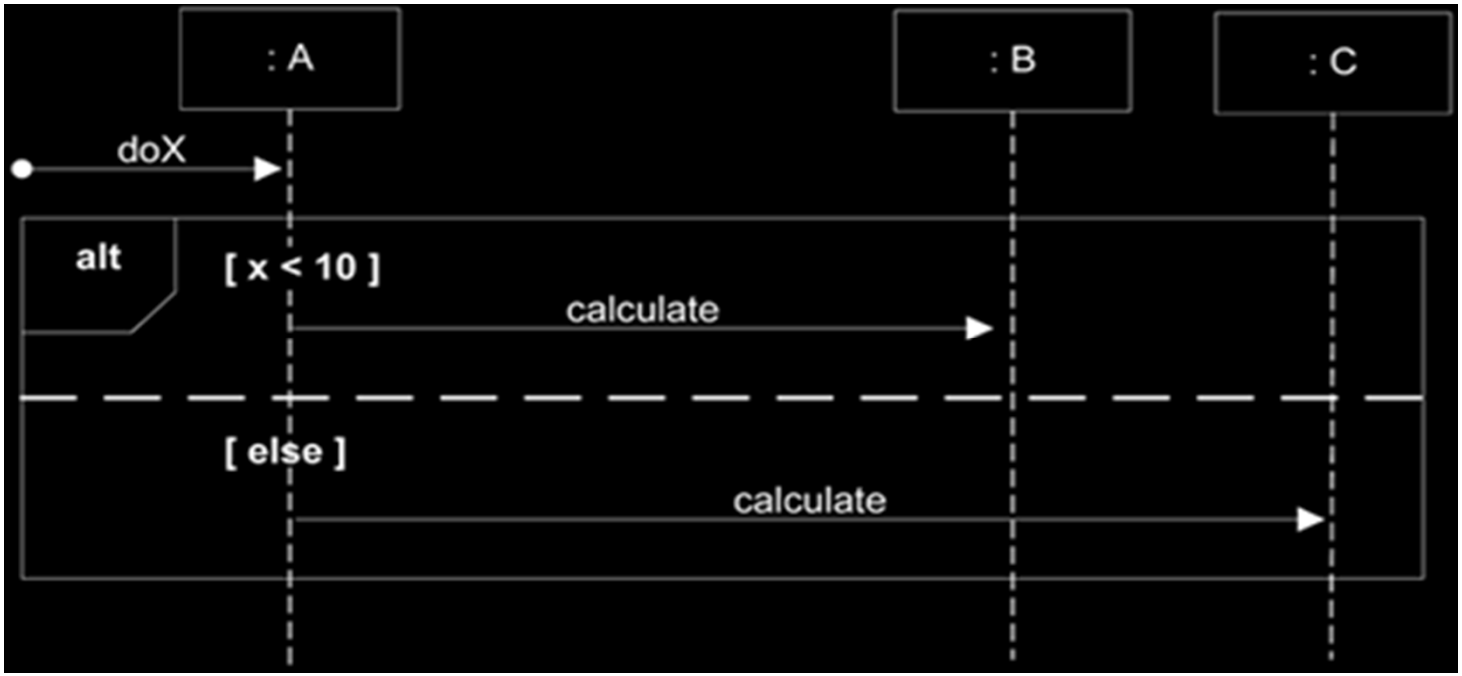
Basic Sequence diagram notation – Conditional Messages

- An OPT frame is placed around one or more messages. Notice that the guard is placed *over* the related lifeline



Basic Sequence diagram notation – Mutually Exclusive Conditional Messages

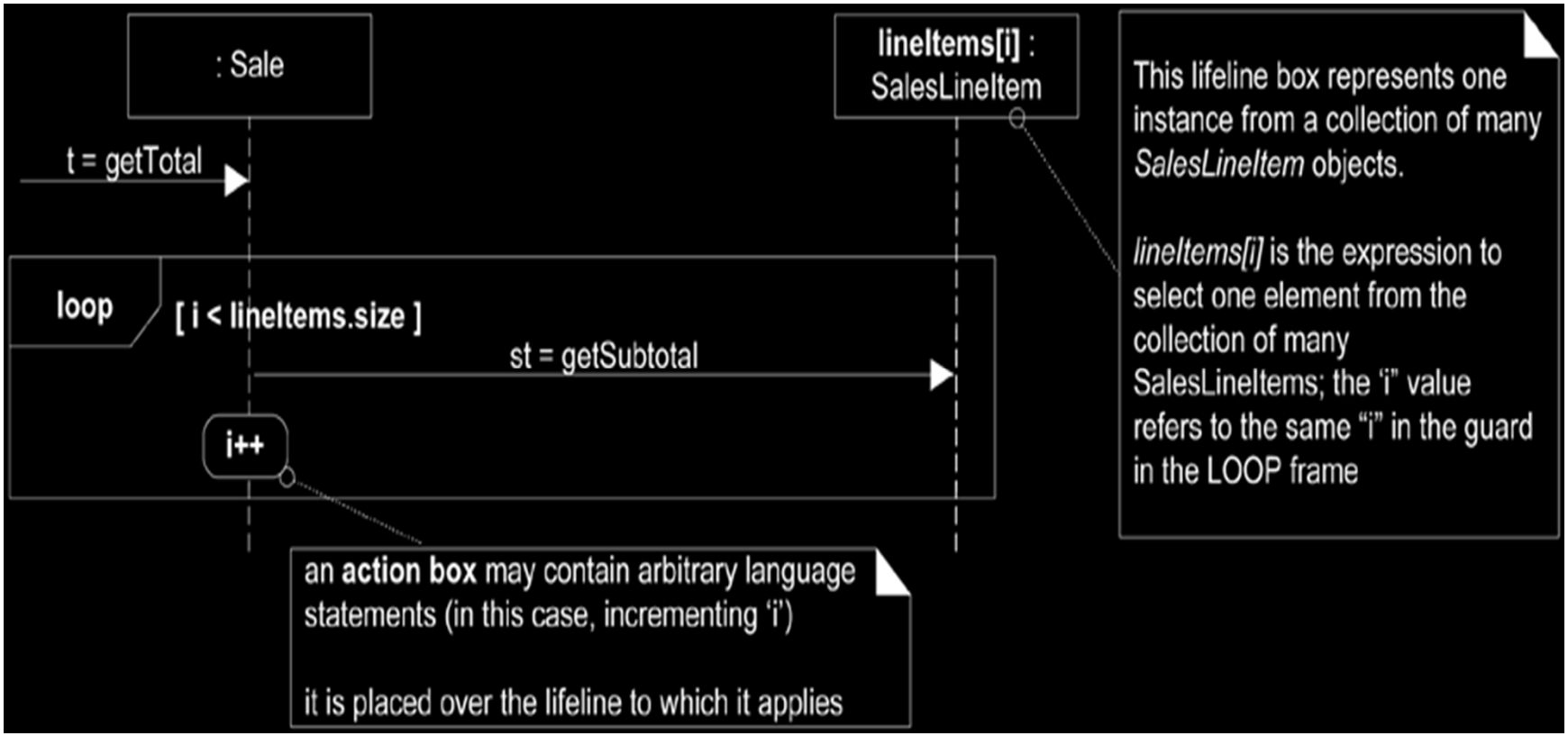
- An ALT frame is placed around the mutually exclusive alternatives



