Back To Course (/batchPage.php?batchId=185)

LIVE BATCHES

Learn

Classroom

Theory

Quiz

Overview Learn Problems Quiz

We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

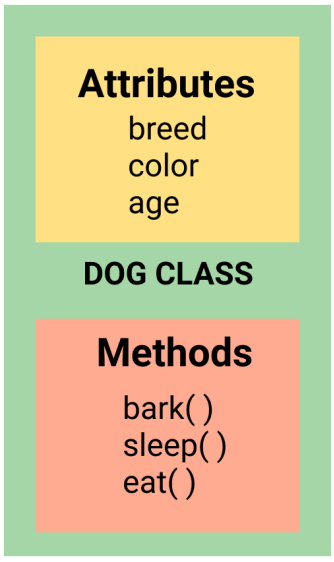
Classroom Theory

Introduction to Object Oriented Programming

Classes and Objects are basic concepts of Object Oriented Programming which revolve around real-life entities.

Class

A class is a user-defined blueprint or prototype from which real-world objects are created. It represents the set of properties or methods that are common to all objects of one type.
We can call class as a collection of objects, which is a logical entity and does not take space in the memory.
Example : Dog Class can be represented as shown in the diagram below.



Object

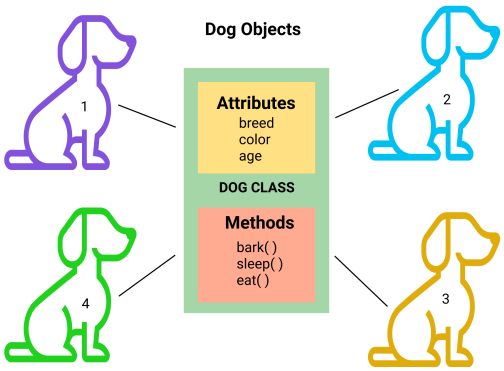
It is a basic unit of Object Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods.
Objects having memory addresses and take up some space. They can interact with each other without knowing the data and code.

An object consists of:-

- **State** : It is represented by the attributes of an object. It also reflects the properties of an object.

- **Behavior** : It is represented by the methods of an object. It also reflects the response of an object with other objects.
- **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Example : Dog objects created from Dog Class

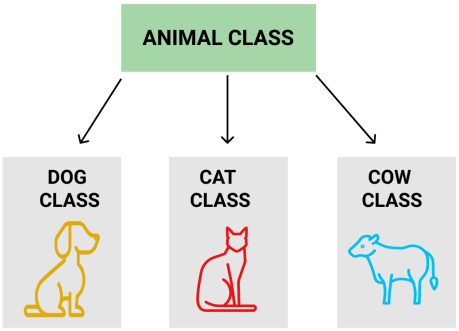


Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature 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.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

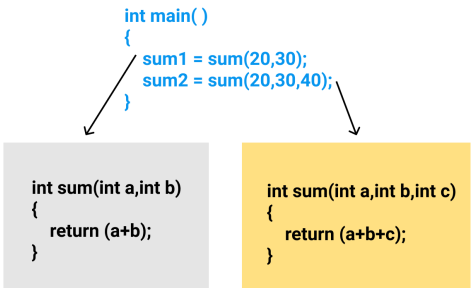
Example : Dog, Cat, and Cow can the be derived classes of Animal base class.



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.
A real-life example of polymorphism is that a person at the same time can have different characteristics. Like a man at the same time can be a father, a husband, and an employee. So the same person posses different behavior in different situations. This is called polymorphism.

Example: Suppose we have to write a function to add some integers, sometimes there are 2 integers and some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. **Ex: A car is viewed as a car rather than its individual components.**
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object and ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a **real-life example** of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying the brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes, etc in the car. This is what abstraction is.
Advantages of Abstraction

- It reduces the complexity of viewing things.
- Avoids code duplication and increases reusability.
- Helps to increase the security of an application or program as only the important details are provided to the user.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from the other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by declaring all the variables in the class as private and writing public methods in the class to set and get the values of the variables.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. He only knows that we are passing the values to a setter method and that the variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only, depending on our requirements. If we wish to make the variables as read-only then we have to omit the setter methods such as setName(), setAge() etc. or if we wish to make the variables as write-only then we have to omit the get methods such as getName(), getAge(), etc.
- **Reusability:** Encapsulation also improves the re-usability and makes the code easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Encapsulation vs Data Abstraction

- Encapsulation is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
- While encapsulation groups together the data and the methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of the implementation.

− Class and Class Members

Classes are the building blocks of Object-Oriented programming. It is a user-defined data-type which contains data members carrying values that describe the characteristics of the entity the class represents. It contains member methods/functions which describe the behavior of the class-entity. Note that the **class is just a blueprint** that has no physical memory associated with it. **When a class is instantiated, an object is created** which is provided a memory/storage location.

Class Definition and Object Instantiation

The syntax for class declaration is as:

```
class < class-name > {
< access-modifier >:
    //data-members
    //constructors
    //member-functions
    //destructors
};   //don't forget the semi-colon here !!
```

Each of the above terms (constructors, destructors, access-modifiers etc.) will be explained in detail afterwards.

```
3  using namespace std;
```

```
4
5  class Employee {  //Class Declaration
6      public:
7          string id, name;
8          int years;  //experience (in years)
9
10         Employee(string id, string name, int years) {
11             this->id = id;
12             this->name = name;
13             this->years = years;
14         }
15
16         void work() {
17             cout << "Employee: " << this->id << " is working\n";
18         }
19 };
20
21 int main()
22 {
23     //Class Instantiation (Direct)
24     Employee emp("GFG123", "John", 3);
25
26     //Class Instantiation (Indirect)
27     Employee *emp_ptr = new Employee("GFG456", "James", 4);
28
29     cout << "Employee ID: " << emp.id << endl;
30     cout << "Name: " << emp.name << endl;
31     cout << "Experience (in years): " << emp.years << endl;
32
```


Run

Output:

```
Employee ID: GFG123
Name: John
Experience (in years): 3
Employee: GFG123 is working

Employee ID: GFG456
Name: James
Experience (in years): 4
Employee: GFG456 is working
```

NOTE: Don't bother about **this** and other keywords which might as of now seem to be unexplained, as they require an additional whole article to explain. We shall cover each of them in detail afterwards.
The statement *Employee emp("GFG123", "John", 3)* inside the *main()* is the object instantiation statement. It is a direct-instantiation of the class. i.e. emp itself is the object stored in RAM, (much like declaring an int, float, struct, etc.). Upon execution of the statement, storage is allocated according to the size of the class and thereafter the constructor - Employee(...) inside the class is run to initialize the data members. We shall study in depth more object initialization formats (there are multiple ways to do so), and more about constructors in later sections.
The statement *Employee *emp_ptr = new Employee("GFG456", "James", 4)* is the indirect way of instantiating class-objects. Here, we use a pointer to point to the object created. We use the **new** keyword and refer to the member entities using **->** operator.

Data Members

Data Members are the identifiers that describe the characteristics and hold important data related to the class. In our Employee class, we have *id, name, years* as the data members. Data members can be anything from primitive data-types (int, char, double, etc.) to collections (arrays, strings, etc.) to user-defined data-types (structs, unions, even other class objects, etc.). Anything that can hold data value can be a data-member.

Member Functions

Member Functions/Methods are class-functions/methods which describe the behavior of the class. i.e. what kind of operation we can perform with objects of this class. e.g. In our Employee example, *work()* is a member function. Member functions in C++ can be defined inside/outside the class, however, it is mandatory to have prototype declaration inside the class if go for an outside-the-class definition. e.g.

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  class Employee {  //Class Declaration
6      public:
7          string id, name;
```

```
        string id, name;
        int years;  //experience (in years)

        Employee(string id, string name, int years) {
            this->id = id;
            this->name = name;
            this->years = years;
        }

        //Prototype Declaration
        void work();
};

//Outside-class definition
void Employee::work() {
    cout << "Employee: " << this->id << " is working\n";
}

int main()
{
    //Class Instantiation (Direct)
    Employee emp("GFG123","John",3);

    emp.work();
```

Run

Output:

```
Employee: GFG123 is working
```

As we observe, while providing the definition for a class member-function we use the **scope-resolution operator ::** with the **class-name::function-name** format to differentiate it from normal global-scoped functions.

− Access Modifiers & Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes, etc in the car. This is what abstraction is.
Abstraction using Classes: We can implement Abstraction in C++ using classes. Classes help us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Modifiers

There are 3 types of access modifiers in C++, which we discuss in detail below:

- **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes 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.

