Lecture 12

Unified Modeling Language

Topics

- Class Diagrams
- Sequence diagrams

Unified Modeling Language

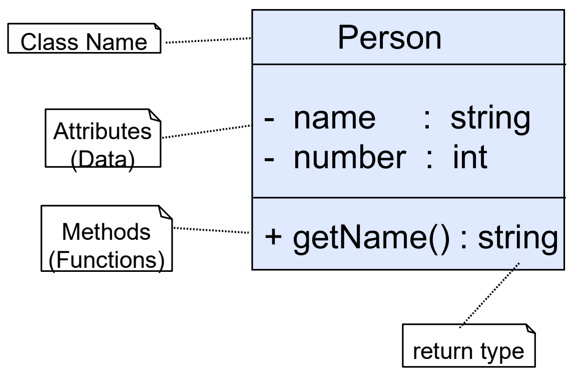
- UML is a graphical language for visualizing, specifying, constructing, and documenting of object-oriented software.
- UML diagrams can be used before coding for design purpose.
- They can also be used after coding for documentation purpose.
- Different kinds of diagrams are used in different phases of software development.
- Basic Diagrams :
  − Class Diagram
  − Sequence Diagram

Class Diagrams

- A class diagram shows the structure of classes and connections between classes.
- A box containing three sections is used to represent a class.
- Access mode specifier symbols are used for member data and functions.
  - is private, + is public, # is protected

Class Diagram

Comments



C++ Code

```
class  Person
{
private:
     string  name;
     int       number;

public:
     string  getName();
};
```

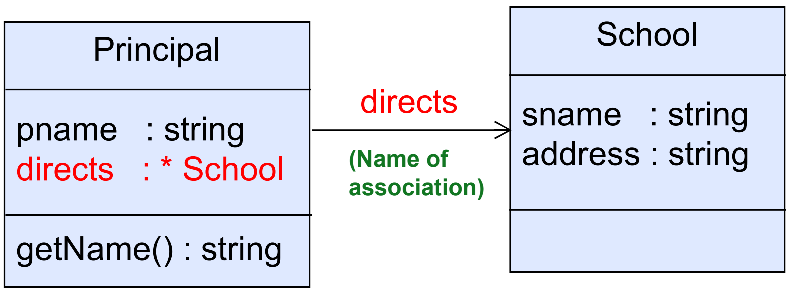
Connections Between Classes

The followings are connection types between classes, used in class diagrams.

- Association
- Multiplicity
- Aggregation (HAS-A)
- Composition (HAS-A)
- Inheritance (IS-A)

Association

- Association is a general type of connection.
- Association is usually implemented with a **pointer** variable in a class.

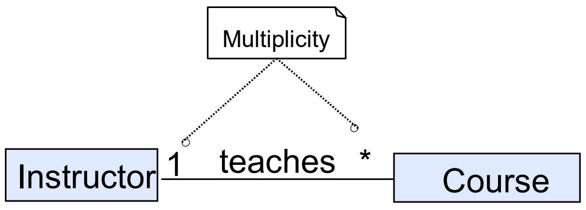


```
class  Principal
{
    string     pname;
    School *  directs;    // pointer
public:
    string  getName() {return  pname;}
};
```

```
class  School
{
   string  sname;
   string  address;
};
```

Multiplicity

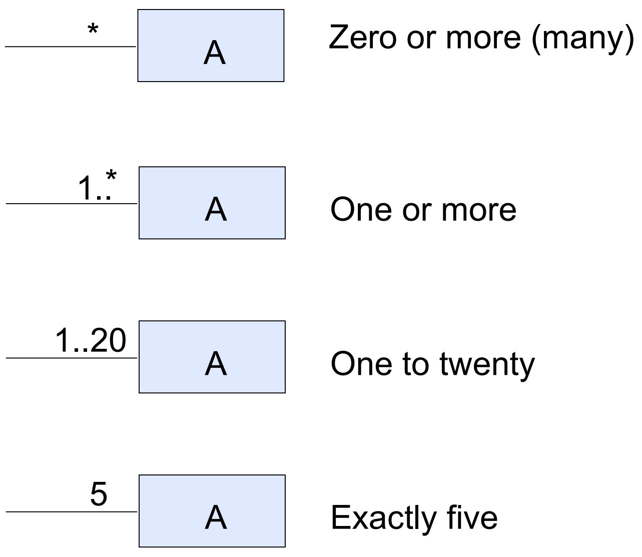
- Multiplicity indicates the number of objects of one class, associated with objects of another class.
- Example:
  An Instructor teaches zero or more Courses.
  A Course is given exactly by one Instructor.



```
#define MAX 5
class  Instructor
{
     string    iname;
     Course  teaches [MAX];  // array
};
```

