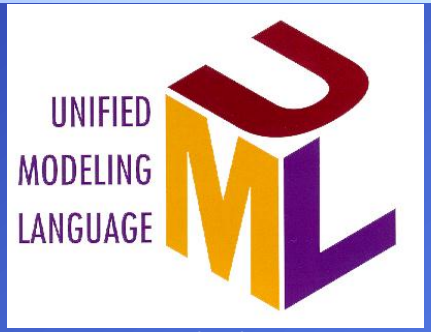
The Unified Modeling Language - UML



The UML is a visual language for specifying, constructing, and documenting the artifacts of a software.

The UML is not a method to design systems, it is used to **visualize** the analysis and the design.

It makes easier to understand and document software systems.

It supports teamwork because UML diagrams are more understandable than the program code.

There are different kinds of UML diagrams, which are used in different phases of a software development process.

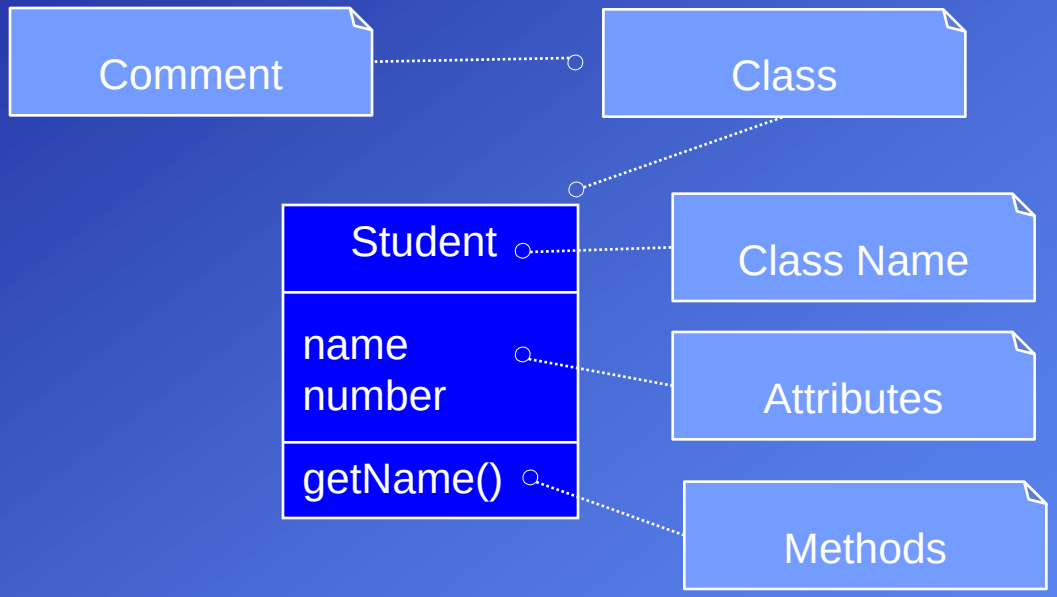
Here, we will discuss three types of these diagrams, which are used in design and coding levels.

The current specification of the UML is available in the Web site of the Object Management Group (OMG).
URL: http://www.omg.org/

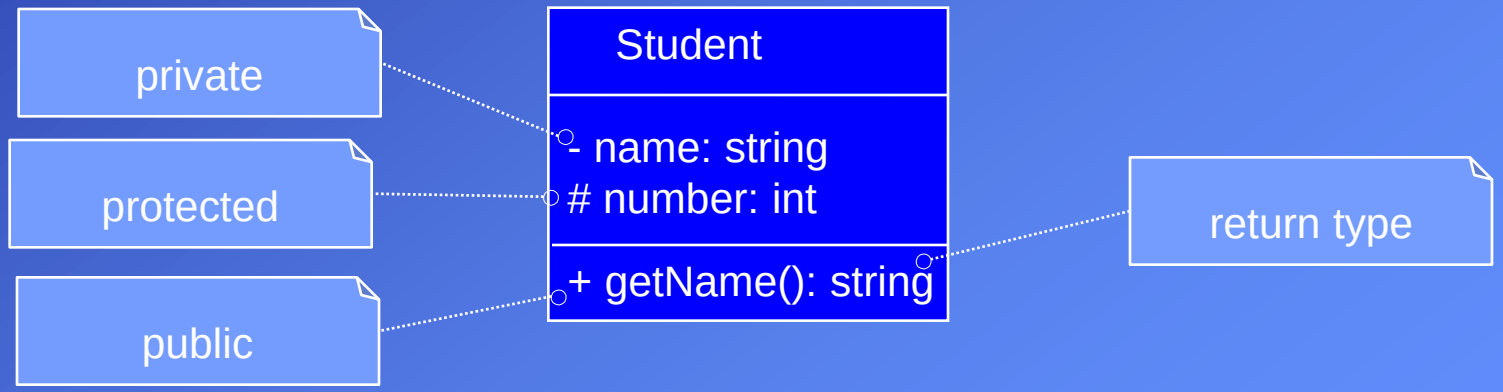
In this course, the current specification of the UML, version 2.x, is used.

Class Diagrams

A class diagram shows the structure of the classes and the relationships between them.



If necessary, access modes and data types may also be shown.



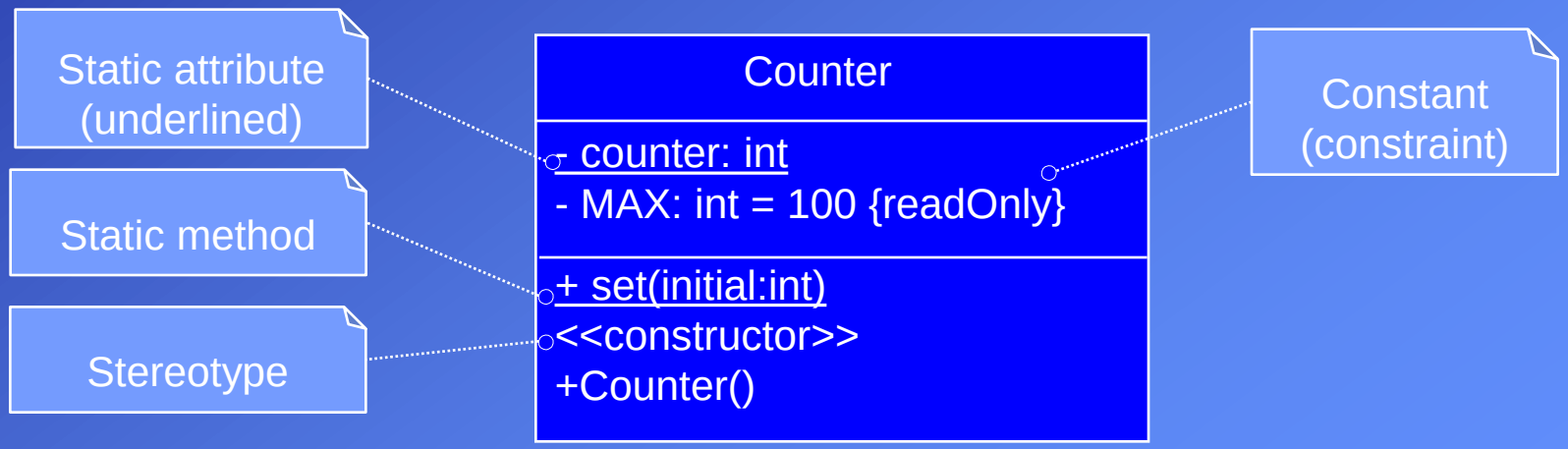
Comments : Comments in UML are placed in dog eared rectangles.

You can use comments to put anything you want in a diagram. You can use comments to add application and program specific details.

Stereotypes: A stereotype is a way of extending the UML in a uniform way, and remaining within the standard.

You indicate a stereotype using: <<stereotype name>>

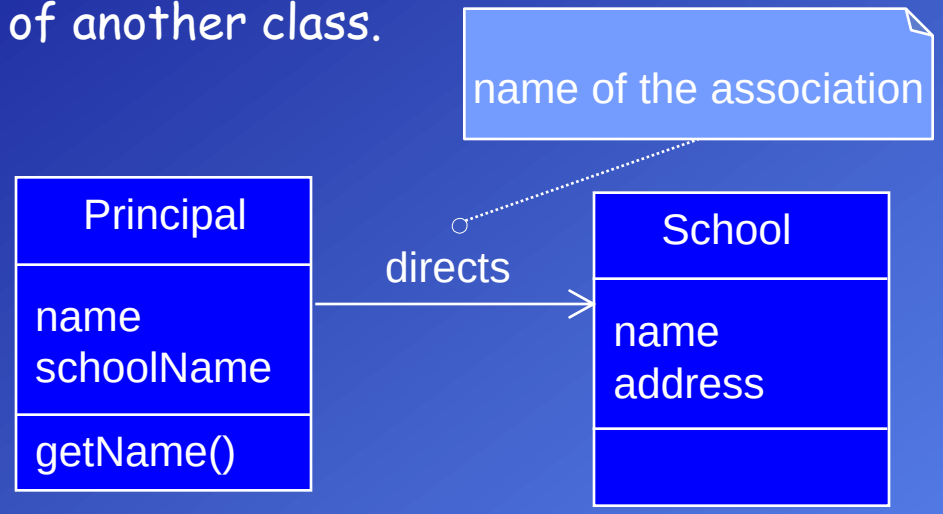
Constraints: A constraint in the UML is a text string in curly braces ({usually language specific}). The UML defines a language (Object Constrain Language – OCL) that you can use for writing constraints.