Basic Sequence diagram notation – Iteration over a collection

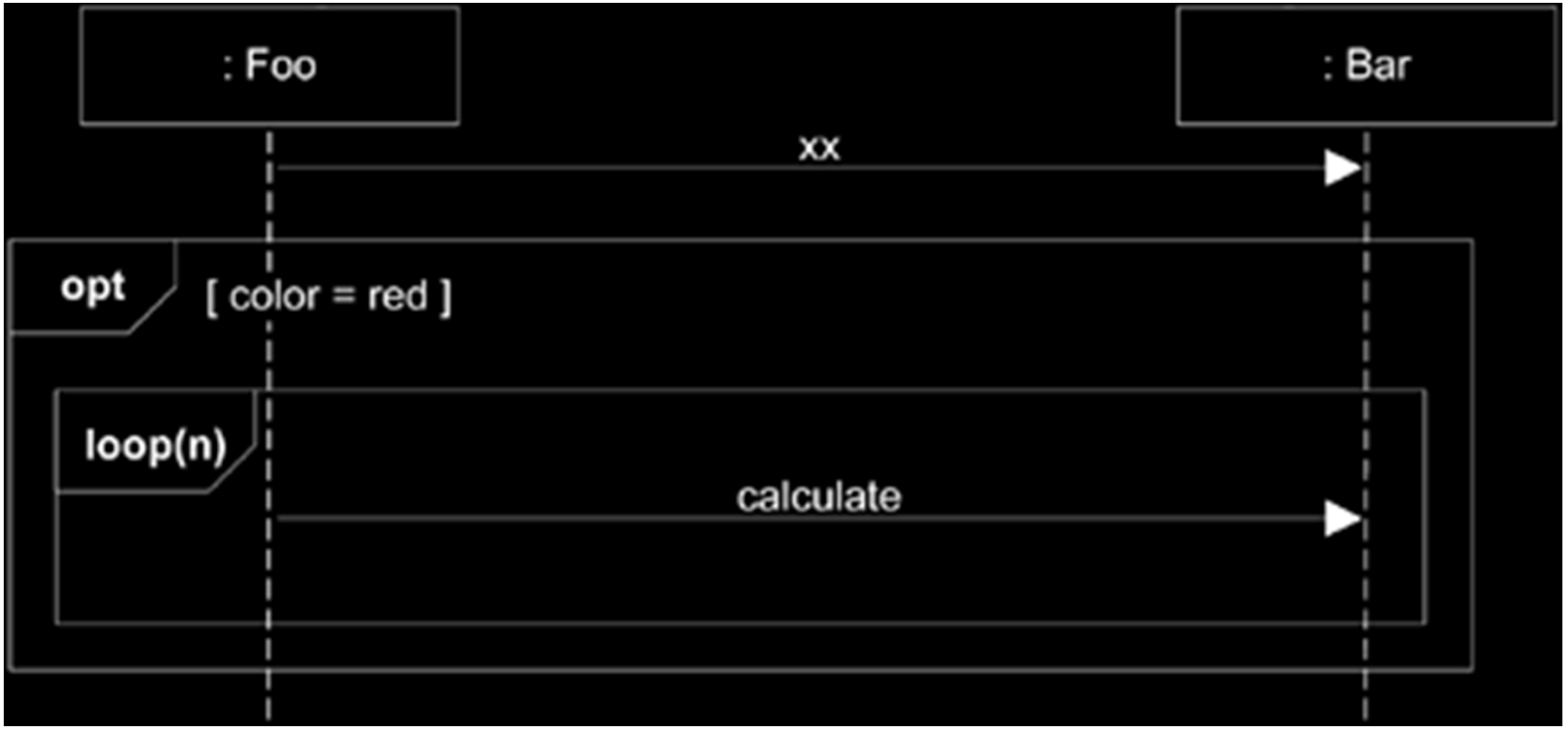
- To iterate over all members of a collection (such as a list or map), sending the same message to each, UML provides two alternatives:

Basic Sequence diagram notation – Iteration over a collection



Basic Sequence diagram notation – Iteration over a collection

Basic Sequence diagram notation – Nesting of Frames



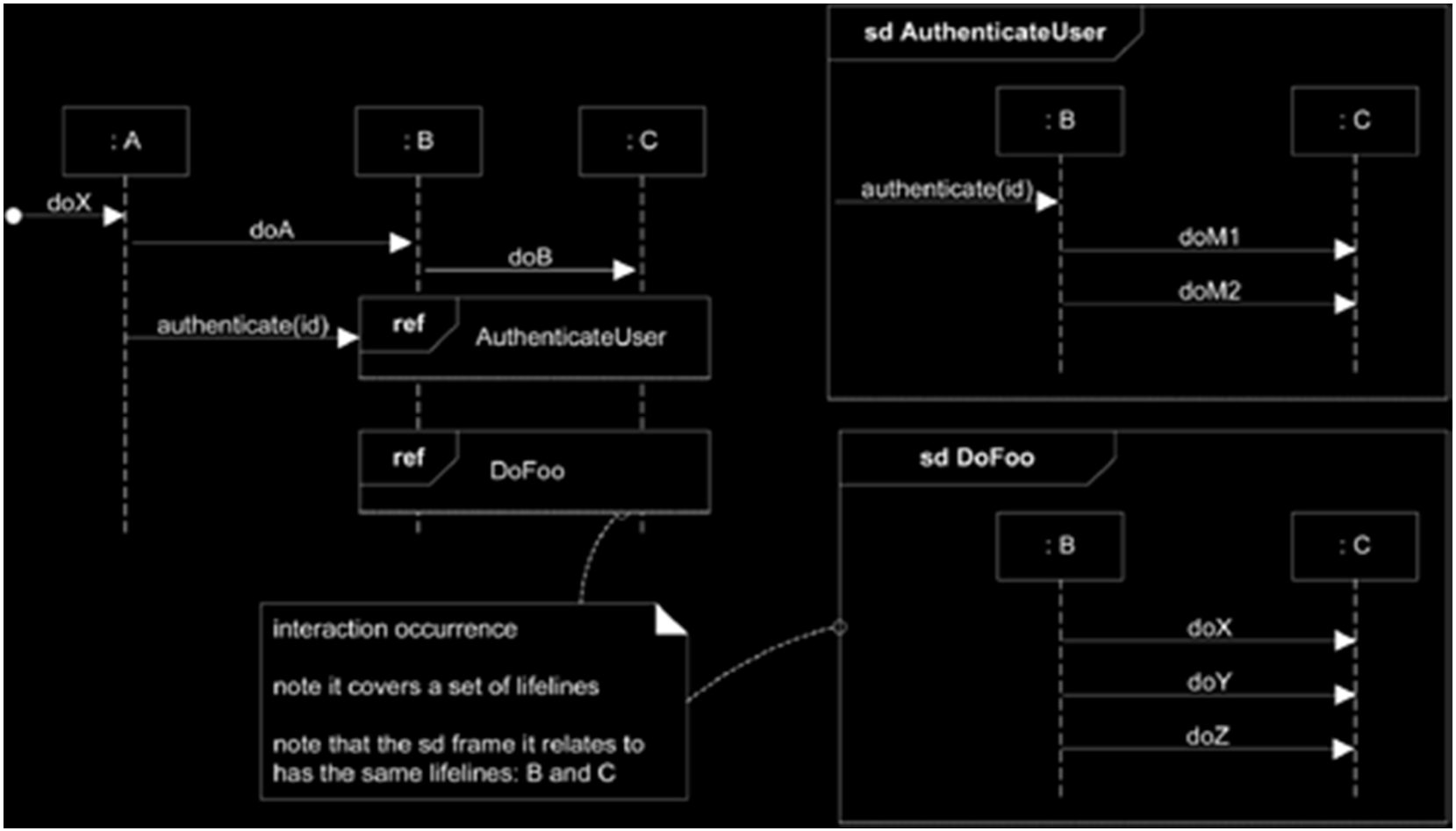
How to relate interaction diagrams

- An **interaction occurrence** (also called an **interaction use**) is a reference to an interaction within another interaction.
- It is useful, for example, when you want to simplify a diagram and factor out a portion into another diagram, or there is a reusable interaction occurrence.

