**Unit-2 : Methods and Polymorphism**

**Constructors- Types of constructors - Static constructor and Copy constructor -Destructor - Polymorphism: Constructor overloading - Method Overloading Operator Overloading - UML Interaction Diagrams -Sequence Diagram** - Collaboration Diagram - Example Diagram

**Constructors in C++:**

**Constructor in C++** is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

Constructor does not have a return value, hence they do not have a return type.

**The prototype of Constructors is as follows:**

class-name (list-of-parameters);

**Constructors can be defined inside or outside the class declaration:-**

The syntax for defining the constructor within the class:

class-name (list-of-parameters)

{

// constructor definition

}

The syntax for defining the constructor outside the class:

class-name: :class-name (list-of-parameters)

{

// constructor definition

}

**Example**

#include <iostream>

using namespace std;

class student {

    int rno;

    char name[10];

    double fee;

public:

    student()

    {

        cout << "Enter the RollNo:";

        cin >> rno;

        cout << "Enter the Name:";

        cin >> name;

        cout << "Enter the Fee:";

        cin >> fee;

    }

    void display()

    {

        cout << endl << rno << "\t" << name << "\t" << fee;

    }

};

int main()

{

    student s; // constructor gets called automatically when

               // we create the object of the class

    s.display();

    return 0;

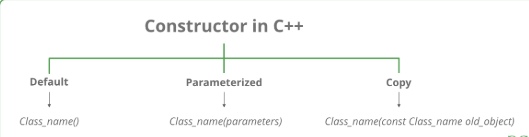
}

* Constructor has same name as the class itself
* Default Constructors don’t have input argument however, Copy and Parameterized Constructors have input arguments
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* It must be placed in public section of class.
* If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

### ****Characteristics of the constructor:****

* The name of the constructor is the same as its class name.
* Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
* Constructors do not return values; hence they do not have a return type.
* A constructor gets called automatically when we create the object of the class.
* Constructors can be overloaded.
* Constructor cannot be declared virtual.
* Constructor cannot be inherited.
* Addresses of Constructor cannot be referred.
* Constructor make implicit calls to **new**and **delete**operators during memory allocation.

### ****Types of Constructors****



[**Default Constructors:**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/) Default constructor is the constructor which doesn’t take any argument. It has no parameters. It is also called a zero-argument constructor.

**EXAMPLE 1**

// Cpp program to illustrate the

// concept of Constructors

#include <iostream>

using namespace std;

class construct {

public:

int a, b;

// Default Constructor

construct()

{

a = 10;

b = 20;

}

};

int main()

{

// Default constructor called automatically

// when the object is created

construct c;

cout << "a: " << c.a << endl << "b: " << c.b;

return 1;

}

**Output**

a: 10

b: 20

**EXAMPLE 2**

#include<iostream>

using namespace std;

class student

{

int rno;

char name[50];

double fee;

public:

student() // Explicit Default constructor

{

cout<<"Enter the RollNo:";

cin>>rno;

cout<<"Enter the Name:";

cin>>name;

cout<<"Enter the Fee:";

cin>>fee;

}

void display()

{

cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

}

};

int main()

{

student s;

s.display();

return 0;

}

**2. Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

**EXAMPLE 1**

// CPP program to illustrate

// parameterized constructors

#include <iostream>

using namespace std;

class Point {

private:

int x, y;

public:

// Parameterized Constructor

Point(int x1, int y1)

{

x = x1;

y = y1;

}

int getX() { return x; }

int getY() { return y; }

};

int main()

{

// Constructor called

Point p1(10, 15);

// Access values assigned by constructor

cout << "p1.x = " << p1.getX()

<< ", p1.y = " << p1.getY();

return 0;

}

**Output**

p1.x = 10, p1.y = 15

**EXAMPLE 2**

#include<iostream>

#include<string.h>

using namespace std;

class student

{

int rno;

char name[50];

double fee;

public:

student(int,char[],double);

void display();

};

student::student(int no,char n[],double f)

{

rno=no;

strcpy(name,n);

fee=f;

}

void student::display()

{

cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

}

int main()

{

student s(1001,"Ram",10000);

s.display();

return 0;

}

* **Uses of Parameterized constructor:**
  1. It is used to initialize the various data elements of different objects with different values when they are created.
  2. It is used to overload constructors.
* **Can we have more than one constructor in a class?**  
         Yes, It is called [Constructor Overloading](https://www.geeksforgeeks.org/constructor-overloading-c/).

### ****3. Copy Constructor:****

A copy constructor is a member function that initializes an object using another object of the same class. A detailed article on [Copy Constructor](https://www.geeksforgeeks.org/copy-constructor-in-cpp/).

Whenever we define one or more non-default constructors( with parameters ) for a class, a default constructor( without parameters ) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

Copy constructor takes a reference to an object of the same class as an argument.

Sample(Sample &t)

{

id=t.id;

}

**// Example: copy constructor**

#include <iostream>

using namespace std;

class Sample

{

int id;

public:

void init(int x)

{

id=x;

}

Sample(){} //default constructor with empty body

Sample(Sample &t) //copy constructor

{

id=t.id;

}

void display()

{

cout<<endl<<"ID="<<id;

}

};

int main()

{

Sample obj1;

obj1.init(10);

obj1.display();

Sample obj2(obj1); copy constructor called

obj2.display();

return 0;

}

**Output**

ID=10

ID=10

**EXAMPLE:**

#include<iostream>

#include<string.h>

using namespace std;

class student

{

int rno;

char name[50];

double fee;

public:

student(int,char[],double);

student(student &t) //copy constructor

{

rno=t.rno;

strcpy(name,t.name);

fee=t.fee;

}

void display();

};

student::student(int no,char n[],double f)

{

rno=no;

strcpy(name,n);

fee=f;

}

void student::display()

{

cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

}

int main()

{

student s(1001,"Manjeet",10000);

s.display();

student manjeet(s); //copy constructor called

manjeet.display();

return 0;

}

OUTPUT

1001 Manjeet 10000

1001 Manjeet 10000

**//copy constructor (member wise initialization)**

#include<iostream>

#include<string.h>

using namespace std;

class student

{

int rno;

char name[50];

double fee;

public:

student(int,char[],double);

student(student &t) //copy constructor (member wise initialization)

{

rno=t.rno;

strcpy(name,t.name);

}

void display();

void disp()

{

cout<<endl<<rno<<"\t"<<name;

}

};

student::student(int no, char n[],double f)

{

rno=no;

strcpy(name,n);

fee=f;

}

void student::display()

{

cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;

}

int main()

{

student s(1001,"Manjeet",10000);

s.display();

student manjeet(s); //copy constructor called

manjeet.disp();

return 0;

}

**OUTPUT**

1001 Manjeet 10000

1001 Manjeet

### Destructor:

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope.  Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

The syntax for defining the destructor within the class

~ class-name()

{

}

The syntax for defining the destructor outside the class

class-name: : ~ class-name()

{

}

**EXAMPLE**

#include <iostream>

using namespace std;

class Test {

public:

Test() { cout << "\n Constructor executed"; }

~Test() { cout << "\n Destructor executed"; }

};

main()

{

Test t;

return 0;

}

**Output**

Constructor executed

Destructor executed

**EXAMPLE 2:**

#include <iostream>

using namespace std;

class Test {

public:

Test() { cout << "\n Constructor executed"; }

~Test() { cout << "\n Destructor executed"; }

};

main()

{

Test t, t1, t2, t3;

return 0;

}

**Output**

Constructor executed

Constructor executed

Constructor executed

Constructor executed

Destructor executed

Destructor executed

Destructor executed

Destructor executed

**EXAMPLE 3**

#include <iostream>

using namespace std;

int count = 0;

class Test {

public:

Test()

{

count++;

cout << "\n No. of Object created:\t" << count;

}

~Test()

{

cout << "\n No. of Object destroyed:\t" << count;

--count;

}

};

main()

{

Test t, t1, t2, t3;

return 0;

}

**Output**

No. of Object created: 1

No. of Object created: 2

No. of Object created: 3

No. of Object created: 4

No. of Object destroyed: 4

No. of Object destroyed: 3

No. of Object destroyed: 2

No. of Object destroyed: 1

### Characteristics of a destructor:-

1. Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.  
2. Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.  
3. Destructor  cannot be declared as static and const;  
4. Destructor should be declared in the public section of the program.  
5. Destructor is called in the reverse order of its constructor invocation.

