OOP Object and class?inheritance, abstraction,encapsulation? primitive and non-primitive datatypes? structure and class? stack and heap? reference type and value type? interface and abstract class? abstract method? final method, variable and class? static method and variable? polymorphism? virtual keyword? pure virtual method? aggregation and composition? shallow copy and deep copy? exception handling? protected members? singleton class? constructor and destructor presidence? overload and override? upcasting and downcasting? stack overflow?

Inheritance:

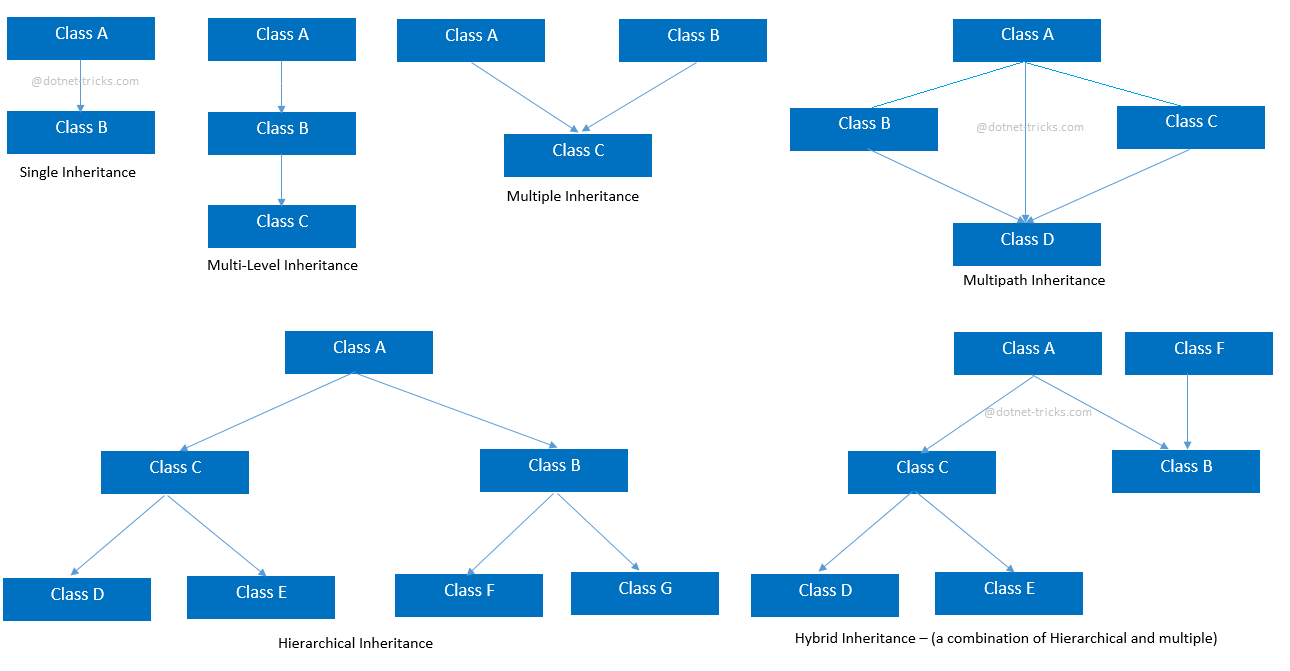
Inheritance is a mechanism of acquiring the features and behaviors of a class by another class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. Inheritance implements the IS-A relationship.

For example, mammal IS-A animal, dog IS-A mammal; Hence dog IS-A animal as well.

## Different Types of Inheritance

OOPs support the six different types of inheritance as given below :

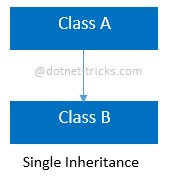
1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Multipath inheritance
5. Hierarchical Inheritance
6. Hybrid Inheritance



### **Single inheritance**

In this inheritance, a derived class is created from a single base class.

In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.



