Relationships Between Objects

In the real world, there are relationships between objects.

Examples:

Students enroll in courses.

Classes have classrooms.

Professors have a list that contains the courses they offer.

The university consists of faculties, and faculties consist of departments.

The dean of the faculty is a professor.

A Ph.D. student is a kind of student.

The objects can cooperate (interact with each other) to perform a specific task.

Examples:

A professor can get the list of the students from the course object.

A student can get her grades from the related course objects.

A university can send an announcement to all faculties, and faculties can distribute this announcement to their departments.

Relationships Between Objects (contd)

In object-oriented design (OOD), we try to **lower the representational gap** between real-world objects and the software components.

This makes it easier to understand what the code is doing.

To represent real-world relationships, we also create relationships among software objects.

Types of relationships in object-oriented design (OOD):

There are two general types of relationships, i.e., **association** and **inheritance**.

- The **association** is also called a "**has-a**" ("uses") relationship.
- The **inheritance** is known as an "**is-a**" relationship.

Example:

A course **has a** classroom.

The dean of the faculty **is a** professor.

Although association itself is not a has-a relationship, its subtypes **aggregation** and **composition** are kinds of the has-a relationship.

In this section, we will cover association, aggregation, and composition.

The Inheritance (is-a) relationship will be covered in the coming sections.

Association ("**uses-a**" relationship):

- Association means objects of class A can send messages to (call methods of) objects of another class B.

  Objects of class A **can use** services given by class B.
- **Objects of A know objects of B**, and they communicate with each other.

  Class A has pointers (or references) to objects of class B.
- The relationship may be unidirectional or bidirectional (where the two objects are aware of each other).

  If the relationship is bidirectional, class B also has pointers (or references) to objects of class A.
- There may be one-to-one, one-to-many, and many-to-many associations between objects.
- The objects that are part of the association relationship can be created and destroyed independently.

  Each of these objects has its own life cycle.
- There is no "owner".

Association (contd):

Example:

Students register for courses.

- A student object can have a list of active courses they registered for.
- A course class can also have a list of the students taking that course (bidirectional).
- A student is associated with multiple courses. At the same time, one course is associated with multiple students (many-to-many).
- A student object can call methods of course classes, for example, to get the grade.
- If there is a bidirectional relation, the course class can also call the methods of the student class.
- Each of these objects has its own life cycle.

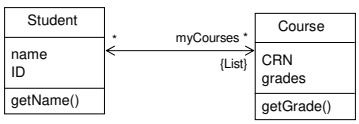
  The department can create new courses. In this case, new students are not created.

  When a course is removed from the department's plan, the students are not destroyed.

  Students can add or drop courses.

Example: Association between students and courses (contd):

UML Notation:

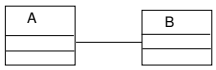


Summary:

An association is a weak "using" relationship between two or more objects in which the objects have their own lifetimes, and **there is no owner**.

UML Class diagrams for association:

Direction of the message flow:

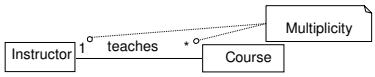


The direction of messages is unspecified. Both may send messages to each other.

Multiplicity:

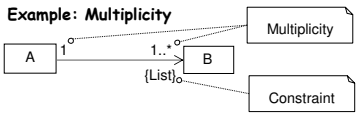
Multiplicity indicates the number of possible combinations of objects of one class associated with objects from another class.

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.

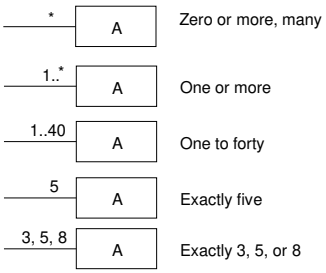


An instructor teaches zero or more courses (read from left to right).
An association may also be read in reverse order.
A course is given exactly by one instructor (read from right to left).

