9. Interaction Diagrams – depicting the dynamic behaviour of the system

A series of diagrams can be used to describe the *dynamic behavior* of an object-oriented system. This is done in terms of a set of messages exchanged among a set of objects within a context to accomplish a purpose. This is often used to model the way a use case is realized through a sequence of messages between objects.

The purpose of Interaction diagrams is to:

- Model interactions between objects
- Assist in understanding how a system (a use case) actually works
- Verify that a use case description can be supported by the existing classes
- Identify responsibilities/operations and assign them to classes

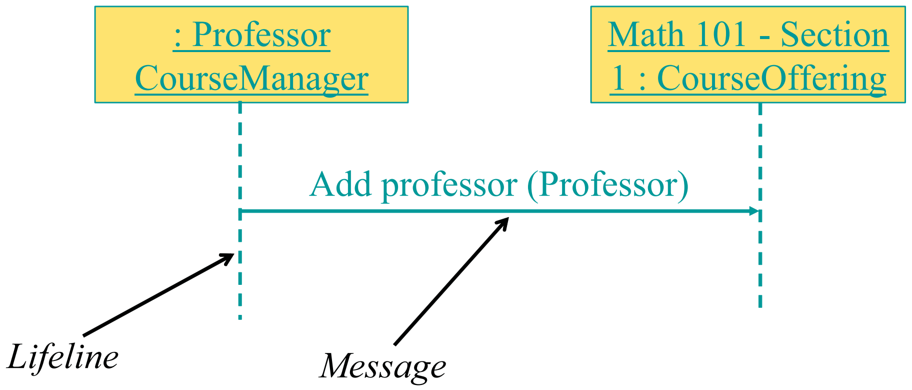
UML provides two different mechanisms to document the dynamic behaviour of the system. These are sequence diagrams which provide a time-based view and *Collaboration Diagrams* which provide an organization-based view of the system's dynamics.

9.1. The Sequence Diagram

Let us first look at Sequence Diagrams. These diagrams illustrate how objects interacts with each other and emphasize time ordering of messages by showing object interactions arranged in time sequence. These can be used to model simple sequential flow, branching, iteration, recursion and concurrency. The focus of sequence diagrams is on objects (and classes) and message exchanges among them to carry out the scenarios functionality. The objects are organized in a horizontal line and the events in a vertical time line.