**static constructor**

A static constructor is used to initialize static data of a class. C++ doesn't have static constructor. But a static constructor can be emulated by using a friend class or nested class as below.

### What’s a Static Constructor?

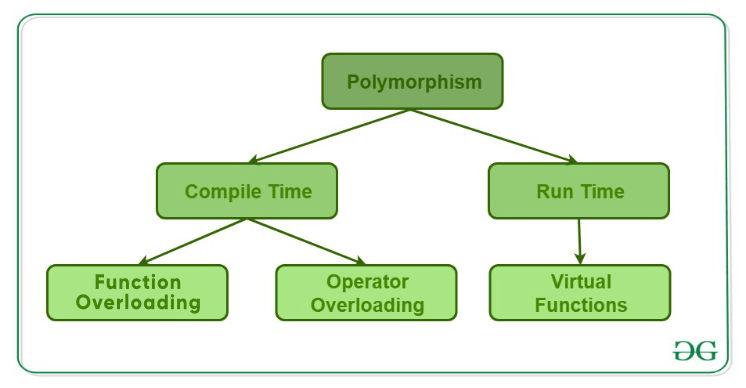
A static constructor is also called a type constructor. Like the ordinary kind of constructor, it’s called “behind-the-scenes” by the compiler, and can’t be called by you. But, instead of being called whenever an instance of your class is created (by a stack allocation, a heap allocation with new, or when some temporary objects are created for function parameters or return values), a static constructor is called the first time a static member variable in your class is used. Then, it is never called again. Its purpose is to initialize your static member variables.

**C++ Polymorphism**

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered one of the important features of Object-Oriented Programming.

## Types of Polymorphism

* **Compile-time Polymorphism.**
* **Runtime Polymorphism.**



## 1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

### A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,**hence this is known as Function Overloading. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions to have the same name but distinct parameters when numerous tasks are listed under one function name. There are certain [Rules of Function Overloading](https://www.geeksforgeeks.org/cpp-polymorphism/) that should be followed while overloading a function.

Below is the C++ program to show function overloading or compile-time polymorphism:

#include <bits/stdc++.h>

using namespace std;

class Geeks {

public:

 // Function with 1 int parameter

void func(int x)

    {

        cout << "value of x is " <<

                 x << endl;

    }

// Function with same name but

    // 1 double parameter

    void func(double x)

    {

        cout << "value of x is " <<

                 x << endl;

    }

 // Function with same name and

    // 2 int parameters

    void func(int x, int y)

    {

        cout << "value of x and y is " <<

                 x << ", " << y << endl;

    }

};

int main()

{

    Geeks obj1;

    // Function being called depends

    // on the parameters passed

    // func() is called with int value

    obj1.func(7);

    // func() is called with double value

    obj1.func(9.132);

    // func() is called with 2 int values

    obj1.func(85, 64);

    return 0;

}

**Output**

value of x is 7

value of x is 9.132

value of x and y is 85, 64

**Explanation:**In the above example, a single function named function **func()** acts differently in three different situations, which is a property of polymorphism.

### B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator ‘+’, when placed between integer operands, adds them and when placed between string operands, concatenates them.

**Unary Operator Overloading**

Unary operator overloading in C++ is polymorphism in which we overload an operator to perform a similar operation with the class objects. We do operator overloading to give operators a user-defined meaning, which is utilized for user-defined data type (object of a class) operations

#include <iostream>

using namespace std;

class unary

{

int x,y,z;

public:

void get(int a,int b,int c);

void disp();

void operator-();

};

void unary::get(int a,int b,int c)

{

x=a;

y=b;

z=c;

}

void unary::disp()

{

cout<<x<<endl;

cout<<y<<endl;

cout<<z<<endl;

}

void unary::operator-()

{

x=-x;

y=-y;

z=-z;

}

int main()

{

unary u;

u.get(10,20,-30);

u.disp();

-u;

u.disp();

return 0;

}

**OUTPUT:**

10

20

-30

-10

-20

30

**Unary operator overloading in matrix:**

#include <iostream>

using namespace std;

class matrix

{

int a[5][5],m,n,i,j;

public:

matrix()

{

}

void get()

{

cout<<"no of rows"<<endl;

cin>>m;

cout<<"no of cols"<<endl;

cin>>n;

for(i=0;i<m;i++)

{

for(j=0;j<n;j++)

{

cin>>a[i][j];

}

}

}

void disp()

{

for(i=0;i<m;i++)

{

for(j=0;j<n;j++)

{

cout<<a[i][j];

}

cout<<endl;

}

}

void operator-()

{

for(i=0;i<m;i++)

{

for(j=0;j<n;j++)

{

a[i][j]=-a[i][j];

}

}

}

};

int main()

{

matrix m;

m.get();

cout<<"the given matrix is"<<endl;

m.disp();

-m;

cout<<"resultant matrix"<<endl;;

m.disp();

return 0;

}

**OUTPUT:**

no of rows

3

no of cols

3

1 2 3 4 5 6 7 8 9

the given matrix is

123

456

789

resultant matrix

-1-2-3

-4-5-6

-7-8-9

**OVERLOADING ASSIGNMENT OPERATOR**

#include <iostream>

using namespace std;

class assign

{

int a;

public:

assign()

{

}

assign(int x)

{

a=x;

}

void operator=(assign);

void disp()

{

cout<<"the value of a is"<<a;

}

};

void assign::operator=(assign as)

{

a=as.a;

}

int main()

{

assign a1(10);

assign a2;

a2=a1;

a2.disp();

return 0;

}

**OUTPUT**

**the value of a is10**

**OVERLOADING SUBSCRIPT[] OPERATOR(UNARY)**

#include <iostream>

using namespace std;

class oversub

{

int a[5];

public:

void operator[](int i);

};

void oversub::operator[](int i)

{

if(i>0 && i<4)

{

cout<<"valid subscript"<<endl;

}

else

{

cout<<"subscript out of range"<<endl;

}

}

int main()

{

oversub s;

s[2];

s[7];

return 0;

}

**OUTPUT**

**valid subscript**

**subscript out of range**

**OVERLOADING FUNCTION CALL () OPERATOR (UNARY)**

#include <iostream>

using namespace std;

class overfun

{

public:

void operator()(int v)

{

cout<<(v<10?"yes":"no");

}

};

int main()

{

overfun of;

of(10);

of(-5);

return 0;

}

OUTPUT

No

Yes

**OVERLOADING INCREMENT OPERATOR (UNARY)-PREFIX**

#include <iostream>

using namespace std;

class incr

{

int count;

public:

incr()

{

count=5;

}

void operator ++()

{

++count;

cout<<count;

}

};

int main()

{

incr i;

++i;

return 0;

}

OUTPUT

6

**OVERLOADING DECREMENT OPERATOR (UNARY)-PREFIX**

#include <iostream>

using namespace std;

class dec

{

int count;

public:

dec()

{

count=5;

}

void operator --()

{

--count;

cout<<count;

}

};

int main()

{

dec i;

--i;

return 0;

}

**OUTPUT**

6

**OVERLOADING INCREMENT OPERATOR (UNARY)-POSTFIX**

#include <iostream>

using namespace std;

class Check

{

private:

int i;

public:

Check()

{

i=0;

}

// Notice int inside barcket which indicates postfix increment.