They are created with two related frames:

- a frame around an entire sequence diagram , labelled with the tag **sd** and a name, such as *AuthenticateUser*
- a frame tagged **ref**, called a **reference**, that refers to another named sequence diagram; it is the actual interaction occurrence

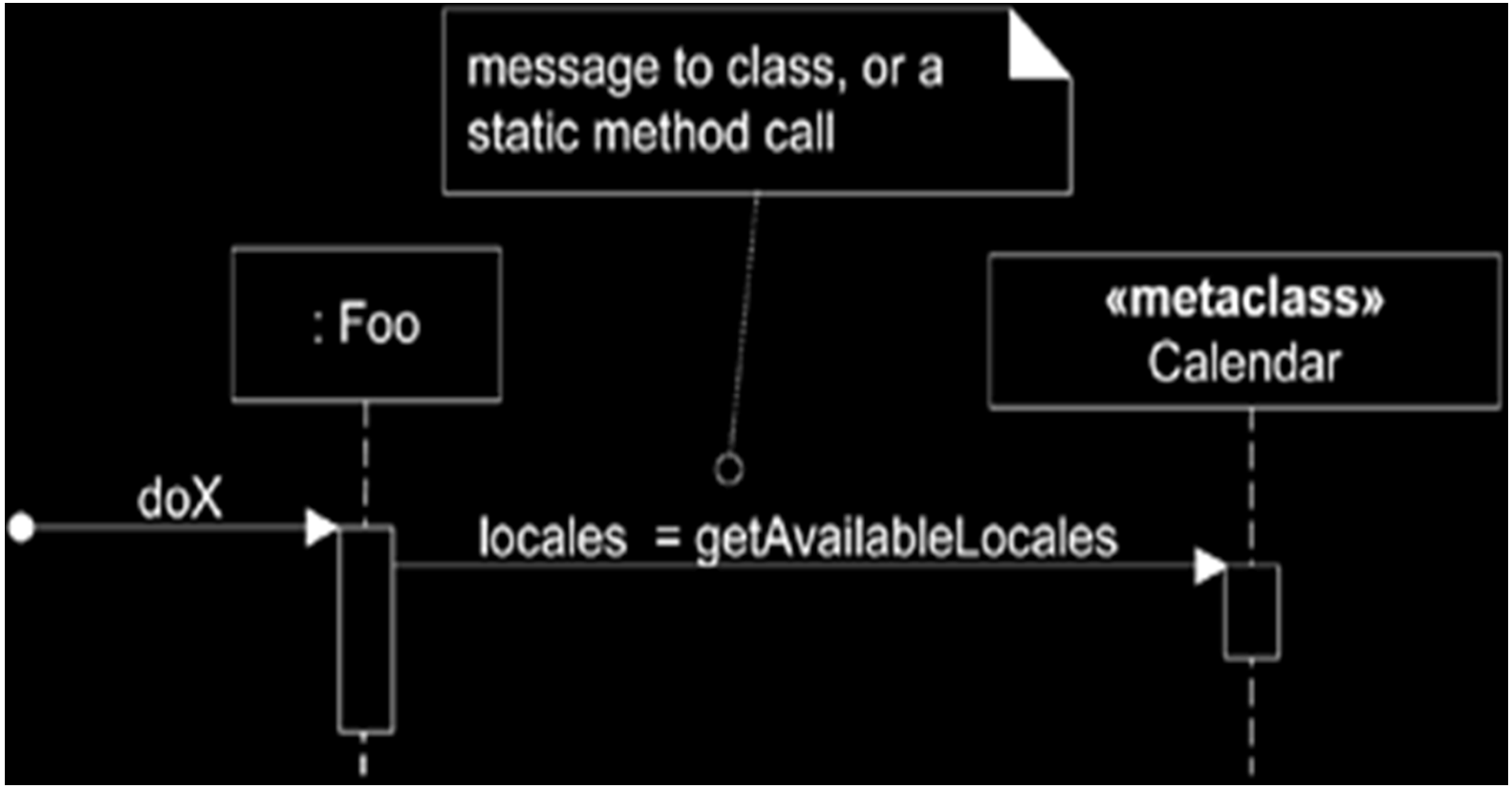
How to relate interaction diagrams



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**
- For example, in Java and Smalltalk, all classes are conceptually or literally *instances* of class *Class*; in .NET classes are instances of class *Type*. The classes *Class* and *Type* are metaclasses, which means their instances are themselves classes. A specific class, such as class *Calendar*, is itself an instance of class *Class*. Thus, class *Calendar* is an instance of a meta class.

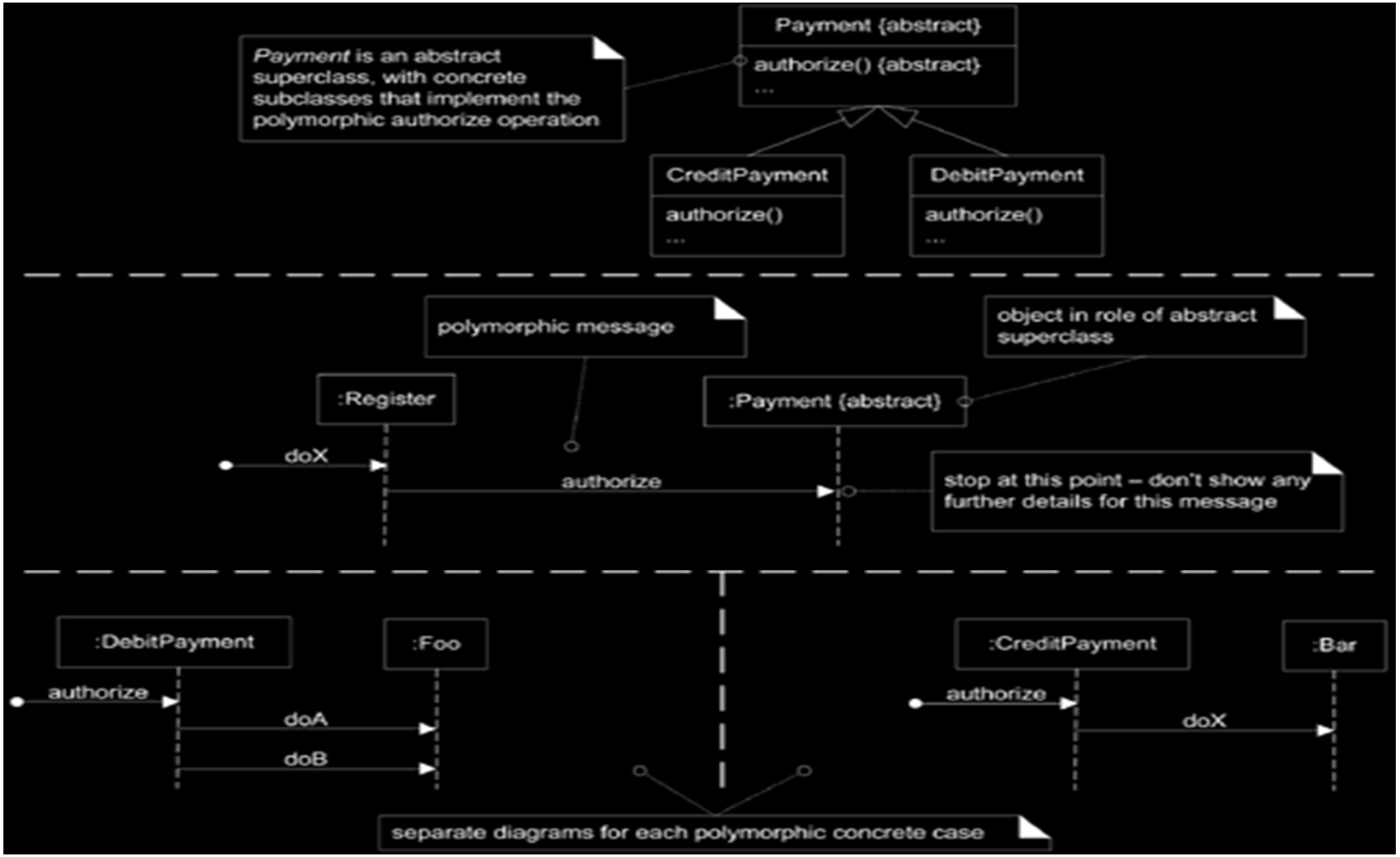
Messages to Classes to Invoke Static (or Class) Methods