Relationship between classes

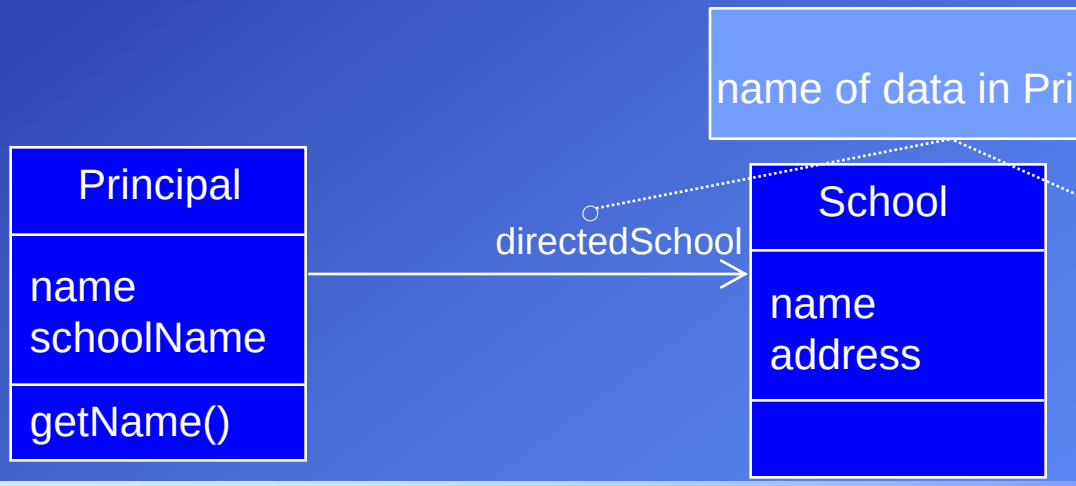
A class diagram also shows the relationships between classes such as association, aggregation, composition, and inheritance.

Association: A general type of relationship. Objects of a class can send messages to objects of another class.



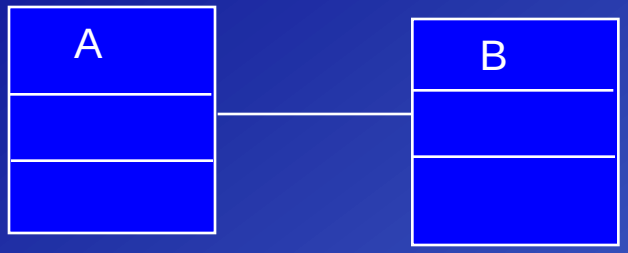
Principal can send messages to School objects.

Association names are used in conceptual perspective (analysis phase).
In software perspective association names are unnecessary.

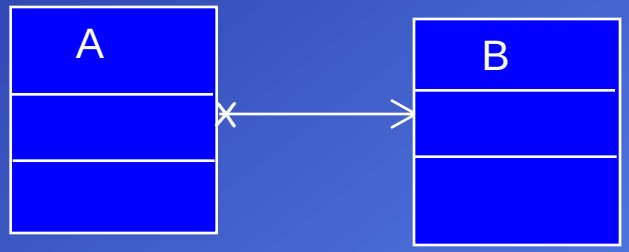



```
class Pricipal{
    private:
        School *directedSchool;
    :
};
```

Direction of the message flow:



Direction of messages is unspecified. Both may send messages to each other.

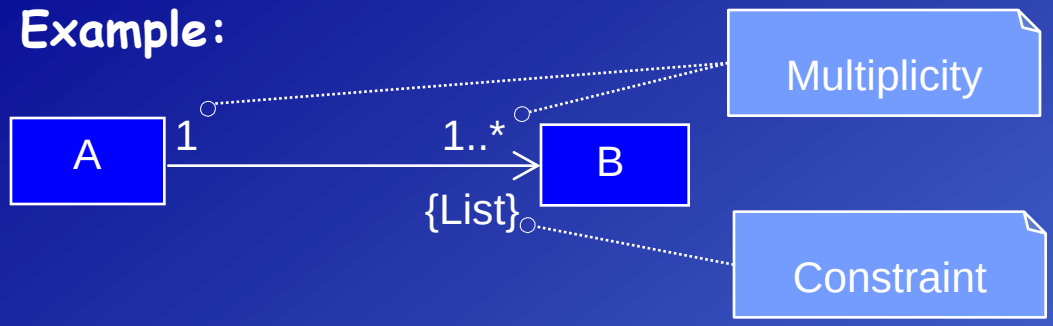


A can send messages to B.
A get a service from B.
B can not send messages to A.

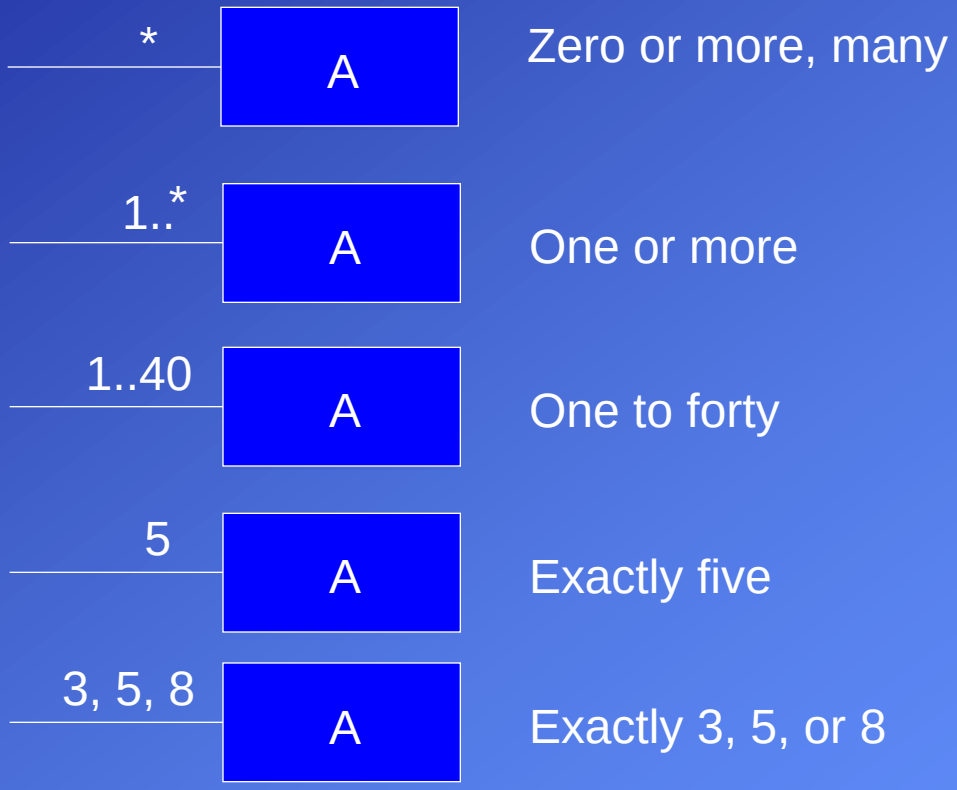
Multiplicity:
Multiplicity indicates the number of any possible combination of objects of one class associated with objects from another class.
In other words, it shows the number of objects from that class, that can be linked at runtime with one instance of the class at the other end of the association line.

An instructor teaches zero or more courses.
An association may also read in reverse order.
A course is given exactly by one instructor.

Example:



One object of class A is associated with one ore more objects of class B at a time.
Class A includes a list that can contain one or more objects of class B.



Aggregation, Composition: Both are a type of association. They are qualified by a " has a" relationship.

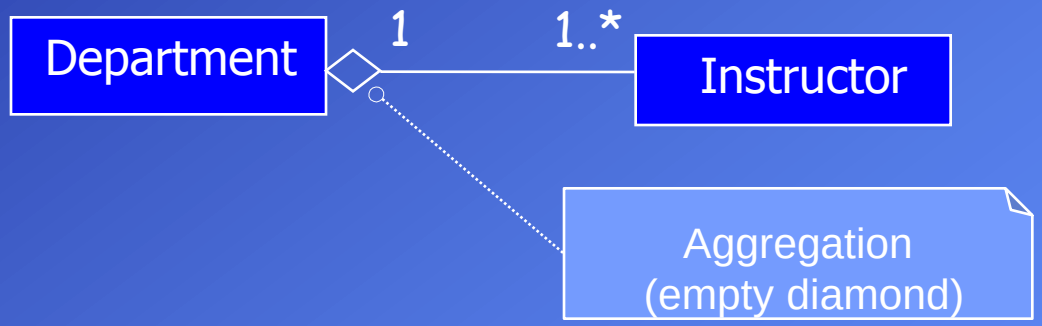
There is a small difference between them.