Example: Multiplicity



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



Aggregation:

- Aggregation is a specialized form of association between two or more objects.
- It indicates a "Whole/Part" ("has-a") relationship.
- Each object has its own life cycle, but ownership also exists.
- The same part-object can belong to multiple objects at a time.
- The whole (i.e., the owner) can exist without the part and vice versa.
- The relation is unidirectional. The whole owns the part(s), but the part does not own the whole.

Example:

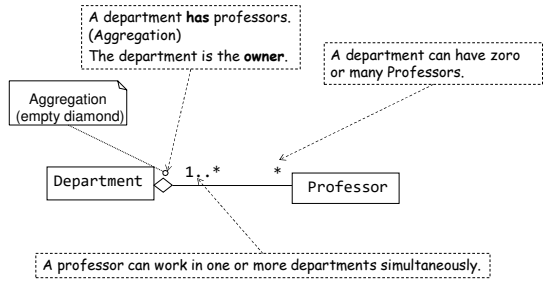
A department of the faculty **has** professors.

- A professor may belong to one or more departments at some universities.
- Parts (professors) can still exist even if the whole (the department) does not exist.
- If all professors retire or resign, the department can still exist and wait for new professors.
- A department may own a professor, but the professor does not own the department.

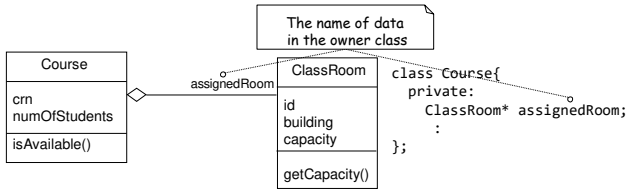
Example (contd):

A department **has** professors.

In UML diagrams, we use an empty diamond to present the aggregation relationship.



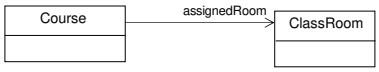
Example: An active course **has a** classroom.



If the owner is clear, we do not need to indicate it on diagrams.

We can also use an arrow to represent the aggregation, like for associations.

Remember: Aggregation is a special type of association where there is an owner.



In C++, the association and aggregation relationships are implemented nearly similarly.

Example: An active course **has a** classroom (contd)

```
class ClassRoom {  // Declaration/definition of the Classroom
public:
    :
   unsigned int getCapacity() const { return m_capacity; }
private:
  std::string m_building;
  std::string m_id;
  unsigned int m_capacity{};  // capacity initialized to zero
};

class Course {
public:
  // Initialize crn, number of students, and the classroom
  Course(const std::string&, unsigned int, const ClassRoom*);
  bool isAvailable() const; // Are there available seats?
private:
  :
  const ClassRoom* m_classRoom;  // The course has a classroom
};
```

Constructer gets the address of the assigned classroom.

Course has a pointer to ClassRoom objects.

Example: An active course **has a** classroom (contd)

```
// Constructor to initialize crn, number of students, and the classroom
Course::Course(const std::string& in_crn, unsigned int in_numOfStudents,
               const ClassRoom* in_classRoom )
            : m_crn{ in_crn },  m_numOfStudents{ in_numOfStudents },
              m_classRoom{ in_classRoom }
{}

bool Course::isAvailable() const {
  return m_classRoom->getCapacity() > m_numOfStudents;
}

int main(){
  ClassRoom classRoom1{ "BBF", "D5102", 100 };    // Classroom is created
  Course BLG252E{ "23135", 110, &classRoom1 };    // Course is created
  if (BLG252E.isAvailable()){
    room_id = BLG252E.getClassRoom()->getId();  // Chain of function calls
   ...
```

The pointer in the Course object points to the ClassRoom object.

A Course object **does not create** or **delete** ClassRoom objects. Each object has its own life cycle.

The Course object calls the method of the ClassRoom object.

See Example e06_1.cpp

Returns the pointer to the ClassRoom object.

getId() of the ClassRoom is called.