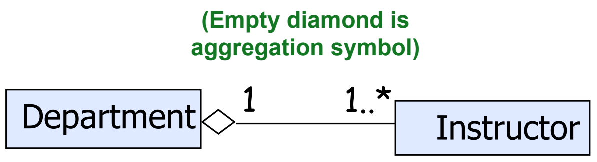
```
class  Course
{
   string           cname;
   Instructor *  givenby;  // pointer
};
```

Multiplicity Examples



Aggregation

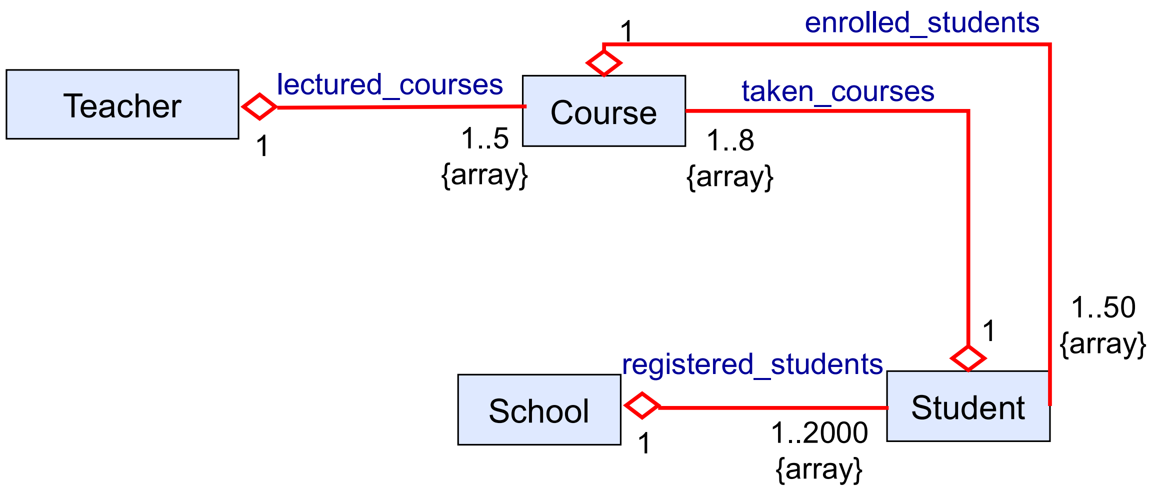
- Aggregation indicates a **Whole / Part** connection.
- Example: Department (whole) has one or more Instructors.
  Department is aggregated with Instructors.
- Parts (Instructors) can still exist, even if the whole (Department) does not exist.



```
#define MAX 30
class  Department
{
    string       dname;
    Instructor  intructors [MAX];
};
```

```
class  Instructor
{
   string            iname;
   Department *   dept;
};
```

Example: Aggregations (With arrays)



Example: Classes with Array Aggregations

```
// Teacher has courses
Class  Teacher
{
   Course  lectured_courses [5];
};
```

```
// Course has students
class  Course
{
   Student  enrolled_students [50];
};
```

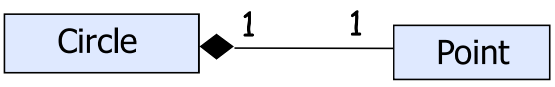
```
// School has students
Class  School
{
   Student  registered_students [2000];
};
```

```
// Student has courses
class  Student
{
   Course  taken_courses [8];
};
```

Composition

- Composition is an association where the Parts cannot exist independently of the Whole object.
- Composition is more strict than Aggregation.
- Example: Circle has-a center Point.
  Circle is composed with a Point.
  A Point can not be used without a Circle (composite object).