Check operator ++ (int)

{

Check temp;

temp.i = i++;

return temp;

}

void Display()

{

cout << "i = "<< i <<endl;

}

};

int main()

{

Check obj,obj1;

obj.Display();

// Assigns value of obj to obj1, only then operator function is called.

obj1 = obj++;

obj.Display();

obj1.Display();

return 0;

}

**OUTPUT**

i = 0

i = 1

i = 0

**OVERLOADING DECREMENT OPERATOR (UNARY)-POSTFIX**

#include <iostream>

using namespace std;

class Check

{

private:

int i;

public:

Check()

{

i=2;

}

// Notice int inside barcket which indicates postfix increment.

Check operator -- (int)

{

Check temp;

temp.i = i--;

return temp;

}

void Display()

{

cout << "i = "<< i <<endl;

}

};

int main()

{

Check obj,obj1;

obj.Display();

// Assigns value of obj to obj1, only then operator function is called.

obj1 = obj--;

obj.Display();

obj1.Display();

return 0;

}

**OUTPUT**

i = 2

i = 1

i = 2

**OVERLOADING BINARY OPERATOR**

An operator which contains two operands to perform a mathematical operation is called the Binary Operator Overloading. It is a polymorphic compile technique where a single operator can perform various functionalities by taking two operands from the programmer or user.

#include <iostream>

using namespace std;

class comp{

float real;

float imag;

public:

comp()

{

real=imag=0;

}

void get()

{

cout<<"enter the real part"<<endl;

cin>>real;

cout<<"enter the imag part"<<endl;

cin>>imag;

}

void disp()

{

cout<<real<<"+i"<<imag<<endl;

}

comp operator +(comp);

comp operator -(comp);

};

comp comp::operator +(comp c)

{

comp temp;

temp.real=real+c.real;

temp.imag=imag+c.imag;

return temp;

}

comp comp::operator -(comp c)

{

comp temp;

temp.real=real-c.real;

temp.imag=imag-c.imag;

return temp;

}

int main()

{

comp c1,c2,c3;

c1.get();

c2.get();

c3=c1+c2;

c3.disp();

return 0;

}

**OUTPUT**

enter the real part

2

enter the imag part

3

enter the real part

3

enter the imag part

2

5+i5

**The list of operators which can be overloaded:**

* Arithmetic Operators like %, +, -, \*, /
* Unary Operators like ++, --,!
* Relational Operators like ==, !=, >=, <=
* Logical operators like && and ||
* Assignment operators like =, +=,\*=, /=,-=, %=
* Subscript operator [ ]
* Bitwise operators like &, |, <<, >>, ^, ~

## List of Operators That Cannot Be Overloaded in C++

The list of operators which cannot be overloaded is as follows:

* **Conditional or Ternary Operator** (?:) cannot be overloaded.
* **Size of Operator** (sizeof) cannot be overloaded.
* **Scope Resolution Operator** (::) cannot be overloaded.
* **Class member selector Operator** (.) cannot be overloaded.
* **Member pointer selector Operator** (.\*) cannot be overloaded.
* **Object type Operator** (typeid) cannot be overloaded.

Among the **in-built** operators which operators cannot be overloaded using the friend function but can be overloaded by member functions are as follows:

* Assignment Operator (=)

Function call Operator (())

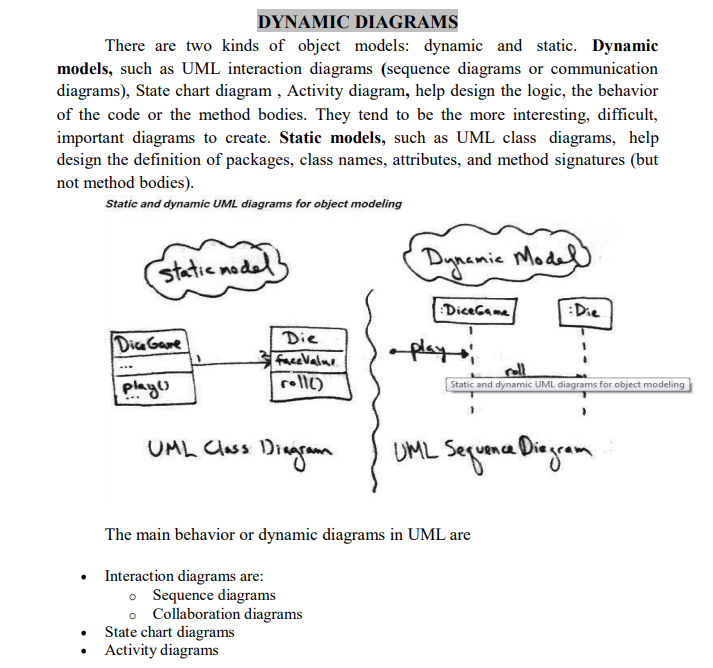
Subscript Operator ([])

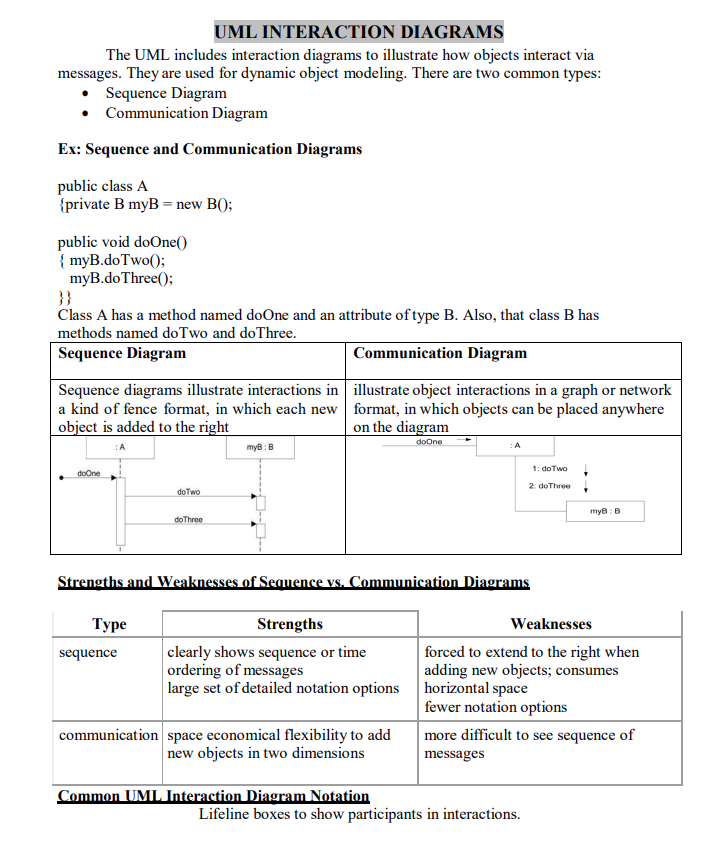
Arrow Operator (->)