```
#include <bits/stdc++.h>
using namespace std;

class Circle
{
    public:
        double radius;

        double compute_area() {
            return 3.14 * radius * radius;
        }
};
```

```
int main()
{
    Circle obj;

    // accessing public data member outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << endl;
    cout << "Area is: " << obj.compute_area();

    return 0;
}

```


Run

Output:

```
Radius is: 5.5
Area is: 94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

- Private: The class members declared as **private** can be accessed only by the 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 (discussed later) are allowed to access the private data members of a class.

```
#include <bits/stdc++.h>
using namespace std;

class Circle
{
    //private data member
    private:
        double radius;

    //public member function
    public:
        double  compute_area() {
            //member function can access private
            //data member radius
            return 3.14 * radius * radius;
        }

};

int main()
{
    Circle obj;

    //trying to access private data member
    //directly outside the class
    obj.radius = 1.5;

    cout << "Area is: " << obj.compute_area();
    return 0;
```

Run

Error generated by the above code:

```
prog.cpp: In function 'int main()':
prog.cpp:9:16: error: 'double Circle::radius' is private
        double radius;
               ^
prog.cpp:27:9: error: within this context
    obj.radius = 1.5;
        ^
```

The output of the above program will be a compile-time error because we are not allowed to access the private data members of a class directly outside the class. If we comment out the *obj.radius = 1.5;* statement, then the program compiles fine. We can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
#include <bits/stdc++.h>
```

```
using namespace std;

class Circle
{
    //private data member
    private:
        double radius;

    //public member functions
    public:
        int getRadius() { return radius; }
        void setRadius(double r) { radius = r; }

        double compute_area() { return 3.14 * radius * radius; }
};

int main()
{
    Circle obj;

    obj.setRadius(5);
    cout << "Radius: " << obj.getRadius() << endl;
    cout << "Area: " << obj.compute_area() << endl;

    return 0;
}
```


Run

Output:

```
Radius: 5
Area: 78.5
```

We can use **getters** and **setter** public functions to indirectly access and manipulate the values of private data-members.

- **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
#include <bits/stdc++.h>
using namespace std;

//Base Class
class Parent
{
    //protected data members
    protected:
        int id_protected;
};

//Derived Class
class Child : public Parent
{
    public:
        void setId(int id) {
            id_protected = id;
        }

        void displayId() {
            cout << "id_protected is: " << id_protected << endl;
        }
};

int main() {

    Child obj;

    //member function of the derived class can
```

Run

Output:

```
id_protected is: 81
```

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

− Constructors

A **Constructor** is a member function of a class that initializes objects of a class. In C++, Constructor is automatically called when an object (instance of the class) is created. It is a special member function of the class.

How constructors are different from a normal member function?

A constructor differs from member-functions in the following ways:

- Constructor has the same name as the Class itself.
- Constructors don't have a return type.
- A Constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Default Constructor

It is not mandatory for the programmer to write a constructor for each class. C++ by default provides a default constructors with no parameters, and no statements for the body. Much like being:

```
Employee() {}   //as per our example
```

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;
};

int main()
{
    Employee emp;
    return 0;
}

```

Run

In the above code, once the object is created, the default constructor is called. We can overload the default constructor (covered below).

Constructor Overloading

We need to first get a grasp of overloading first, which is explained as:
Overloading: Having the same name for a member-function/constructor as long as the list of arguments is different is called overloading. In such a case, depending upon the arguments passed, the appropriate overloaded function is deduced and called. An example of constructor overloading:

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;

    Employee()
    {
        id = "";
        name = "";
        years = 0;
    }

    // Overloaded constructor
    Employee(string id, string name, int years)
```

```
    Employee(string id, string name, int years)
    {
        this->id = id;
        this->name = name;
        this->years = years;
    }

    // Overloaded constructor
    Employee(string id, string name)
    {
        this->id = id;
        this->name = name;
        years = 0;
```

Run

Output:

```
ID:, Name:, Experience: 0
ID: GFG123, Name: John, Experience: 4
ID: GFG456, Name: James, Experience: 0
```

Member Initializer List

Member Initialization List is a new syntactic construct introduced in modern C++, which allows us to write concise initialization code in constructors. The basic syntax is as:

```
Constructor(< arguments >) : < mem1(arg1), mem2(arg2), ...., > {
    //additional code to execute after initialization
}
```

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;

    Employee(string id, string name, int years)
        : id(id), name(name), years(years)
    {
        // extra code to run after initialization
    }

    // does the same as:
    // Employee(string id, string name, int years) {
    // this->id = id;
    // this->name = name;
    // this->years = years;

    //     // extra-code to run after initialization
    // }

    void getDetails()
    {
        cout << "ID: " << id << ", Name: " << name
             << ", Experience: " << years << endl;
    }
};
```

Run

Output:

```
ID: GFG123, Name: John, Experience: 4
```

Constructor Chaining (Delegation)

Constructor chaining/delegation is the process of re-using constructors by others to avoid writing repeated code. This is done by calling one constructor to set common values by other constructors. As an example:

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;

    Employee(string id)
        : id(id)
    {
    }

    // uses constructor1              constructor1 call
    Employee(string id, string name)
        : Employee(id)
    {
        this->name = name;
    }

    // uses constructor2                       constructor2 call
    Employee(string id, string name, int years)
        : Employee(id, name)
    {
        this->years = years;
    }

    void getDetails()
    {
```

Run

Output:

```
ID: GFG123, Name: John, Experience: 4
```

Destructors

Destructors like constructors are special members of a class that is executed once the lifetime of the object expires. It is like the final clean-up code required before deleting the class instance. Unlike JAVA, C++ doesn't perform automatic garbage collection. Hence, it often becomes the responsibility of the developer to de-allocate memory (not required further). A classic example:

```

#include <bits/stdc++.h>
using namespace std;

char* p_chr;

class String {
public:
    char* s;
    int size;

    String(char* c)
    {
        size = strlen(c);
        s = new char[size + 1];
        p_chr = s; // assign to global variable
        strcpy(s, c);
    }
};

void func()
{
    String str("Hello World");
}

int main()
{
    func();
    cout << p_chr << endl;
    return 0;
```

Run

Output:

```
Hello World
```

In the above code, we dynamically create a character array in our constructor, where we copy the string passed as an argument. Since it is a dynamically-allocated memory, once, the lifetime of *str* object expires (inside the *func()* call), still the memory is not de-allocated, as in main(), when we print *p_chr*, we get the string. Thus, we can see that proper clean-up of pointer references doesn't occur. Thus, we need destructors where we would explicitly de-allocate the memory and perform other clean-up code. Syntax of destructor:

```
~ Classname { //clean-up code }
```

The above code with destructors:

```
#include <bits/stdc++.h>
using namespace std;

char* p_chr;

class String {
public:
    char* s;
    int size;

    String(char* c)
    {
        size = strlen(c);
        s = new char[size + 1];
        p_chr = s;
        strcpy(s, c);
    }

    // Destructor
    ~String()
    {
        delete[] s;
    }
};

void func()
{
    String str("Hello World");
}
```

Run

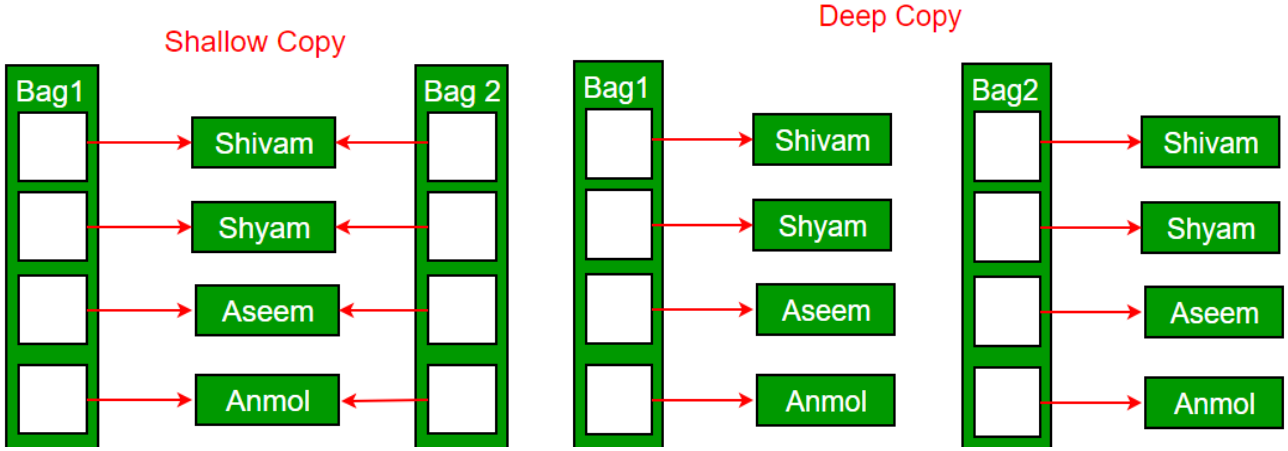
The above code prints nothing because *p_chr* reference is gone due to destructor call.

Copy Constructor

A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
Classname (const Classname &object);
```

Copy constructor, in general, is not required to be defined by the user, as the compiler automatically provides a default copy constructor. However, this default copy constructor performs a shallow copy only (i.e. copy values only). This results in pointer variables pointing the same instances upon copy. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection.etc.



As an example:

```
#include <bits/stdc++.h>
using namespace std;

class Array {
public:
    int n;
    int* ref;

    Array(int n)
        : n(n)
    {
        ref = new int[n];

        for (int i = 0; i < n; i++)
            *(ref + i) = i;
    }
};

int main()
{
    Array arr1(10);

    // copy constructor called
    // at this point
    Array arr2 = arr1;

    // changing n-value in 2nd instance
    arr2.n = 5;

```

Run

Output:

```
n-value of 1st instance: 10
Array values of 1st instance:
0 2 4 6 8 10 12 14 16 18
```

In the above code, we find the value of member *n* for *t1* not modified as it is not a pointer value. Thus, upon copying new instance of *n* got created for *t2*. Any change to n in *t2* didn't change *t1.n*. However, *t1.ref* is a pointer. So, upon copy-constructor call, the address value got copied to *t2.ref*, and thus, any change at t2.ref (as we here are multiplying by 2), gets reflected at t1.ref also because both of them are pointing to the same array. This is an example of a shallow-copy. To fix this, we write our custom copy-constructor:

```
#include <bits/stdc++.h>
using namespace std;

class Array {
public:
    int n;
    int* ref;