(Filled diamond is composition symbol)

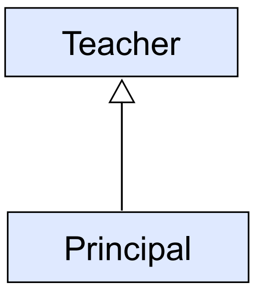


```
class  Circle
{
     int       radius;
     Point  center;   // Composition
};
```

```
class  Point
{
   int  x, y;
};
```

Inheritance

- **Inheritance Diagram:** An empty arrow points from derived class to base class, which is being extended.
- Example: Principal is-a Teacher.
  Principal class is derived from Teacher class.



```
// Base class
class  Teacher  {};
```

```
// Derived class
class  Principal  :  public  Teacher  {};
```

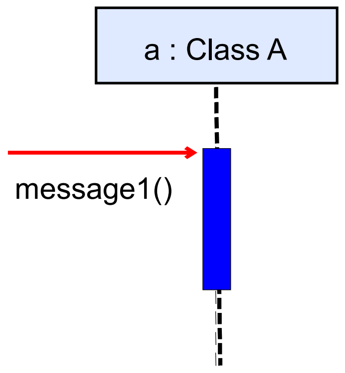
Topics

- Class Diagrams
- Sequence Diagrams

Sequence Diagrams

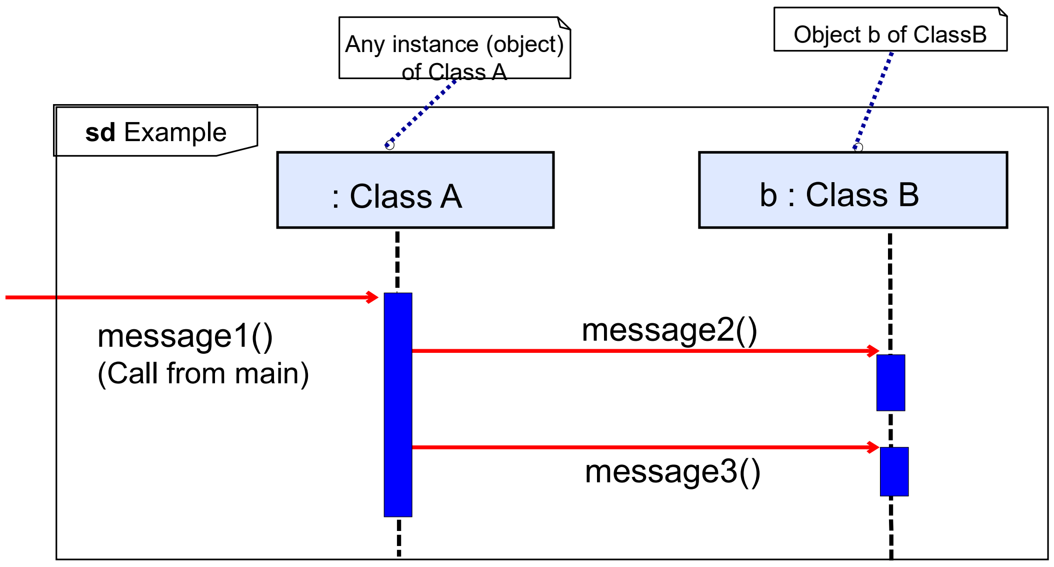
- Sequence diagrams illustrate interactions between objects.
- Horizontal axis represents the sequence ordering of messages (member functions).
- Vertical axis represents timeline chronology of objects.
- Dashed vertical line represents timeline of an object.
- Solid box (bar) over timeline of object represents a member function call of the object. It is the duration that the member function is on the system call stack.

Example:
- Class name is A.
- Object name is a.
- Messages are member functions of classes.
- In example, message1() is the member function of class A.