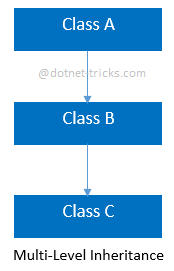
### **Syntax for Single Inheritance**

* 1. *//Base Class*
  2. class A
  3. {
  4. public void fooA()
  5. {
  6. *//TO DO:*
  7. }
  8. }
  10. *//Derived Class*
  11. class B : A
  12. {
  13. public void fooB()
  14. {
  15. *//TO DO:*
  16. }
  17. }

### **Multi-level inheritance**

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

In the given example, class c inherits the properties and behavior of class B and class B inherits the properties and behavior of class B. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.



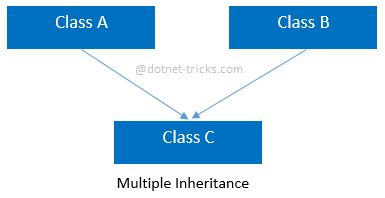
### **Syntax for Multi-level Inheritance**

* 1. *//Base Class*
  2. class A
  3. {
  4. public void fooA()
  5. {
  6. *//TO DO:*
  7. }
  8. }
  10. *//Derived Class*
  11. class B : A
  12. {
  13. public void fooB()
  14. {
  15. *//TO DO:*
  16. }
  17. }
  19. *//Derived Class*
  20. class C : B
  21. {
  22. public void fooC()
  23. {
  24. *//TO DO:*
  25. }
  26. }

### **Multiple inheritance**

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by [.NET](https://www.dotnettricks.com/learn/netframework)Languages like C#, F# etc. and Java Language.

In the given example, class c inherits the properties and behavior of class B and class A at same level. So, here A and Class B both are the parent classes for Class C.



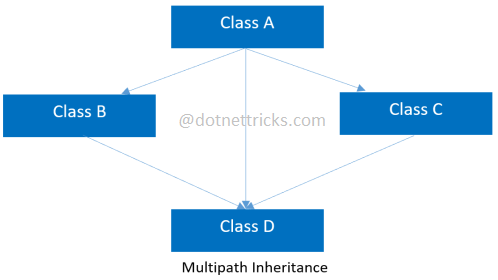
### **Syntax for Multiple Inheritance**

* 1. *//Base Class*
  2. class A
  3. {
  4. public void fooA()
  5. {
  6. *//TO DO:*
  7. }
  8. }
  10. *//Base Class*
  11. class B
  12. {
  13. public void fooB()
  14. {
  15. *//TO DO:*
  16. }
  17. }
  19. *//Derived Class*
  20. class C : A, B
  21. {
  22. public void fooC()
  23. {
  24. *//TO DO:*
  25. }
  26. }

### **Multipath inheritance**

In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes. This inheritance is not supported by .NET Languages like C#, F# etc.

In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherits the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it Multipath inheritance.



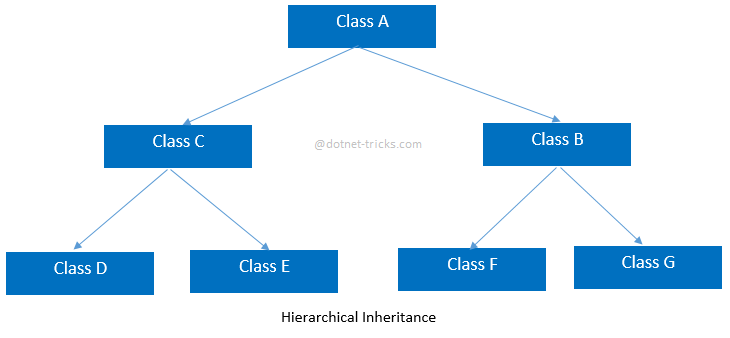
### **Syntax for Multipath Inheritance**

* 1. *//Base Class*
  2. class A
  3. {
  4. public void fooA()
  5. {
  6. *//TO DO:*
  7. }
  8. }
  10. *//Derived Class*
  11. class B : A
  12. {
  13. public void fooB()
  14. {
  15. *//TO DO:*
  16. }
  17. }
  19. *//Derived Class*
  20. class C : A
  21. {
  22. public void fooC()
  23. {
  24. *//TO DO:*
  25. }
  26. }
  28. *//Derived Class*
  29. class D : B, A, C
  30. {
  31. public void fooD()
  32. {
  33. *//TO DO:*
  34. }
  35. }

### **Hierarchical Inheritance**

In this inheritance, more than one derived classes are created from a single base class and futher child classes act as parent classes for more than one child classes.