Polymorphic Messages

- Polymorphism is fundamental to *OO* design.
- 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

Polymorphic Messages

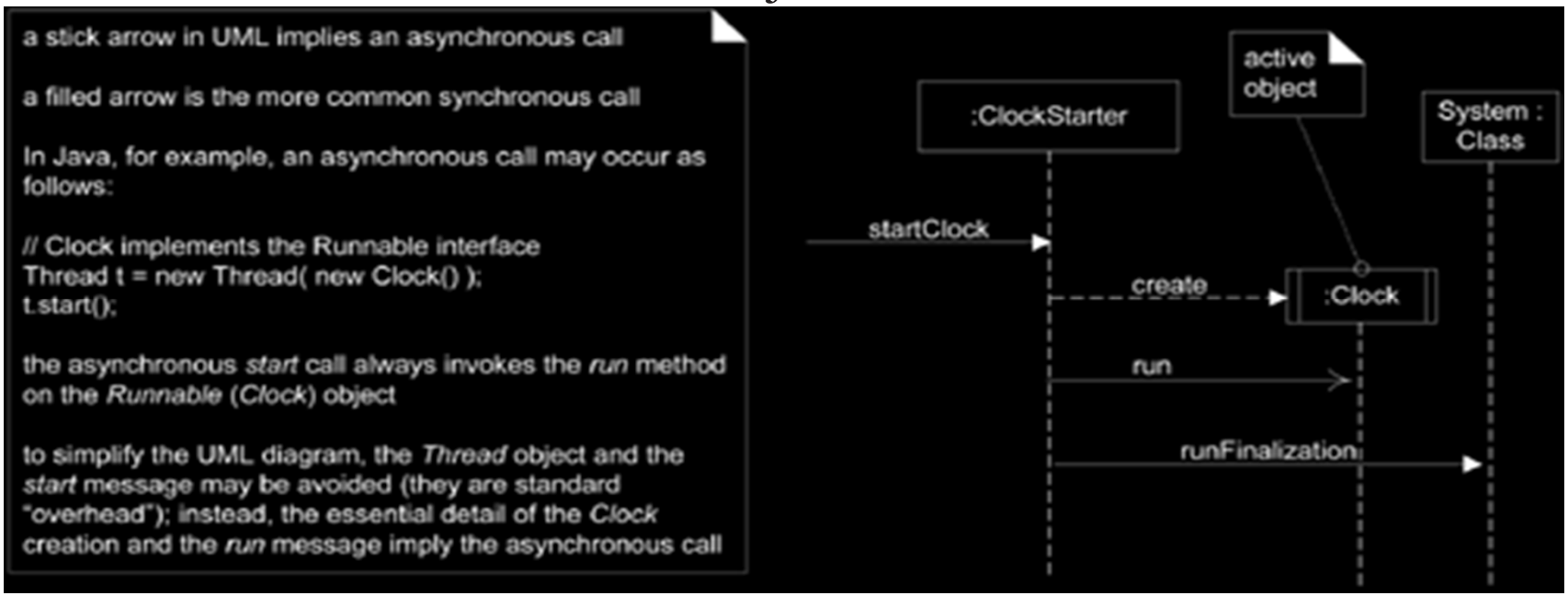


Asynchronous and Synchronous calls

- 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

Asynchronous and Synchronous calls

- An object such as the *Clock* is also known as an active object—each instance runs on and controls its own thread of execution.
- In the *UML*, it may be shown with double vertical lines on the left and right sides of the lifeline box. The same notation is used for an active class whose instances are active objects.

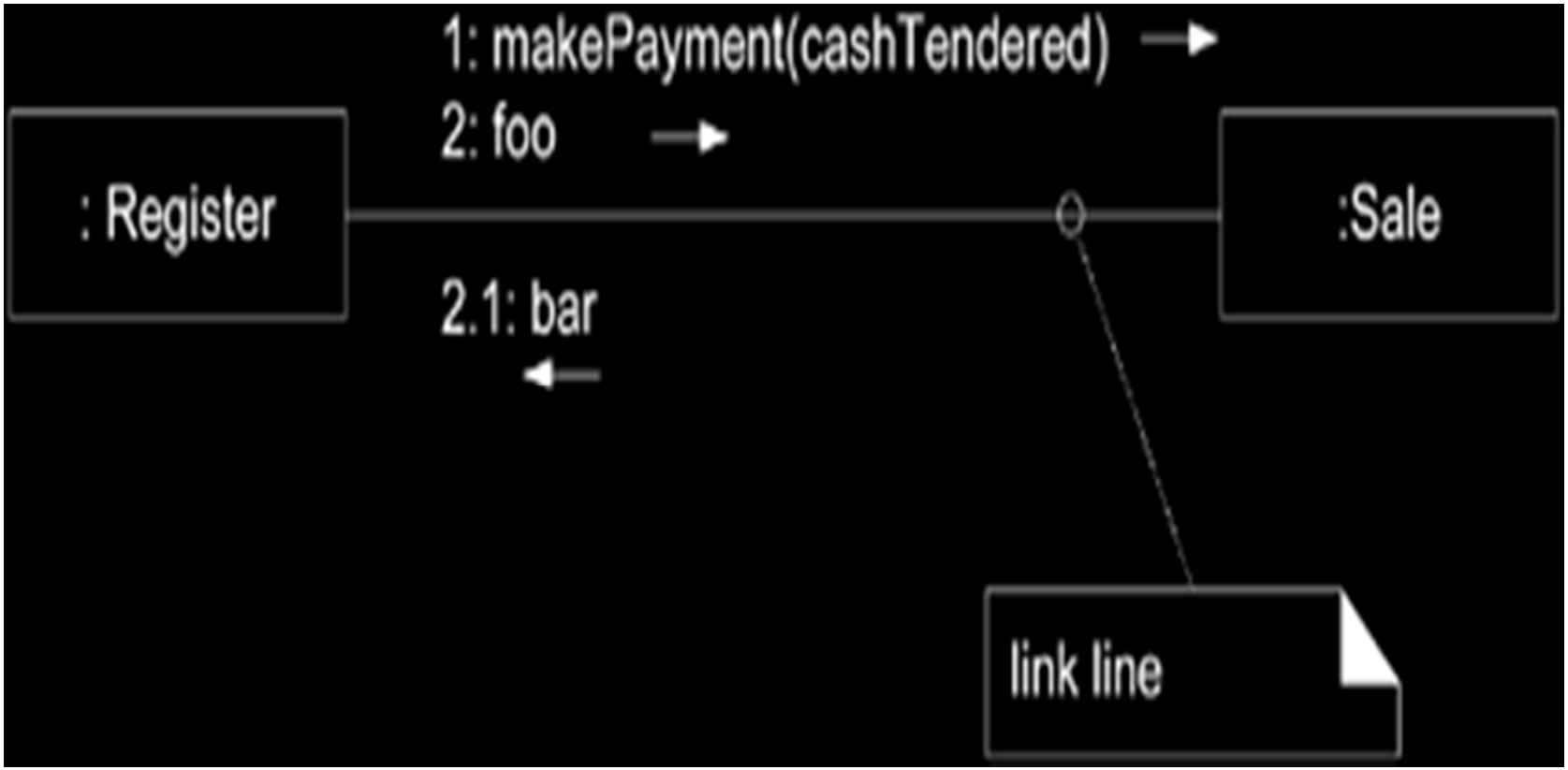


Basic Communication diagram notation

Links

- A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible
- More formally, a link is an instance of an association. For example, there is a link—or path of navigation—from a *Register* to a *Sale*, along which messages may flow, such as the *makePayment* message.

