

Inheritance

Access Modifier

1. Public
2. Private
3. Protected

<pre>Public Class A { Int i , j; void abc () { S=u+t; } Int x , y }</pre>	<ol style="list-style-type: none">1. Particular member access modifier2. Access modifier block3. Class declaration as a public
--	--

Concept Of Inheritance

- Way-1 : calling parent class component from main method using child class object reference
- Way-2 : calling parent class member from child class

	Mode of inheritance		
parent class access modifier	Public	Protected	private
Public	w1 ,w 2 Publicly	W1 , w2 Protectedly	W1 , w2 Privately (2 nd generation is locked in multi level)
Protected	W1 , w2 protectedly	W1 , w2 Protectedly	W1 , w2 Privately (2 nd generation is locked in multi level)
Private	W1 , W2 Not possible	W1 , W2 Not possible	W1 , W2 Not possible

Mode Of inheritance

- Public Mode
- Private Mode
- Protected Mode

Public mode	Protected mode	Private Mode
<pre>class classOne{ public: }; class classTwo : public classOne{ } };</pre>	<pre>class classOne{ public: }; class classTwo : Protected classOne{ } };</pre>	<pre>class classOne{ public: }; class classTwo : private classOne{ } };</pre>

Types Of Inheritance

1. **Single Inheritance**
 2. **Multiple Inheritance**
 3. **Multilevel Inheritance**
 4. Hybrid Inheritance
 5. Hierarchical Inheritance
1. Single

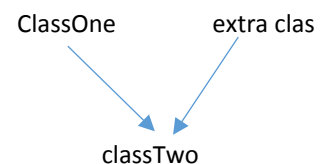
```
class classOne{
public:
};

class classTwo : public classOne{
}
};
```

2. Multiple Inheritance

```
class classOne{
public:
};

class extraClass{
public:
```



```
};

class classTwo : public classOne , extraClass{
}

};
```

3. Multi Level Inheritance

```
class classOne{
public:

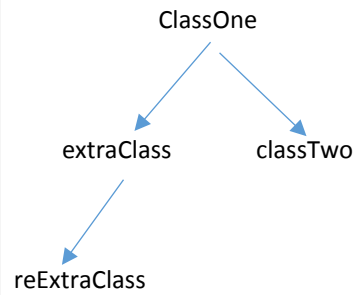
};

class extraClass : public classOne{
public:
};

Class reExtra : public extraClass {
Public:
};

class classTwo : public classOne {
}

};
```



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 and protected members of other classes.

We can declare a friend class in C++ by using the **friend** keyword.

class1 class2 class3 classN

Public

Protected

Private

Real life case

- Custom content
- Page - owner/ admin - moderation
- Reliable second mail , id

[NB: details of friend class, friend function – Book]

Access Modifiers

1. Public
2. Private
3. Protected

Ways of mentioning or using an access modifier

<pre>Class A { public : Int i , j; void bc () { S=u+t; } public: Int x , y }</pre>	<pre>public Class B { Int i , j; void ab () { S=u+t; } Int x , y }</pre>	<pre>Class C { Public Int i , Private int j; Public int ac () { B on; on.abc(); S=u+t; } Int x , y }</pre>	<pre>Int main () { C obj; Obj.ac; }</pre>
--	--	--	---

Let us now look at each one of these access modifiers in detail:

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.

	<pre> public Class B { Int i , j; void ab () { S=u+t; } Int x , y } </pre>	<pre> Class C { Public Int i , Private int j; Public int ac () { B on; on.ab(); } Int x , y } </pre>	<pre> Int main () { C obj; Obj.ac; } </pre>
--	--	--	---

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 the class.

<pre> Class A { private: Int i , j; void bc () { S=u+t; } Void nm() { Cout << I // private member are </pre>		<pre> Class C { Public Int i , Private int j; Public int ac () { A on; on.bc(); // A class member is private , so not accessible } } </pre>	<pre> Int main () { C obj; Obj.ac; } </pre>
---	--	--	---

accessible inside the class }		Int x , y }	
private: Int x , y }			

Encapsulation (Getter Setter Method)



Encapsulate

Private:

Int pet;

Int speed;

Int getPet()

{}

Int setSet()

{}

Polimorphism

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 a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

1. Compile Time Polimorphism

- Method Overloading
- Operator Overloading

2. Runtime Polimorphism

- Method overriding
- Virtual function

Based on the functionality, you can categorize polymorphism into two types:

● Compile-Time Polymorphism:

When the relationship between the definition of different functions and their function calls, is determined during the compile-time, it is known as compile-time polymorphism. This type of polymorphism is also known as static or early binding polymorphism. All the methods of compile-time polymorphism get called or invoked during the compile time.

You can implement compile-time polymorphism using function overloading and operator overloading. Method/function overloading is an implementation of compile-time polymorphism where the same name can be assigned to more than one method or function, having different arguments or signatures and different return types. Compile-time polymorphism has a much faster execution rate since all the methods that need to be executed are called during compile time. However, it is less preferred for handling complex problems since all the methods and details come to light only during the compile time. Hence, debugging becomes tougher.