Composition:

- The Composition is also a specialized form of association and a specialized form of aggregation.
  Composition is a **strong** kind of "has-a" relationship.
- It is also called a "part-of" or "belongs-to" relationship.

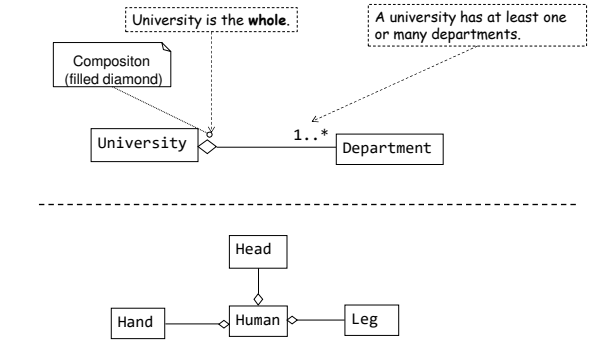
Examples:

- University is composed of departments, or departments are parts of a university.
- A rectangle is composed of four points.
- Rooms belong to a house.

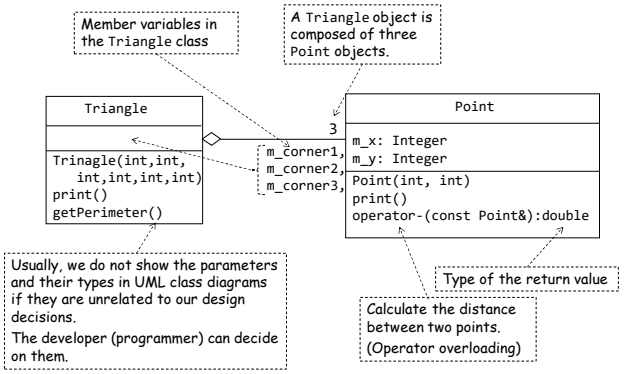
- The objects' lifecycles are tied.
  The part object (room) cannot exist without the owner/whole (house).
  When the owner object is deleted, the part objects are also deleted.
  Sometimes, the owner can still exist without some parts (members).
  The whole and part objects are created together.
  Constructors in C++ will ensure the creation of the parts when the owner is created.
- The relation is unidirectional.

Compositon (contd):

In UML diagrams, the composition is represented by a filled diamond arrowhead.



Example: A triangle is composed of three `Point` objects.



Example: A triangle is composed of three `Point` objects (contd).

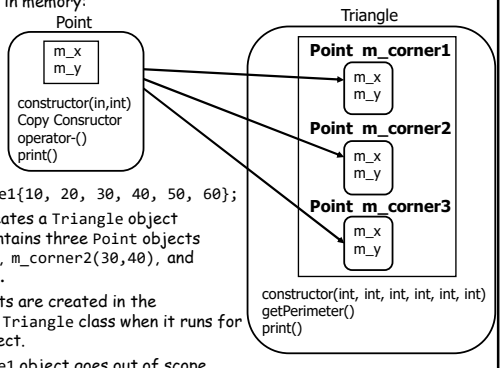
```
class Triangle {
public:
                // Constructor with three points
  Triangle(const Point&, const Point&, const Point&);
               // Constructor with the coordinates of three corners
  Triangle(int, int, int, int, int, int);
              // Calculates and returns the perimeter of the triangle
  double getPerimeter()const;
  void print()const;      // Prints the corners
private:
             // Corners of the triangle
   Point m_corner1, m_corner2, m_corner3;  // Composition
};
```

`Point` objects (parts) are created in the constructors of the `Triangle` object (whole, owner).

When the `Triangle` objects are destroyed, `Point` objects contained by them are also destroyed.

Example: A triangle is composed of three `Point` objects (contd).

`Triangle` objects in memory:



`Triangle triangle1{10, 20, 30, 40, 50, 60};`

This statement creates a `Triangle` object `tirangle1` that contains three `Point` objects `m_corner1(10,20)`, `m_corner2(30,40)`, and `m_corner3(50,60)`.