Basic Communication diagram notation

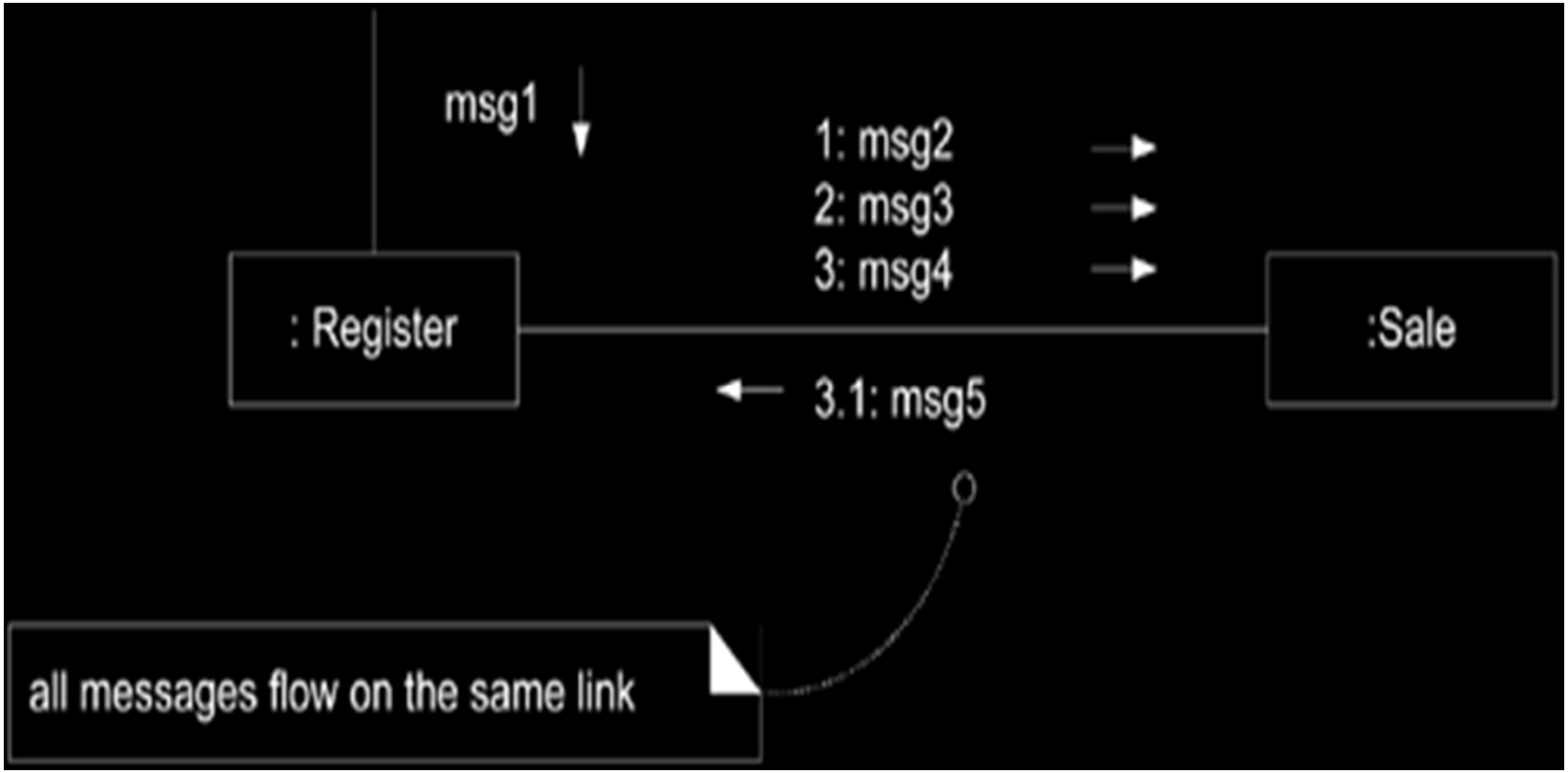


Basic Communication diagram notation

Messages

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

Basic Communication diagram notation



Basic Communication diagram notation

Messages to "self" or "this"

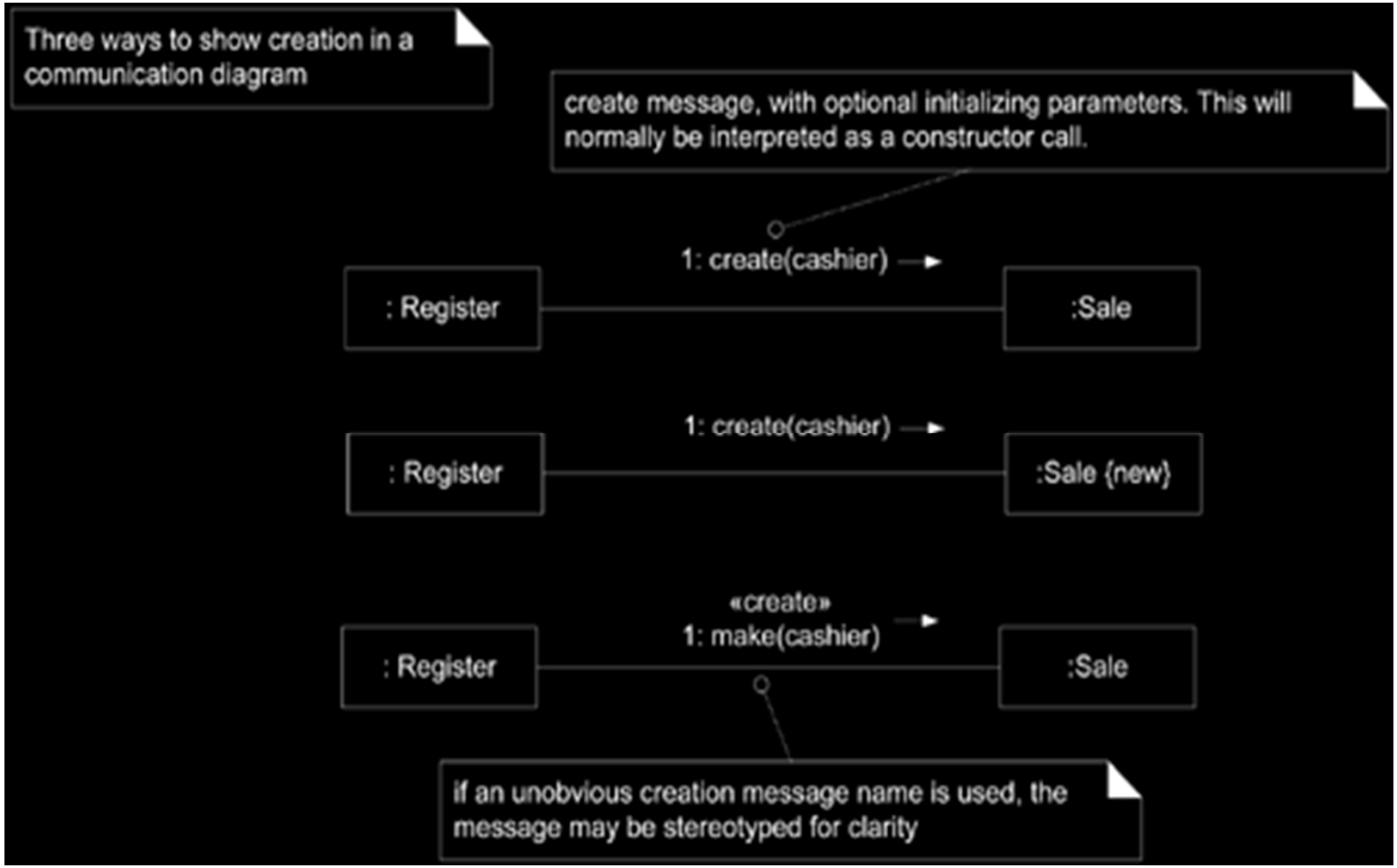
- A message can be sent from an object to itself. This is illustrated by a link to itself, with messages flowing along the link.