The implementation of compile-time polymorphism is achieved in two ways:

- Function overloading

- Operator overloading

Runtime Polymorphism

In runtime polymorphism, the compiler resolves the object at run time and then it decides which function call should be associated with that object. It is also known as dynamic or late binding polymorphism. This type of polymorphism is executed through [virtual functions](#) and function overriding. All the methods of runtime polymorphism get invoked during the run time.

Method overriding is an application of run time polymorphism where two or more functions with the same name, arguments, and return type accompany different classes of the same structure. This method has a comparatively slower execution rate than compile-time polymorphism since all the methods that need to be executed are called during run time. Runtime polymorphism is known to be better for dealing with complex problems since all the methods and details turn up during the runtime itself.

The implementation of run time polymorphism can be achieved in two ways:

- Function overriding
- Virtual functions

Function Overriding : (reference - Programiz)

In [C++ inheritance](#), we can have the same function in the base class as well as its derived classes.

When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.

So, different functions are executed depending on the object calling the function.

This is known as **function overriding** in C++. For example,

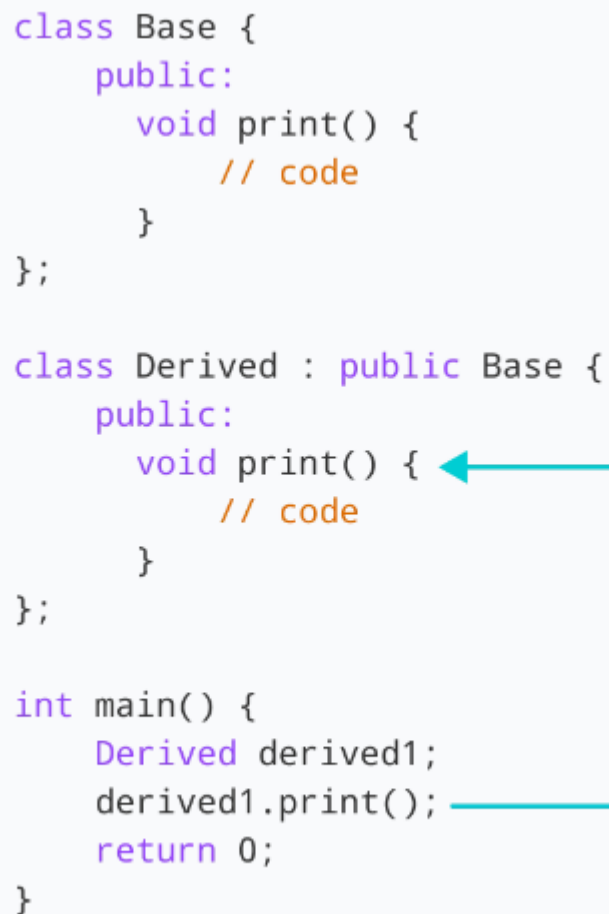
Example :

1. Animal code (in Program file)
- 2, Code-2

```
class Base {
    public:
        void print() {
            // code
        }
};

class Derived : public Base {
    public:
        void print() { ←
            // code
        }
};

int main() {
    Derived derived1;
    derived1.print(); ←
    return 0;
}
```



Here, the same function `print()` is defined in both Base and Derived classes. So, when we call `print()` from the Derived object `derived1`, the `print()` from Derived is executed by overriding the function in Base.

As we can see, the function was overridden because we called the function from an object of the Derived class. Had we called the `print()` function from an object of the Base class, the function would not have been overridden.

3, Code -3

C++ Access Overridden Function to the Base Class

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
int main() {  
    Derived derived1, derived2;  
  
    derived1.print();  
  
    derived2.Base::print();  
  
    return 0;  
}
```

The diagram consists of two teal arrows. The first arrow originates from the `derived1.print();` line in the `main()` function and points to the `void print() {` line inside the `Derived` class. The second arrow originates from the `derived2.Base::print();` line in the `main()` function and points to the `void print() {` line inside the `Base` class.

4, Code- 4

Call overridden function from derived class .

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
            Base::print();  
        }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```

The diagram illustrates the execution flow of the provided C++ code. It features three main components: the `Base` class, the `Derived` class, and the `main` function. A red arrow originates from the `derived1.print();` line in the `main` function, pointing to the `void print()` method of the `Derived` class. Another red arrow starts from the `Base::print();` line within the `Derived::print()` method, pointing to the `void print()` method of the `Base` class. This visualizes the call stack where the `main` function calls the `Derived` method, which in turn calls the `Base` method.

5, Code- 5

Calling overridden function using pointer

```
using namespace std;

class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* ptr = &derived1;