    Array(int n)
        : n(n)
    {
        ref = new int[n];

        for (int i = 0; i < n; i++)
            *(ref + i) = i;
    }

    // copy-contructor definition
    Array(const Array& obj)
        : n(obj.n)
    {
        ref = new int[n];

        for (int i = 0; i < n; i++)
            *(ref + i) = *(obj.ref + i);
    }
};

int main()
```

Run

Output:

```
n-value of 1st instance: 10
Array values of 1st instance:
0 1 2 3 4 5 6 7 8 9
```

We can see that in our copy-constructor, we re-create a dynamic memory for the array and then copy the values. This results in a deep-copy. Meaning changes in *arr2* doesn't affect *arr1*.

'this' pointer

To understand *'this'* pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as *'this'*.
The *'this'* pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. *'this'* pointer is a constant pointer that holds the memory address of the current object. *'this'* pointer is not available in static member functions as static member functions can be called without any object (with class name).
For class X, the type of this pointer is 'X* const'. Also, if a member function of X is declared as const, then the type of *'this'* pointer is 'const X *const'.

Following are the situations where *'this'* pointer is used:

When local variable's name is same as member's name

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
    public:
        string id, name;
        int years;

        //'this' keyword here retrieves the object's
        //instance variables: id, name, years hidden
        //by their same-name local counterparts
        Employee(string id, string name, int years) {
            this->id = id;
            this->name = name;
            this->years = years;
        }

        //here we don't need to use 'this' keyword
        //explicitly as their are no local variables
        //with the same name. So, compiler automatically
        //deduces it as instance variables
        void printDetails() {
            cout << "ID: " << id
                 << ", Name: " << name
                 << ", Experience: " << years;
        }
};

int main()
```

Run

Output:

```
ID: GFG123, Name: John, Experience: 4
```

For constructors, initializer list can also be used when parameter name is same as member's name.

To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
    private:
        string id, name;
        int years;

    public:
        Employee setId(string id) {
            this->id = id;
            return *this;
        }

        Employee setName(string name) {
            this->name = name;
            return *this;
        }

        Employee setYears(int years) {
            this->years = years;
            return *this;
        }

        void printDetails() {
            cout << "ID: " << id
                 << ", Name: " << name
                 << ", Experience: " << years;
        }
```

Run

Output:

```
ID: GFG123, Name: John, Experience: 4
```

In the above code, each time we call the setter methods, the employee instance we are referring to is returned using the *'this'* pointer. We can thus re-use this instance to chain more method calls.

− Static data members and methods

We looked at the static keyword earlier in the respect of functions. How declaring an identifier as static gives its lifetime scope of the program, such that it retains its value even after successive function calls. But, static has a different meaning when it comes to classes.

Static data-members

Declaring a data member in a class as static gives it class-scope. i.e. The variable no longer remains specific and bound to one particular object instance. Thereafter, changing the value from one instance reflects over all the instances. As an example:

```
#include <bits/stdc++.h>
using namespace std;

class Test {
    public:
        static int x;
};

/*static members need to be
defined outside the class*/
```

```
12  int Test::x = 1;
13
14  int main()
15  {
16      Test t1, t2;
17
18      cout << "Access from instance: " << t1.x << endl
19           << "Access from Class: " << Test::x << endl;
20
21      Test::x = 5;
22
23      cout << "t1.x: " << t1.x << endl
24           << "t2.x: " << t2.x << endl
25           << "Test::x: " << Test::x << endl;
26
27      return 0;
28  }
29
```


Run

Output:

```
Access from instance: 1
Access from Class: 1
t1.x: 5
t2.x: 5
Test::x: 5
```

As we can see from the above program, any change made to the member *x*, it gets reflected over both instances *t1* and *t2*. We can also access static members using Class-name and scope-resolution ~ *Test::x*.

NOTE: When we declare a static variable inside a class, we are just telling the compiler the existence of such a variable. It is treated as a variable with a global scope and is initialized only when the program starts. No memory is allocated at that point. Thus, we can't directly initialize a static data-member along with the declaration. i.e. *static int x = 1;* is reported as a compilation error. We must explicitly initialize it outside the class. Another thing to note is that this initializer statement works regardless of whether we declare the static variable as public, private or protected.

Static member-functions

Static Member Functions are similar to the static data-members implying that they too have class-scope. In addition to that, they are allowed to access only other static fields (data & member). However, they can be called using object instances. (i.e. *obj.static_method()* is valid). Some important points regarding static member functions are given below:

- Static member-functions can't access non-static data-members.
- Static member-functions can't access other non-static member-functions.
- Static member-functions have no *this pointer, as it is not associated with any particular instance.
- There is no concept of a static constructor.
- Static member-functions are useful for accessing static data-members which are not declared public.

As an example:

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  class Test {
6      private:
7          static int x;
8
9      public:
10         //set private static integer x
11         static void setX(int x) { Test::x = x; }
12
13         //get private static integer x
14         static int getX() { return x; }
15 };
16
17 //Initializer statement is valid
18 //even though variable is declared
19 //private
20 int Test::x = 1;
21
22 int main()
23 {
24     Test::setX(5);
25     cout << Test::getX();
```

```
    cout << test::getX();

    return 0;
}

```

Run

LIVE BATCHES

Output:

```
5
```

- Friend function in C++

For the sake of data-hiding, we emphasize the use of a private modifier for critical data-members. However, there might be some situations where multiple classes need to work together closely, so much so that they require access to each other's private members too. Well, we can solve the problem by using getters and setters, but it would be tedious to do so for all the private members. Instead, we use the ***friend*** keyword to provide access to the private fields to outside entities (global functions, other class-member functions etc.).

Friend Class

A *Friend Class* can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private members of other classes.e.g.A LinkedList class may be allowed to access private members of Node.

```
#include <bits/stdc++.h>
using namespace std;

class Node
{
    private:
        int key;
        Node *next;
        Node(int key) : key(key), next(nullptr) {}

    public:
        friend class LinkedList;
};

class LinkedList
{
    public:
        Node *root;

        LinkedList(int key) {
            root = new Node(key);
        }

        void insert(int key) {
            Node *trav = root;
            while (trav->next != nullptr)
                trav = trav->next;
            trav->next = new Node(key);
        }
```

Run

Output:

```
0 1 2 3 4
```

As we can see in the above code, *LinkedList* has access to all the private fields of *Node* class, because it has been declared as a friend inside the Node class.

Friend Function

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

- A Member Function of another class

```
#include <bits/stdc++.h>
using namespace std;
```

```
using namespace std;

//forward-declaration is
//necessary for usage in A
//as B is not defined yet
class B;

class A
{
    public:
        void showB(B &x);
};

class B
{
    private:
        int b;
    public:
        B() : b(0) {}

        //Friend function Declaration
        friend void A::showB(B &x);
};

//Friend Member Function Definition
void A::showB(B &x)
{
    cout << "B::b = " << x.b;
```

Run

Output:

```
B::b = 0
```

- A global Function

```
#include <bits/stdc++.h>
using namespace std;

class A
{
    private:
        int a;
    public:
        A() : a(0) {}

        //global friend function
        friend void showA(A&);
};

void showA(A &x) {
    std::cout << "A::a=" << x.a;
}

int main()
{
    A a;
    showA(a);
    return 0;
}
```

Run

Output:

```
A::a=0
```

- Anonymous Objects in C++

Anonymous objects are created without assigning a reference to them. Thus, they can be used only once (i.e. in the same statement only).
e.g.

```
#include <bits/stdc++.h>
using namespace std;

class Math {
    public:
        int add(int x, int y) { return x + y; }
        int mul(int x, int y) { return x * y; }
};

int main()
{
    cout << Math().add(5,6) << endl;
    cout << Math().mul(5,6) << endl;
    return 0;
}

```

Run

Output:

```
11
30
```

In the above code, in each of the *cout* statements, a new instance of the *Math* class is instantiated and then the said operation is performed. Thus, there is no way to reference those objects afterword since there is no reference attached to them.

- Operator Overloading in C++

In C++, we can make operators work for user-defined classes. This means 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 overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc. An example for the case of complex numbers addition:

```
#include <bits/stdc++.h>
using namespace std;

class Complex {
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i =0)  {real = r;   imag = i;}

        // This is automatically called when '+' is used with
        // between two Complex objects
        Complex operator + (Complex const &obj) {
             Complex res;
             res.real = real + obj.real;
             res.imag = imag + obj.imag;
             return res;
        }
        void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}

```

Run

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions? Operator functions are the same as normal functions. The only difference is that the name of an operator function is always the operator keyword followed by the symbol of operator and operator functions are called when the corresponding operator is used.
Following is an example of global operator function.

```
#include <bits/stdc++.h>
using namespace std;

class Complex
{
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i =0)  {real = r;   imag = i;}
        void print() { cout << real << " + i" << imag << endl; }

        // The global operator function is made friend of this class so
        // that it can access private members
        friend Complex operator + (Complex const &, Complex const &);
};

Complex operator + (Complex const &c1, Complex const &c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}


int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Run

Output:

```
12 + i9
```

Can we overload all operators? Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

- . (dot)
- :: (scope-resolution)
- ?: (ternary-operator)
- sizeof

Important points about operator overloading

1. For operator overloading to work, at least one of the operands must be an user-defined class object.
2. *Assignment Operator* - Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor).
3. *Conversion Operator* - We can also write conversion operators that can be used to convert one type to another type. As an example:

```
#include <bits/stdc++.h>
using namespace std;

class Fraction
{
    int num, den;
    public:
        Fraction(int n,  int d) : num(n), den(d) {}

        // conversion operator: return float value of fraction
        operator float() const {
            return float(num) / float(den);
        }
```

```
};

int main()
{
    Fraction f(2,5);
    float val = f;
    cout << val;
    return 0;
}

```

Run

Output:

```
0.4
```

4. Overloaded conversion operators must be a member method. Other operators can either be a member method or a global method.
5. Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

```
#include <bits/stdc++.h>
using namespace std;

class Point
{
    private:
        int x, y;
    public:
        Point(int i=0, int j=0) : x(i), y(j) {}
        void print() {
            cout << endl << " x = " << x << ", y = " << y;
        }
};

int main() {
    Point t(20, 20);
    t.print();
    t = 30;   // Member x of t becomes 30
    t.print();
    return 0;
}

```

Run

Output:

```
x = 20, y = 20
x = 30, y = 0
```

- Overloading vs Overriding

Overloading

Overloading is a feature that allows a class to have multiple methods with the same name, the only difference lies in their list of arguments. i.e. The argument list for each of the methods differ, and it helps the compiler or the run-time environment to identify which method to call depending upon the parameters passed. Constructors can be overloaded too.