These `Point` objects are created in the constructor of the `Triangle` class when it runs for the `triangle1` object.

When the `triangle1` object goes out of scope, these `Point` objects will also be destroyed.

Example: A triangle is composed of three `Point` objects (contd)

The author of the `Triangle` class must call the constructors of the `Point` class properly to create point objects.

In this example, we assume that the `Point` class has only the following two constructors:

```
Point(int, int);       // Constructor to initialize x and y coordinates
Point(const Point&);   // Copy Constructor
```

The constructors of the Tringle class must call one of these constructors.

```
// Constructor with the coordinates of three corners
Triangle::Triangle(int corner1_x, int corner1_y, int corner2_x,
                   int corner2_y, int corner3_x, int corner3_y)
  :m_corner1{ corner1_x, corner1_y }, m_corner2{ corner2_x, corner2_y },
   m_corner3{ corner3_x, corner3_y }
{}
```

The constructor of the `Point` is called.

This constructor takes the x and y coordinates of three corner points (six integers) and calls the constructor of the `Point` class three times, once for each corner point.

Example: A triangle is composed of three Point objects (contd)

The author of the Triangle class must call the constructors of the Point class properly to create point objects.

```
// Constructor with three points
Triangle::Triangle(const Point& in_corner1, const Point& in_corner2,
                   const Point& in_corner3)
          :m_corner1{ in_corner1 }, m_corner2{ in_corner2 },
           m_corner3{ in_corner3 }
{}
```

The copy constructor of the Point is called.

This constructor takes references to three existing point objects and calls the copy constructor of the Point class three times, once for each corner point. The member points of the triangle are created as copies of the input points.

Since the Point class does not contain a default constructor in this example, the author of the Triangle class **cannot** create corner points as follows:

```
// Constructor that calls the default constructor of the Point
Triangle::Triangle():m_corner1{}, m_corner2{}, m_corner3{}  //Error!
{}
    or
Triangle::Triangle(){} //Error! If the Point does not contain a default constructor
```


Example: A triangle is composed of three Point objects (contd)

Objects of the Triangle can use public methods of its member points to fulfill its tasks.

```
// Calculates and returns the perimeter of the triangle
double Triangle::getPerimeter()const {
  return (m_corner2 - m_corner1) + (m_corner3 - m_corner2)
        +(m_corner1 - m_corner3);
}

// Prints the corners
void Triangle::print()const {
  cout << "Corners of the triangle:" << endl;
  m_corner1.print();
  m_corner2.print();
  m_corner3.print();
}
```

See Example e06_2.cpp

Default Constructors and Destructors in composition:

Remember, when the programmer does not write a constructor, the compiler provides a default default constructor.

The default default constructor of the whole (owner) calls the default constructor of the parts.

See Example e06_3a.cpp

If the Part class contains a programmer-written default constructor, the default default constructor of the Whole calls it automatically.

See Example e06_3b.cpp

When the whole object goes out of scope, the destructors are called in reverse order, i.e., the Whole object is destroyed first, then the member objects (parts).

See Example e06_3c.cpp

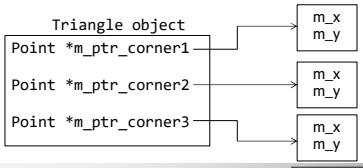
Dynamic Member objects (Pointers as members)

Instead of automatic objects, data members of a class may also be pointers to objects of other classes (parts).

Example: The Triangle class contains pointers to Point objects.

```
class Triangle {
   :
private:
                              // Pointers to corners of the triangle
 Point *m_ptr_corner1, *m_ptr_corner2, *m_ptr_corner3;
};
```

Now, only the pointers (addresses) of Point objects are included in the objects of the Triangle.



Dynamic Member objects (Pointers as members) (contd)

If the relationship is aggregation, the owner should get the addresses of its members as parameters to its constructors.

If the relationship is composition, the whole must create and initialize part objects (memory allocation) in the constructor.