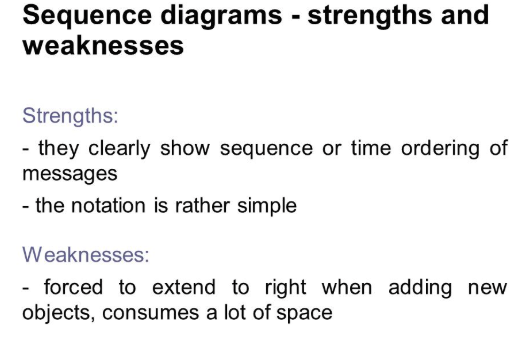
**UML INTERACTION DIAGRAM**

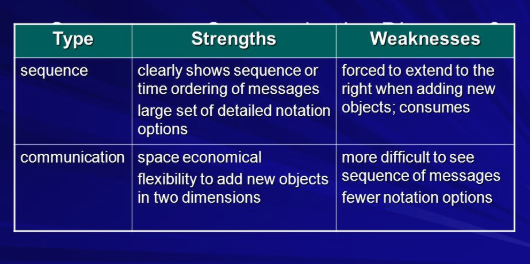
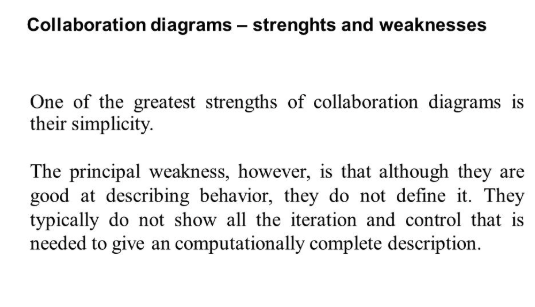
[**UML**](https://en.wikipedia.org/wiki/Unified_Modeling_Language)**Sequence Diagrams** are interaction diagrams that detail how operations are carried out. They capture the interaction between objects in the context of a collaboration. Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when.

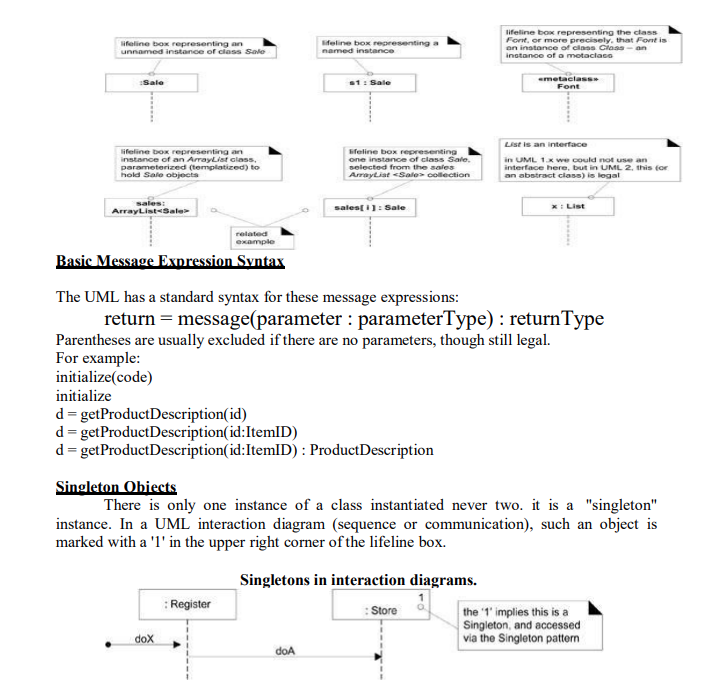
A **collaboration diagram**, also known as a communication diagram, is an illustration of the relationships and interactions among software [objects](https://www.techtarget.com/searchapparchitecture/definition/object) in the Unified Modeling Language ([UML](https://www.techtarget.com/searchsoftwarequality/definition/Unified-Modeling-Language)). These diagrams can be used to portray the dynamic behavior of a particular [use case](https://www.techtarget.com/searchsoftwarequality/definition/use-case) and define the role of each object.