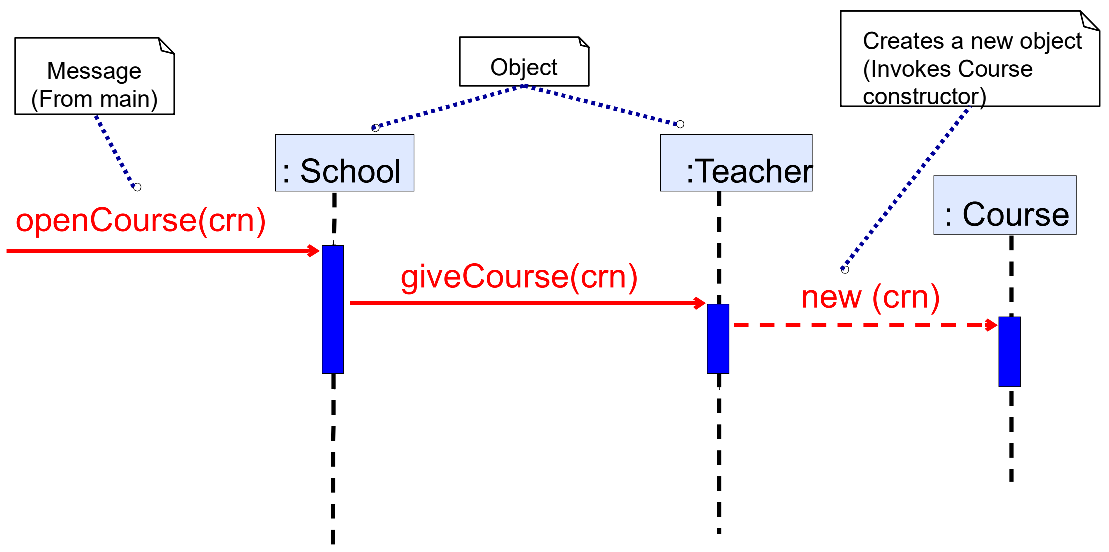


Example1: Sequence Diagram

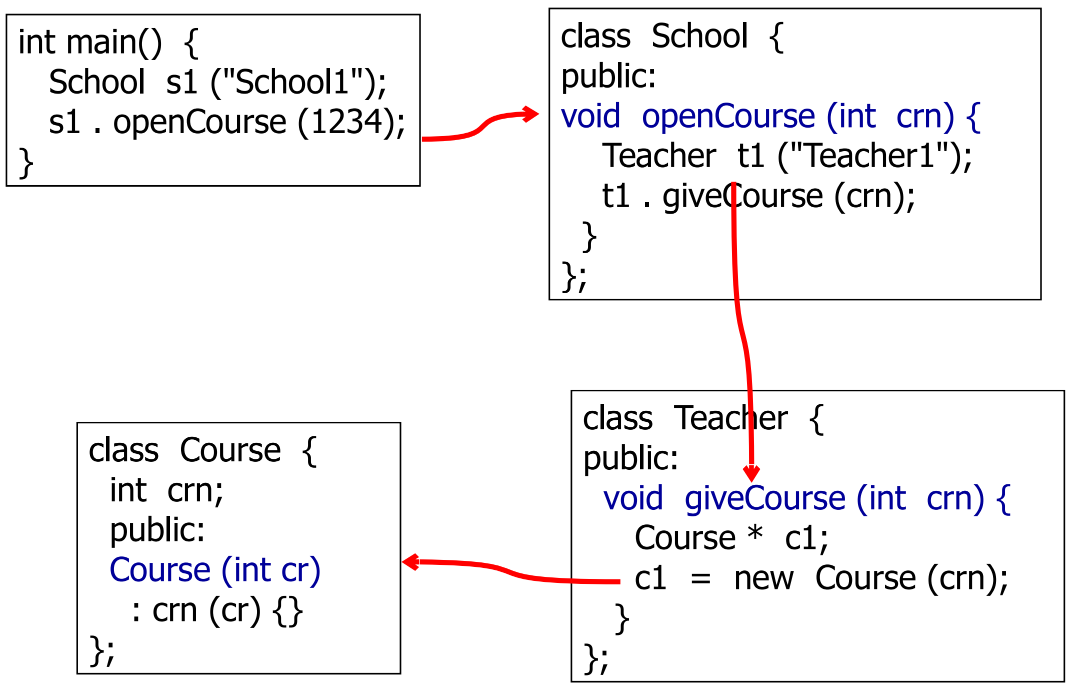
- message1 is member function of Class A.
- message2 and message3 are member functions of Class B.