Example: The Triangle class contains pointers to Point objects.

Since the relationship is composition, the member objects must be created in the constructor of the Triangle.

```
// Constructor with the coordinates of three corners
Triangle::Triangle(int corner1_x, int corner1_y,
                   int corner2_x,int  corner2_y, int corner3_x, int corner3_y)
       :m_ptr_corner1{ new Point{corner1_x,corner1_y} },
        m_ptr_corner2{ new Point{corner2_x,corner2_y} },
        m_ptr_corner3{ new Point{corner3_x,corner3_y} }
{}
```

The constructor of the Point is called.

Dynamic Member objects (Pointers as members) (contd)

If the relationship is composition and memory is allocated in the constructor, then these memory locations must be released (in most cases) in the destructor.

Example: The Triangle class contains pointers to Point objects.

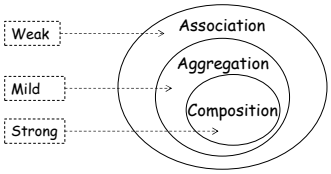
```
// Destructor
Triangle::~Triangle()
{
   delete m_ptr_corner1;
   delete m_ptr_corner2;
   delete m_ptr_corner3;
}
```

The destructor of the Point is called.

See Example e06_4.cpp

Summary: Asscociation, Aggregation, Compoisiton

Property	Association	Aggregation	Composition
Relationship type	Otherwise unrelated	Whole/part	Whole/part
Relationship verb	Uses-a	Has-a	Part-of
Members can belong to multiple classes	Yes	Yes	No
Members' existence managed by owner	No	No	Yes
Directionality	Unidirectional or bidirectional	Unidirectional	Unidirectional

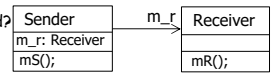


Visibility Between Objects

Visibility means that one object can "see" or have reference to another object.

To send a message to another object, the sender must have a reference or a pointer to the receiver object.

How can the Sender call Receiver's mR() method?



The sender must "see" the receiver.

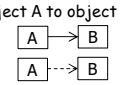
```
m_r.mR();
m_rPtr->mR();
```

During the design of a system as a set of interacting objects, it is necessary to ensure that the required visibility is achieved between objects to support message interaction.

Types of visibility:

There are four ways that visibility can be established from object A to object B:

- **Attribute visibility:** B is an attribute of A.
- **Parameter visibility:** B is a parameter of a method of A.
- **Local visibility:** B is a (non-parameter) local object in a method of A.
- **Global visibility:** B is in the global space of A.



Types of visibility:

Example:

In the example `e06_1.cpp`, the `Course` class has a pointer to its classroom.

```
class Course{
  private:
    ClassRoom* assignedRoom;  // The course has a classroom
     :
};
```

In the main function, we create the object of the `ClassRoom` and send it to the constructor of the `Course` object to establish the attribute visibility from the `Course` object to the `ClassRoom` object.

Now the `Course` object can "see" the `ClassRoom` object.

```
ClassRoom classRoom1{ "BBF", "D5102", 100 }  // ClassRoom object is created
Course BLG252E{ "23135", 110, &classRoom1 }; // Visiblity
```

Example:

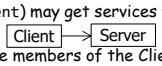
In the examples `e06_2.cpp` and `e06_4.cpp` corner points of the `Triangle` are created in the constructor of the `Triangle` class.

There is attribute visibility from the `Triangle` to the corner objects.

Example: Parameter visibility

Sending this as an argument to establish visibility:

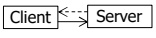
In an object-oriented program, a class (`Client`) may get services from another class (`Server`) by calling its methods.



The Server class may also need to access the members of the Client class to give these services.

If this is the case, the `Client` object can send its address (`this`) to the `Server` object to enable the `Server` to (see) access the `public` members of the `Client` object. Now, we have a bidirectional association (visibility)

Example:



We have a class called `GraphicTools` that contains tools that `Point` objects can use.