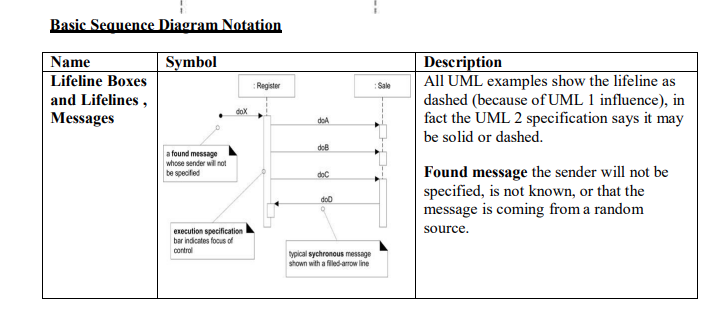


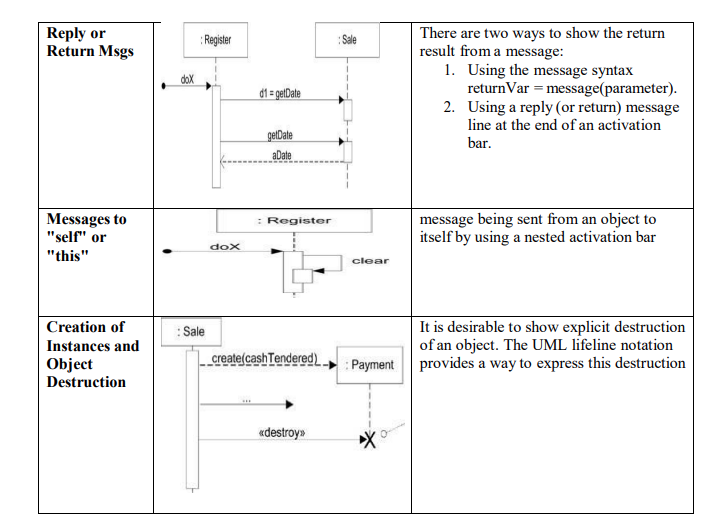


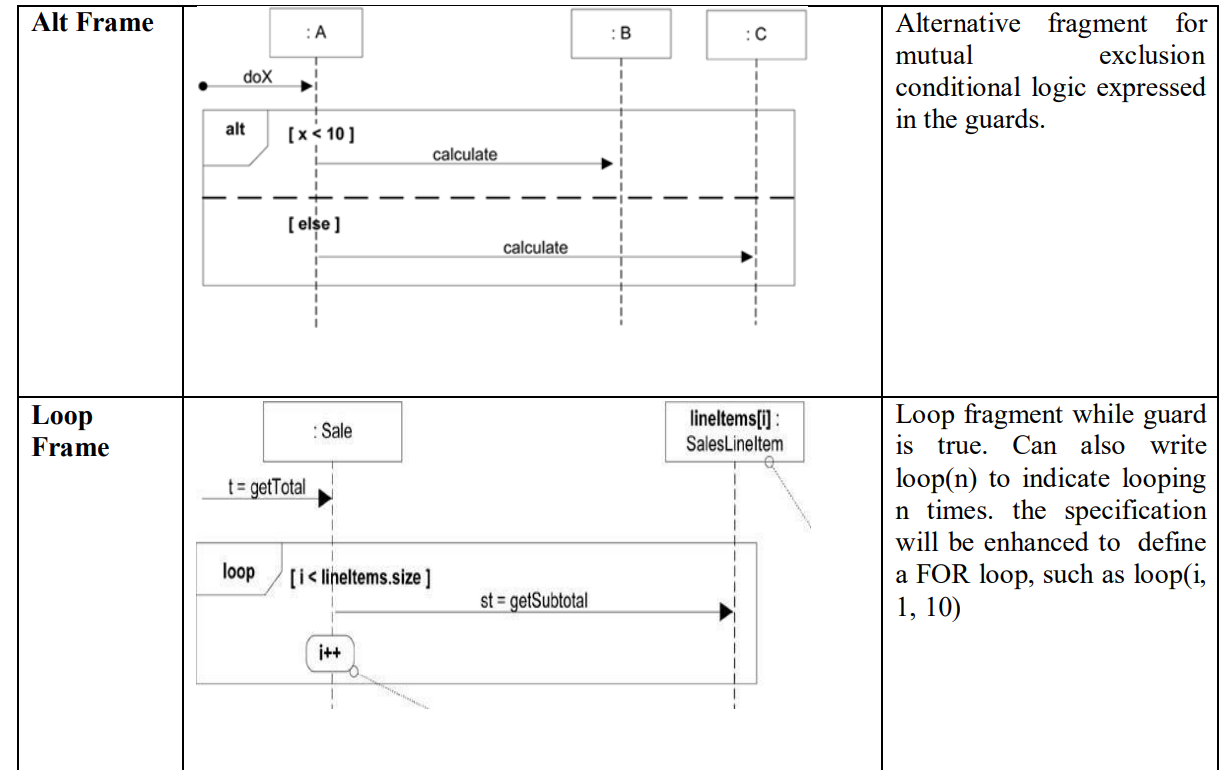
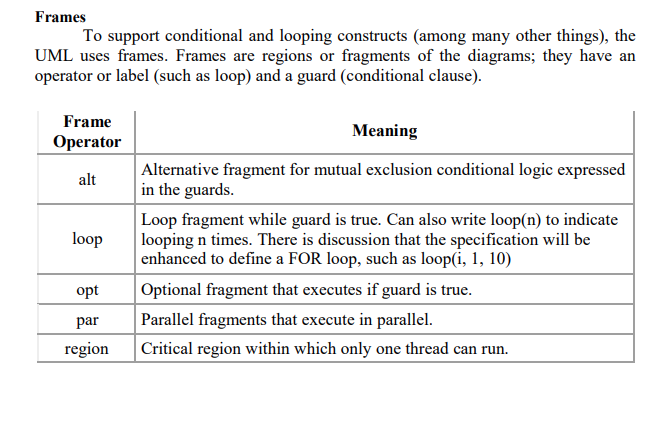


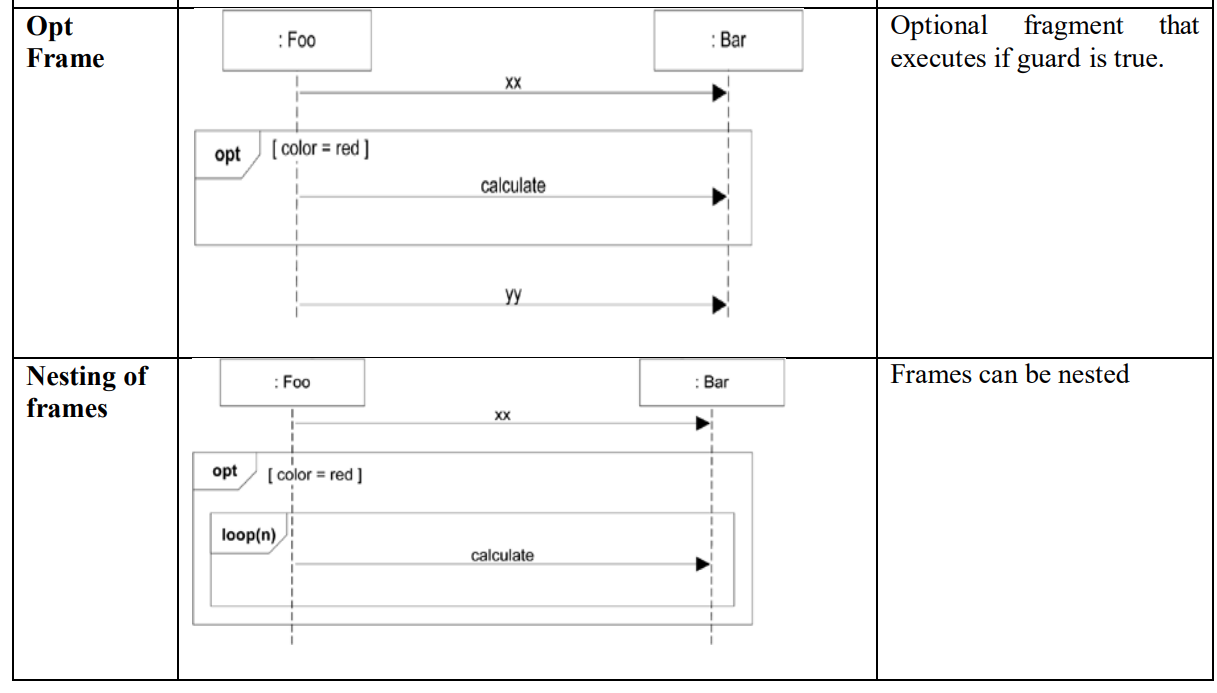


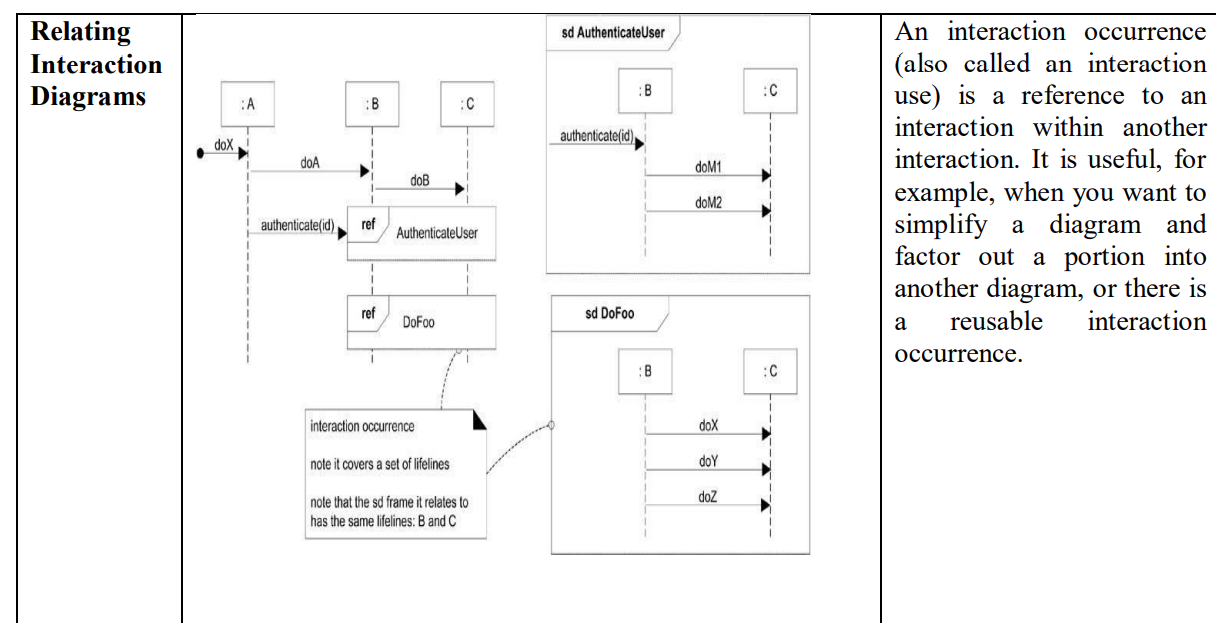


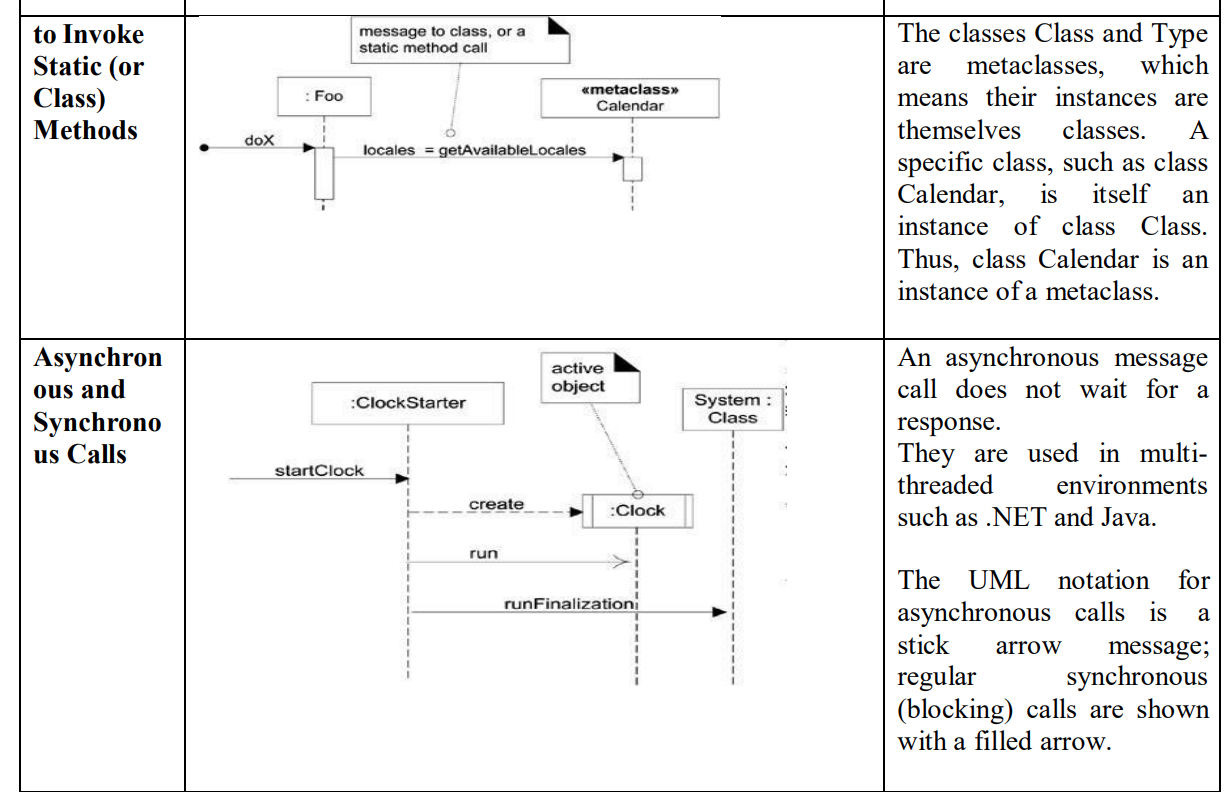


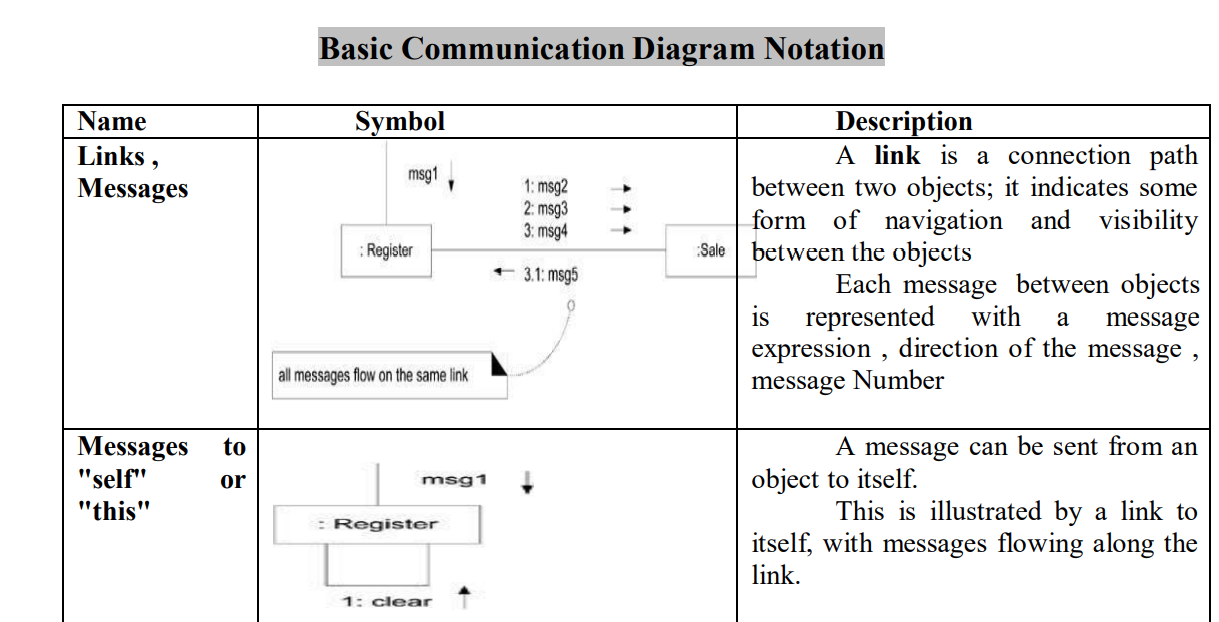


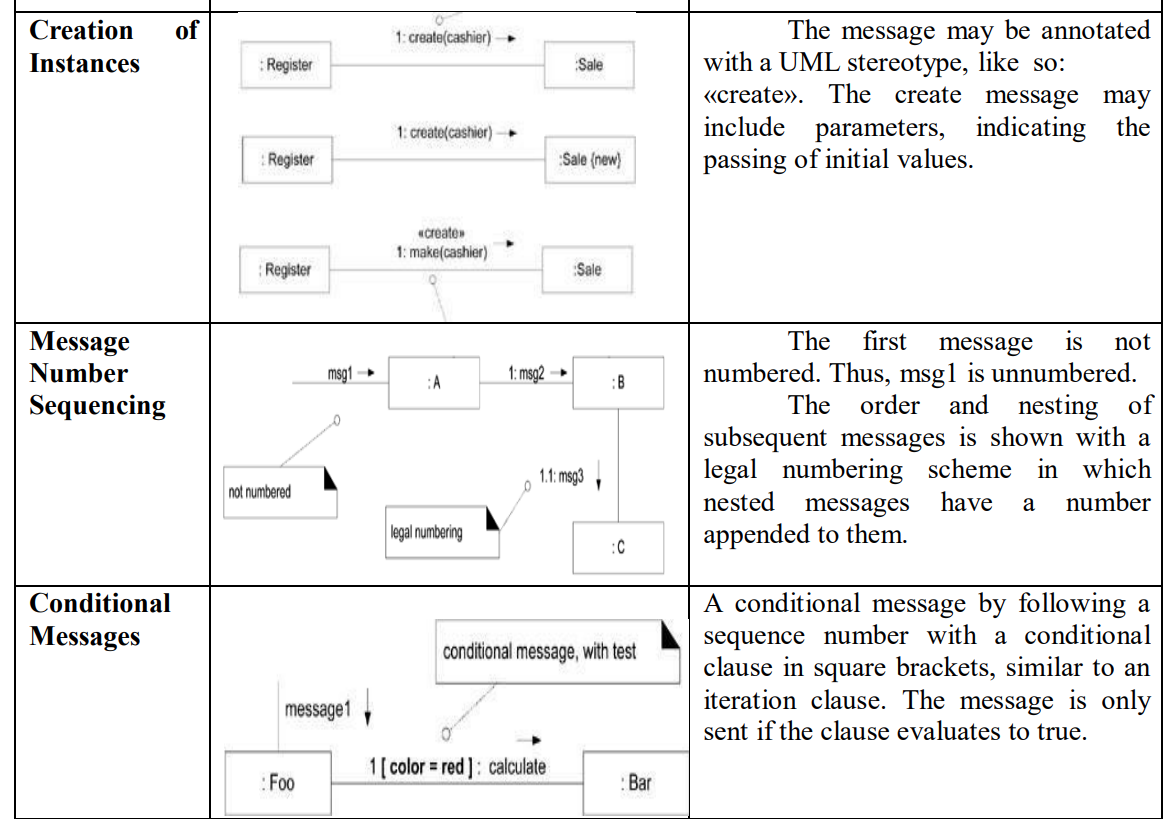


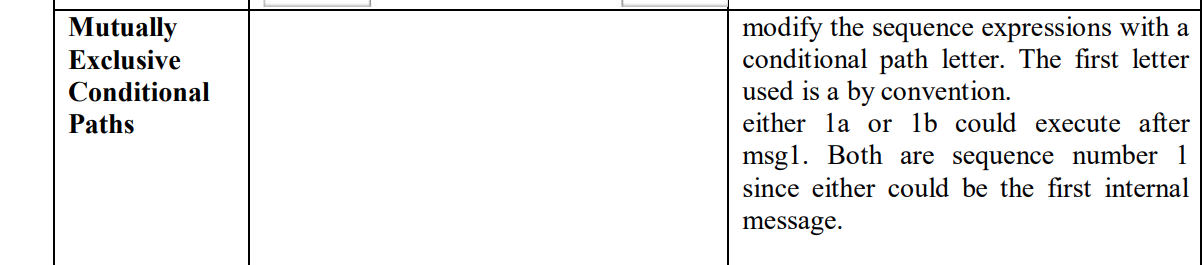


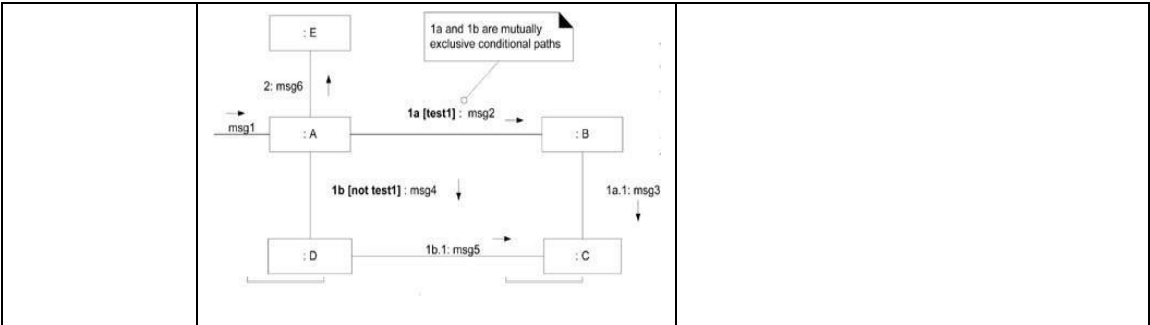


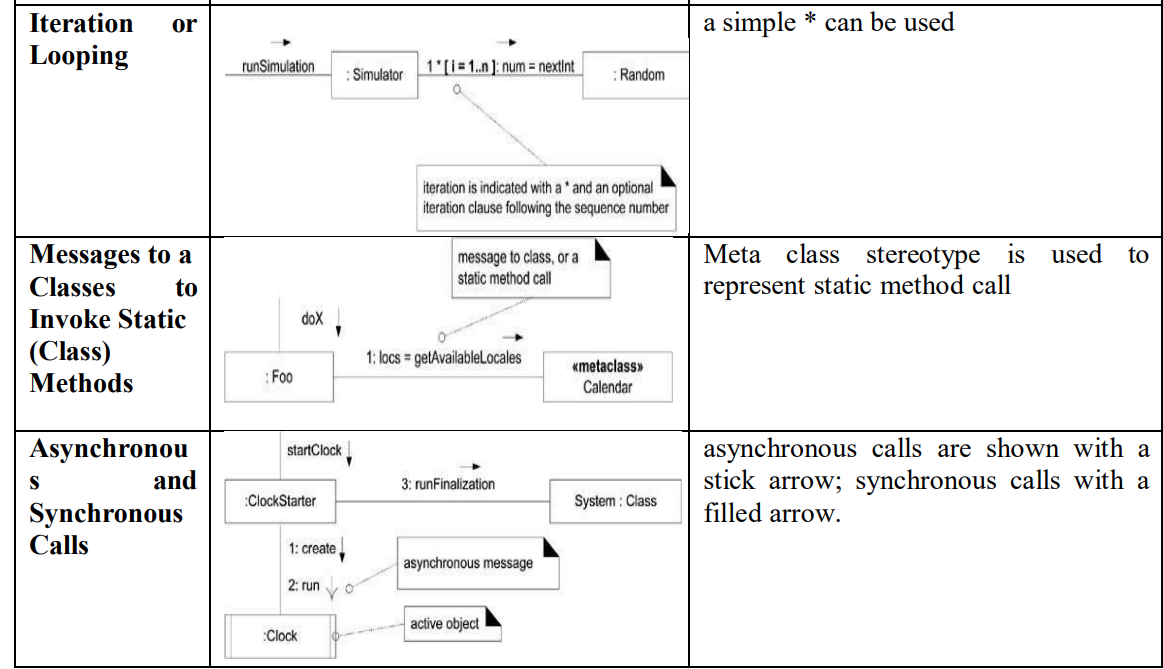


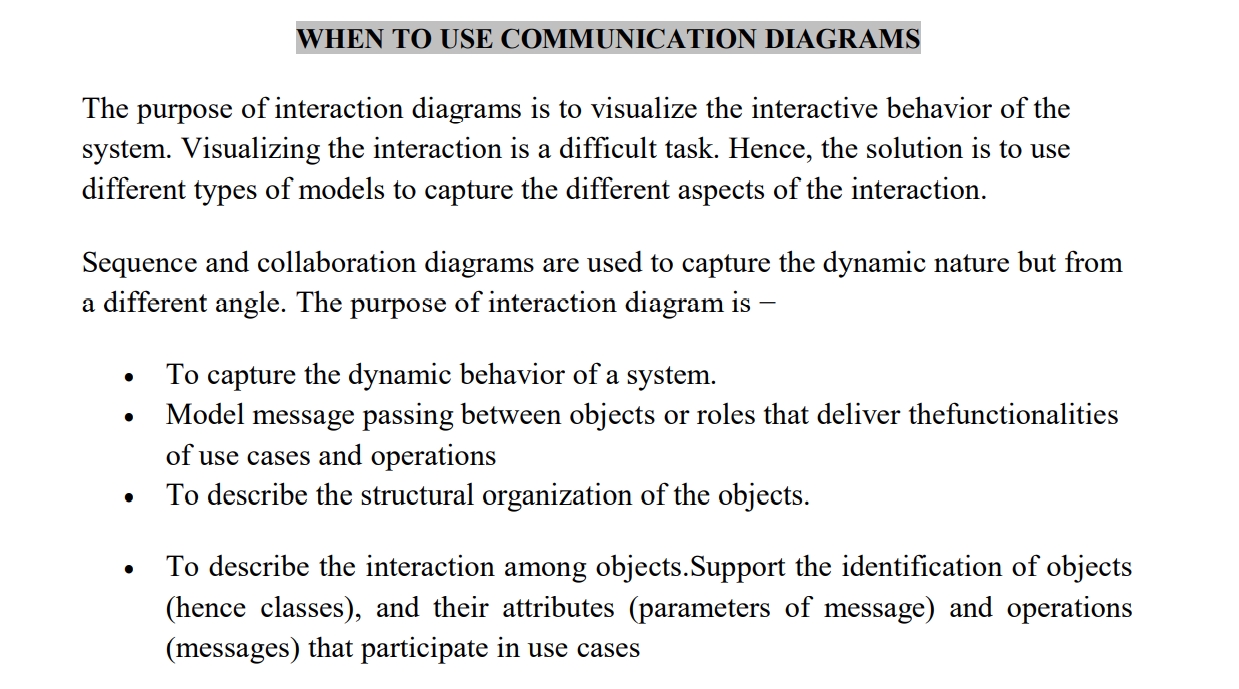




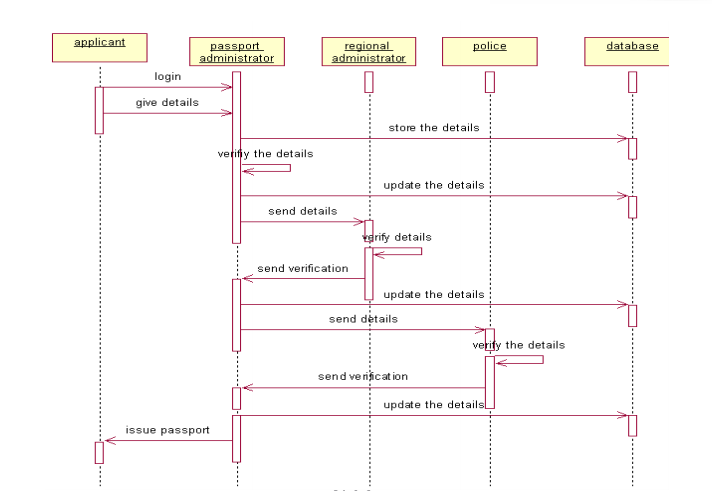




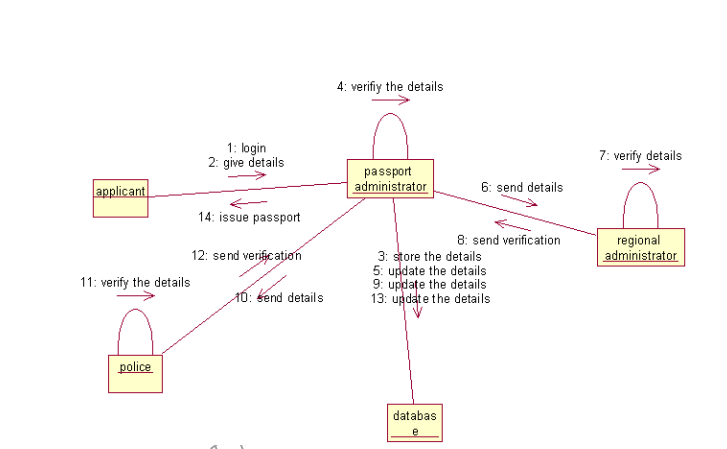




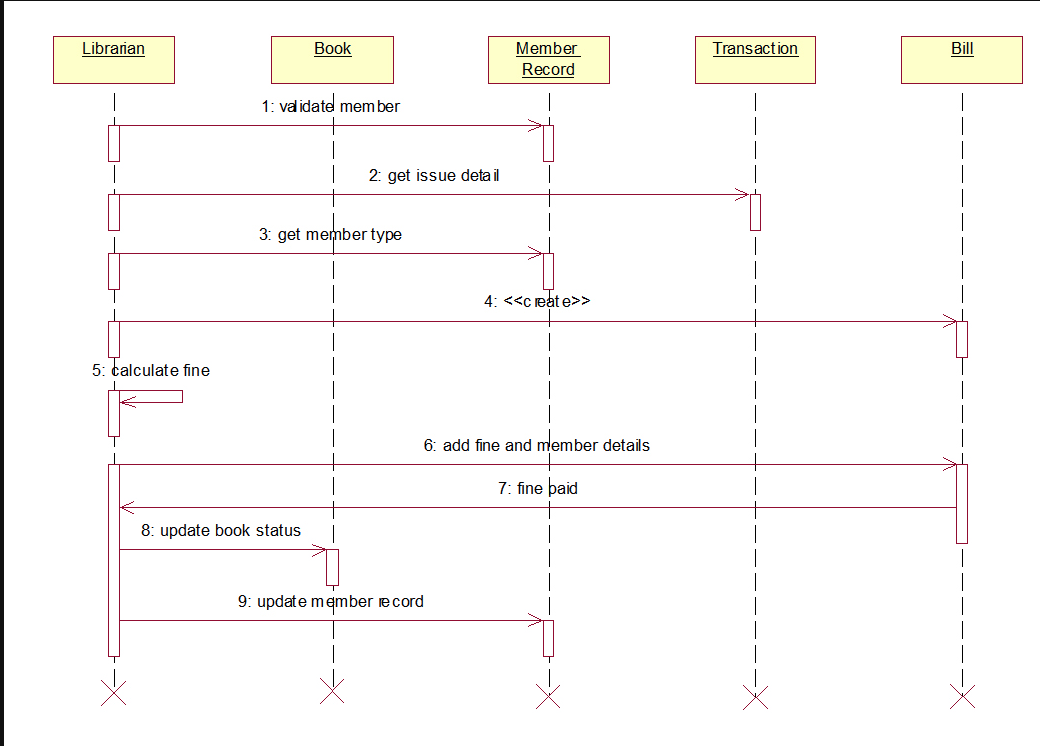
**SEQUENCE DIAGRAM FOR PASSPORT AUTOMATION SYSTEM**



**COLLABORATION DIAGRAM FOR PASSPORT AUTOMATION SYSTEM**



**SEQUENCE DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM**



**COLLABORATION DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM**