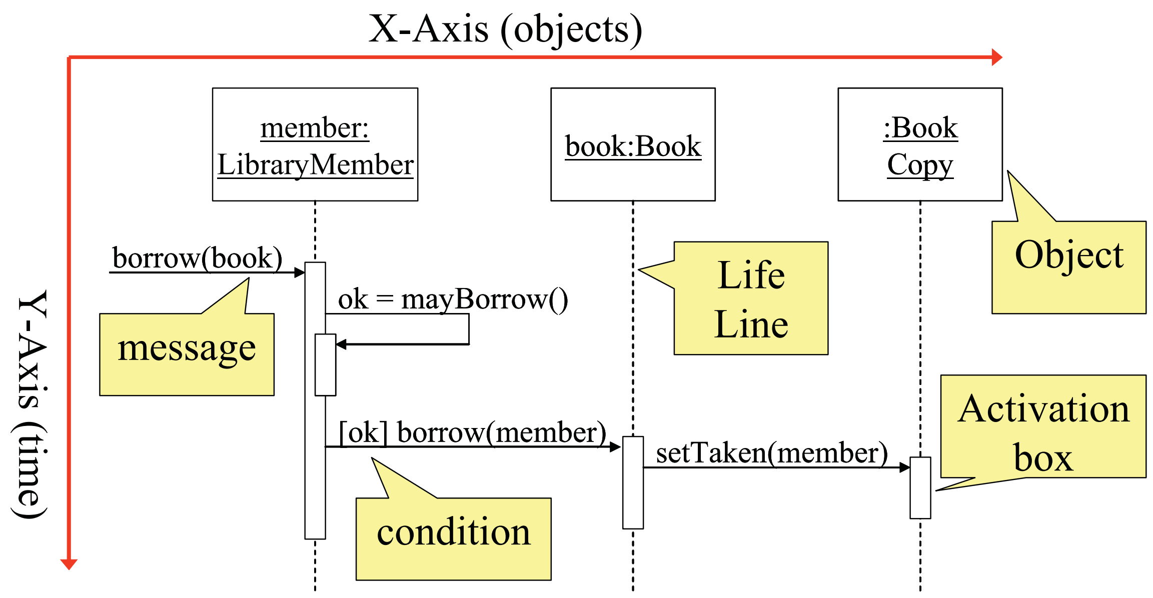
9.1.1. The Notation

Following diagram illustrates the notation used for drawing sequence diagrams.


: Professor CourseManager
Math 101 - Section 1 : CourseOffering
Add professor (Professor)
Lifeline
Message

The boxes denote objects (or classes), the solid lines depict messages being sent from one object to the other in the direction of the arrow, and the dotted lines are called life-lines of objects. The life line represents the object's life during interaction. We will discuss this in more detail later.

These concepts are further elaborated with the help of the following sequence diagram.



As shown above, in a sequence diagram, objects (and classes) are arranged on the X-Axis (horizontally) while time is shown on the Y-Axis (vertically). The boxes on the life-line are called activation boxes and show for how long a particular message will be active, from its start to finish. We can also show if a particular condition needs to occur before a message is invoked simply by putting the condition in a box before the message. For example, object *member:LibraryMember* sends a message to object *book:book* if the value of *ok* is true.

The syntax used for naming objects in a sequence diagram is as follows:

- syntax: [instanceName][:className]
- Name classes consistently with your class diagram (same classes).
- Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.

An interaction between two objects is performed as a message sent from one object to another. It is most often implemented by a simple operation call. It can however be an actual message sent through some communication mechanism, either over the network or internally on a computer.

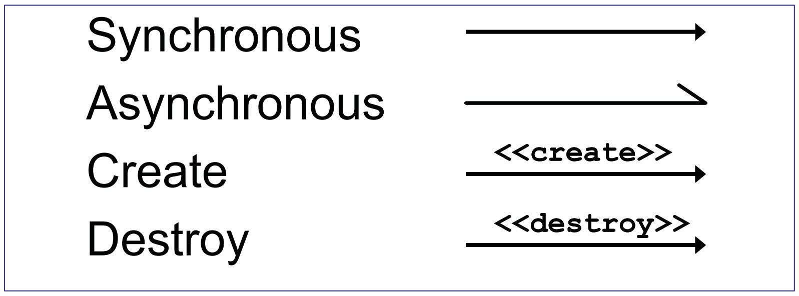
If object obj_1 sends a message to another object obj_2 an association must exist between those two objects. There has to be some kind of structural dependency. It can either be that obj_2 is in the global scope of obj_1, or obj_2 is in the local scope of obj_1 (method argument), or obj_1 and obj_2 are the same object.

A message is represented by an arrow between the life lines of two objects. Self calls are also allowed. These are the messages that an object sends to itself. This notation allows self calls. In the above example, object *member:LibraryMember* sends itself the *mayBorrow* message. A message is labeled at minimum with the message name.
Arguments and control information (conditions, iteration) may also be included. It is preferred to use a brief textual description whenever an *actor* is the source or the target of a message.

The time required by the receiver object to process the message is denoted by an *activation-box.*

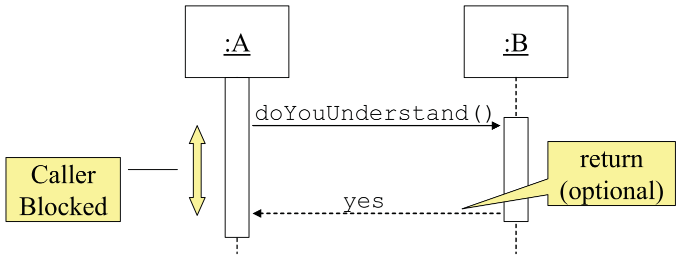
9.1.2. Message Types

Sequence diagrams can depict many different types of messages. These are: synchronous or simple, asynchronous, create, and destroy. The following diagram shows the notation and types of arrows used for these different message types.