    // call function of Base class using ptr
    ptr->print();

    return 0;
}
```

Virtual Function

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:

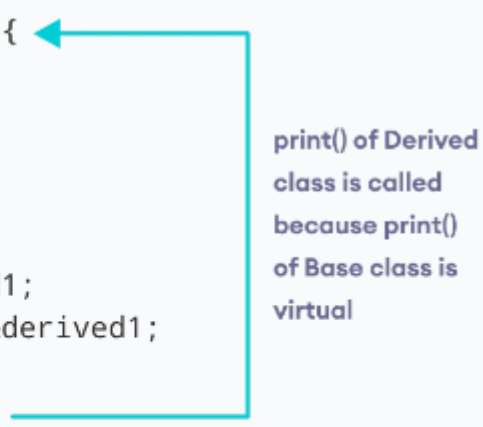
```
class Base {
public:
    virtual void print() {
        // code
    }
};

class Derived : public Base {
public:
    void print() {
        // code
    }
};

int main() {
    Derived derived1;
    Base* base1 = &derived1;

    base1->print();

    return 0;
}
```



print() of Derived class is called because print() of Base class is virtual

Later, if we create a pointer of Base type to point to an object of Derived class and call the print() function, it calls the print() function of the Base class.

In other words, the member function of Base is not overridden.

In order to avoid this, we declare the print() function of the Base class as virtual by using the virtual keyword.

C++ override Identifier

C++ 11 has given us a new identifier `override` that is very useful to avoid bugs while using virtual functions.

This identifier specifies the member functions of the derived classes that override the member function of the base class.

For example,

```
class Base {  
    public:  
        virtual void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() override {  
            // code  
        }  
};
```

If we use a function prototype in `Derived` class and define that function outside of the class, then we use the following code:

```
class Derived : public Base {  
    public:  
        // function prototype  
        void print() override;  
};
```

```
// function definition
void Derived::print() {

    // code

}
```

Use of C++ override

When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.

Using the override identifier prompts the compiler to display error messages when these mistakes are made.

Otherwise, the program will simply compile but the virtual function will not be overridden.

Some of these possible mistakes are:

- **Functions with incorrect names:** For example, if the virtual function in the base class is named `print()`, but we accidentally name the overriding function in the derived class as `pint()`.
- **Functions with different return types:** If the virtual function is, say, of `void` type but the function in the derived class is of `int` type.
- **Functions with different parameters:** If the parameters of the virtual function and the functions in the derived classes don't match.
- No virtual function is declared in the base class.

Use of C++ Virtual Functions

Suppose we have a base class `Animal` and derived classes `Dog` and `Cat`.

Suppose each class has a data member named `type`. Suppose these variables are initialized through their respective constructors.

```
class Animal {

    private:

        string type;

        ... ..

    public:

        Animal(): type("Animal") {}

}
```



```

    ... ..
};

class Dog : public Animal {

    private:

        string type;

        ... ..

    public:

        Animal(): type("Dog") {}

        ... ..

};

```

```

class Cat : public Animal {

    private:

        string type;

        ... ..

    public:

        Animal(): type("Cat") {}

        ... ..

};

```

Now, let us suppose that our program requires us to create two public functions for each class:

1. `getType()` to return the value of `type`
2. `print()` to print the value of `type`

We could create both these functions in each class separately and override them, which will be long and tedious.

Or we could make `getType()` **virtual** in the `Animal` class, then create a single, separate `print()` function that accepts a pointer of `Animal` type as its argument. We can then use this single function to override the virtual function.

```

class Animal {

    ... ..

    public:

```

```

... ..

virtual string getType {...}

};

... ..

```

```

... ..

void print(Animal* ani) {

    cout << "Animal: " << ani->getType() << endl;

}

```

This will make the code **shorter**, **cleaner**, and **less repetitive**.

Example 2: C++ virtual Function Demonstration

```

// C++ program to demonstrate the use of virtual function

#include <iostream>#include <string>using namespace std;
class Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Animal() : type("Animal") {}

        // declare virtual function
        virtual string getType() {
            return type;
        }
};

class Dog : public Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Dog() : type("Dog") {}

        string getType() override {
            return type;
        }
};

```

```

class Cat : public Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Cat() : type("Cat") {}
        string getType() override {
            return type;
        }
};

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();

    print(animal1);
    print(dog1);
    print(cat1);

    return 0;
}

```

Output

```
Animal: Animal
```

```
Animal: Dog
```

```
Animal: Cat
```

Here, we have used the virtual function `getType()` and an `Animal` pointer `ani` in order to avoid repeating the `print()` function in every class.

```

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

```

In `main()`, we have created 3 `Animal` pointers to dynamically create objects of `Animal`, `Dog` and `Cat` classes.