```
#include <bits/stdc++.h>
using namespace std;

class Math
{
    public:
        //Overloaded add() methods
        static int add(int x, int y) { return x + y; }
        static void add(int a[], int b[], int sum[], int n) {
            for (int i=0;i<n;i++)
                //use the 1st form to get addition of
                //two numbers
                sum[i] = add(a[i], b[i]);
        }
```

```
16
17          //Overloaded mul() methods
18          static int mul(int x, int y) { return x * y; }
19          static void mul(int a[], int b[], int prod[], int n) {
20              for (int i=0;i<n;i++)
21                  //use the 2nd form to get product
22                  //of two numbers
23                  prod[i] = mul(a[i], b[i]);
24          }
25  };
26
27  int main()
28  {
29      int a[] = {1, 2, 3, 4, 5};
30      int b[] = {9, 8, 7, 6, 5};
```

Run

Output:

```
10 10 10 10 10
9 16 21 24 25
```

As shown in the above program, we have 2 sets of overloaded functions, namely integer addition-array addition and integer product-array-product (sort of dot product). Depending on the type of arguments passed, the compiler decides which one of the set of overloaded functions it needs to call.

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  class Math
6  {
7      public:
8          //Overloaded add() methods
9          static int add(int x, int y) {
10             cout << "1st form called: ";
11             return x + y;
12         }
13
14         static double add(int x, double y) {
15             cout << "2nd form called: ";
16             return x + y;
17         }
18
19         static double add(double x, double y) {
20             cout << "3rd form called: ";
21             return x + y;
22         }
23 };
24
25 int main()
26 {
27     cout << Math::add(2, 3) << endl;
28     cout << Math::add(2, 3.0) << endl;
29     cout << Math::add(2.0, 3.0) << endl;
30     return 0;
```

Run

Output:

```
1st form called: 5
2nd form called: 5
3rd form called: 5
```

NOTE: Return-type is not a factor of uniqueness: Having the list of arguments identical with different return-types doesn't remove ambiguity. Hence, the following declarations are invalid:

```
1
2  int add(int x, int y) { ... }
3  double add(int x, int y) { ... }
4
```

Overriding

Inheritance allows Derived Classes to inherit Base class data-members as well as member functions. Thus, if we call base class function with derived class instance, it would run perfectly:

```
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};

class Derived : public Base {

};

int main()
{
    Base b;
    Derived d;

    b.whoami();
    d.whoami();

    return 0;
}
```

Run

Output:

```
I'm Base Class!!
I'm Base Class!!
```

In the above program, we see no definition of *whoami()* method inside the Derived Class. So when we call *d.whoami()*, the compiler first looks into the Derived Class for its definition. Then, it looks inside its Parent class, where it finds the definition and that version is called. However, suppose we want to change the behavior of the inherited method inside our Derived Class. This feature provided by OOP is called Overriding. To do that, we provide the full method definition as usual:

```
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};

class Derived : public Base {
    public:
        void whoami() {
            cout << "I'm Derived Class!!\n";
        }
};

int main()
{
    Base b;
    Derived d;

    b.whoami();
    d.whoami();

    return 0;
}
```

Run

Output:

```
I'm Base Class!!
I'm Derived Class!!
```

Sometimes, we don't want to replace the complete functionality of the inherited methods, but add some extra functionality to it. We can do so by calling the Base Class's method from the Derived class and then continue with our overriding added statements.

```
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};

class Derived : public Base {
    public:
        void whoami() {
            // call Base class's version
            Base::whoami();
            cout << "I'm Derived Class!!\n";
        }
};

int main()
{
    Derived d;

    d.whoami();

    return 0;
}

```

Run

Output:

```
I'm Base Class!!
I'm Derived Class!!
```

We call the Base class's *whoami()* using the scope-resolution operator with the classname preceding it.

- Virtual Functions & Polymorphism

Run-time Polymorphism

Consider a situation where we have Derived Class which has overridden some method of the Base Class. Polymorphism allows us to have a Base Class reference a Derived Class Object. Then, in such a case, which function to call (Base or Derived) is decided at run-time, as the compiler is unable to resolve which one to call during compilation. Below is a classic example of Run-time Polymorphism:

```
#include <bits/stdc++.h>
using namespace std;

class Base
{
    public:
        void whoami() {
            cout << "I'm Base\n";
        }
};

class Derived: public Base
{
    public:
        void whoami() {
            cout << "I'm Derived\n";
        }
};

int main(void)
{
```

```
    Base *b_ptr = new Derived;

    //run-time polymorphism
    b_ptr->whoami();
    return 0;
}

```

Run

Output:

```
I'm Base
```

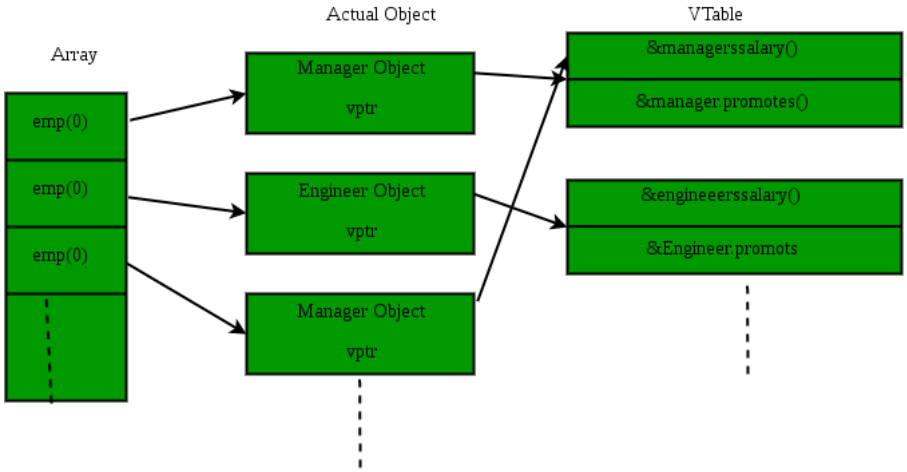
In the above code, since, the pointer reference is of Base-type, so at runtime, it is resolved to call the *whoami()* version of the Base class. Therefore, *"I'm Base"* is printed. But, suppose we want to call the Derived Class's version of the function. Here, comes the play of **virtual** keyword.

Virtual Functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,.. etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. e.g. we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```

class Employee
{
    public:
        virtual void raiseSalary()
        {  /* common raise salary code */  }

        virtual void promote()
        { /* common promote code */ }
};

class Manager: public Employee
{
    public:
        virtual void raiseSalary()
        {  /* Manager specific raise salary code, may contain
              increment of manager-specific incentives*/  }

        virtual void promote()
        { /* Manager specific promote */ }
};

// Similarly, there may be other types of employees

// We need a very simple function to increment the salary of all employees
// Note that emp[] is an array of pointers and actual pointed objects can
// be any type of employees. This function should ideally be in a class
// like Organization, we have made it global to keep things simple
void globalRaiseSalary(Employee *emp[], int n)
{
```

like *globalRaiseSalary()*, there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.
Virtual functions are so useful that later languages like Java kept all the methods as virtual by default.

How does compiler do this magic of late resolution?
Compiler maintains two things to this magic :



vtable - A table of function pointers. It is maintained per class.
vptr - A pointer to vtable. It is maintained per object.

Compiler adds additional code at two places to maintain and use *vptr*.
1) Code in every constructor. This code sets vptr of the object being created. This code sets *vptr* to point to *vtable* of the class.
2) Code with polymorphic function call (e.g. *bp->show()* in above code). Wherever a polymorphic call is made, compiler inserts code to first look for *vptr* using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, address of derived derived class function *show()* is accessed and called.

- Multiple Inheritance and Diamond Problem

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class.
The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

```
#include <bits/stdc++.h>
using namespace std;

class A
{
    public:
      A() { cout << "A's constructor called" << endl; }
};

class B
{
    public:
      B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A  // Note the order
{
    public:
      C() { cout << "C's constructor called" << endl; }
};

int main()
{
    C c;
    return 0;
}

```

Run

Output:

```
B's constructor called
A's constructor called
C's constructor called
```

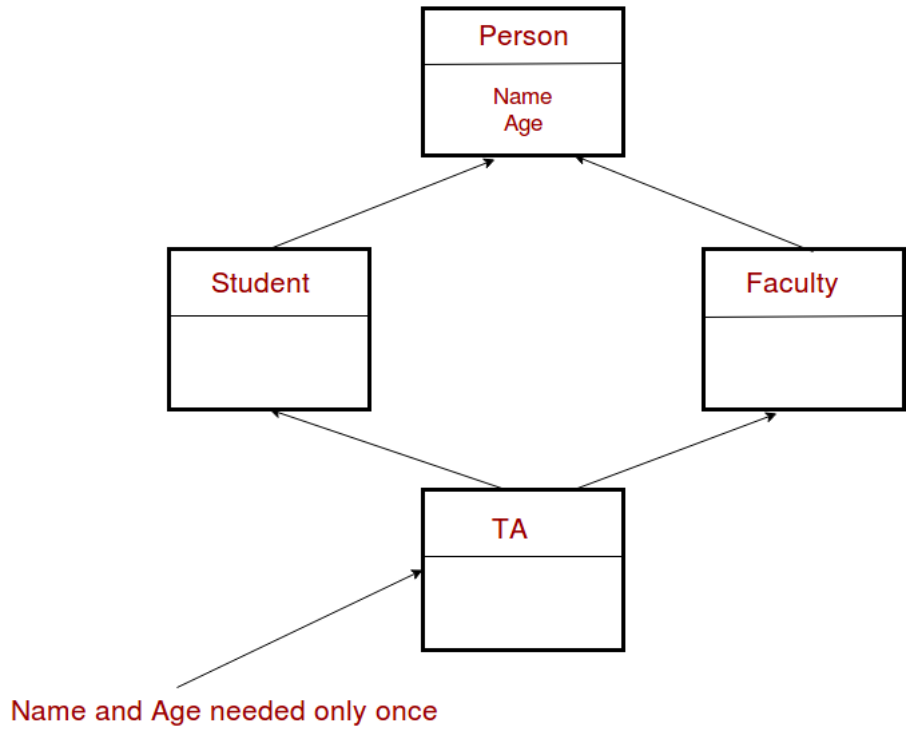
The Diamond Problem The diamond problem occurs when two superclasses of a class have a common base class. As an example, in the following diagram, the TA class gets two copies of all attributes of the Person class that leads to an ambiguity.

```
#include <bits/stdc++.h>
using namespace std;

class Person
{
    // Data members of person
    public:
        Person(int x) {
            cout << "Person::Person(int ) called" << endl;
        }
};

class Faculty : public Person
{
    // data members of Faculty
    public:
        Faculty(int x) : Person(x) {
            cout << "Faculty::Faculty(int) called" << endl;
        }
};

class Student : public Person
{
    // data members of Student
    public:
        Student(int x) : Person(x) {
            cout << "Student::Student(int) called" << endl;
        }
};
```

Run

Output:

```
Person::Person(int ) called
Faculty::Faculty(int) called
Person::Person(int ) called
Student::Student(int) called
TA::TA(int) called
```



In the above program, the constructor of the *Person* is called two times. The destructor of *Person* will also be called two times when the object *ta* is destructed. So object *ta* has two copies of all members of *Person*, this causes ambiguity. The solution to this problem is using

virtual classes. We make the classes *Faculty* and *Student* as virtual base classes to avoid two copies of *Person* in the *TA* class. As an example, consider the following program:

```
#include <bits/stdc++.h>
using namespace std;

class Person {
    public:
        Person(int x) { cout << "Person::Person(int) called" << endl; }
        Person() { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
    public:
        Faculty(int x) : Person(x)   {
            cout << "Faculty::Faculty(int) called" << endl;
        }
};

class Student : virtual public Person {
    public:
        Student(int x) : Person(x) {
            cout << "Student::Student(int) called" << endl;
        }
};

class TA : public Faculty, public Student  {
    public:
        TA(int x) : Student(x), Faculty(x) {
            cout << "TA::TA(int) called" << endl;
        }
};
```

Run

Output:

```
Person::Person() called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

In the above program, the constructor of the *Person* is called once. One important thing to note in the above output is the default constructor of *Person* is called. When we use the virtual keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call the parameterized constructor

How to call the parameterized constructor of the *Person* class? The constructor has to be called in *TA* class:

```
#include <bits/stdc++.h>
using namespace std;

class Person
{
    public:
        Person(int x) { cout << "Person::Person(int) called" << endl; }
        Person() { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person
{
    public:
        Faculty(int x) : Person(x) {
            cout << "Faculty::Faculty(int) called" << endl;
        }
};

class Student : virtual public Person
{
    public:
        Student(int x) : Person(x) {
            cout << "Student::Student(int) called" << endl;
        }
};

class TA : public Faculty, public Student  {
```

```
29      public:
30          TA(int x) : Student(x), Faculty(x), Person(x) {
```

Run

Output:

```
Person::Person(int) called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through a parent class. It is allowed only when the virtual keyword is used.

− Structs vs Classes in C++

In C++, a structure is the same as a class except for a few differences. The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details. Following are the points that expound on this difference:

1. Members of a class are private by default and members of a struct are public by default. e.g.

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  struct A {
6      int x; // x is public
7  };
8
9  class B {
10     int x; // x is private
11 };
12
13 int main()
14 {
15     A a;
16     B b;
17
18     cout << a.x << endl;
19     cout << b.x << endl;
20
21     return 0;
22 }
23
```

Run

Error generated by above code:

```
prog.cpp: In function 'int main()':
prog.cpp:10:9: error: 'int B::x' is private
     int x; // x is private
         ^
prog.cpp:19:15: error: within this context
     cout << b.x << endl;
               ^
```

Above code generates compilation-error because we tried to access data-member *x* for instance *b* (which is private).

2. When deriving a **struct from a class/struct**, default access-specifier for a base class/struct is **public**. And when **deriving a class**, the default access specifier is **private**.

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  class Base
6  {
7      public:
8          int x; // x is public
9  };
10
11 class Derived1 : Base
12 //equivalent to private Base
```

```
{
};

struct Derived2 : Base
//equivalent to public Base
{
};

int main()
{
    Derived1 d1;  //class
    Derived2 d2;  //struct

    cout << d1.x << endl;
    cout << d2.x << endl;

    return 0;
}
```


Run

Error generated by above code:

```
prog.cpp: In function 'int main()':
prog.cpp:8:13: error: 'int Base::x' is inaccessible
        int x; // x is public
            ^
prog.cpp:26:16: error: within this context
    cout << d1.x << endl;
               ^
```

The above code generates compilation error because of the access statement *d1.x*. Since we didn't specify the access-modifier for the Base class, *x* became private in Derived Class.

Company-wise (https://practice.geeksforgeeks.org/company-tags/)

Topic-wise (https://practice.geeksforgeeks.org/topic-tags/)

How to begin? (https://practice.geeksforgeeks.org/faq.php)

Write Interview Experience (https://www.geeksforgeeks.org/write-interview-experience/)

Internships (https://www.geeksforgeeks.org/internship/)

Videos (https://www.geeksforgeeks.org/how-to-contribute-videos-to-geeksforgeeks/)

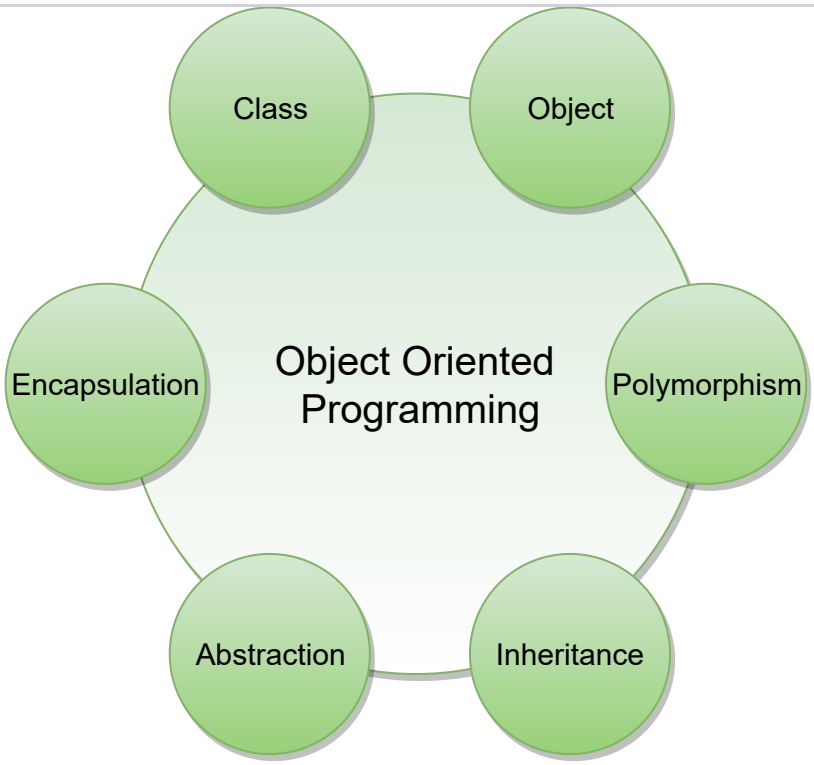
(https://in.linkedin.com/company/geeksforgeeks)

Object-Oriented Basics

Object-oriented programming (OOP) is a style of programming that focuses on using objects to design and build applications. Contrary to procedure-oriented programming where programs are designed as blocks of statements to manipulate data, OOP organizes the program to combine data and functionality and wrap it inside something called an “Object”.

If you have never used an object-oriented programming language before, you will need to learn a few basic concepts before you can begin writing any code. This chapter will introduce some basic concepts of OOP:

- **Objects:** Objects represent a real-world entity and the basic building block of OOP. For example, an Online Shopping System will have objects such as shopping cart, customer, product item, etc.
- **Class:** Class is the prototype or blueprint of an object. It is a template definition of the attributes and methods of an object. For example, in the Online Shopping System, the Customer object will have attributes like shipping address, credit card, etc., and methods for placing an order, canceling an order, etc.



OOP basics

The four principles of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism.

- **Encapsulation:** Encapsulation is the mechanism of binding the data together and hiding it from the outside world. Encapsulation is achieved when each object keeps its state private so that other objects don't have direct access to its state. Instead, they can access this state only through a set of public functions.

- **Abstraction:** Abstraction can be thought of as the natural extension of encapsulation. It means hiding all but the relevant data about an object in order to reduce the complexity of the system. In a large system, objects talk to each other, which makes it difficult to maintain a large code base; abstraction helps by hiding internal implementation details of objects and only revealing operations that are relevant to other objects.

- **Inheritance:** Inheritance is the mechanism of creating new classes from existing ones.

- **Polymorphism:** Polymorphism (from Greek, meaning “many forms”) is the ability of an object to take different forms and thus, depending upon the context, to respond to the same message in different ways. Take the example of a chess game; a chess piece can take many forms, like bishop, castle, or knight and all these pieces will respond differently to the ‘move’ message.

Next → (/courses/grokking-the-object-oriented-design-interview/m28YxDZ5kBr)

Mark as Completed

OO Analysis and Design

Stuck? Get help on DISCUSS (https://discuss.educative.io/c/grokking-the-object-oriented-design-interview-design-gurus/object-oriented-design-and-uml-object-oriented-basics)

Send feedback

20 Recommendations

OO Analysis and Design

OO Analysis and Design is a structured method for analyzing and designing a system by applying object-oriented concepts. This design process consists of an investigation into the objects constituting the system. It starts by first identifying the objects of the system and then figuring out the interactions between various objects.

The process of OO analysis and design can be described as:

1. Identifying the objects in a system;
2. Defining relationships between objects;
3. Establishing the interface of each object; and,
4. Making a design, which can be converted to executables using OO languages.

We need a standard method/tool to document all this information; for this purpose we use UML. UML can be considered as the successor of object-oriented (OO) analysis and design. UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are a representation of object-oriented concepts only. Thus, before learning UML, it is essential to understand OO concepts.

Let’s find out how we can model using UML.

← Back (/courses/grokking-the-object-oriented-design-interview/qVDnLQ75r5G)

Object-Oriented Basics

Mark as Completed

Next → (/courses/grokking-the-object-oriented-design-interview/xV63Bo7kp6n)

What is UML?

Stuck? DISCUSS

Class Diagram

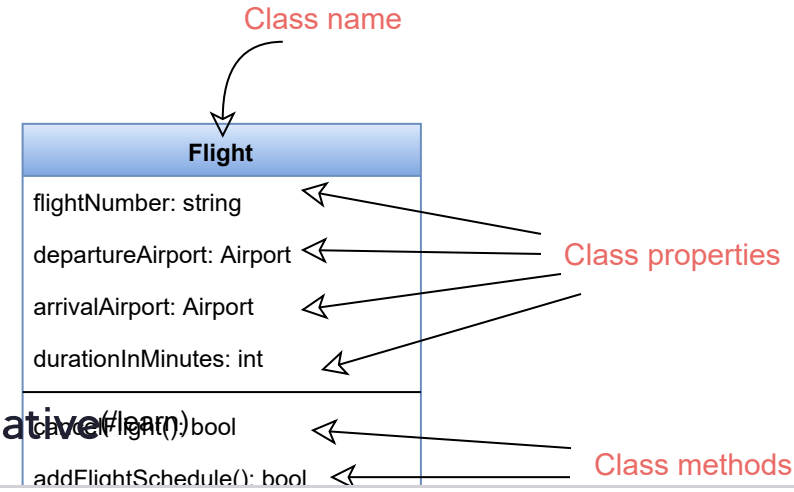
Class diagram is the backbone of object-oriented modeling - it shows how different entities (people, things, and data) relate to each other. In other words, it shows the static structures of the system.

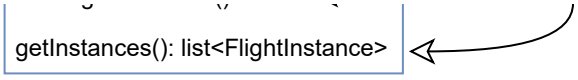
A class diagram describes the attributes and operations of a class and also the constraints imposed on the system. Class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams that can be mapped directly to object-oriented languages.

The purpose of the class diagram can be summarized as:

1. Analysis and design of the static view of an application;
2. To describe the responsibilities of a system;
3. To provide a base for component and deployment diagrams; and,
4. Forward and reverse engineering.

A class is depicted in the class diagram as a rectangle with three horizontal sections, as shown in the figure below. The upper section shows the class's name (Flight), the middle section contains the properties of the class, and the lower section contains the class's operations (or "methods").





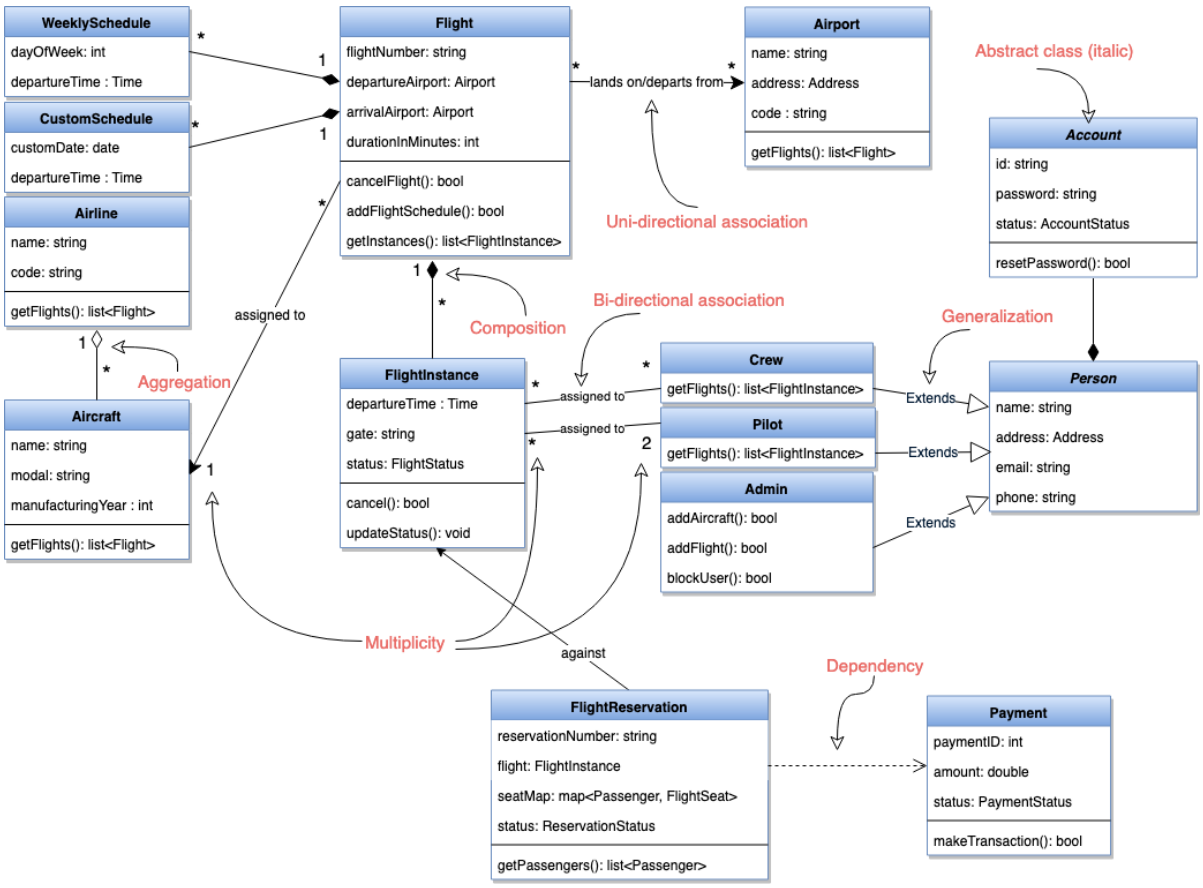
These are the different types of relationships between classes:

Association: If two classes in a model need to communicate with each other, there must be a link between them. This link can be represented by an association. Associations can be represented in a class diagram by a line between these classes with an arrow indicating the navigation direction.

- By default, associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship. In the diagram below, the association between Pilot and FlightInstance is bi-directional, as both classes know each other.
- By contrast, in a uni-directional association, two classes are related - but only one class knows that the relationship exists. In the below example, only Flight class knows about Aircraft; hence it is a uni-directional association

Multiplicity Multiplicity indicates how many instances of a class participate in the relationship. It is a constraint that specifies the range of permitted cardinalities between two classes. For example, in the diagram below, one FlightInstance will have two Pilots, while a Pilot can have many FlightInstances. A ranged multiplicity can be expressed as “0...*” which means “zero to many" or as “2...4” which means “two to four”.

We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association. The below diagram, demonstrates that a FlightInstance has exactly two Pilots but a Pilot can have many FlightInstances.



Sample class diagram for flight reservation system

Aggregation: Aggregation is a special type of association used to model a "whole to its parts" relationship. In a basic aggregation relationship, the lifecycle of a PART class is independent of the WHOLE class's lifecycle. In other words, aggregation implies a relationship where the child can exist independently of the parent. In the above diagram, Aircraft can exist without Airline.

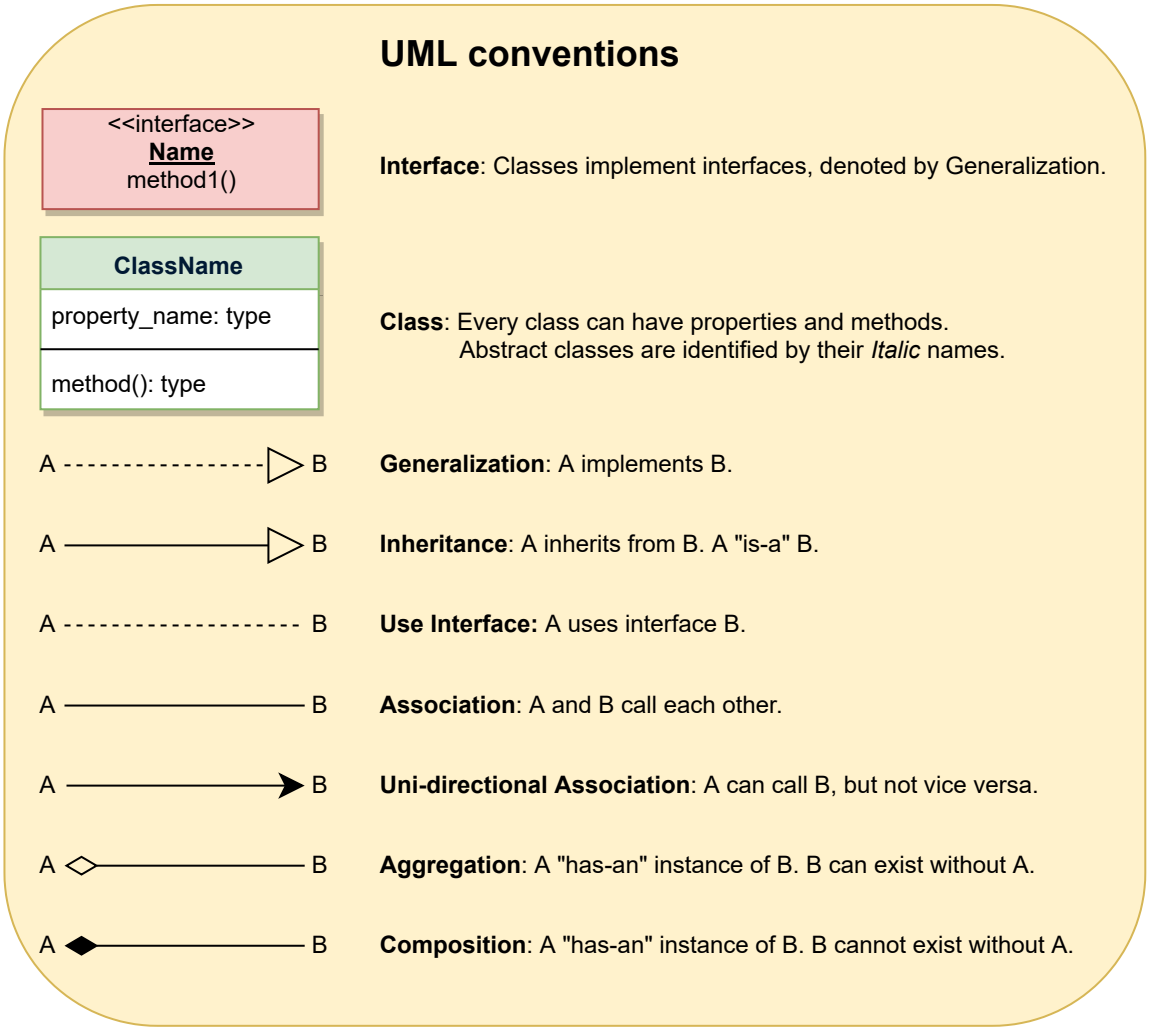
Composition: The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In other words, Composition implies a relationship where the child cannot exist independent of the parent. In the above example, WeeklySchedule is composed in Flight which means when Flight lifecycle ends, WeeklySchedule automatically gets destroyed.

Generalization: Generalization is the mechanism for combining similar classes of objects into a single, more general class. Generalization identifies

commonalities among a set of entities. In the above diagram, Crew, Pilot, and Admin, all are Person.

Dependency: A dependency relationship is a relationship in which one class, the client, uses or depends on another class, the supplier. In the above diagram, FlightReservation depends on Payment.

Abstract class: An abstract class is identified by specifying its name in *italics*. In the above diagram, both Person and Account classes are abstract classes.



← Back (/courses/grokking-the-object-oriented-design-interview/7npMYmzm8DA)

Use Case Diagrams

Mark as Completed

Next → (/courses/grokking-the-object-oriented-design-interview/7nX38BMK9NO)

Sequence diagram

educative (/learn)

Stuck? Get help on DISCUSS (https://discuss.educative.io/c/grokking-the-object-oriented-design-interview-design-gurus/object-oriented-design-and-uml-class-diagram)

Send feedback

31 Recommendations

What is UML?

UML stands for Unified Modeling Language and is used to model the Object-Oriented Analysis of a software system. UML is a way of visualizing and documenting a software system by using a collection of diagrams, which helps engineers, businesspeople, and system architects understand the behavior and structure of the system being designed.

Benefits of using UML:

1. Helps develop a quick understanding of a software system.
2. UML modeling helps in breaking a complex system into discrete pieces that can be easily understood.
3. UML's graphical notations can be used to communicate design decisions.
4. Since UML is independent of any specific platform or language or technology, it is easier to abstract out concepts.
5. It becomes easier to hand the system over to a new team.

Types of UML Diagrams: The current UML standards call for 14 different kinds of diagrams. These diagrams are organized into two distinct groups: structural diagrams and behavioral or interaction diagrams. As the names suggest, some UML diagrams analyze and depict the structure of a system or process, whereas others describe the behavior of the system, its actors, and its building components. The different types are broken down as follows:

Structural UML diagrams

- Class diagram
- Object diagram
- Package diagram
- Component diagram
- Composite structure diagram
- Deployment diagram
- Profile diagram

Behavioral UML diagrams

- Use case diagram
- Activity diagram
- Sequence diagram
- State diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram

In this course, we will be focusing on the following UML diagrams:

- **Use Case Diagram:** Used to describe a set of user scenarios, this diagram, illustrates the functionality provided by the system.