In the given example, class A has two childs class B and class D. Further, class B and class C both are having two childs - class D and E; class F and G respectively.



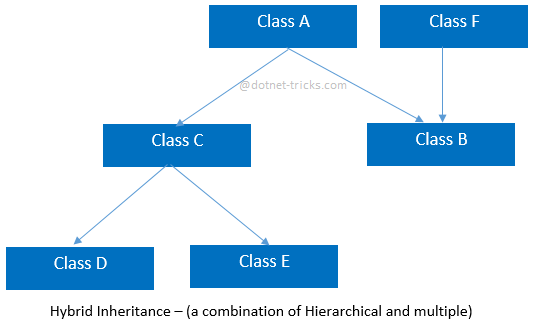
### **Syntax for Hierarchical Inheritance**

* 1. *//Base Class*
  2. class A
  3. {
  4. public void fooA()
  5. {
  6. *//TO DO:*
  7. }
  8. }
  10. *//Derived Class*
  11. class B : A
  12. {
  13. public void fooB()
  14. {
  15. *//TO DO:*
  16. }
  17. }
  19. *//Derived Class*
  20. class C : A
  21. {
  22. public void fooC()
  23. {
  24. *//TO DO:*
  25. }
  26. }
  28. *//Derived Class*
  29. class D : C
  30. {
  31. public void fooD()
  32. {
  33. *//TO DO:*
  34. }
  35. }
  37. *//Derived Class*
  38. class E : C
  39. {
  40. public void fooE()
  41. {
  42. *//TO DO:*
  43. }
  44. }
  46. *//Derived Class*
  47. class F : B
  48. {
  49. public void fooF()
  50. {
  51. *//TO DO:*
  52. }
  53. }
  55. *//Derived Class*
  56. class G :B
  57. {
  58. public void fooG()
  59. {
  60. *//TO DO:*
  61. }
  62. }

### **Hybrid inheritance**

This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.

Since .NET Languages like[C#](https://www.dotnettricks.com/learn/csharp), F# etc. does not support multiple and multipath inheritance. Hence hybrid inheritance with a combination of multiple or multipath inheritances is not supported by .NET Languages.



### **Syntax for Hybrid Inheritance**

* 1. *//Base Class*
  2. class A
  3. {
  4. public void fooA()
  5. {
  6. *//TO DO:*
  7. }
  8. }
  10. *//Base Class*
  11. class F
  12. {
  13. public void fooF()
  14. {
  15. *//TO DO:*
  16. }
  17. }
  19. *//Derived Class*
  20. class B : A, F
  21. {
  22. public void fooB()
  23. {
  24. *//TO DO:*
  25. }
  26. }
  28. *//Derived Class*
  29. class C : A
  30. {
  31. public void fooC()
  32. {
  33. *//TO DO:*
  34. }
  35. }
  37. *//Derived Class*
  38. class D : C
  39. {
  40. public void fooD()
  41. {
  42. *//TO DO:*
  43. }
  44. }
  46. *//Derived Class*
  47. class E : C
  48. {
  49. public void fooE()
  50. {
  51. *//TO DO:*
  52. }
  53. }

### **Advantages of Inheritance**

1. Reduce code redundancy.
2. Provides code reusability.
3. Reduces source code size and improves code readability.
4. The code is easy to manage and divided into parent and child classes.
5. Supports code extensibility by overriding the base class functionality within child classes.

### **Disadvantages of Inheritance**

1. In Inheritance base class and child class, both are tightly coupled. Hence If you change the code of parent class, it will affect all the child classes.
2. In a class hierarchy, many data members remain unused and the memory allocated to them is not utilized. Hence it affects the performance of your program if you have not implemented inheritance correctly.

Private:

The private members of a base class can only be accessed by members of that base class .

Public:

The public members of a base class can be accessed by members of that base class, members of its derived class as well as the members which are outside the base class and derived class.

Protected:

The protected members of a base class can be accessed by members of base class as well as members of its derived class.