```
// dynamically create objects using Animal pointers  
  
Animal* animal1 = new Animal();  
  
Animal* dog1 = new Dog();  
  
Animal* cat1 = new Cat();
```

We then call the `print()` function using these pointers:

1. When `print(animal1)` is called, the pointer points to an `Animal` object. So, the virtual function in `Animal` class is executed inside of `print()`.
2. When `print(dog1)` is called, the pointer points to a `Dog` object. So, the virtual function is overridden and the function of `Dog` is executed inside of `print()`.
3. When `print(cat1)` is called, the pointer points to a `Cat` object. So, the virtual function is overridden and the function of `Cat` is executed inside of `print()`.

Class Diagram

UML Class Notation

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the **only mandatory information**.*



Class Name:

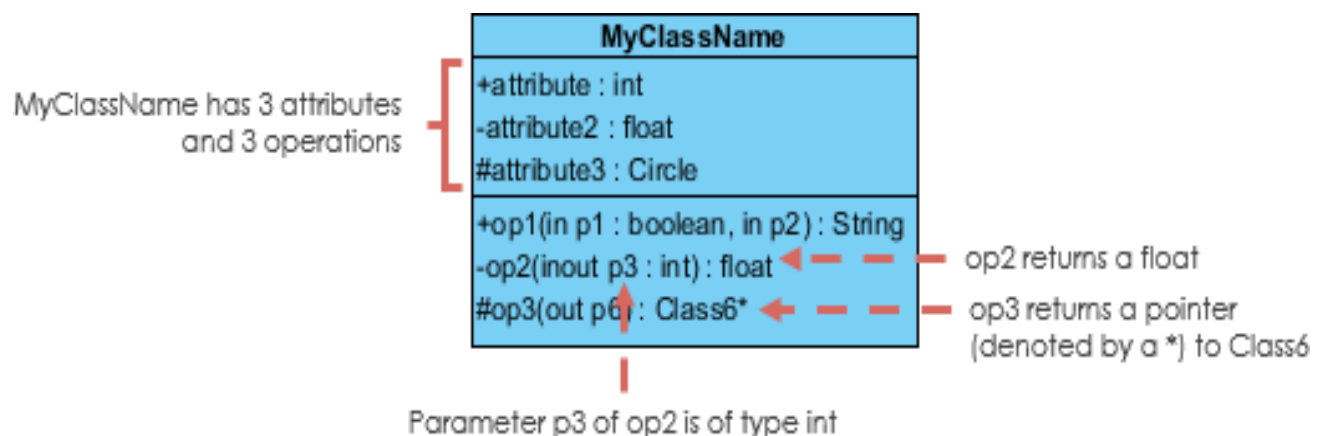
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

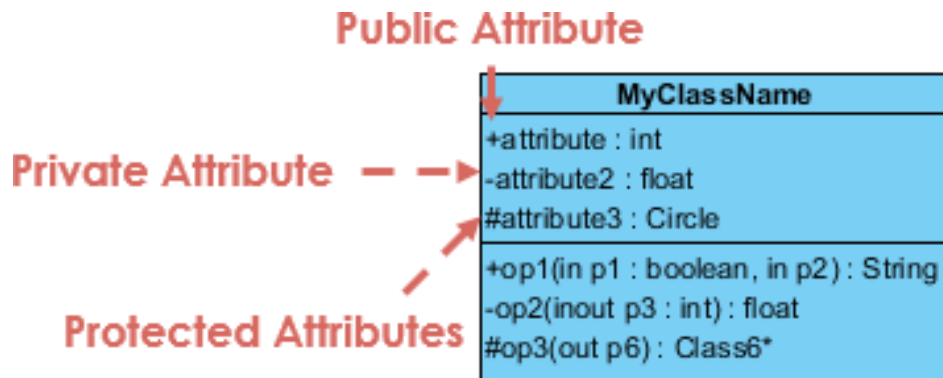
Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



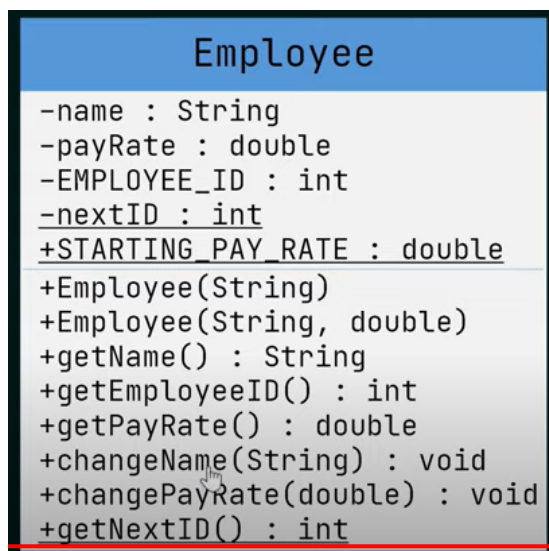
Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations

Static and Final member



Static - members are underlined (nextID)

Cons - members are written in capital letter (STRATING_PAY_RATE)

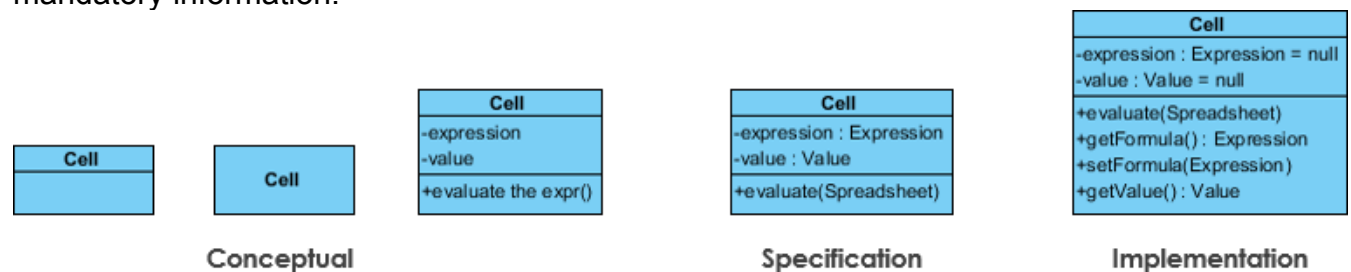
Perspectives of Class Diagram

The choice of perspective depends on how far along you are in the development process. During the formulation of a **domain model**, for example, you would seldom move past the **conceptual perspective**. **Analysis models** will typically feature a mix of **conceptual and specification perspectives**. **Design model** development will typically start with heavy emphasis on the **specification perspective**, and evolve into the **implementation perspective**.

A diagram can be interpreted from various perspectives:

- **Conceptual**: represents the concepts in the domain
- **Specification**: focus is on the interfaces of Abstract Data Type (ADTs) in the software
- **Implementation**: describes how classes will implement their interfaces
-

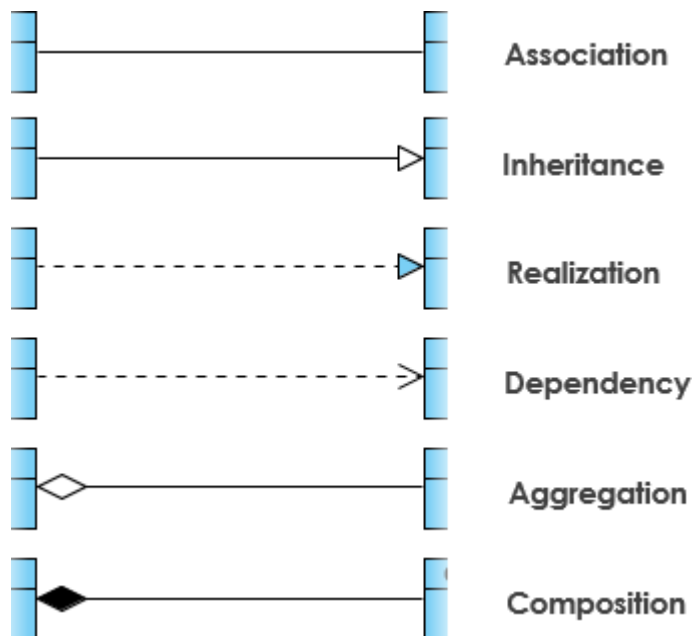
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?

If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

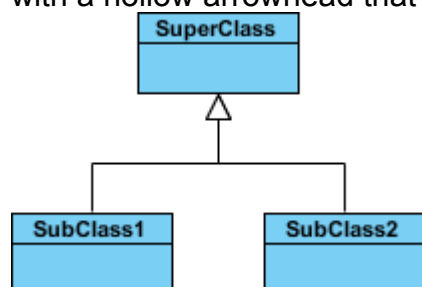


Inheritance (or Generalization):

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

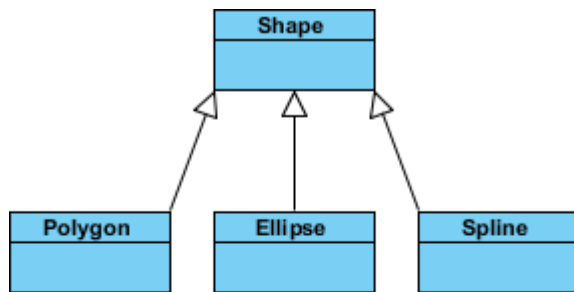
- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.

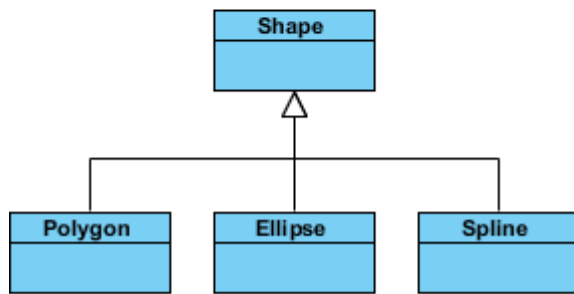


Inheritance Example - Shapes

The figure below shows an inheritance example with two styles. Although the connectors are drawn differently, they are semantically equivalent.



Style 1: Separate target



Style 2: Shared target

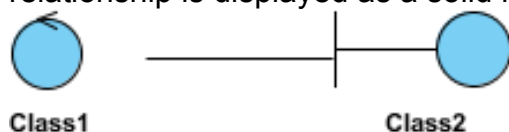
Association

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.

Simple Association

- A structural link between two peer classes.
- There is an association between Class1 and Class2

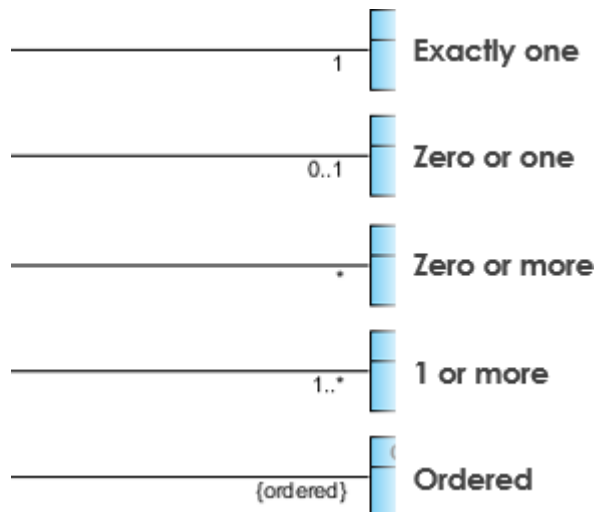
The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



Cardinality

Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

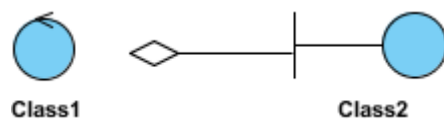


Aggregation

A special type of association.

- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.



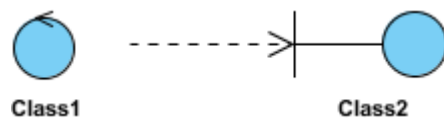
Dependency

An object of one class might use an object of another class in the code of a method.

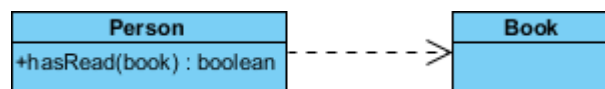
If the object is not stored in any field, then this is modeled as a dependency relationship.

- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.



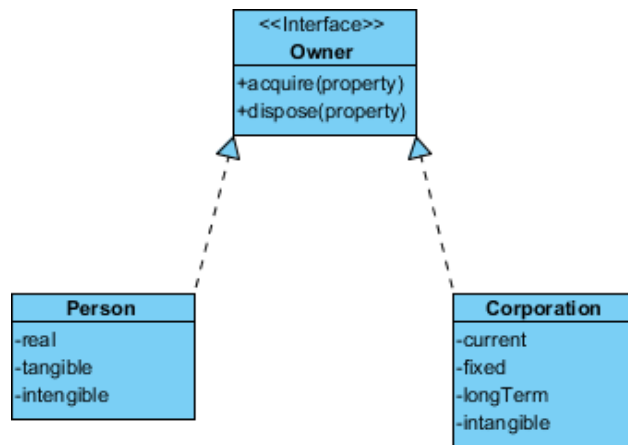
The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



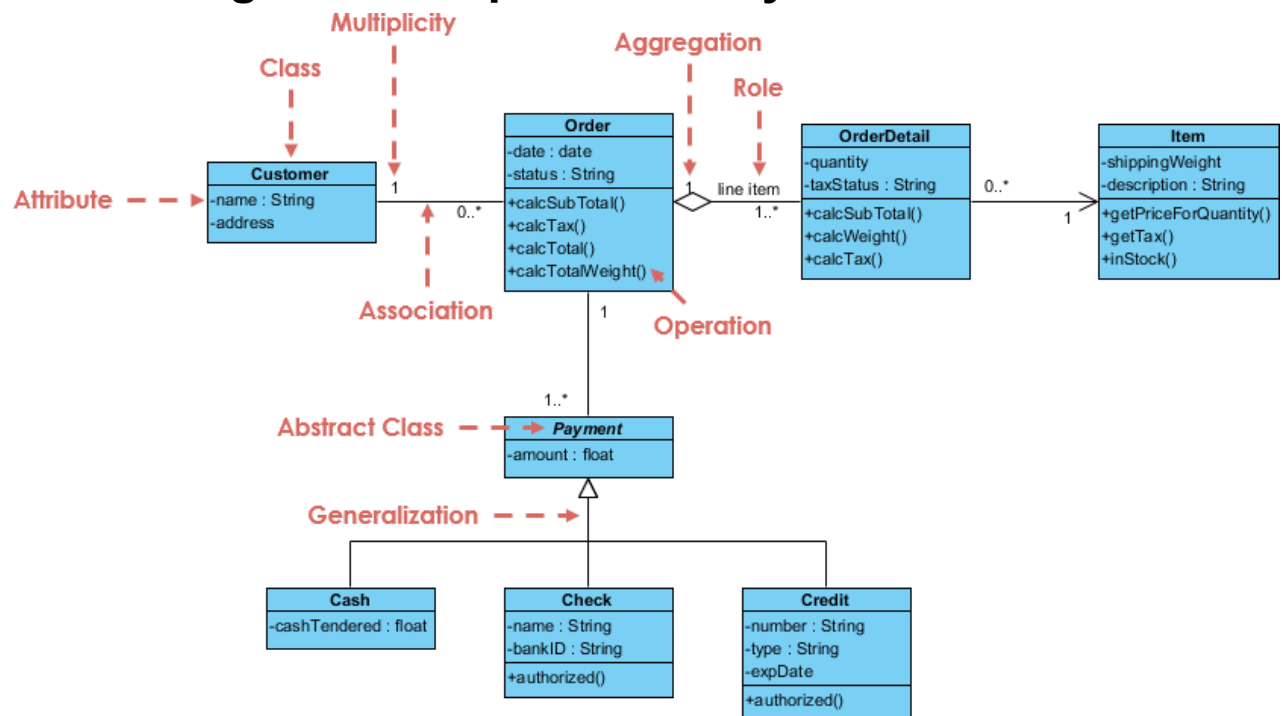
Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.

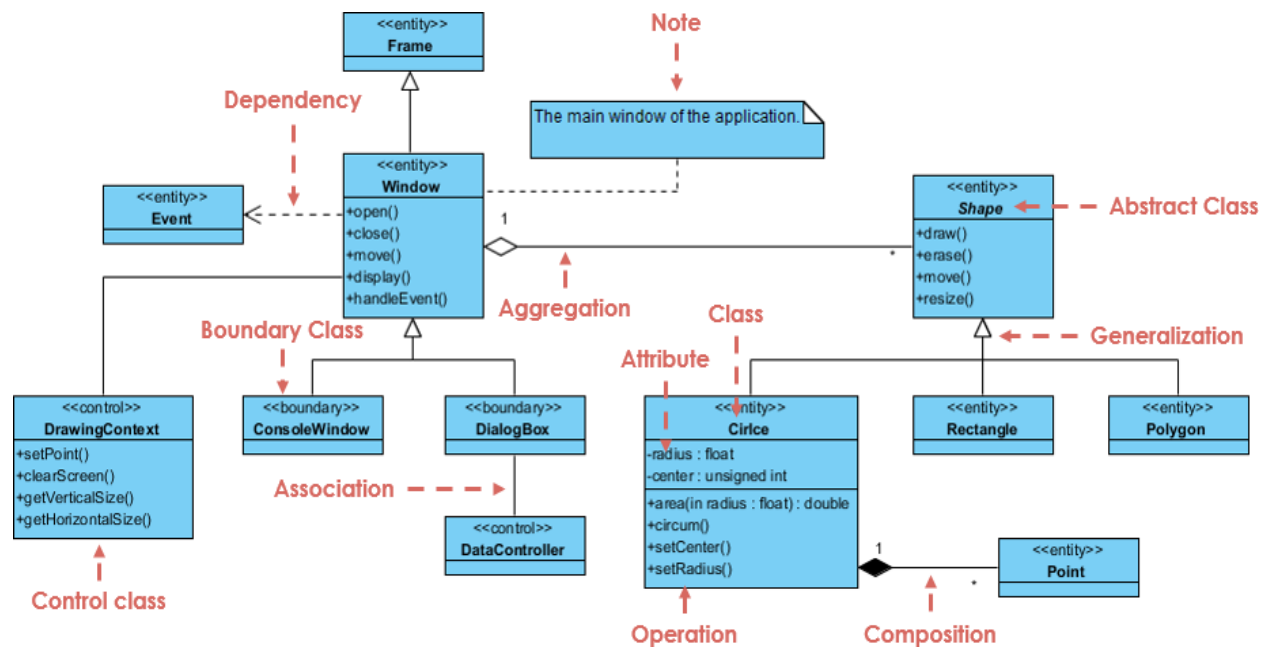


Class Diagram Example: Order System



Class Diagram Example: GUI

A class diagram may also have notes attached to classes or relationships.



Resources : (Class Diagram)

1. For study -

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

2. For Practice -

[OOP Exercises - Java Programming Tutorial \(ntu.edu.sg\)](http://www.ntu.edu.sg)

(Posted in google classroom)

3. Some Question :

- Draw a class diagram for a library management system

File System / File Handle

1. Connect with files
 2. Open a file from your code
 3. Open an existing from code
 4. Writing on a file
 5. Read from an existing file documents
 6. Changing file document
 7. Search in a file
 8. Alphabetically search
 9. Count
-
-

Modes of opening a file :

1. Read - r
2. Write - w
3. Append - a

23/11/2023

OOP keyword

1. this
2. Static
3. Final
4. const
5. Super

Static Keyword

Resource - reference book + geeks for geeks

The static keyword has different meanings when used with different types. We can use static keywords with:

1. **Static Variables:** Variables in a function, Variables in a class
2. **Static Members of Class:** Class objects and Functions in a class Let us now look at each one of these uses of static in detail.

1. Static Variables

Static variables in a Function: When a variable is declared as static, space for it **gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call. This is useful for implementing any application where the previous state of function needs to be stored.

Code - static.cpp

Static variables in a class: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are shared by the objects**. There can not be multiple copies of the same static variables for different objects. Also because of this reason static variables can not be initialized using constructors.