Example2: Sequence Diagram



Example: Classes and Member function calls



```
int main()  {
   School  s1 ("School1");
   s1 . openCourse (1234);
}
```

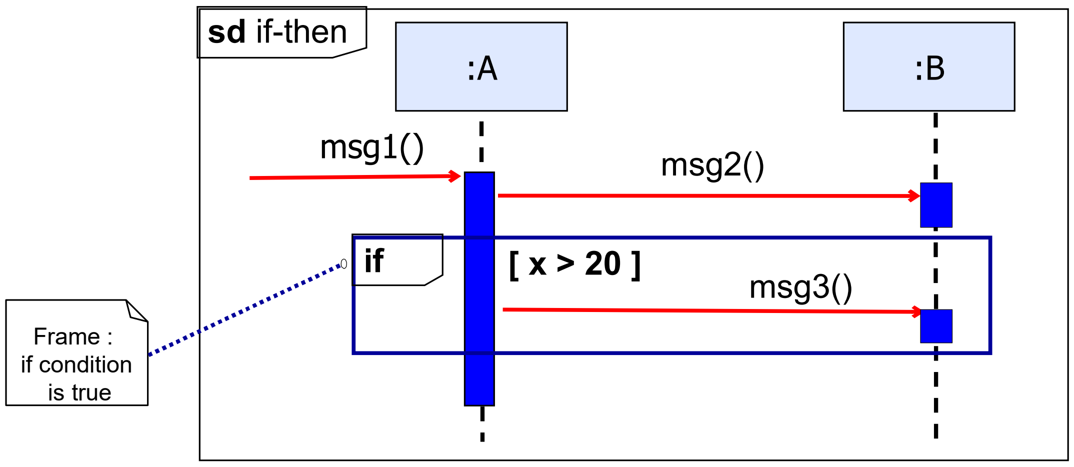
```
class  School  {
public:
void  openCourse (int  crn) {
    Teacher  t1 ("Teacher1");
    t1 . giveCourse (crn);
  }
};
```

```
class  Teacher  {
public:
  void  giveCourse (int  crn) {
     Course *  c1;
     c1  =  new  Course (crn);
   }
};
```

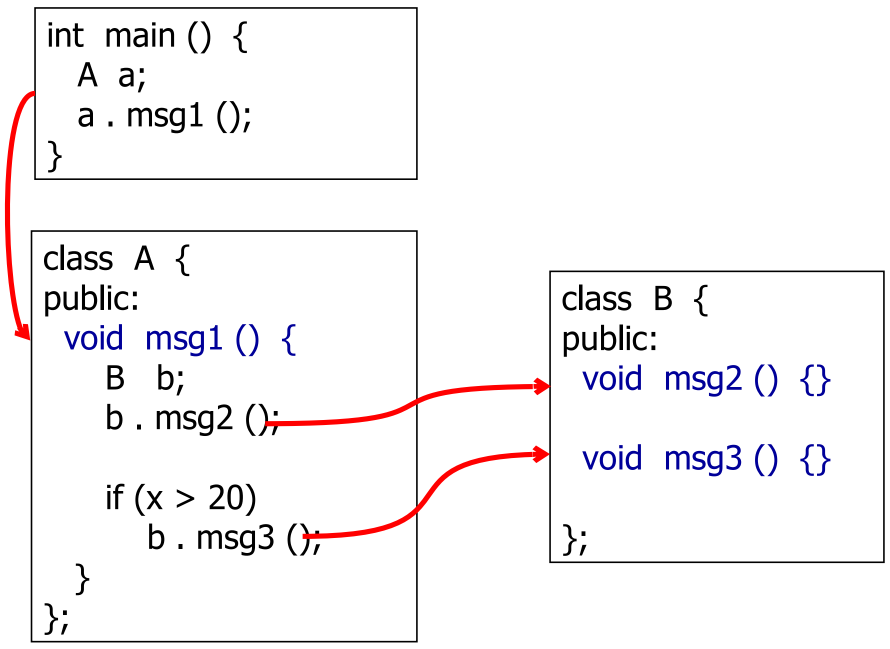
```
class  Course  {
  int  crn;
  public:
  Course (int cr)
    : crn (cr) {}
};
```

Conditional Messages (if)

- To support conditional and looping constructs, frames are used.
- Frames are regions of the diagrams;
  they have a label (such as loop or if) and a condition.
- In order to illustrate conditional messages,
  an **if** frame is placed around one or more messages.