9.1.2.1. Synchronous Messages

Synchronous messages are "call events" and are denoted by the full arrow. They represent nested flow of control which is typically implemented as an operation call. In case of a synchronous message, the caller waits for the called routine to complete its operation before moving forward. That is, the routine that handles the message is completed before the caller resumes execution. Return values can also be optionally indicated using a dashed arrow with a label indicating the return value. This concept is illustrated with the help of the following diagram.



While modeling synchronous messages, the following guidelines should be followed:

- Don't model a return value when it is obvious what is being returned, e.g. getTotal()
- Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
- Prefer modeling return values as part of a method invocation, e.g.

```
ok = isValid()
```

9.1.2.2. Asynchronous messages

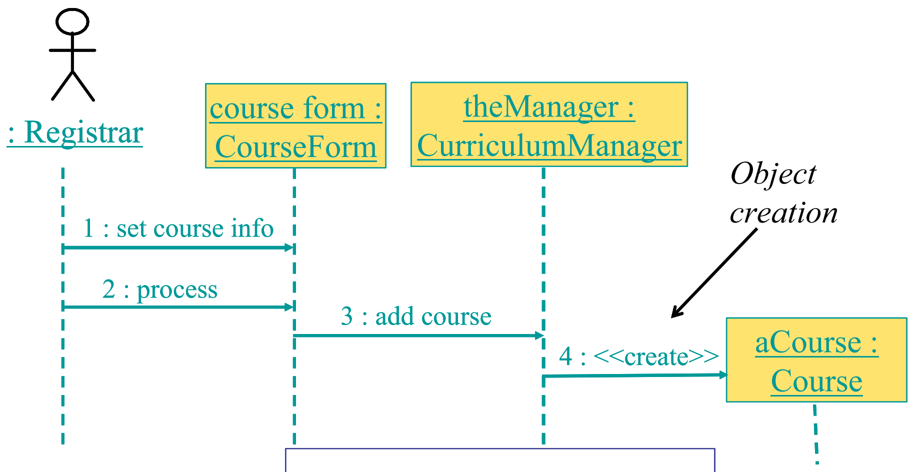
Asynchronous messages are "signals," denoted by a half arrow. They do not block the caller. That is, the caller does not wait for the called routine to finish its operation for continuing its own sequence of activities. This occurs in multi-threaded or multi-processing applications where different execution threads may pass information to one another by sending asynchronous messages to each other. Asynchronous messages typically perform the following actions:

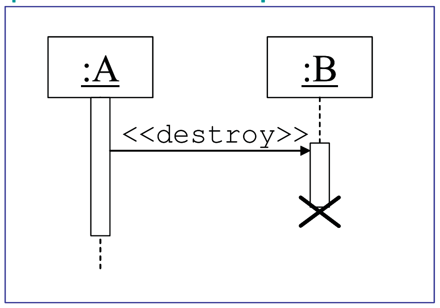
- Create a new thread
- Create a new object

- Communicate with a thread that is already running

9.1.2.3. Object Creation and Destruction

An object may create another object via a <<create>> message. Similarly an object may destroy another object via a <<destroy>> message. An object may also destroy itself. One should avoid modeling object destruction unless memory management is critical. The following diagrams show object creation and destruction. It is important to note the impact of these activities on respective life lines.





9.1.3. Sequence diagrams and logical complexity

It is important to judiciously use the sequence diagrams where they actually add value. The golden principle is to keep it small and simple. It is important to understand that the diagrams are meant to make things clear. Therefore, in order to keep them simple, special attentions should be paid to the conditional logic. If it is simple then there is no harm in adding it to the diagram. On the other hand if the logic is complex then we should draw separate diagrams like flow charts.