In short:

**private**: base

**protected**: base + derived

**public**: base + derived + any other member

**1) Public Inheritance**:

a. Private members of Base class are not accessible in Derived class.

b. Protected members of Base class remain protected in Derived class.

c. Public members of Base class remain public in Derived class.

So, other classes can use public members of Base class through Derived class object.

**2) Protected Inheritance**:

a. Private members of Base class are not accessible in Derived class.

b. Protected members of Base class remain protected in Derived class.

c. Public members of Base class too become protected members of Derived class.

So, other classes can't use public members of Base class through Derived class object; but they are available to subclass of Derived.

**3) Private Inheritance**:

a. Private members of Base class are not accessible in Derived class.

b. Protected & public members of Base class become private members of Derived class.

So, no members of Base class can be accessed by other classes through Derived class object as they are private in Derived class. So, even subclass of Derived class can't access them.

class A

{

public:

int x;

protected:

int y;

private:

int z;

};

class B : public A

{

// x is public

// y is protected

// z is not accessible from B

};

class C : protected A

{

// x is protected

// y is protected

// z is not accessible from C

};

class D : private A // 'private' is default for classes

{

// x is private

// y is private

// z is not accessible from D

};

**Encapsulation** is one of the fundamental concepts in**object-oriented** programming (**OOP**). It describes the idea of bundling data and methods that work on that data within one unit, e.g., a class in Java.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

**Abstraction** means hide the internal information or hide the unnecessary information in oop object of any class provide abstraction all function can be access from one object how this function do work we do not need to know we only need to know how object wiil the function and which parameters will be used.

**Cohesion and coupling:**

Cohesion must high and coupling must low for good design.

Coupling is interrelation between modules or classes and cohesion is interrelation between components of module or methods.

<https://stackoverflow.com/questions/3085285/difference-between-cohesion-and-coupling?rq=1>

**Early binding and late binding**

Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

By default early binding happens in C++.

Late binding (discussed below) is achieved with the help of [virtual keyword](https://www.geeksforgeeks.org/virtual-function-cpp/))

<https://www.codesdope.com/cpp-virtual-and-abstract/>

**Diamond problem:**

<https://www.cprogramming.com/tutorial/virtual_inheritance.html>

**Order of constructor call and destructive call in inheritance:**

<https://www.geeksforgeeks.org/order-constructor-destructor-call-c/>

<https://www.studytonight.com/cpp/order-of-constructor-call.php>

**Compile time and runtime:**

* Programmer writes a code into high level language that is known as source code. At that time when compiler convert the source into low level language or into executable form that is known as compile time. These errors can be occurred into compile time
* Syntax errors
* Typechecking errors
* Compiler crashes (Rarely)

Runtime time is that time when executable program is run into machine.

* Logical error
* Division by zero
* Dereferencing a null pointer
* Running out of memory

<http://net-informations.com/python/iq/checking.htm>

<https://pc.net/helpcenter/answers/compile_time_vs_runtime>

**Association & Aggregations & Composition:**

<https://stackoverflow.com/questions/885937/what-is-the-difference-between-association-aggregation-and-composition>

<https://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html>

#include <iostream>

using namespace std;

class Animals

{

public:

void sound1()

{

cout << "This is parent class" << endl;

}

};

class Dogs

{

Animals n\*;

public:

void sound()

{

n->sound1();

}

};

int main()

{

Dogs d;

d.sound(); // early binding

return 0;

}

**Up casting & Down Casting:**

**Reference and Pointer:**

Pointer contain or store the address the of another variable.

Pointer has its own memory.

Reference contains address of a variable. It can be called as alternative name of a variable. .Reference variable is another or second name of original variable.it does not has its own memory its allocate on the memory of original variable.

Reference variable can refers only one variable.

When we declare the reference variable then we must have to initialize the with existing variable. i.e **int &y = x;**

int main()

{

int x=9;

int \*p;

p=&x;

int &y = x;

cout << x<<endl << " Reference gives same value as original variable" << y << endl<<"address of variable"<< p << endl<<"Address of pointer:"<<&p<< endl<<"value who pointer point: "<<\*p<< endl<<"Address of refererance"<<&y;

return 0;

}

<https://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in>

<https://www.thecrazyprogrammer.com/2016/12/difference-between-reference-and-pointer.html>