Example: Classes and Member function calls with if



```
int  main () {
   A  a;
   a . msg1 ();
}
```

```
class  A  {
public:
  void  msg1 ()  {
      B   b;
      b . msg2 ();

      if (x > 20)
          b . msg3 ();
   }
};
```

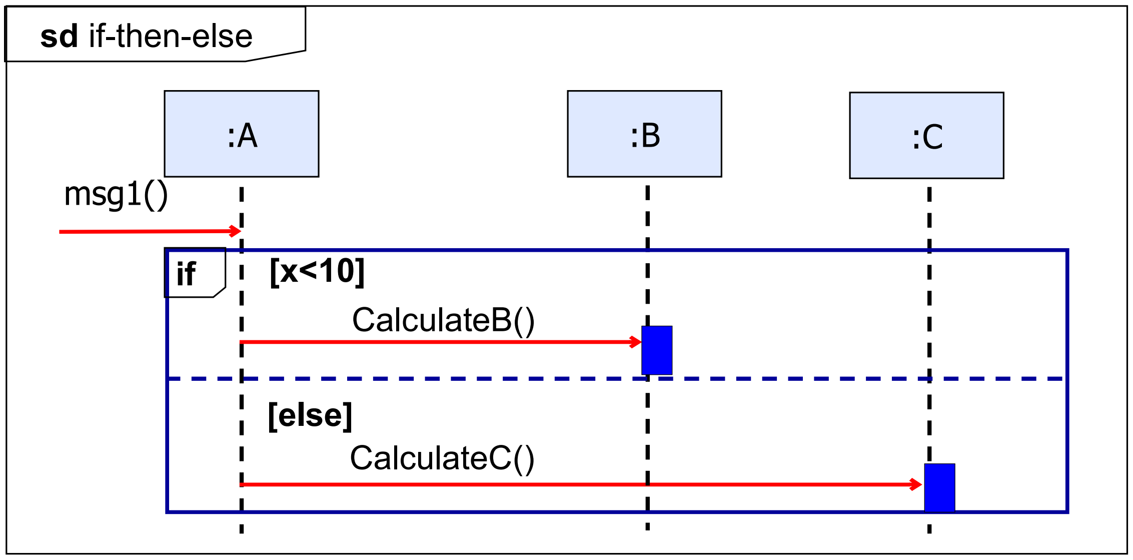
```
class  B  {
public:
  void  msg2 ()  {}

  void  msg3 ()  {}

};
```