Aggregation: It indicates a "Whole/Part" relationship.

A department of the faculty has instructors.

Parts (instructors) can still exist even if the whole (the department) does not exist.

The same part-object can belong to more than one objects a time.



Composition: Composition is a strong kind of aggregation where the parts cannot exist independently of the "whole" object.

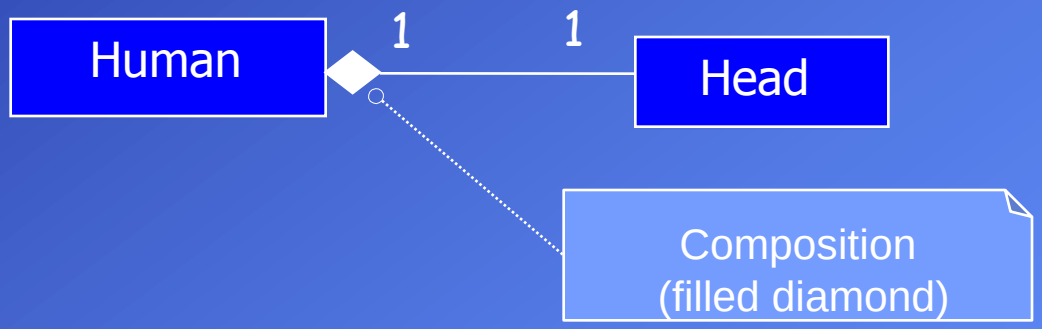
Examples: A human has a head. A car has an engine.

A composition relation implies that:

a) An instance of the part belongs to only one composite object.

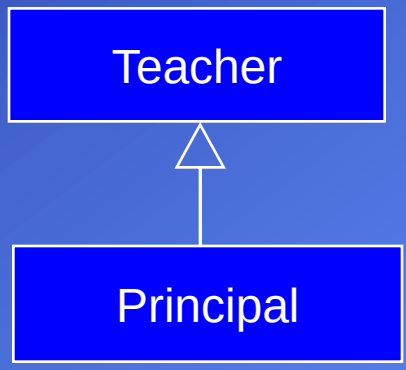
b) An instance of the part must belong to one composite object. It can not exist without the whole-object.

c) The composite is responsible for the creation and deletion of its parts. If the composite is destroyed its parts must also be destroyed.

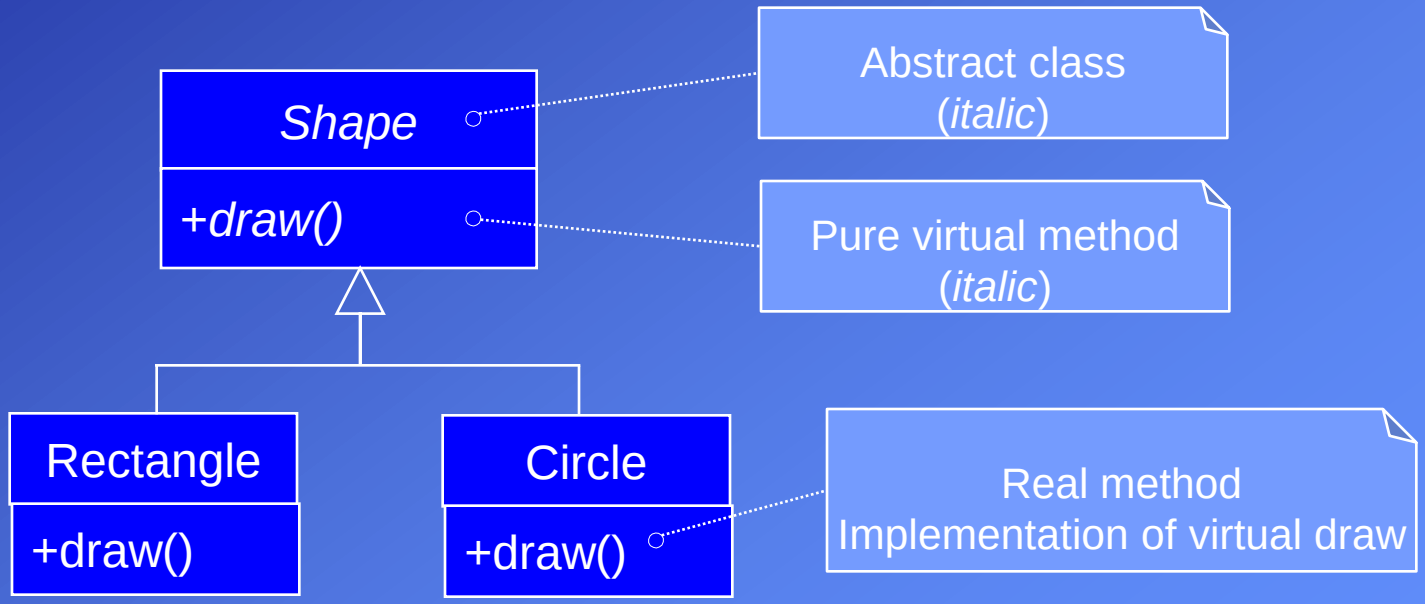


Inheritance: The white triangular arrow should point towards the class being extended.

The arrow should point upwards. This is not a rule of UML, but it feels more logical and easier to read in this form.

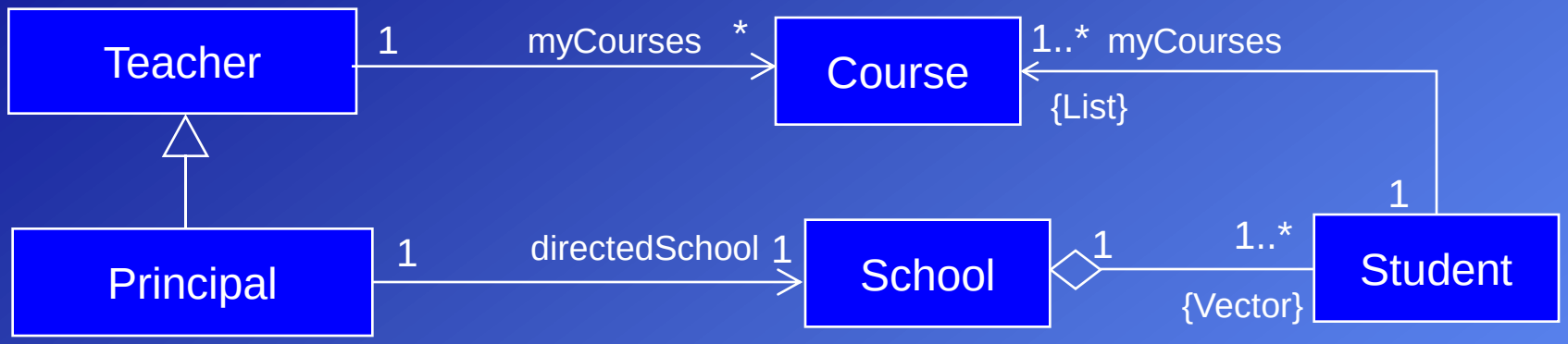


Abstract classes and pure virtual (abstract) methods are written with italic fonts.



Example:

Partial class diagram of a part of a system.




```
Class Teacher {
    private:
      Course * myCourses; //may be a linked list
    :
};

class Principal:public Teacher{
  private:
     School directedSchool;
     // or
     School *directedSchool;
};
```

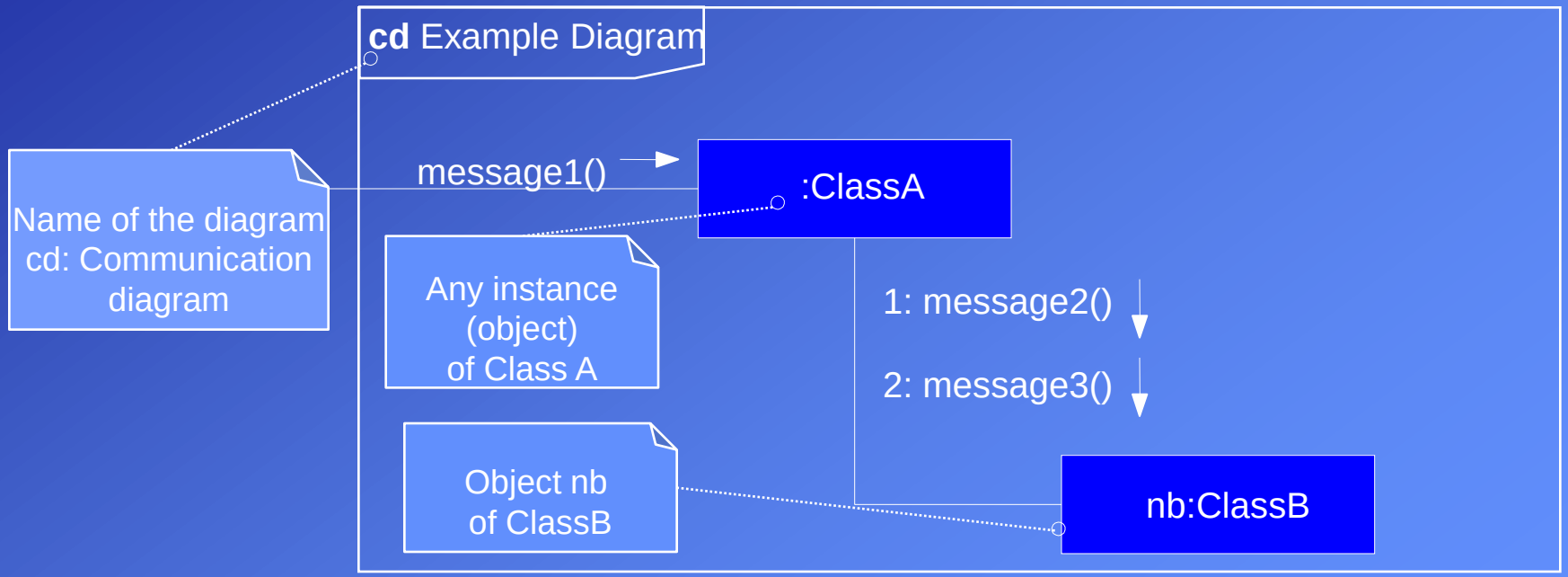
```
Class School {
    private:
        vector<Student*> students;
     :
};

class Student{
  private:
      list<Course*> myCourses;
      :
};
```

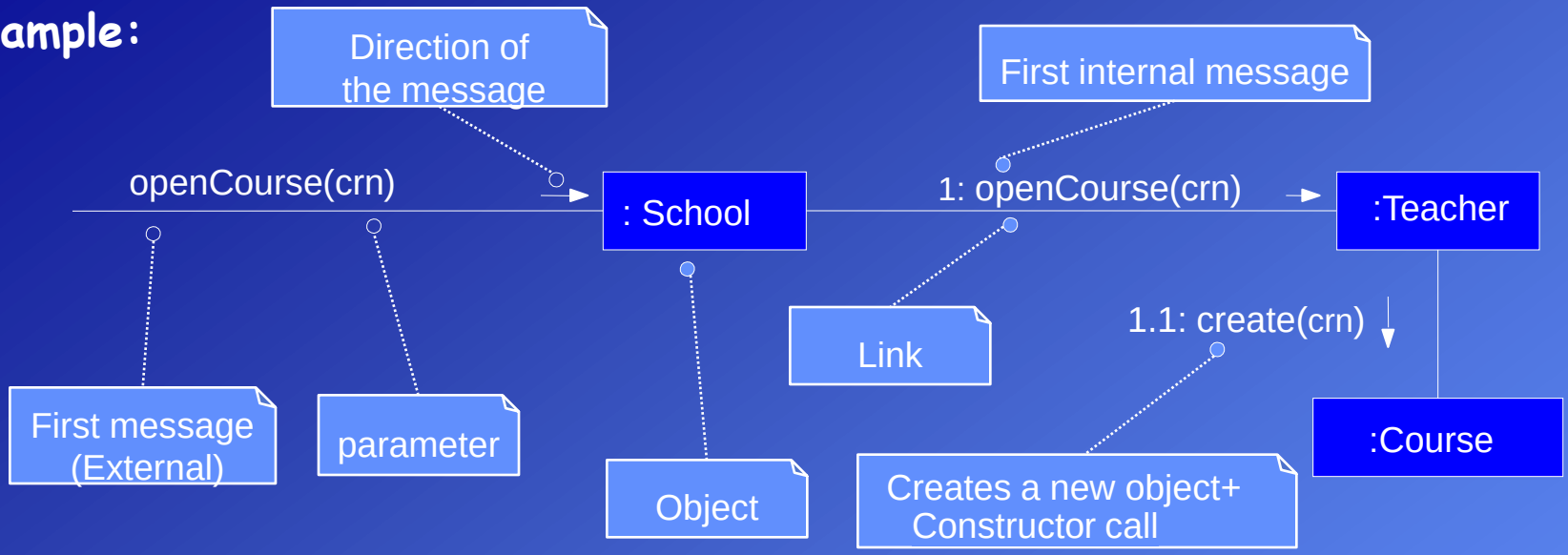
UML Interaction Diagrams

Interaction diagrams illustrate how objects interact via messages.
There are two common types: **communication** and **sequence** interaction diagrams.
Both can express similar interactions.
Sequence diagrams are more notationally rich, but communication diagrams have their use as well, especially for wall sketching.

Communication diagrams:
They illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram.



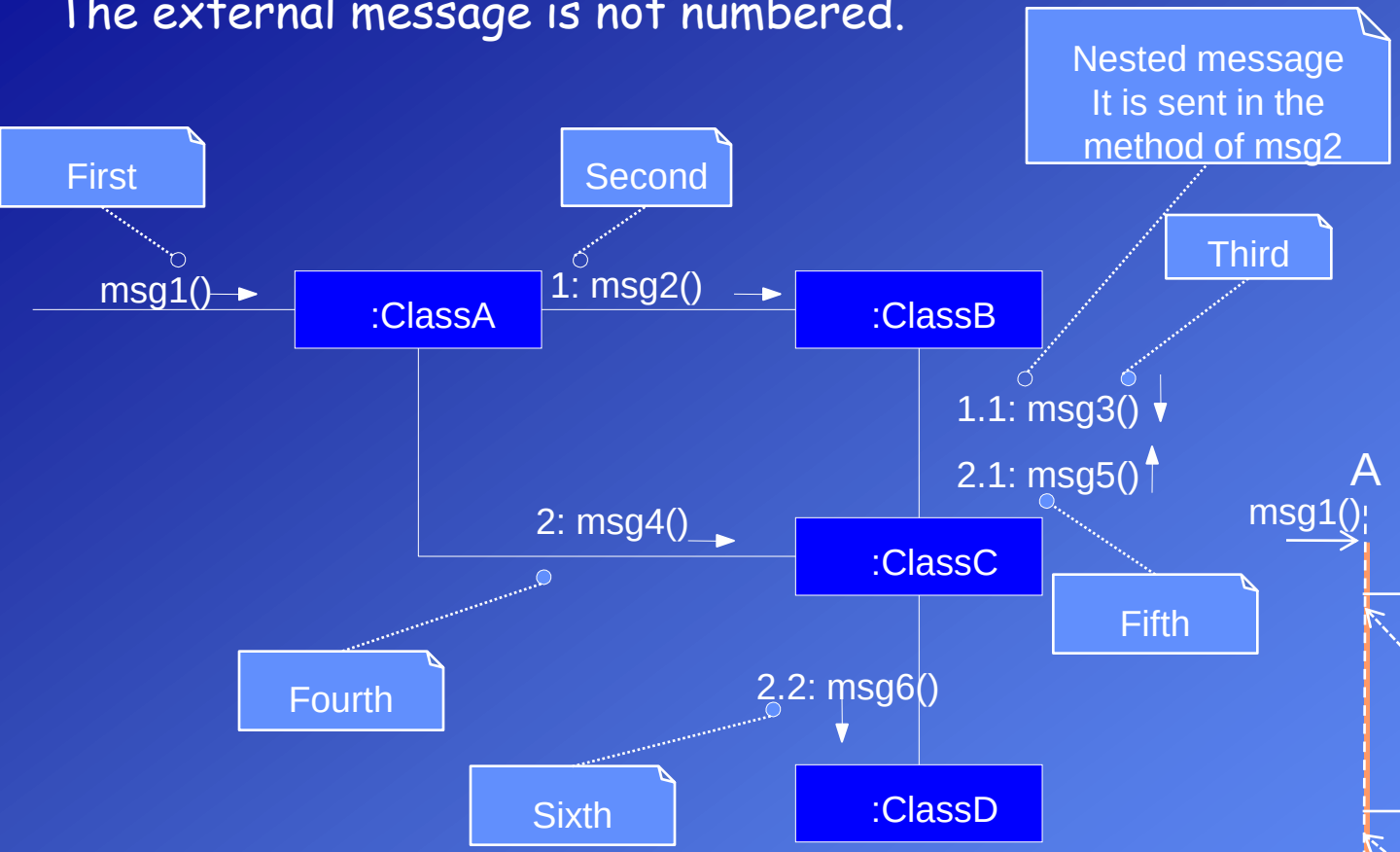
Example:

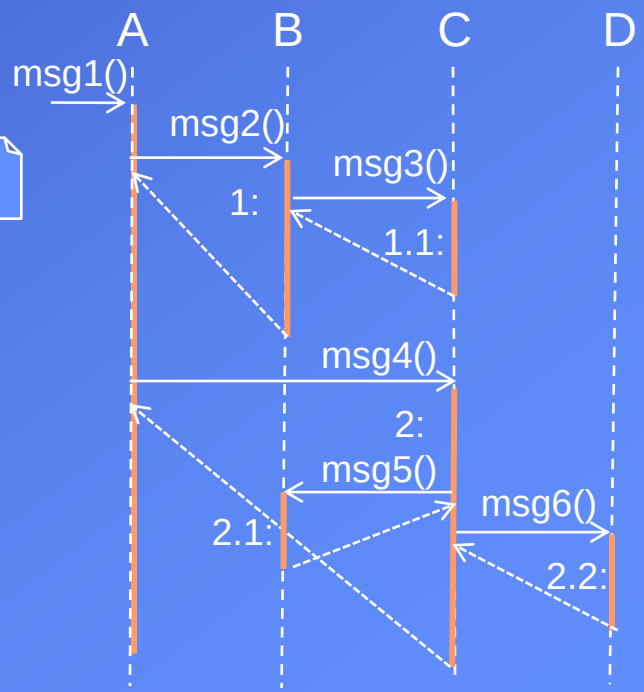



```
class Teacher{
  private:
      Course * myCourse;
  public:
     void openCourse(int crn){                    // openCourse method of the Teacher
             myCourse = new Course(crn);          // An object of type Course is created
             // Other operations ...
     }
     // Other members ...
};
```

Sequence numbers of messages:

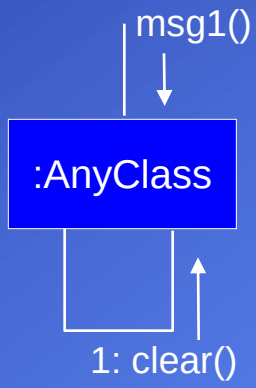
The external message is not numbered.





Messages to "self" or "this":

A message can be sent from an object to itself.

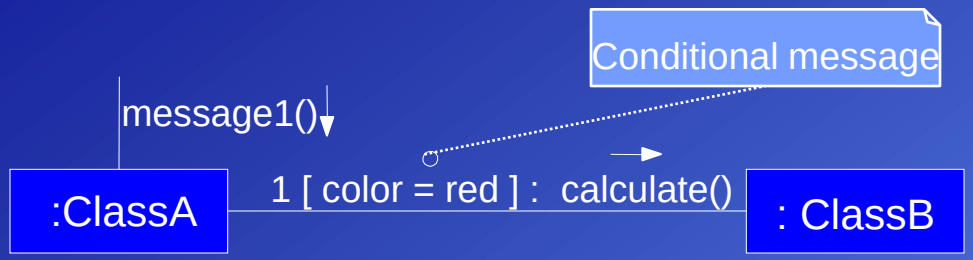


Creation of Instances:

Any message can be used to create an instance, but there is a convention in the UML to use a message named *create* for this purpose (some use new).
If another message name is used, the message may be annotated with a stereotype, like so: *«create»*.
The *create* message may include parameters, indicating the passing of initial values. This indicates, a constructor call with parameters.

Conditional Messages:

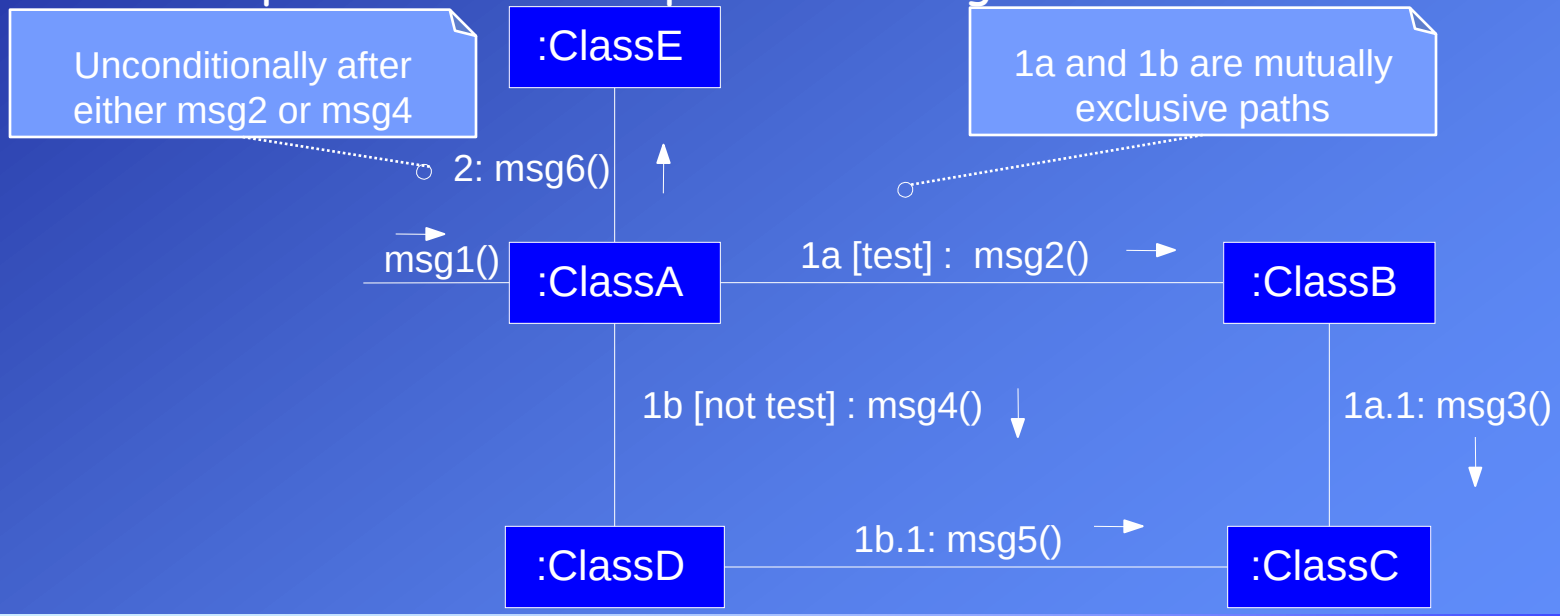
The message is only sent if the clause evaluates to *true*.



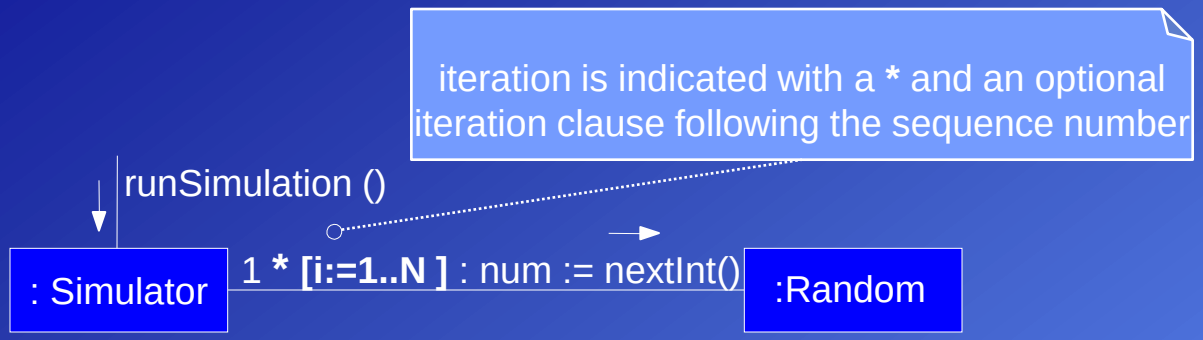
Mutually Exclusive Paths:

Message flows between objects may follow different paths according to some conditions.

In the example there are two path according to condition "test" : a or b .