- **Class Diagram:** Used to describe structure and behavior in the use cases, this diagram provides a conceptual model of the system in terms of entities and their relationships.
- **Activity Diagram:** Used to model the functional flow-of-control between two or more class objects.
- **Sequence Diagram:** Used to describe interactions among classes in terms of an exchange of messages over time.

← Back (/courses/grokking-the-object-oriented-design-interview/m28YxDZ5kBr)

OO Analysis and Design

Mark as Completed

Next → (/courses/grokking-the-object-oriented-design-interview/7npMYmzm8DA)

Use Case Diagrams

Stuck? Get help on DISCUSS (https://discuss.educative.io/c/grokking-the-object-oriented-design-interview-design-gurus/object-oriented-design-and-uml-what-is-uml)

Send feedback

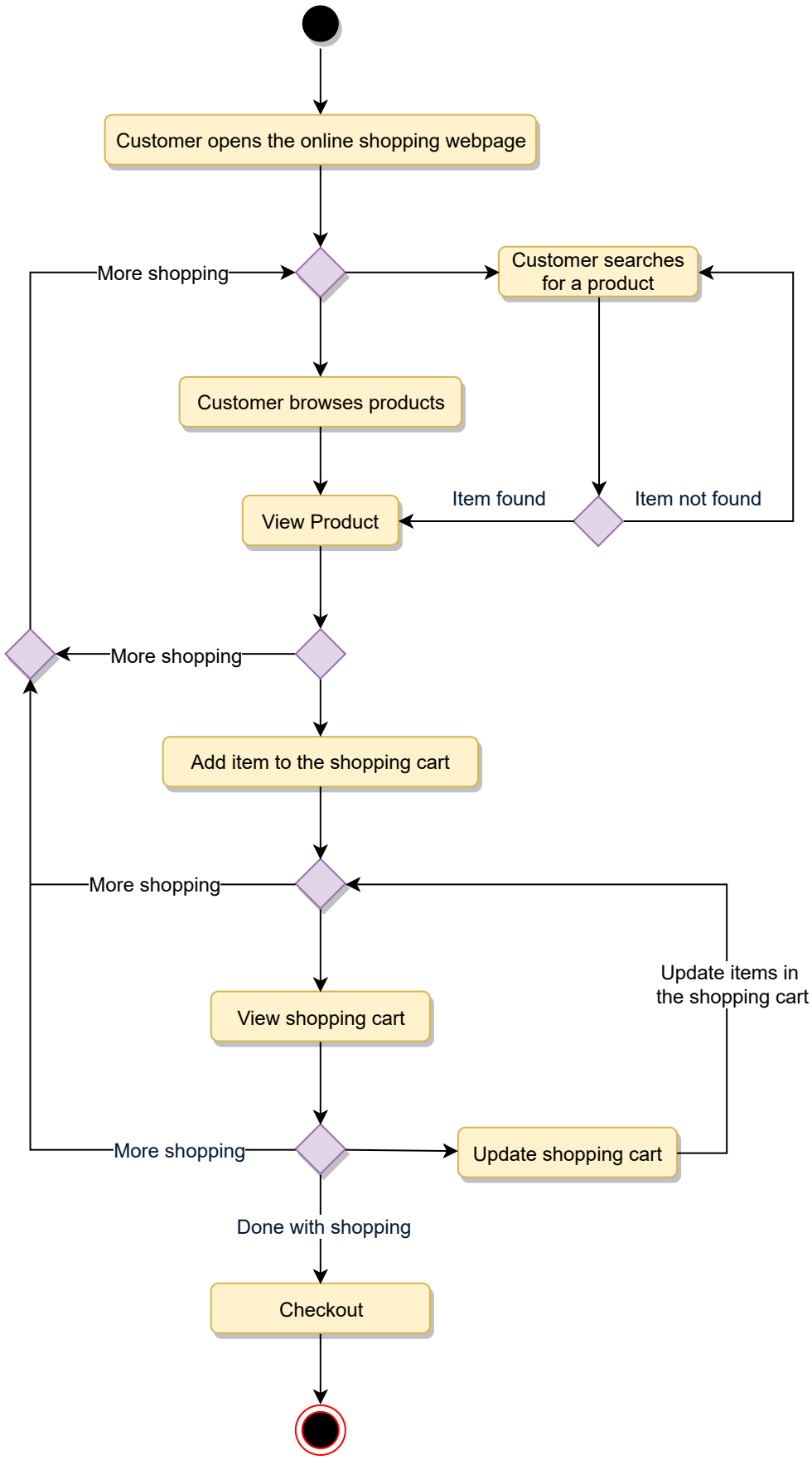
10 Recommendations

Activity Diagrams

We use Activity Diagrams to illustrate the flow of control in a system. An activity diagram shows the flow of control for a system functionality; it emphasizes the condition of flow and the sequence in which it happens. We can also use an activity diagram to refer to the steps involved in the execution of a use case.

Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operations.

Following is an activity diagram for a user performing online shopping:



Sample activity diagram for online shopping

What is the difference between Activity diagram and Sequence

diagram?

educative(/learn)

Activity diagram captures the process flow. It is used for functional modeling. A functional model represents the flow of values from external inputs, through operations and internal data stores, to external outputs. **Sequence diagram** tracks the interaction between the objects. It is used for dynamic modeling, which is represented by tracking states, transitions between states, and the events that trigger these transitions.

← Back (/courses/grokking-the-object-oriented-design-interview/7nX38BMK9NO)

Sequence diagram

Mark as Completed

Next → (/courses/grokking-the-object-oriented-design-interview/RMlM3NgjAyR)

Design a Library Management System

Stuck? Get help on DISCUSS (https://discuss.educative.io/c/grokking-the-object-oriented-design-interview-design-gurus/object-oriented-design-and-uml-activity-diagrams)

Send feedback

10 Recommendations

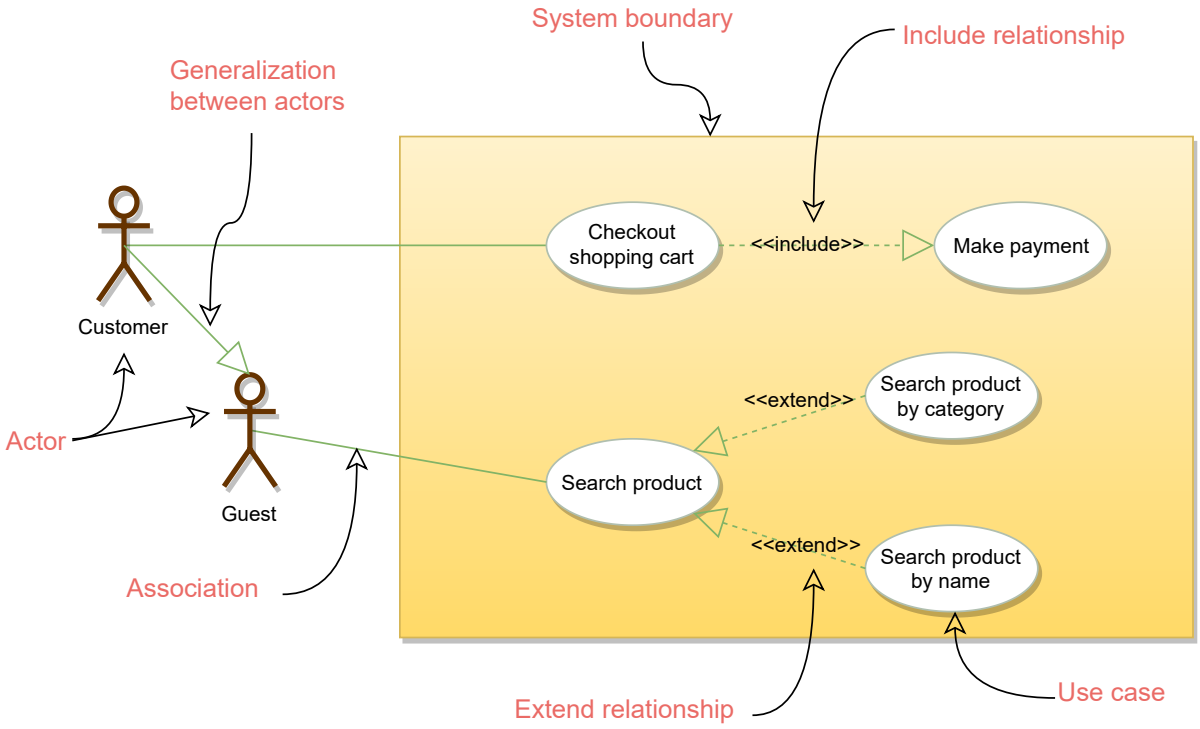
Use Case Diagrams

Use case diagrams describe a set of actions (called use cases) that a system should or can perform in collaboration with one or more external users of the system (called actors). Each use case should provide some observable and valuable result to the actors.

1. Use Case Diagrams describe the high-level functional behavior of the system.
2. It answers what system does from the user point of view.
3. Use case answers ‘What will the system do?’ and at the same time tells us ‘What will the system NOT do?’.

A use case illustrates a unit of functionality provided by the system. The primary purpose of the use case diagram is to help development teams visualize the functional requirements of a system, including the relationship of “actors” to the essential processes, as well as the relationships among different use cases.

To illustrate a use case on a use case diagram, we draw an oval in the middle of the diagram and put the name of the use case in the center of the oval. To show an actor (indicating a system user) on a use-case diagram, we draw a stick figure to the left or right of the diagram.



Sample use-case diagram for online shopping system

The different components of the use case diagram are:

- **System boundary:** A system boundary defines the scope and limits of the system. It is shown as a rectangle that spans all use cases of the system.
- **Actors:** An actor is an entity who performs specific actions. These roles are the actual business roles of the users in a given system. An actor interacts with a use case of the system. For example, in a banking system, the customer is one of the actors.
- **Use Case:** Every business functionality is a potential use case. The use case should list the discrete business functionality specified in the problem statement.
- **Include:** Include relationship represents an invocation of one use case by another use case. From a coding perspective, it is like one function being called by another function.
- **Extend:** This relationship signifies that the extended use case will

work exactly like the base use case, except that some new steps will be inserted in the extended use case.

← Back (/courses/grokking-the-object-oriented-design-interview/xV63Bo7kp6n)

What is UML?

Mark as Completed

Next → (/courses/grokking-the-object-oriented-design-interview/g7Lw3O0A2Aj)

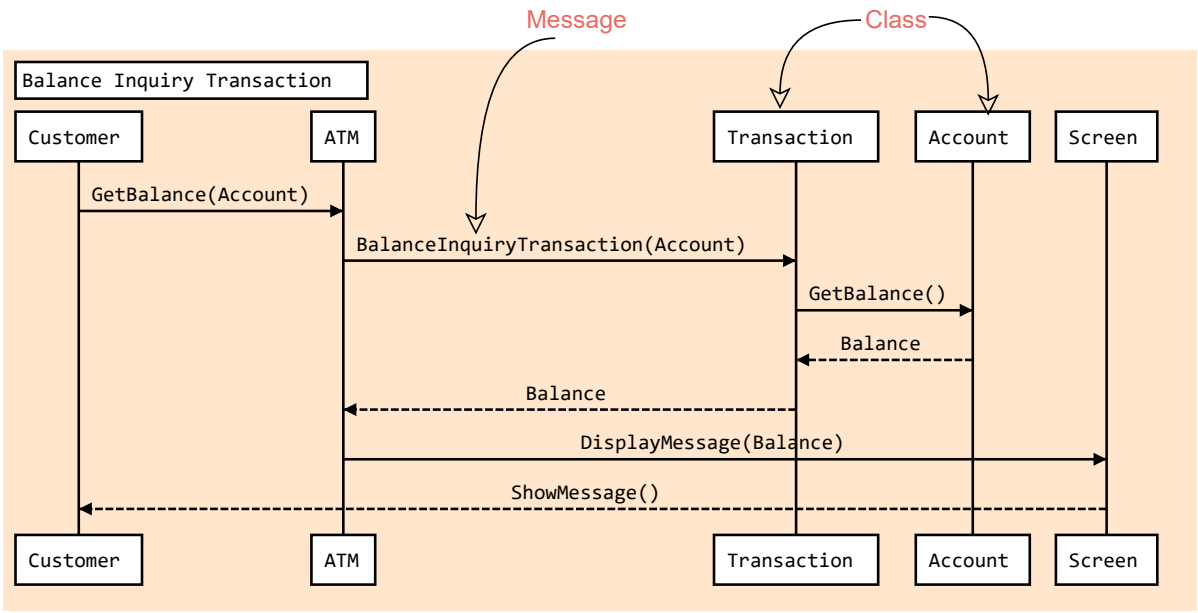
Class Diagram

Stuck? Get help on DISCUSS (https://discuss.educative.io/c/grokking-the-object-oriented-design-interview-design-gurus/object-oriented-design-and-uml-use-case-diagrams)

Send feedback

9 Recommendations

Sequence diagram



Sample sequence diagram for ATM balance inquiry

← Back (/courses/grokking-the-object-oriented-design-interview/g7Lw3O0A2Aj)

Class Diagram

Mark as Completed

Next → (/courses/grokking-the-object-oriented-design-interview/B8RPL3VEI8N)

Activity Diagrams

Stuck? Get help on DISCUSS (https://discuss.educative.io/c/grokking-the-object-oriented-design-interview-design-gurus/object-oriented-design-and-uml-sequence-diagram)

Send feedback

12 Recommendations