For example, the method `distanceFromZero` of the `GraphicTools` calculates the distance of a `Point` object from zero (`0,0`).

We assume that the `Point` class does not have the ability to calculate distances.

The `Point` class may contain a pointer to the object of the `GraphicTools`.

The `distanceFromZero` method of the `GraphicTools` can get the reference to a `Point` object for which the distance is calculated.

Now, both of the objects can see each other.

Example: Parameter visibility (contd)


```
class Point {
public:
  //Constructor receives the address of the GraphicTools object for visibility
  Point(int, int, GraphicTools*);
private:
  GraphicTools * m_toolPtr;     // Visibility to GrpahicsTool
};
:
double Point::distanceFromZero() const {
  return m_toolPtr->distanceFromZero(*this); // sending this for visiblity
}
```

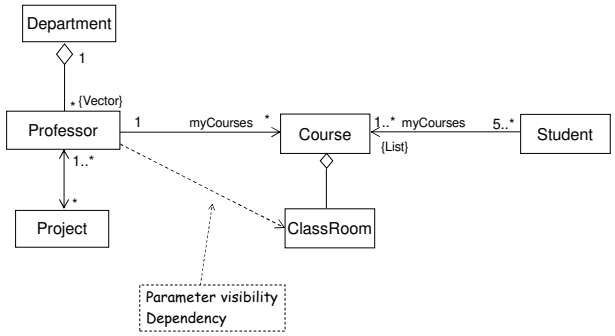
The methods of the `Point` can access methods of the `GraphicTools`.
Since the method sends `this` pointer, the method of the `GraphicTools` can also access methods of the `Point` class (bidirectional association).

Parameter visibility

```
double GraphicTools::distanceFromZero(const Point& in_point) const {
  double local_x = in_point.getX();   // Can call methods of the Point
  double local_y = in_point.getY();
  return sqrt(local_x * local_x + local_y * local_y);
}
```

See Example `e06_5.cpp`

Example: Partial class diagram of an exemplary software system for a school.



Smart pointers:

Industrial software systems generally comprise many collaborating objects linked together using pointers and references.

All these objects must be **created**, linked together (visibility), and **destroyed** at the end.

The Standard Library of C++ includes **smart pointers**, which ensure all objects are deleted in a timely manner.

A smart pointer is a wrapper class template that owns a raw pointer and overloads necessary operators like * and -> .

Smart pointers are used like standard pointers.

Unlike standard pointers, they can destroy objects automatically when necessary.

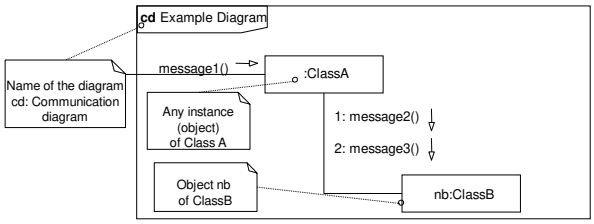
C++ Standard Library smart pointers:

- `std::unique_ptr<type>`: It ensures the object is deleted if it is not referenced anymore.
- `std::shared_ptr<type>`: It is used when an object has (shared by) multiple owners. It is a reference-counted smart pointer.
  The raw pointer is not deleted until all `shared_ptr` owners have gone out of scope or given up ownership.
  We will cover smart pointers in detail in Chapter 10.

UML Interaction Diagrams

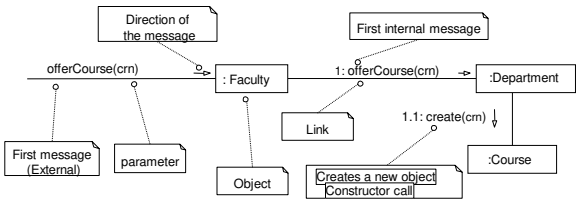
Interaction diagrams illustrate how **objects** interact via messages in **runtime**.
There are two common types: **communication** and **sequence** interaction diagrams.
Both can express similar interactions.
Sequence diagrams are more notationally rich, but communication diagrams also have their use,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