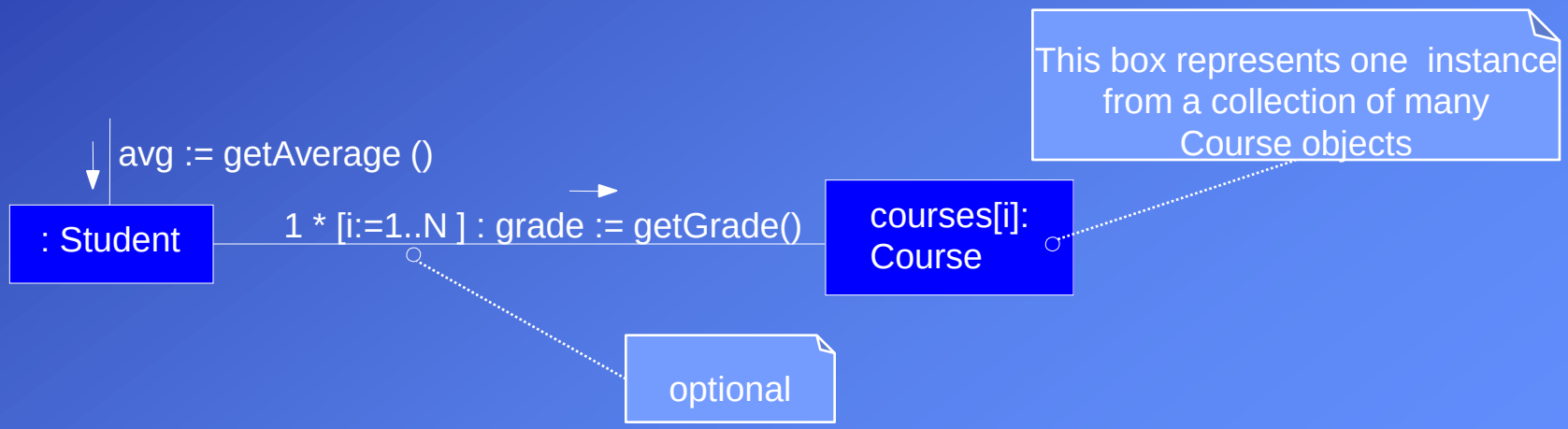
Iteration or Looping:



Iteration Over a Collection (Multiobject):

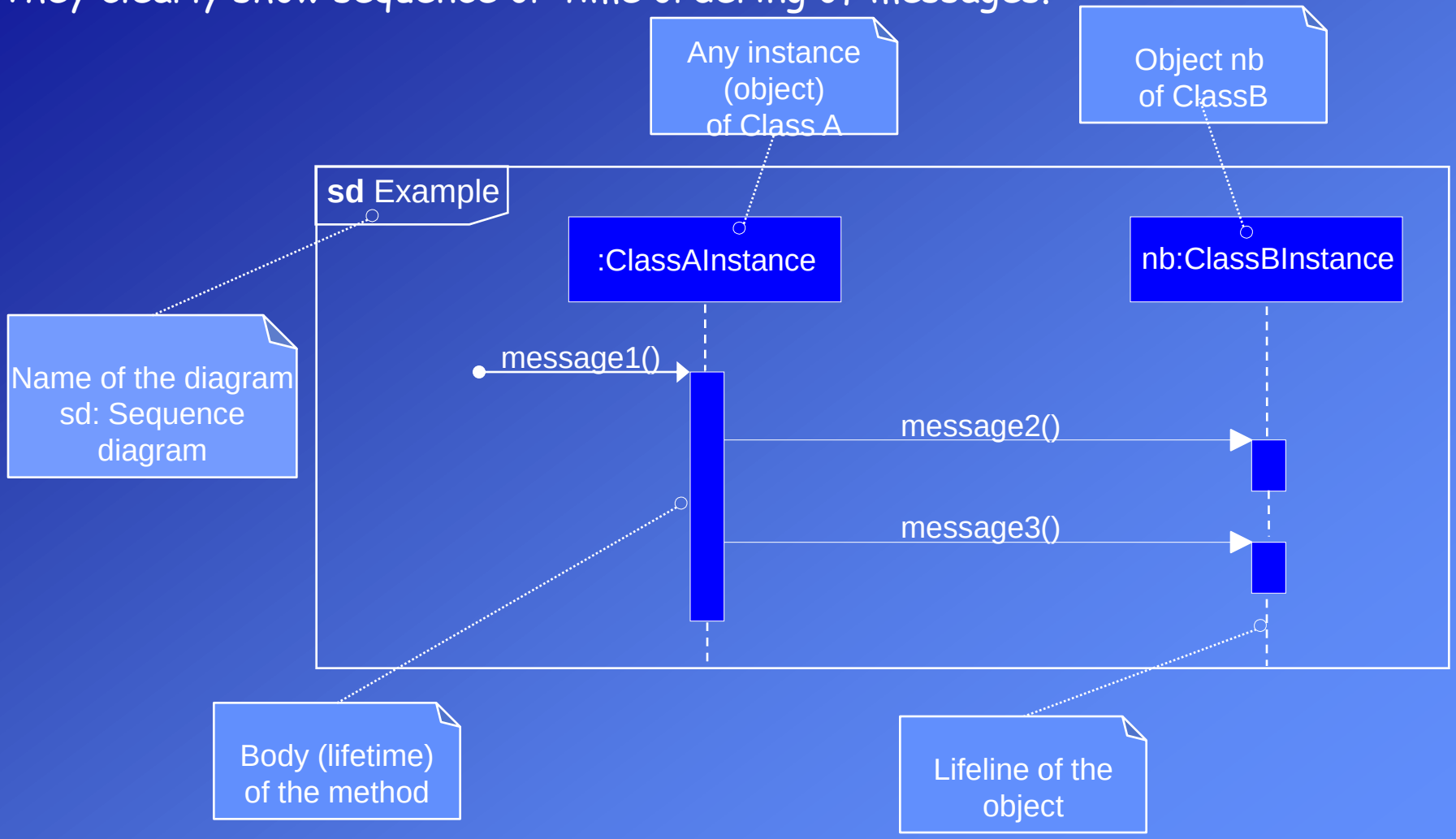
A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each.

In the UML, the term "**multiobject**" is used to denote a set of instances.

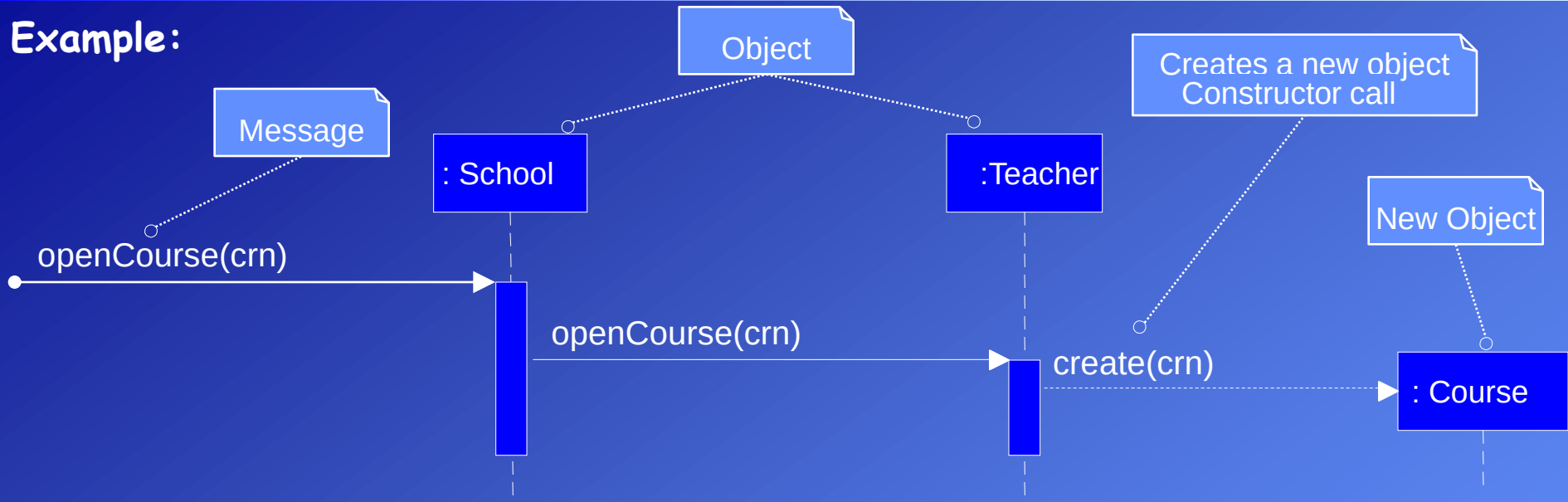


Sequence diagrams:

Sequence diagrams also illustrate the interactions between objects.
They clearly show sequence or time ordering of messages.

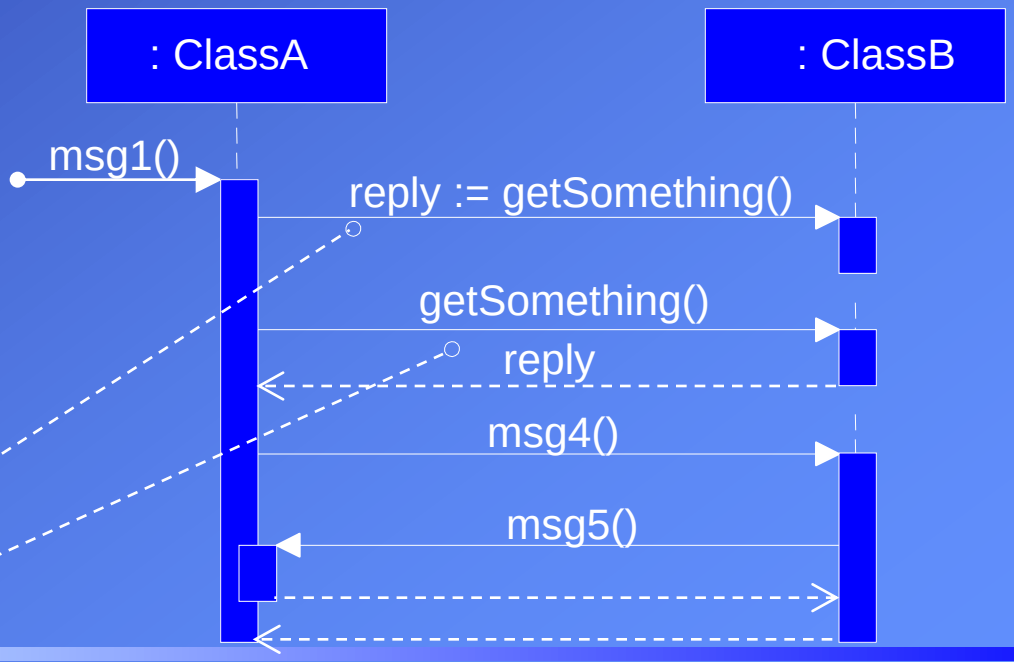


Example:



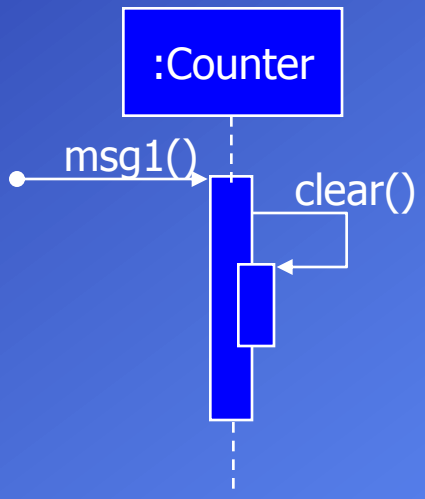
Illustrating Reply or Returns:

A sequence diagram may optionally show the return from a message as a dashed open-arrowed line at the end of an activation box.
There are two ways to show the return result from a message:
1. Using the message syntax:
   returnVar := message (parameter)
2. Using a reply (return) message line.



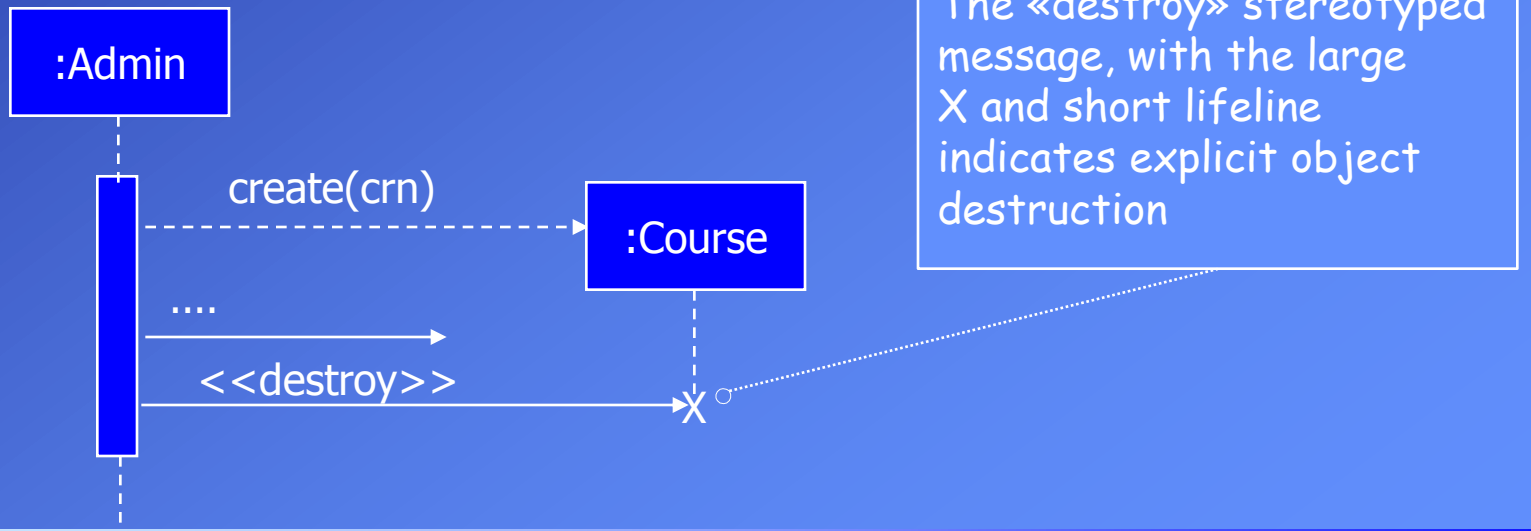
Messages to "self" or "this":

A message can be sent from an object to itself.



Object Destruction:

In some circumstances it is desirable to show explicit destruction of an object (as in C++, which does not have garbage collection).
In this case delete operator is used and destructor of the target object is called.

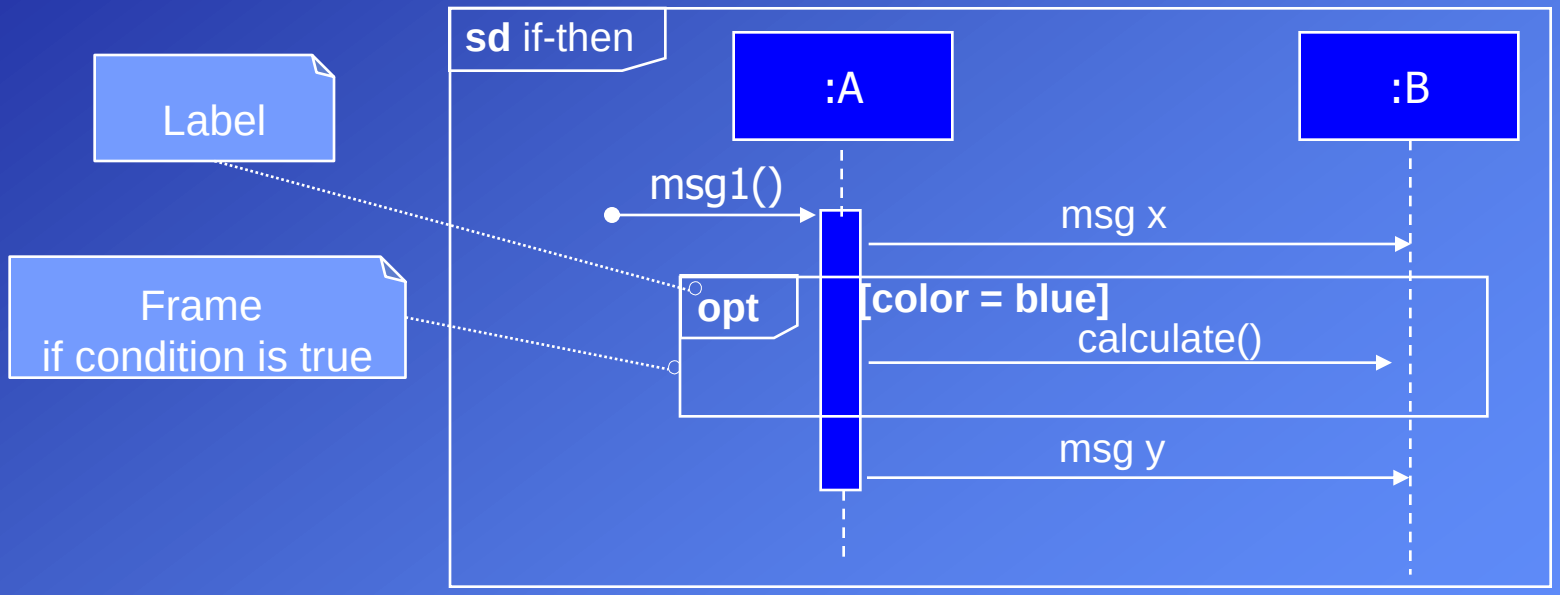


Conditional Messages:

To support conditional and looping constructs, the UML uses frames.