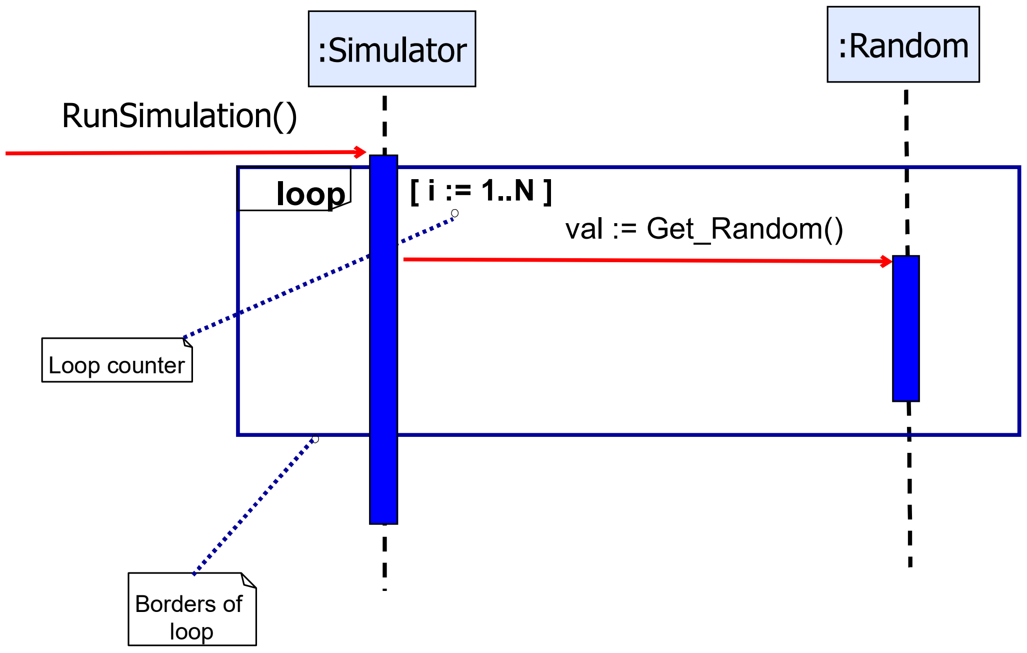
If-then-else Branches

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

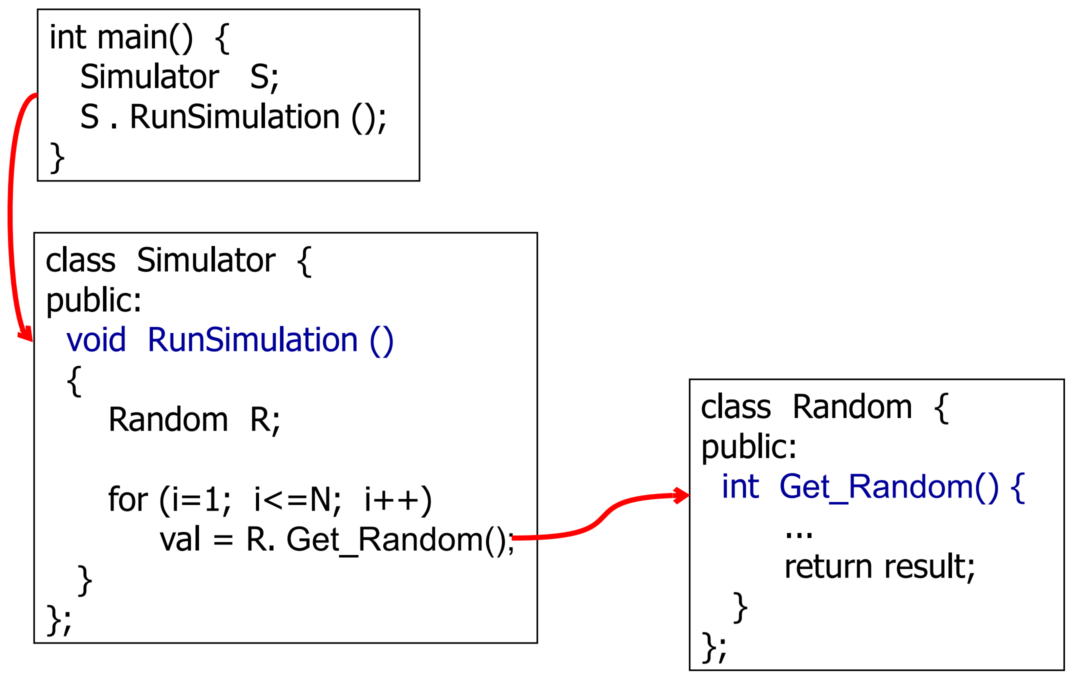


Looping

A **loop** frame is placed around the messages that belong to a loop block.



Example: Function calling with looping



```
int main()  {
   Simulator   S;
   S . RunSimulation ();
}
```

```
class  Simulator  {
public:
  void  RunSimulation ()
  {
      Random  R;

      for (i=1;  i<=N;  i++)
           val = R. Get_Random();
   }
};
```

```
class  Random  {
public:
  int  Get_Random() {
        ...
        return result;
   }
};
```