Basic Communication diagram notation

Creation of Instances

- Any message can be used to create an instance, but the convention in the *UML* is to use a message named *create* for this purpose (some use *new*).
- The message may be annotated with a UML stereotype, like so: «*create*». The *create* message may include parameters, indicating the passing of initial values. This indicates, for example, a constructor call with parameters in Java.
- Furthermore, the UML tagged value *{new}* may optionally be added to the lifeline box to highlight the creation. Tagged values are a flexible extension mechanism in the *UML* to add semantically meaningful information to a *UML* element.

Basic Communication diagram notation

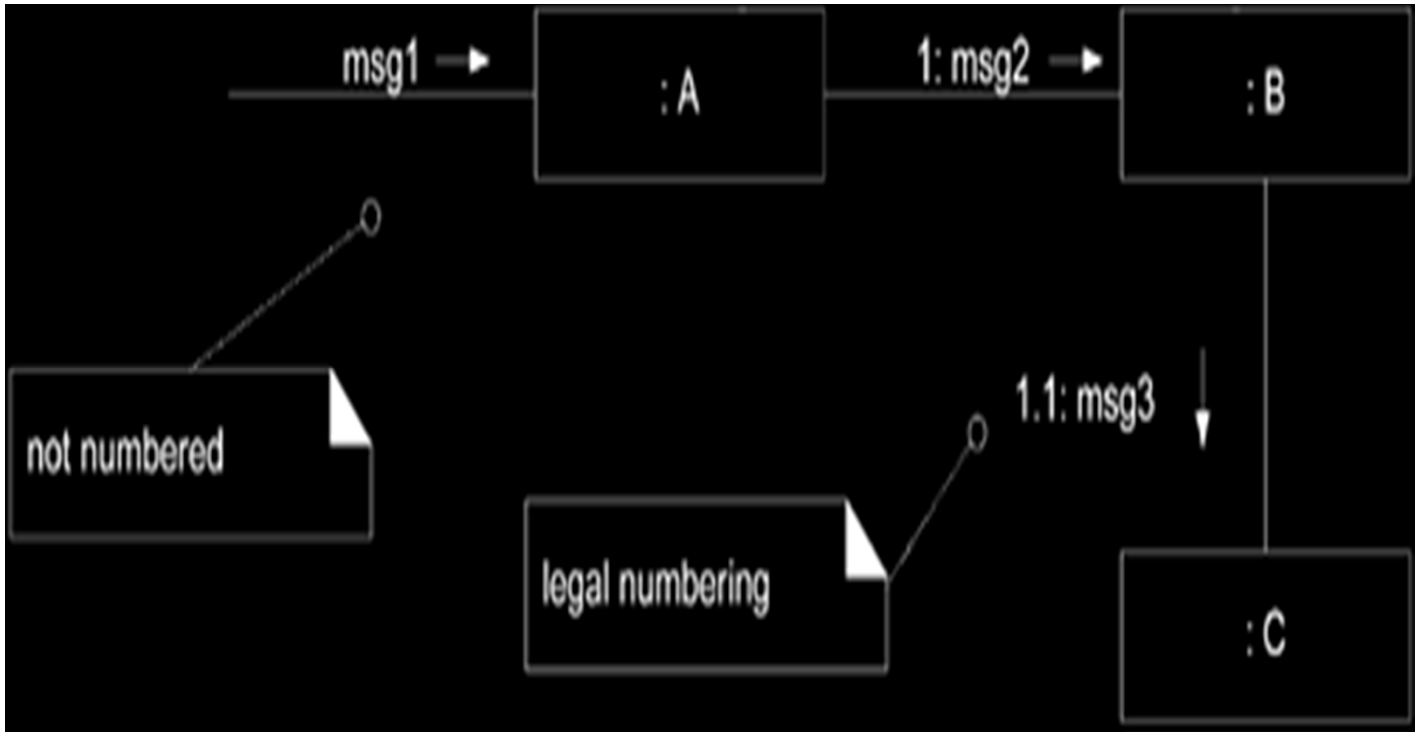


Basic Communication diagram notation

Message Number Sequencing

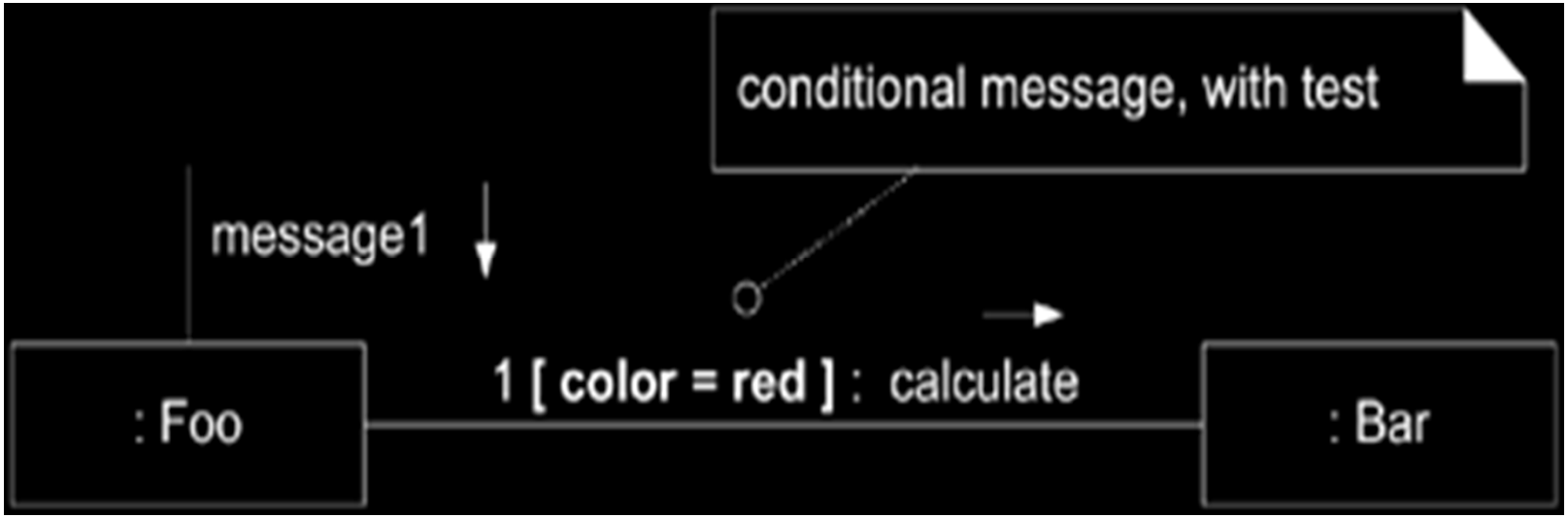
- The order of messages is illustrated with sequence numbers
- The numbering scheme is:
  1. The first message is not numbered. Thus, *msg1* is unnumbered.
  2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them. You denote nesting by prepending the incoming message number to the outgoing message number.

Basic Communication diagram notation



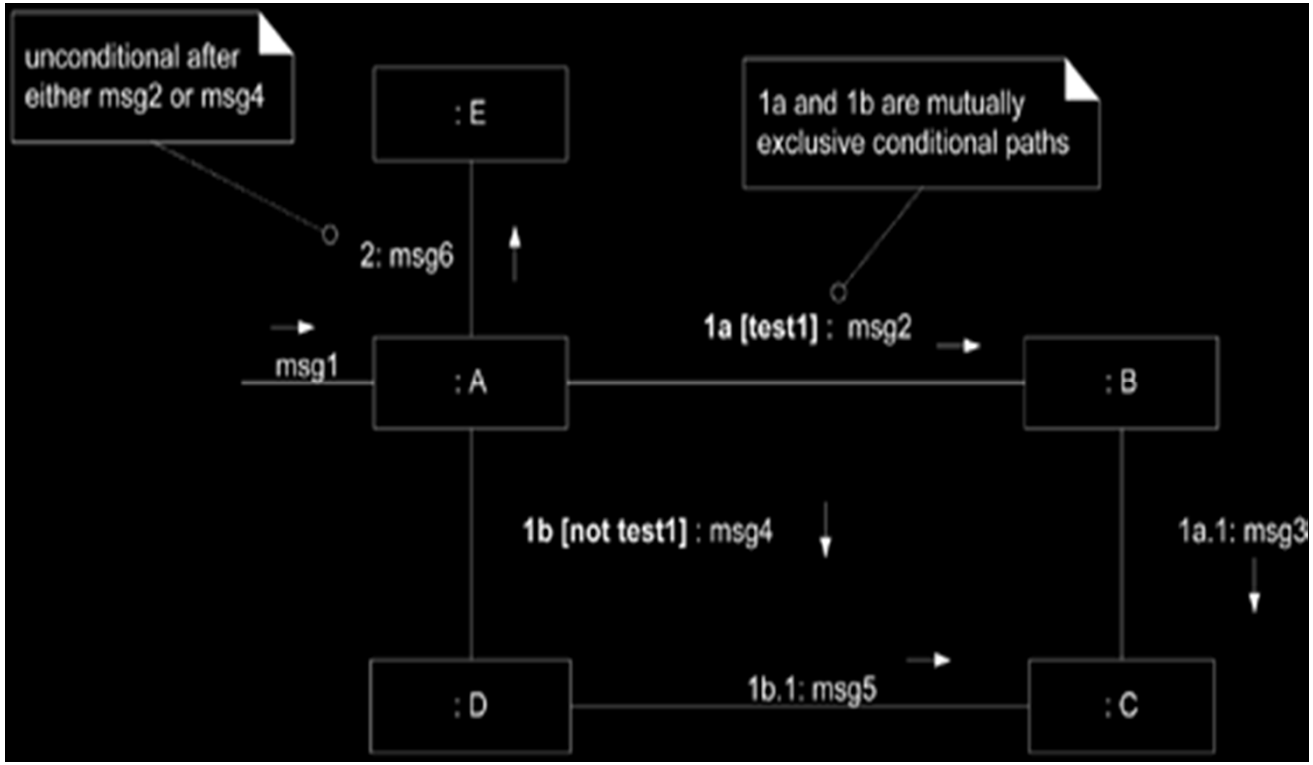
Basic Communication diagram notation

Conditional Messages



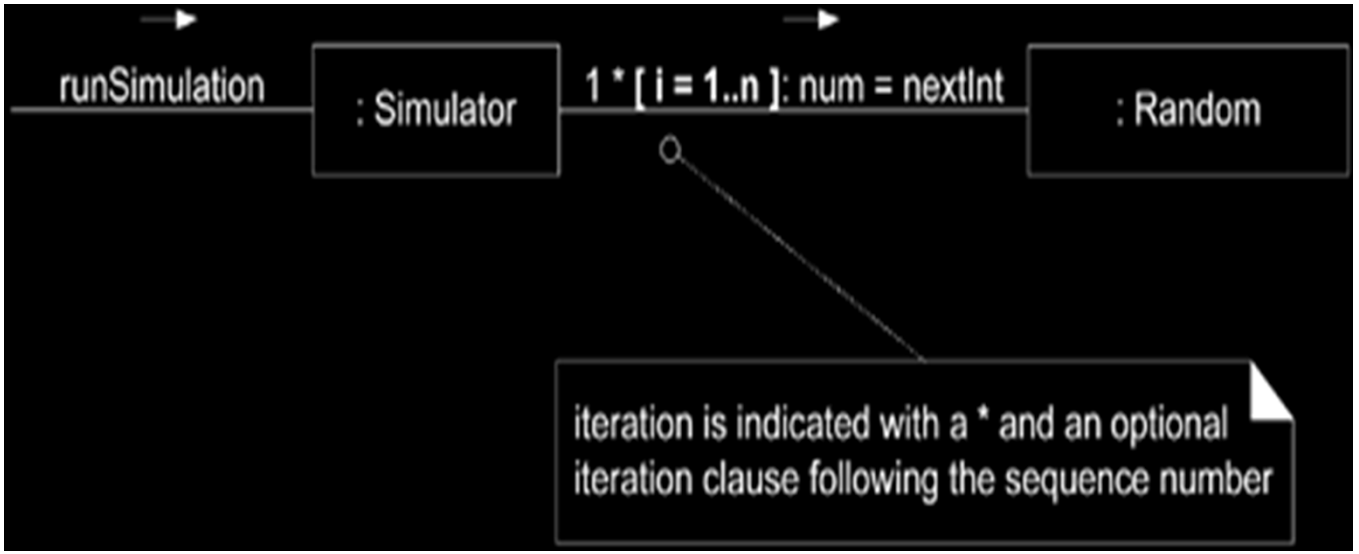
Basic Communication diagram notation

Mutually Exclusive Conditional Paths



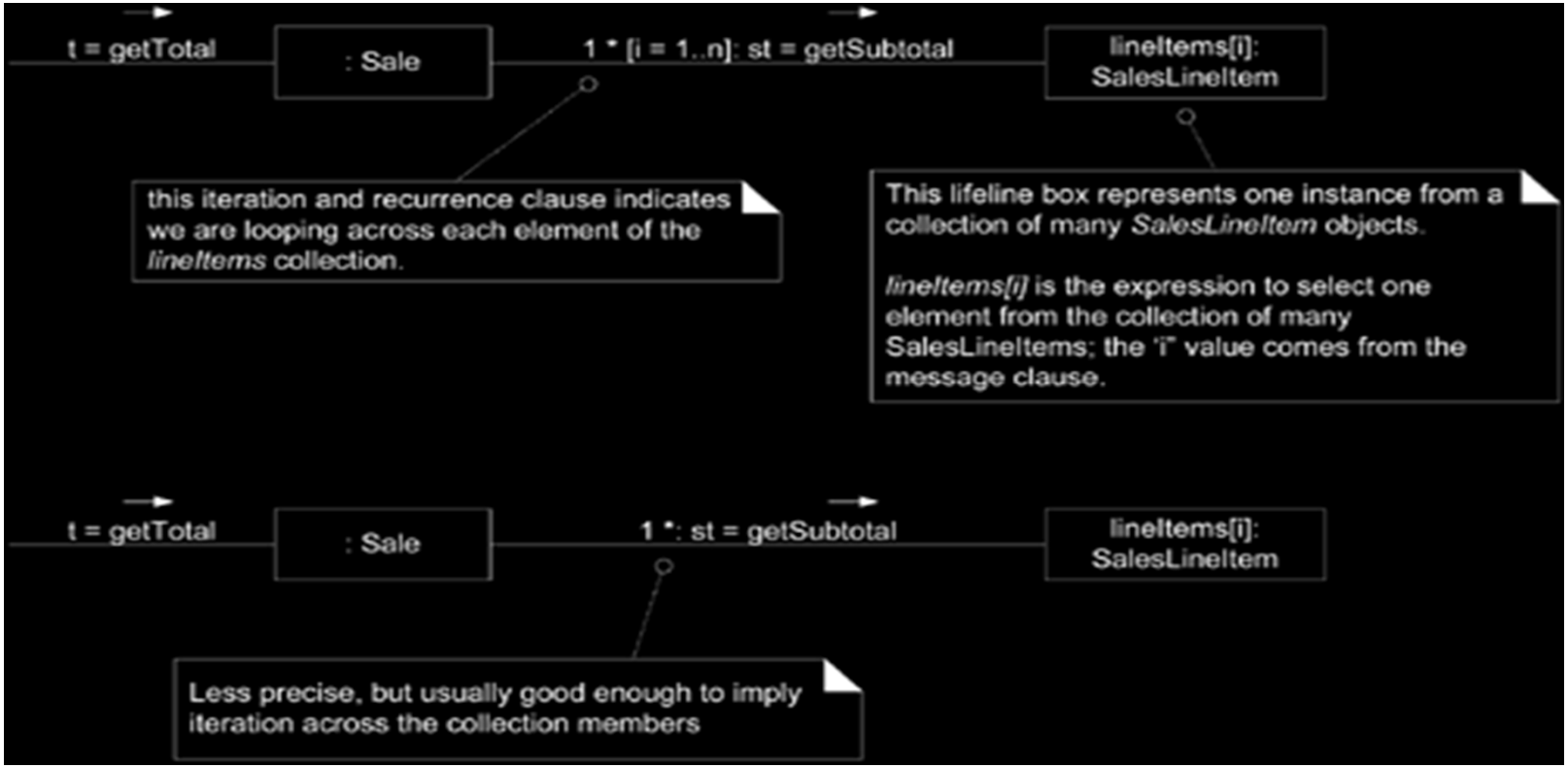
Basic Communication diagram notation

Iteration or Looping



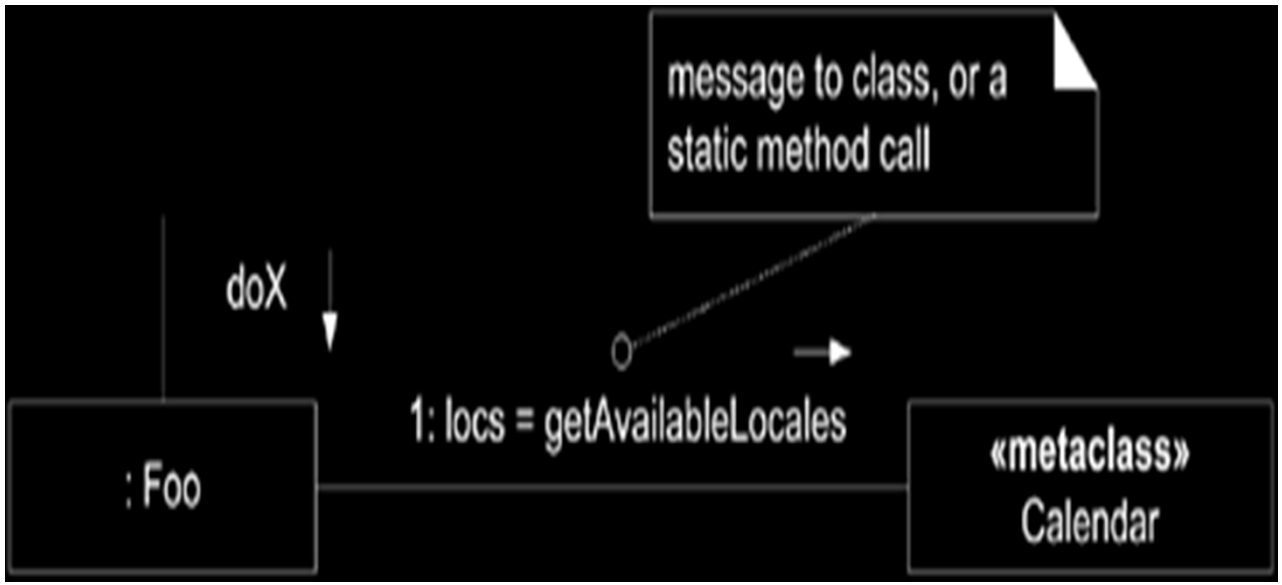
Basic Communication diagram notation

Iteration Over a Collection



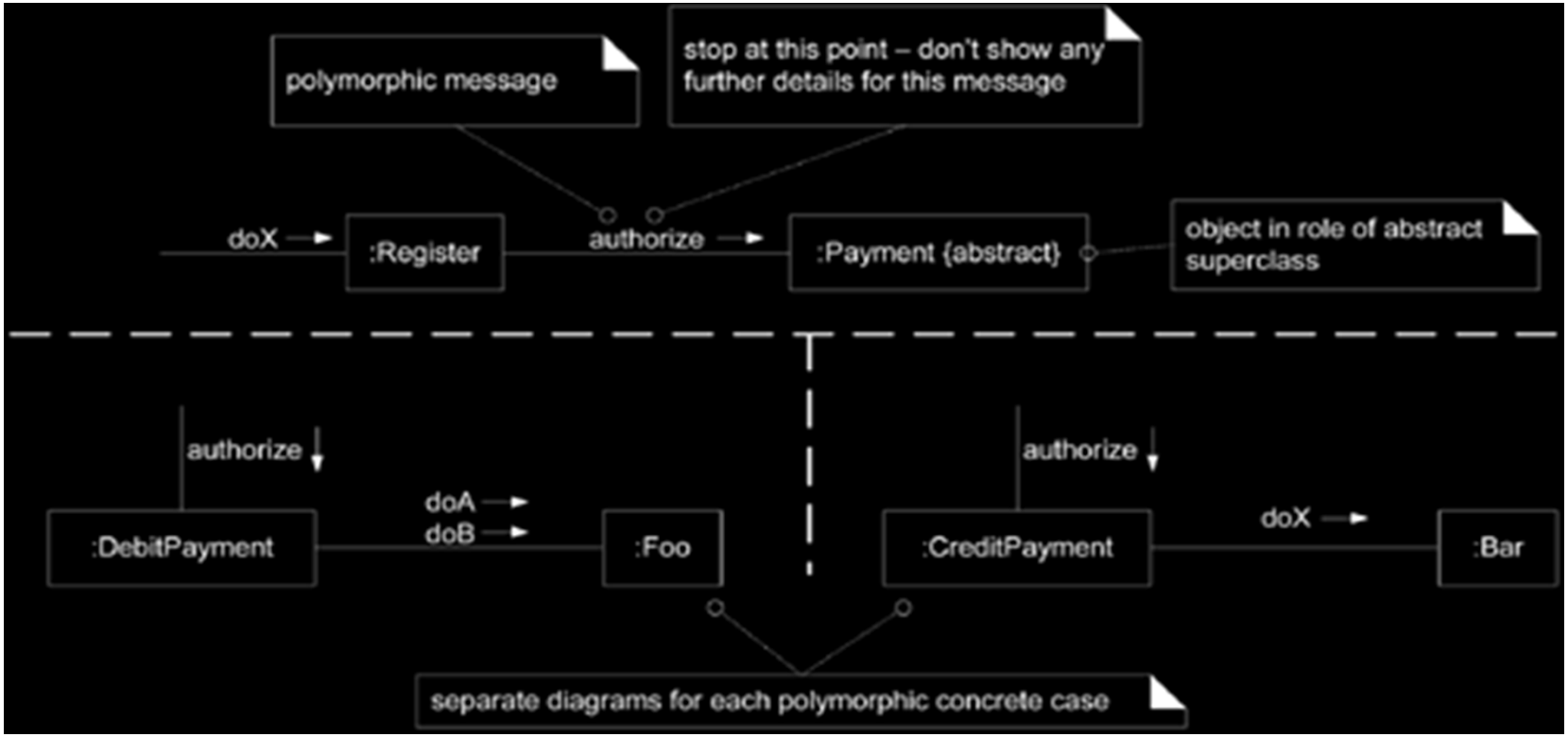
Basic Communication diagram notation

Messages to a Classes to Invoke Static (Class) Methods



Basic Communication diagram notation

Polymorphic Messages and Cases



Basic Communication diagram notation

Asynchronous and Synchronous Calls