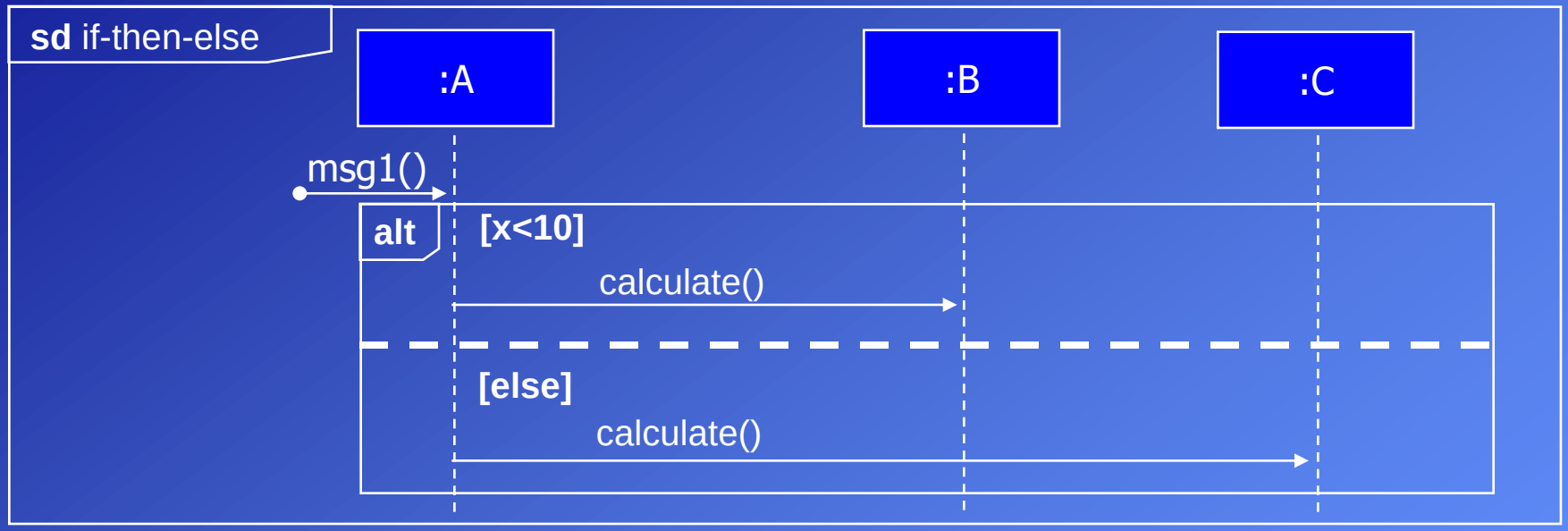
Frames are regions or fragments of the diagrams; they have an operator or label (such as loop or opt) and a guard (conditional clause).

In order to illustrate conditional messages an **opt** frame is placed around one or more messages.

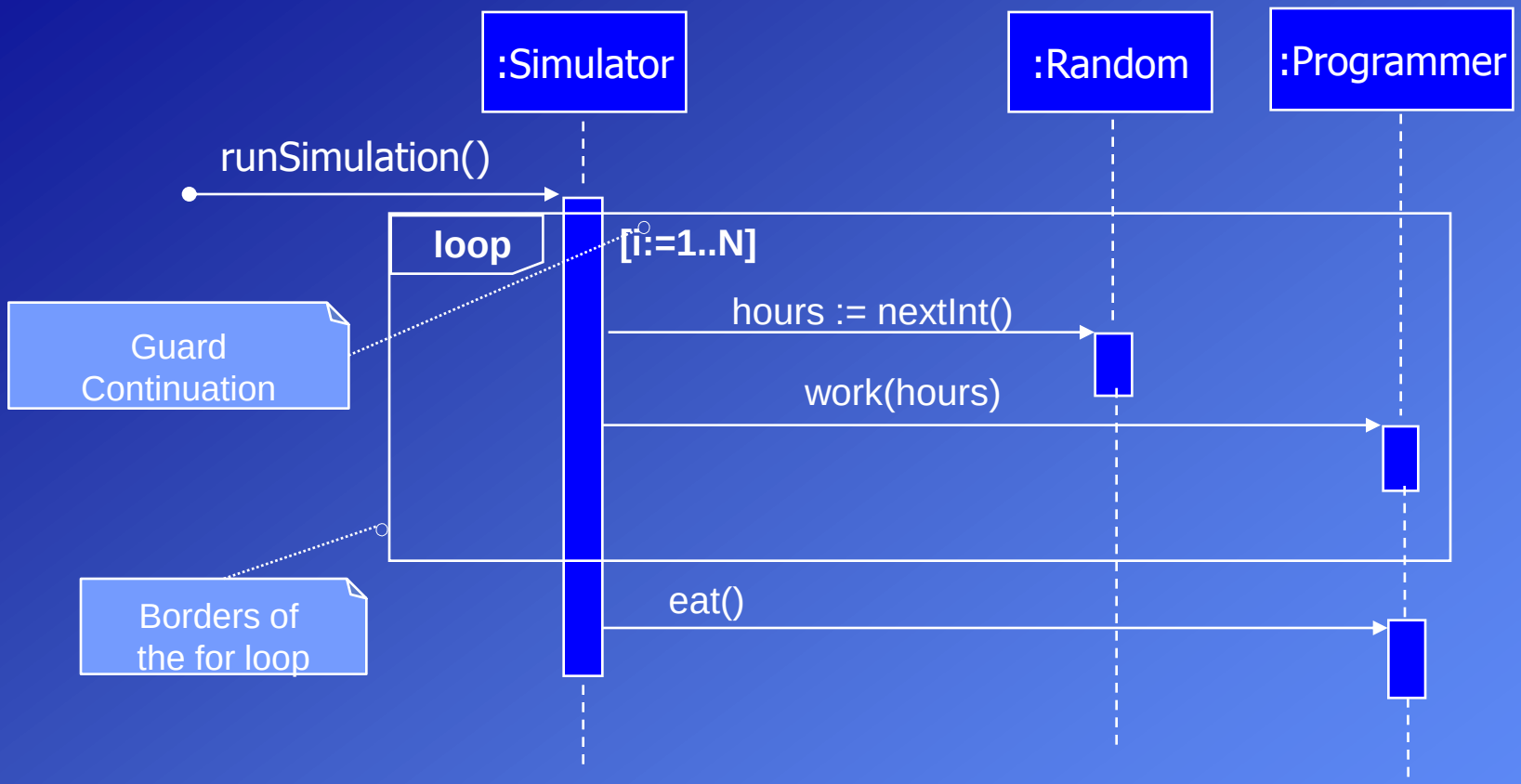


Mutually Exclusive Conditional Messages :

An **alt** frame is placed around the mutually exclusive alternatives.

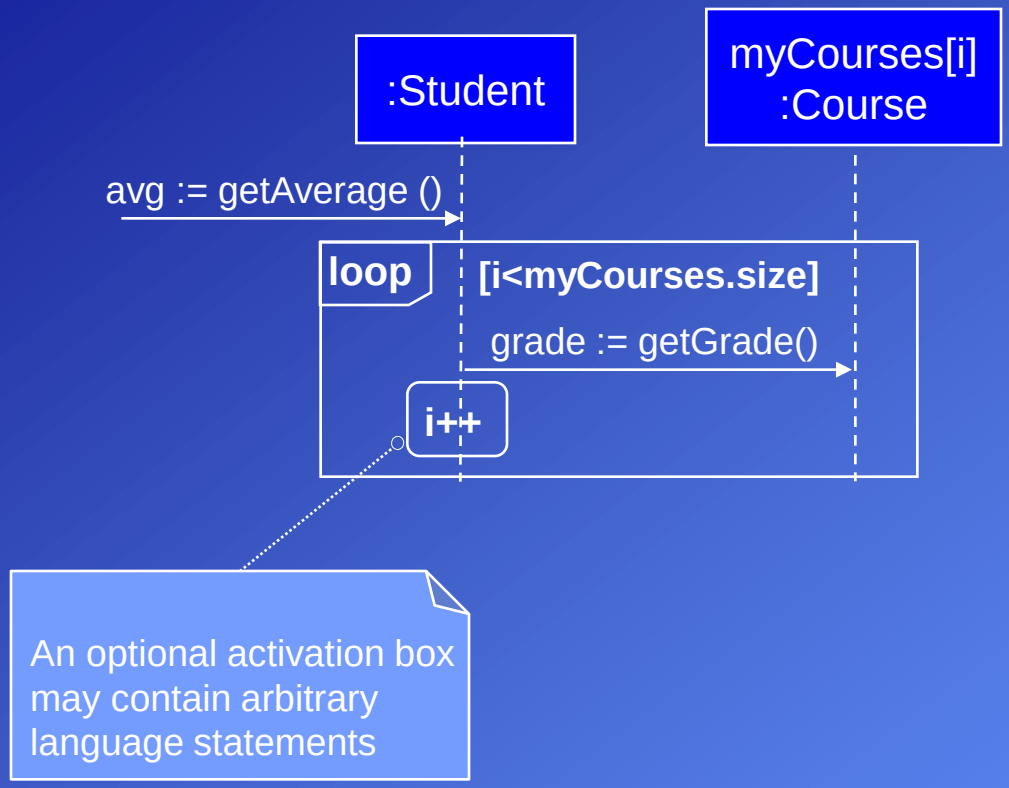


Looping:



Iteration Over a Collection (Multiobject):

A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each.
In the UML, the term "**multiobject**" is used to denote a set of instances.



Interaction of diagrams :

Reference frames are used to simplify a diagram and factor out a portion into another diagram, or if there is a reusable interaction occurrence.

It is like calling subroutines.