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



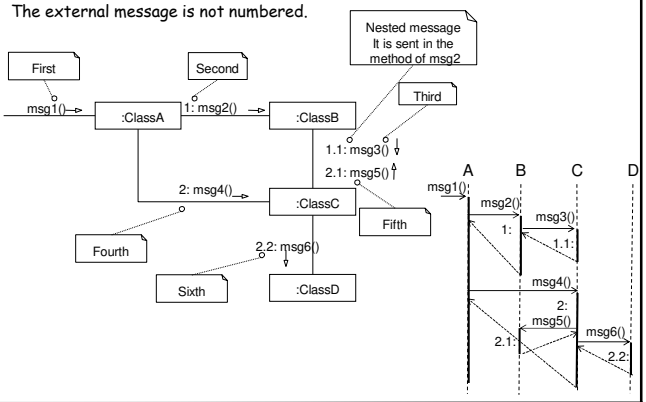
Example:

A communication diagram that presents a message flow about offering a new course at the beginning of a semester.



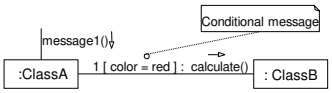
Sequence numbers of messages:

The external message is not numbered.

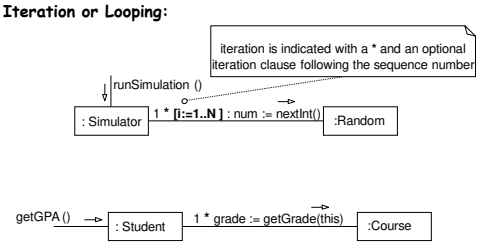


Conditional Messages:

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

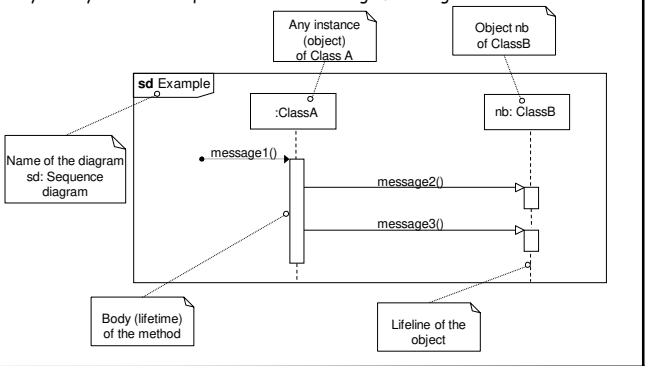


Iteration or Looping:

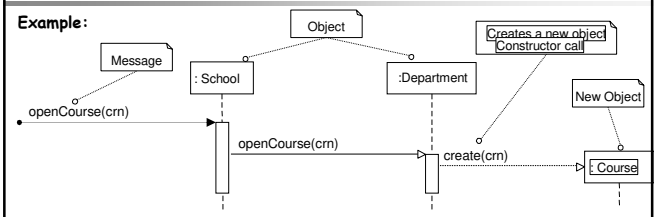


Sequence diagrams:

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

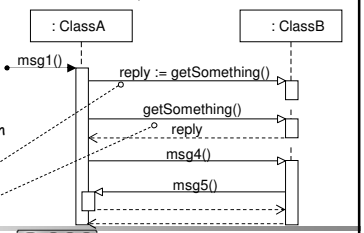


Example:



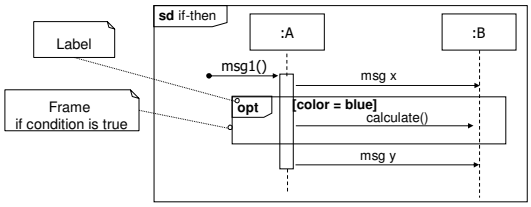
Illustrating Reply or Returns:

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.



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).
To illustrate conditional messages, an **opt** frame is placed around one or more messages.



Looping: