STARK REPAIRS CUSTOMER SERVICE SIMULATION

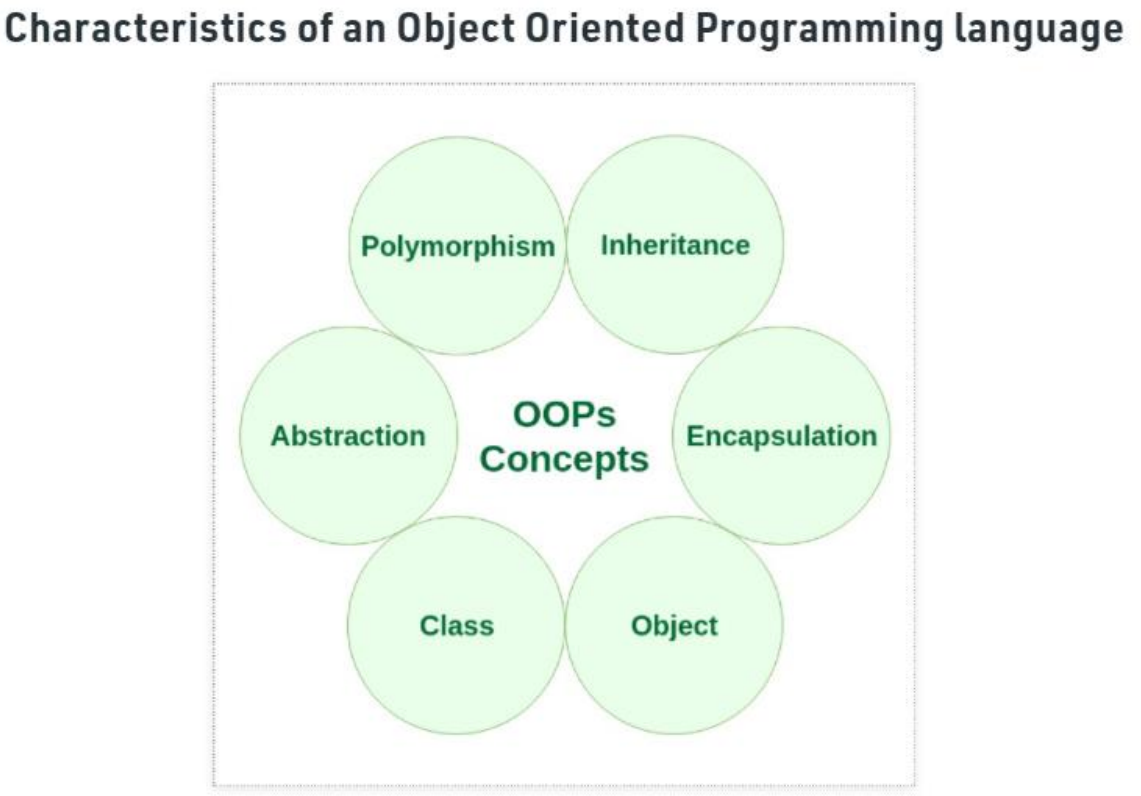
A software simulation to avail customer service from Stark Repairs. The software provides repairing services for different electronic items. It has login and signup pages, query and feedback options and a payment gateway. it also generates a receipt with all of user's personal information, payment and servicing details.

Details of the Project:

1. **Class Startup**: Encapsulates all the classes and runs the simulation.
2. **Class Signup**: associated with Startup class
   Inputs the personal details, email ID and password from the user.
   Class Login: associated with Startup Class and inherits from Signup Class.
   Accepts the already signed up email ID and password.
3. **Class Disp_Services**
   Displays the available services
4. **Class Choose_Service**
   Gives the option to choose the services required by the user.
5. **Class Service_Details**
   Inputs the user's address and preferred date and time for the service.
6. **Class Payment**
   Gives the user 4 modes of payment.
7. **Class Query**
   Inputs the query of the use, if any.
8. **Class End**
   Displays the Log out message.

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

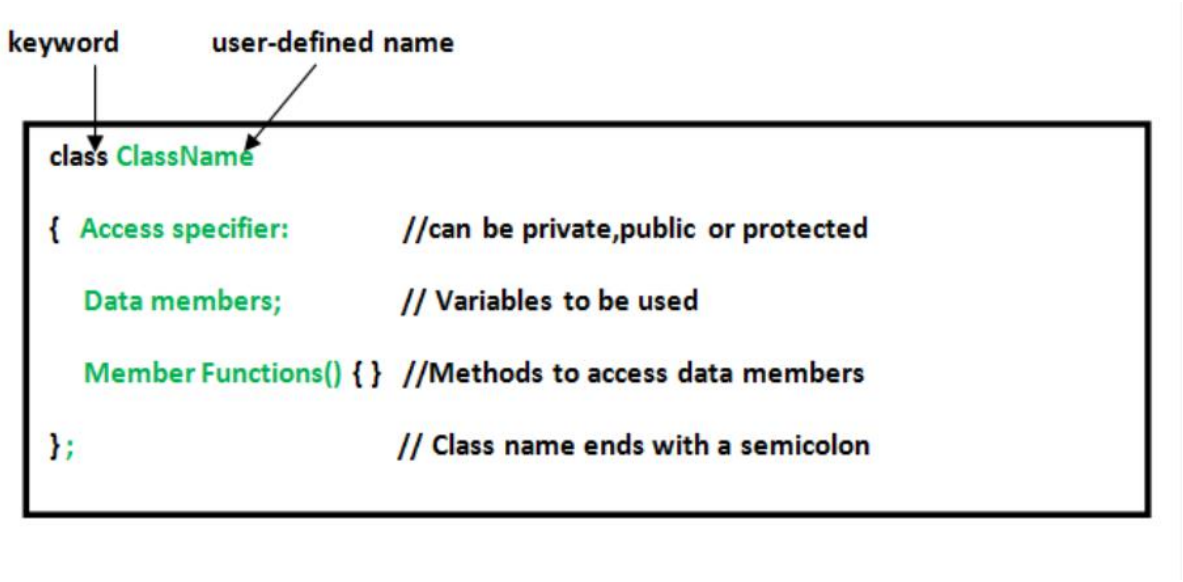


CLASS

A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

A Class is a user defined data-type which has data members and member functions.

Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behaviour of the objects in a Class.



OBJECT:

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class:

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

Declaring Objects:

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Accessing Data Members:

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.
This access control is given by Access modifiers in C++. There are three access modifiers:

1. **Public:** All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
2. **Private:** The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.
3. **Protected:** Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of its class unless with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

INHERITANCE

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Modes of Inheritance

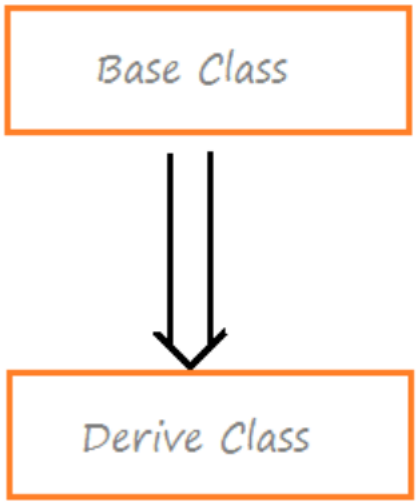
1. **Public mode**
   If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**
   If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**
   If we derive a sub class from a private base class. Then both public member and protected members of the base class will become Private in derived class.

Access in Base Class	Base Class Inherited as	Access in Derived Class
Public	Public	Public
Protected		Protected
Private		No Access
Public	Protected	Protected
Protected		Protected
Private		No Access
Public	Private	Private
Protected		Private
Private		No Access

Types of Inheritance

1. Single Inheritance

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Single Inheritance

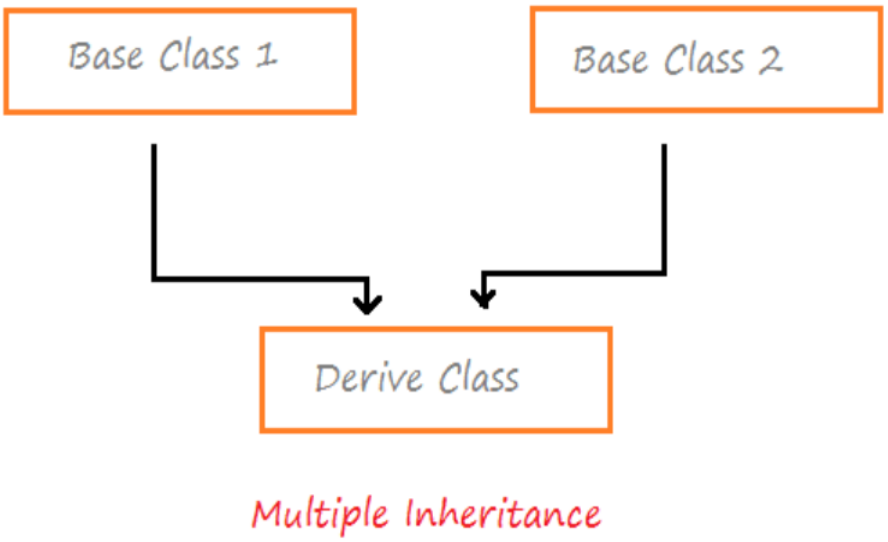
Syntax:

```
class Base {
   //class members
};

class Derive : <access_specifier> Base {
   //class members
};
```

2. Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.



Multiple Inheritance

Syntax:

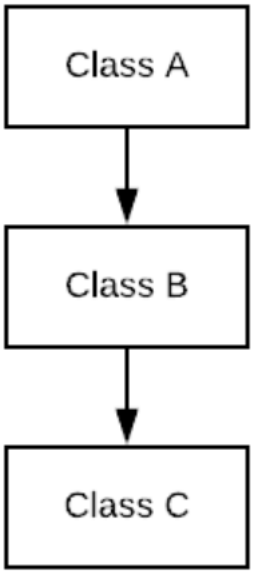
```
class Base1 {
   //Statements
};

class Base1 {
   //Statements
};

class Derive : <access_specifier> Base1, <access_specifier> Base2 {
   //Statements
};
```

3. Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class.



Syntax:

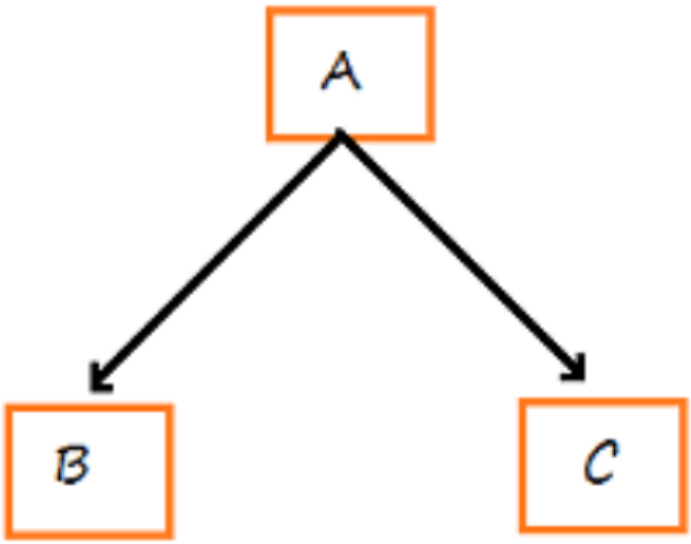
```
class Base_Class{
   //Member Methods
};

class Derived_Class_I: public Base_Class{
   //Member Methods
};

class Derived_Class II: public Derived_Class_I{
   //Member Methods
};
```

4. Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class.



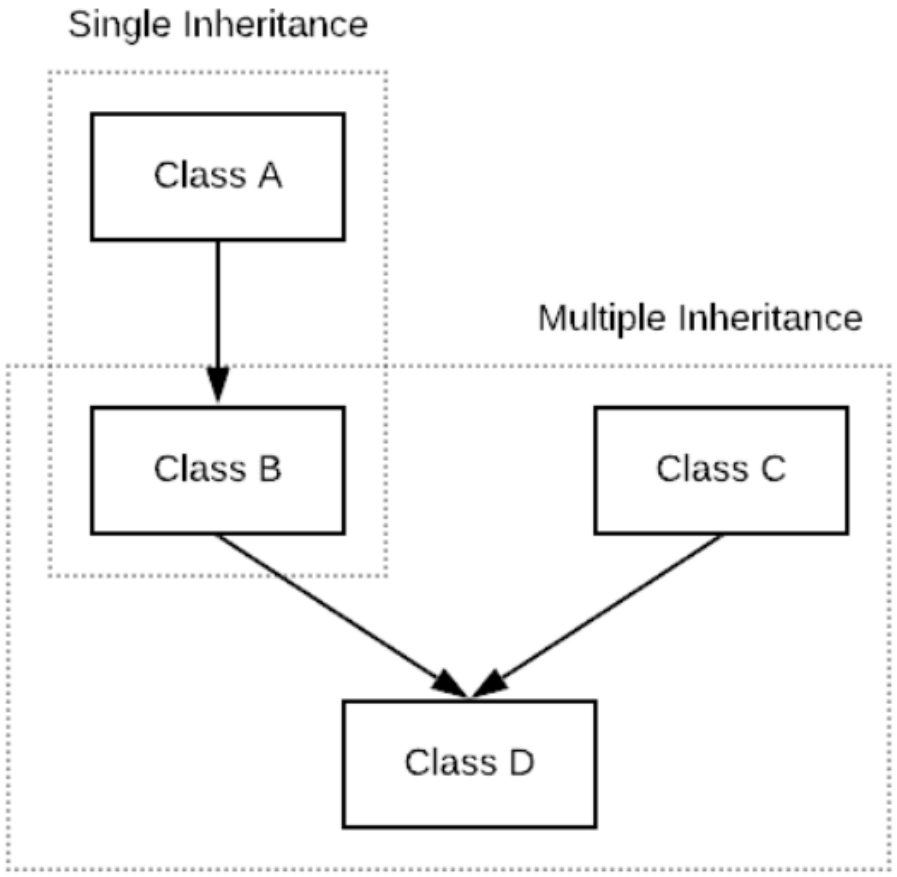
Heirarchical Inheritance

Syntax:

```
class A // base class
{
   ..............
};
class B : access_specifier A // derived class from A
{
   ...........
} ;
class C : access_specifier A // derived class from A
{
   ...........
} ;
class D : access_specifier A // derived class from A
{
   ...........
} ;
```

5. Hybrid Inheritance

hybrid inheritance is a combination of two or more types of inheritance.



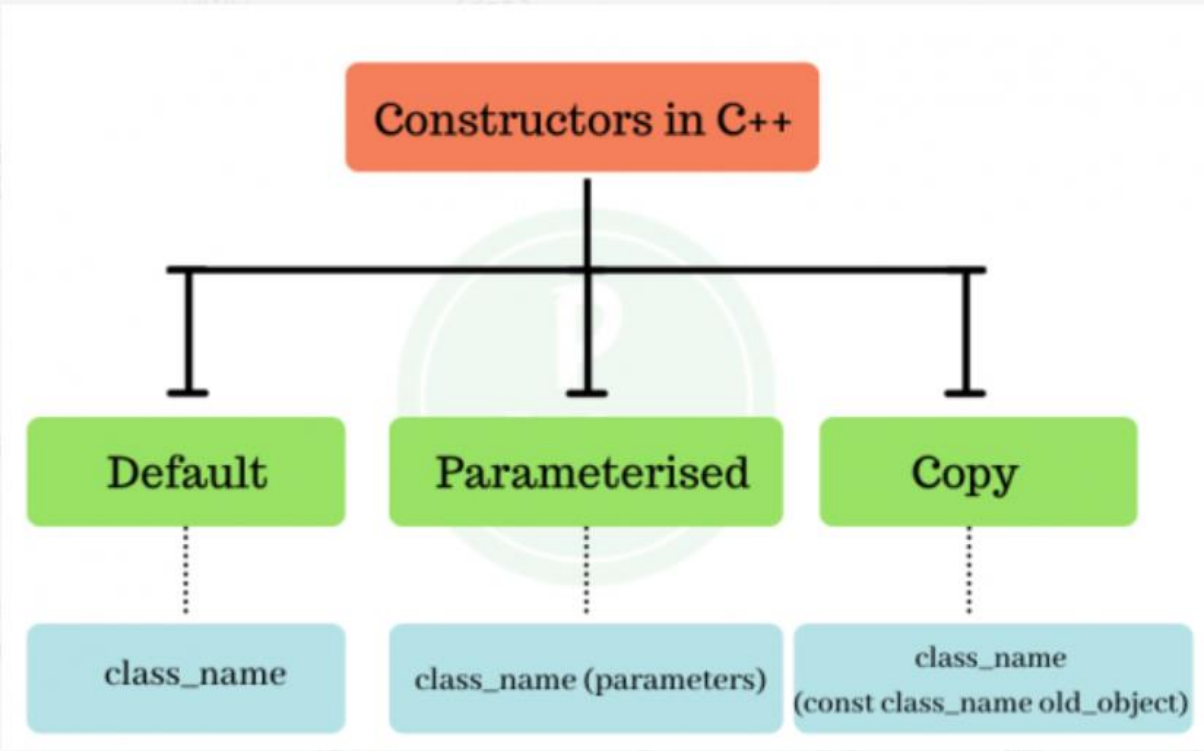
Syntax:

```
class A
{
    .....
};
class B : public A
{
    .....
} ;
class C
{
    .....
};
 class D : public B, public C
{
    .....
};
```

CONSTRUCTORS

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object (instance of class) create. It is special member function of the class because it does not have any return type.

Types of Constructors



1. **Default Constructors:**
   Default constructor is the constructor which doesn't take any argument. It has no parameters.

   Syntax:

```
class construct
{
public:
```

```
   // Default Constructor
   construct()
   {
      .....
   }
};
```

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.

Syntax:

```
class construct
{
public:

   // Parameterized Constructor
   construct(parameters)
   {
      .....
   }
};
```

3. Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class.

FRIEND

Friend Class: A friend class can access private and protected members of other class in which it is declared as

friend. It is sometimes useful to allow a particular class to access private members of other class.

Syntax:

```
class ClassB;

class ClassA {
   // ClassB is a friend class of ClassA
   friend class ClassB;
   ... .. ...
}

class ClassB {
   ... .. ...
}
```

Friend Function: Like friend class, a friend function can be given a special grant to access private and protected members. A friend function can be:

a) A member of another class

b) A global function

Syntax:

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

UML DIAGRAM

Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding.

We prepare UML diagrams to understand the system in a better and simple way. A single diagram is not enough to cover all the

aspects of the system. UML defines various kinds of diagrams to cover most of the aspects of a system.

You can also create your own set of diagrams to meet your requirements. Diagrams are generally made in an incremental and iterative way.

There are two broad categories of diagrams and they are again divided into subcategories −

- Structural Diagrams
- Behavioural Diagrams

1. Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

2. Behavioural Diagrams

Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered.

Behavioural diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.

CLASS DIAGRAM

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

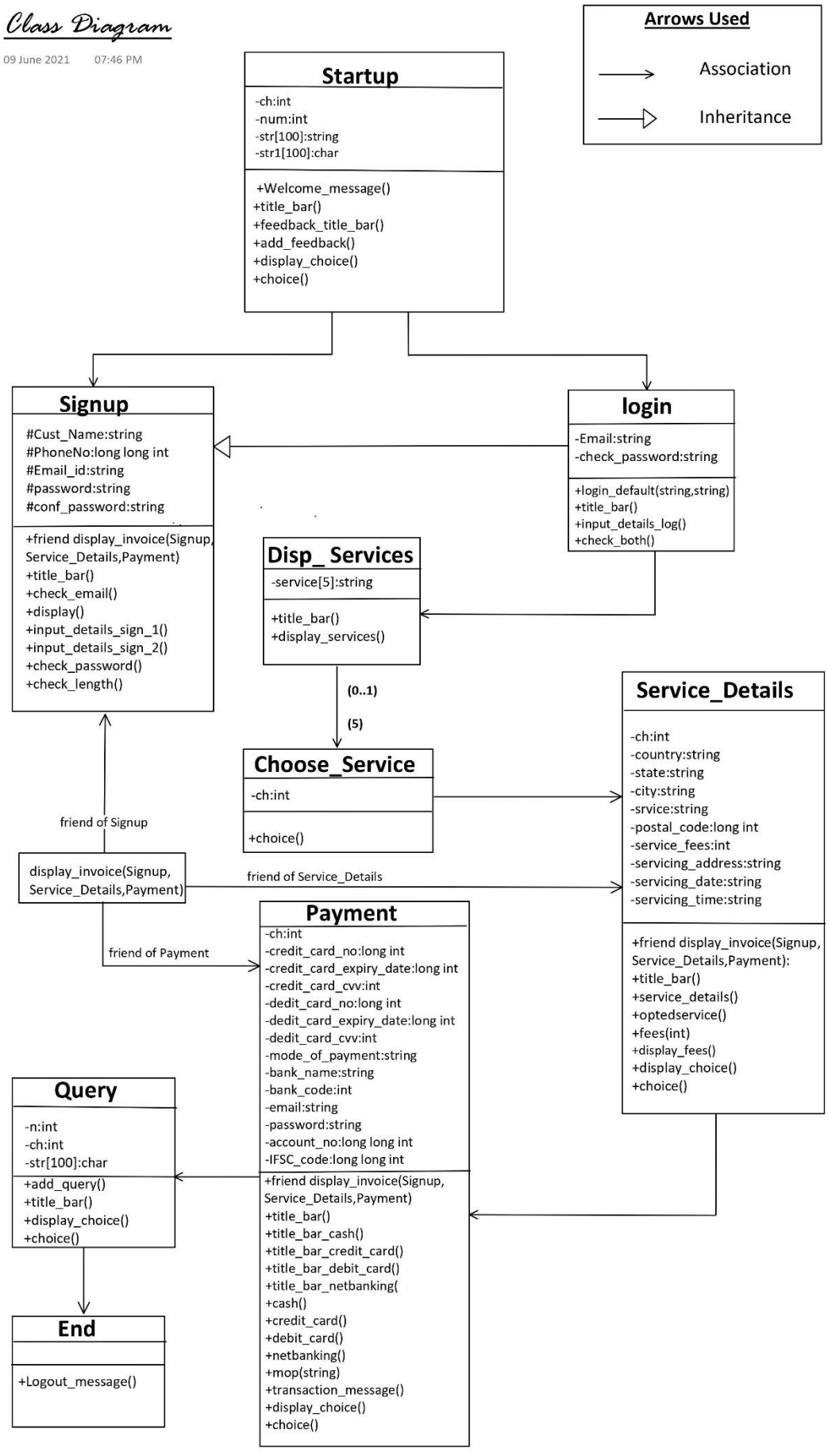
The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

CLASS DIAGRAM FOR OUR PROJECT

Class Diagram

09 June 2021 07:46 PM



Vital components of a Class Diagram

The class diagram is made up of three sections-

- **Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:
    1. Capitalize the initial letter of the class name.
    2. Place the class name in the centre of the upper section.
    3. A class name must be written in bold format.
    4. The name of the abstract class should be written in italics format.
- **Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:

1. The attributes are written along with its visibility factors, which are public (+), private (-) and protected (#).
    2. The accessibility of an attribute class is illustrated by the visibility factors.
    3. A meaningful name should be assigned to the attribute, which will explain its usage inside the class.

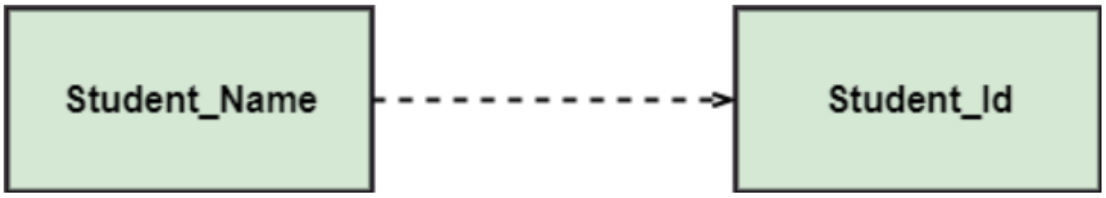
- **Lower Section:** The lower section contains methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.

Relationships

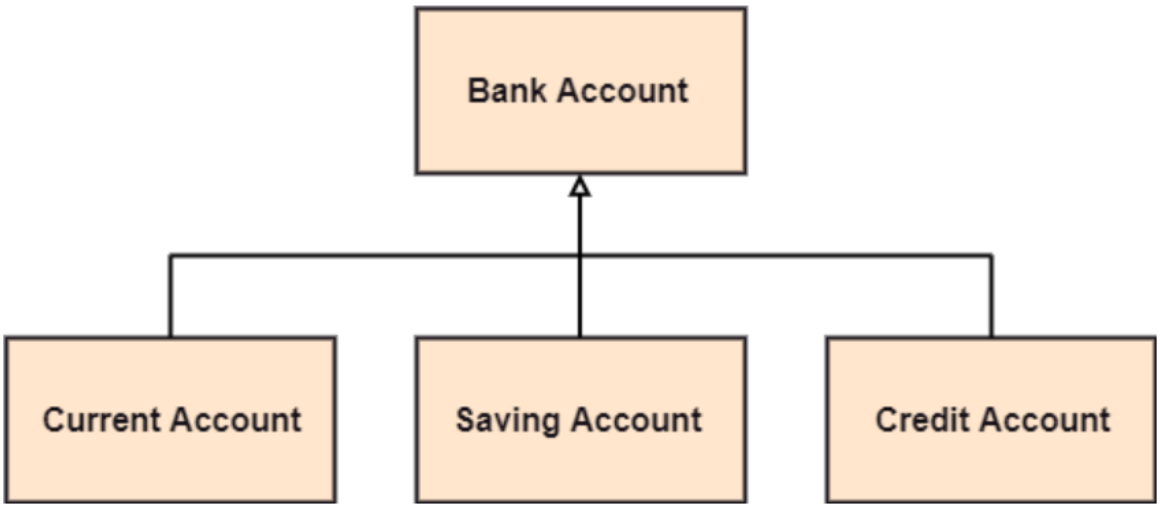
In UML, relationships are of three types:

- **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship.

In the following example, Student_Name is dependent on the Student_Id.



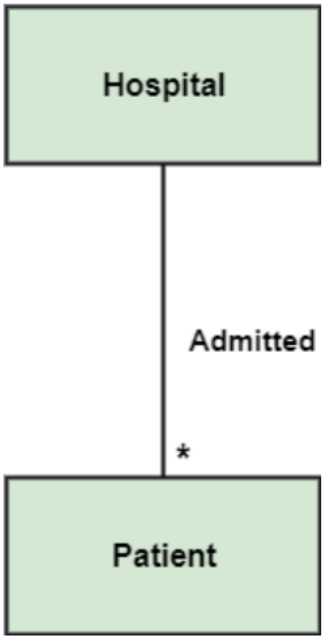
- **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.
  For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.



- **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship.
  For example, a department is associated with the college.

Multiplicity: It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.

For example, multiple patients are admitted to one hospital.



Aggregation: An aggregation is a subset of association, which represents has a relationship. It is more specific than association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.

The company encompasses a number of employees, and even if one employee resigns, the company still exists.

Composition: The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.

A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.

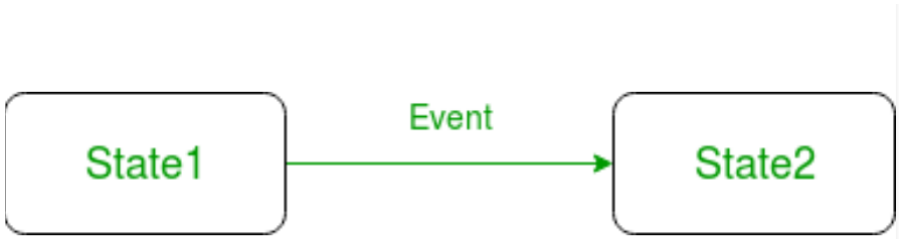
STATE DIAGRAM

A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's
a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams. These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli. We can say that each and every class has a state but we don't model every class using State diagrams. We prefer to model the states with three or more states.

Basic components of a state chart diagram

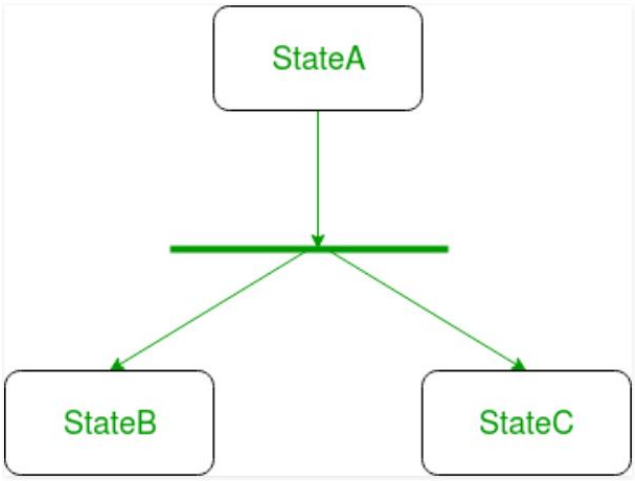
1. **Initial state** – We use a black filled circle represent the initial state of a System or a class.

2. **Transition** – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.

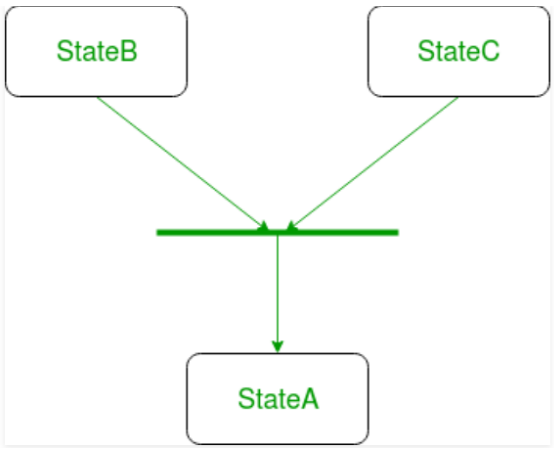


3. **State** – We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.

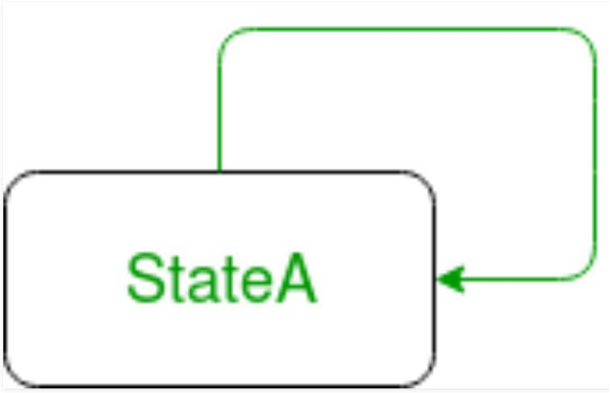
4. **Fork** – We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



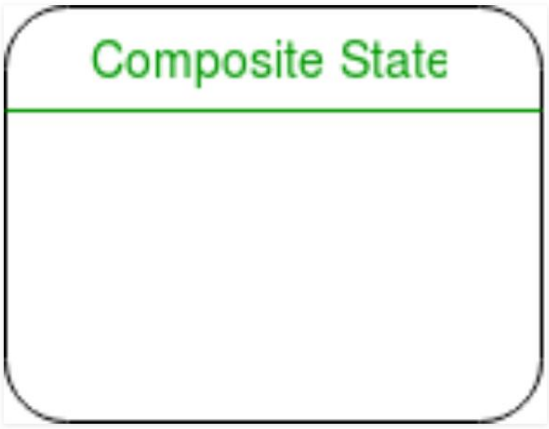
5. **Join** – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



6. **Self-transition** – We use a solid arrow pointing back to the state itself to represent a self-transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self-transitions to represent such cases.

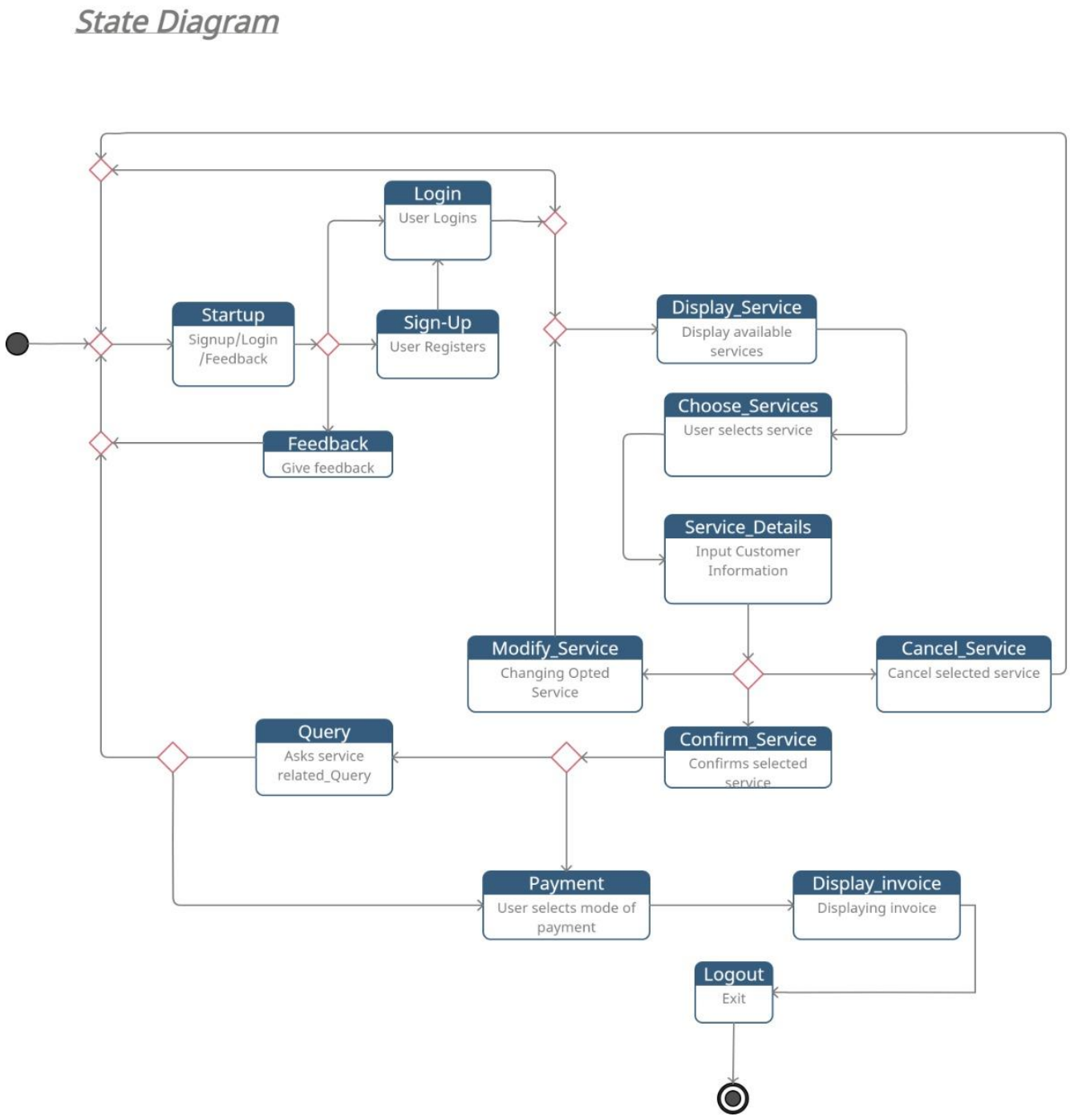


7. **Composite state** – We use a rounded rectangle to represent a composite state also.We represent a state with internal activities using a composite state.



8. **Final state** – We use a filled circle within a circle notation to represent the final state in a state machine diagram.

STATE DIAGRAM FOR OUR PROJECT



SEQUENCE DIAGRAM

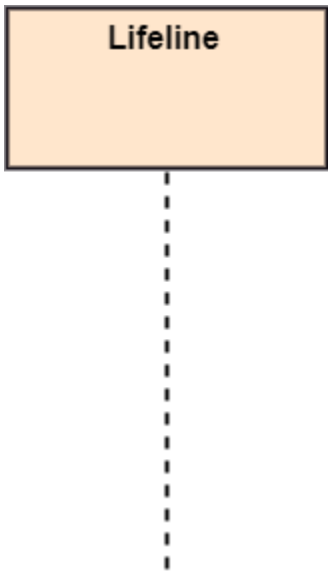
The sequence diagram represents the flow of messages in the system and is also termed as an event diagram. It helps in envisioning several dynamic scenarios. It portrays the

communication between any two lifelines as a time-ordered sequence of events, such that these lifelines took part at the run time. In UML, the lifeline is represented by a vertical bar, whereas the message flow is represented by a vertical dotted line that extends across the bottom of the page. It incorporates the iterations as well as branching.

Notations of a Sequence Diagram

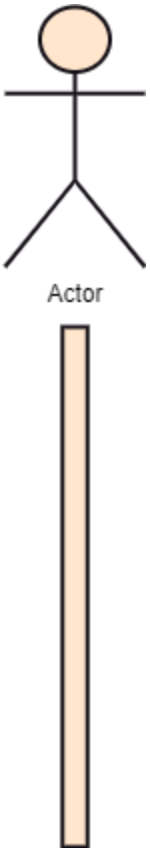
1. Lifeline

An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.



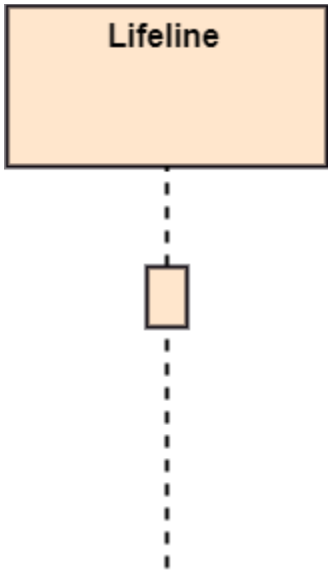
2. Actor

A role played by an entity that interacts with the subject is called as an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa.



3. Activation

It is represented by a thin rectangle on the lifeline. It describes that time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the initiation and the completion time, each respectively.

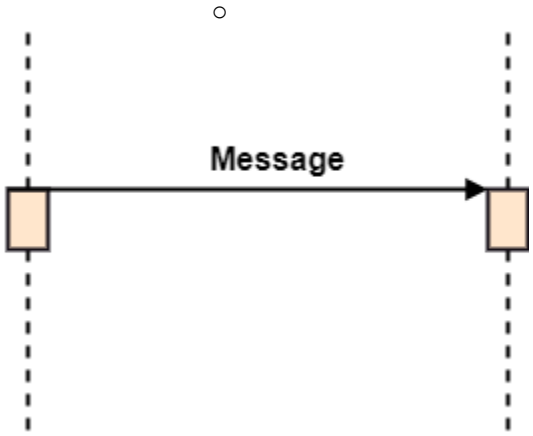


4. Messages

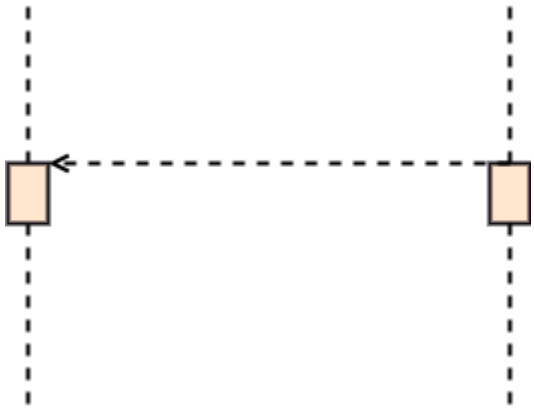
The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

Following are types of messages enlisted below:

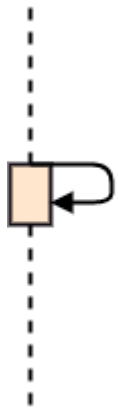
- **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



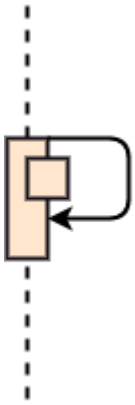
- **Return Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.



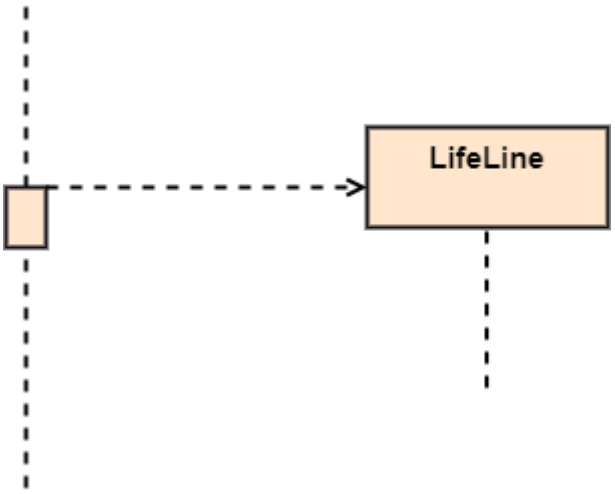
- **Self-Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



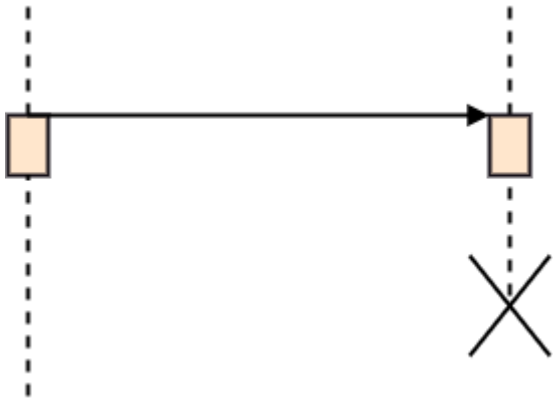
- **Recursive Message:** A self-message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self-message as it represents the recursive calls.



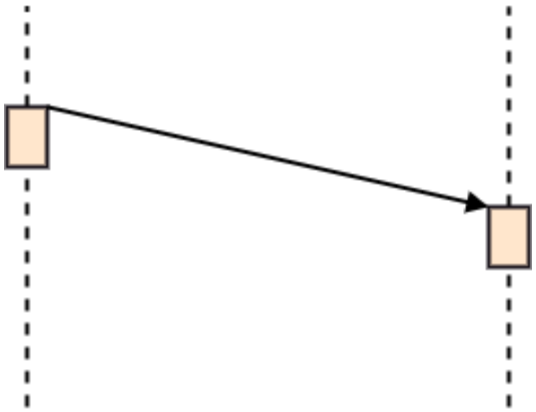
- **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- **Destroy Message:** It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



- Duration Message: It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modelling a system.



SEQUENCE DIAGRAM FOR OUR PROJECT

Sequence Diagram