Code - static2.cpp (**Error**)

Code - static3.cpp (correct way)

Error Reason (static2.cpp)
You can see in the above program that we have tried to create multiple copies of the static variable i for multiple objects. But this didn't happen.

Solve (static3.cpp)
So, a static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class as shown in static3.cpp

2. Static Members of Class:

1 Class objects as static:

2 Static functions in a class:

2.1 Class objects as static: Just like variables, objects also when declared as static have a scope till the lifetime of the program. Consider the below program where the object is non-static. (MyClass obj();)

Code - staticObject1.cpp

Code - StaticObject2.cpp

Description - staticObject1.cpp

In the above program, the object is declared inside the if block as non-static. So, the scope of a variable is inside the if block only. So when the object is created the constructor is invoked and soon as the control of if block gets over the destructor is invoked as the scope of the object is inside the if block only where it is declared. Let us now see the change in output if we declare the object as static

Description - staticObject2.cpp

You can clearly see the change in output. Now the destructor is invoked after the end of the main. This happened because the scope of static objects is throughout the lifetime of the program.

[for better understanding read the following . source - static object in OOP - geeks for geeks - <https://www.geeksforgeeks.org/static-objects-in-cpp/>]

An object becomes static when a **static** keyword is used in its declaration. Static objects are initialized only once and live until the program terminates. They are allocated storage in the data segment or BSS segment of the memory.

C++ supports two types of static objects:

1. Local Static Objects
2. Global Static Objects.

Syntax:

```
Test t;                // Stack based object
static Test t1;        // Static object
```

The first statement when executes creates an object on the stack means storage is allocated on the stack. Stack-based objects are also called automatic objects or local objects. The second statement creates a static object in the data segment or BSS segment of the memory.

Local Static Object

Local static objects in C++ are those static objects that are declared inside a block. Even though their lifespan is till the termination of the program, their scope is limited to the block in which they are declared.

Example:

- C++

```
// C++ program to demonstrate the
// usage of static keyword
#include <iostream>
class Test {
```



```

public:
    Test() { std::cout << "Constructor is executed\n"; }
    ~Test() { std::cout << "Destructor is executed\n"; }
};

void myfunc()
{
    // local static object declaration
    static Test obj;
}

int main()
{
    std::cout << "main() starts\n";
    myfunc(); // calling function with static object
    std::cout << "main() terminates\n";
    return 0;
}

```

Output

main() starts

Constructor is executed

main() terminates

Destructor is executed

If we observe the output of this program closely, we can see that the destructor for the local object named **obj** is not called after it goes out of scope because the local object **obj** is static with its lifetime till the end of the program. That is why its destructor will be called when main() terminates.

Now, what happens when we remove static in the above program?

As an experiment if we remove the static keyword from the global function myfunc(), we get the output shown below:

main() starts

Constructor is called

Destructor is called

main() terminates

This is because the object **obj** is now a stack-based object and it is destroyed when it goes out of scope and its destructor will be called.

Global Static Object

Global static objects are those objects that are declared outside any block. Their scope is the whole program and their lifespan is till the end of the program.

Example:

- C++

```

// C++ program to demonstrate a global static keyword
#include <iostream>
class Test {
public:
    int a;
    Test()
    {
        a = 10;
    }
}

```

```

        std::cout << "Constructor is executed\n";
    }
    ~Test() { std::cout << "Destructor is executed\n"; }
};
static Test obj;
int main()
{
    std::cout << "main() starts\n";
    std::cout << obj.a;
    std::cout << "\nmain() terminates\n";
    return 0;
}

```

Output:

```

Constructor is executed

main() starts

10

main() terminates

Destructor is executed

```

We can see that the global static object is constructed even before the main function.

When are static objects destroyed?

As seen in both examples, static objects are always destroyed at the end of the program whether their scope is local or global. This property allows them to retain their value between multiple function calls and we can even use a pointer to them to access them out of scope.

Whether you're preparing for your first job interview or aiming to upskill in this ever-evolving tech landscape, [GeeksforGeeks Courses](#) are your key to success. We provide top-quality content at affordable prices, all geared towards accelerating your growth in a time-bound manner. Join the millions we've already empowered, and we're here to do the same for you.

2.2 Static functions in a class: Just like the static data members or static variables inside the class, static member functions also do not depend on the object of the class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator. **Static member functions are allowed to access only the static data members or other static member functions**, they can not access the non-static data members or member functions of the class.

A static function in C++ is a member function of a class that is associated with the class itself rather than with an instance or object of the class. This means that a static function can be called without creating an instance of the class.

Code - StaticFunction.cpp

Code - StaticFunction2.cpp

Destructor in CPP

What is a destructor?

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

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

The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory

Code : Des.cpp