**inheritance,**

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

**JAVA:** (extends)

<http://www.tutorialspoint.com/java/java_inheritance.htm>

class Calculation{

int z;

public void addition(int x, int y){

z = x+y;

System.out.println("The sum of the given numbers:"+z);

}

public void Substraction(int x,int y){

z = x-y;

System.out.println("The difference between the given numbers:"+z);

}

}

public class My\_Calculation extends Calculation{

public void multiplication(int x, int y){

z = x\*y;

System.out.println("The product of the given numbers:"+z);

}

public static void main(String args[]){

int a = 20, b = 10;

My\_Calculation demo = new My\_Calculation();

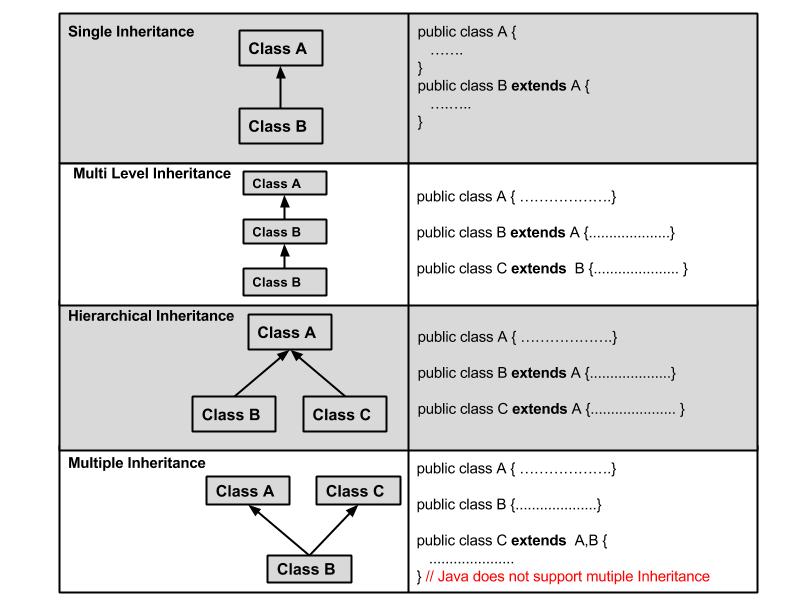
demo.addition(a, b);

demo.Substraction(a, b);

demo.multiplication(a, b);

}

}



A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal −

public class extends Animal, Mammal{}

Howevxer, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.

**C++** (class derived-class: access-specifier base-class)

<http://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm>

#include <iostream>

using namespace std;

// Base class

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

A derived class inherits all base class methods with the following exceptions:

* Constructors, destructors and copy constructors of the base class.
* Overloaded operators of the base class.
* The friend functions of the base class.

## Type of Inheritance:

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

* **Public Inheritance:** When deriving a class from a **public** base class,**public** members of the base class become **public** members of the derived class and **protected** members of the base class become**protected** members of the derived class. A base class's **private**members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
* **Protected Inheritance:** When deriving from a **protected** base class,**public** and **protected**members of the base class become **protected**members of the derived class.
* **Private Inheritance:** When deriving from a **private** base class,**public** and **protected** members of the base class become **private**members of the derived class.

## Multiple Inheritances:

A C++ class can inherit members from more than one class and here is the extended syntax:

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

#include <iostream>

using namespace std;

// Base class Shape

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Base class PaintCost

class PaintCost

{

public:

int getCost(int area)

{

return area \* 70;

}

};

// Derived class

class Rectangle: public Shape, public PaintCost

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

// Print the total cost of painting

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Total area: 35

Total paint cost: $2450

**C#**

<http://www.tutorialspoint.com/csharp/csharp_inheritance.htm>

Syntax:

<acess-specifier> class <base\_class>

{

...

}

class <derived\_class> : <base\_class>

{

...

}

using System;

namespace InheritanceApplication

{

class Shape

{

public void setWidth(int w)

{

width = w;

}

public void setHeight(int h)

{

height = h;

}

protected int width;

protected int height;

}

// Derived class

class Rectangle: Shape

{

public int getArea()

{

return (width \* height);

}

}

class RectangleTester

{

static void Main(string[] args)

{

Rectangle Rect = new Rectangle();

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

Console.WriteLine("Total area: {0}", Rect.getArea());

Console.ReadKey();

}

}

}

## Multiple Inheritance in C#

**C# does not support multiple inheritance**. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this:

using System;

namespace InheritanceApplication

{

class Shape

{

public void setWidth(int w)

{

width = w;

}

public void setHeight(int h)

{

height = h;

}

protected int width;

protected int height;

}

// Base class PaintCost

public interface PaintCost

{

int getCost(int area);

}

// Derived class

class Rectangle : Shape, PaintCost

{

public int getArea()

{

return (width \* height);

}

public int getCost(int area)

{

return area \* 70;

}

}

class RectangleTester

{

static void Main(string[] args)

{

Rectangle Rect = new Rectangle();

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

Console.WriteLine("Total area: {0}", Rect.getArea());

Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));

Console.ReadKey();

}

}

}

**Polymorphism,**

<http://www.tutorialspoint.com/cplusplus/cpp_polymorphism.htm>

<http://www.tutorialspoint.com/java/java_polymorphism.htm>

<http://www.tutorialspoint.com/csharp/csharp_polymorphism.htm>

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Virtual Function

Pure Virtual Function

**Abstraction,**

http://www.tutorialspoint.com/cplusplus/cpp\_data\_abstraction.htm

**Encapuslation,**

http://www.tutorialspoint.com/cplusplus/cpp\_data\_encapsulation.htm

**Information Hiding**

**composition vs aggregation**

Compositions:

* Typically use normal member variables
* Can use pointer values if the composition class automatically handles allocation/deallocation
* Responsible for creation/destruction of subclasses

Aggregations:

* Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
* Can use reference values that point to an object that lives outside the scope of the aggregate class
* Not responsible for creating/destroying subclasses

**static memory vs dynamic memory**

**function overloading vs function overriding**

**Deep Copy vs Shallow Copy**

**Mutable vs Immutable**

**Threading, Multithreading**

**Design patterns, Singleton**

**Three Tier Architecture**

**MVC Architecture**

Instance vs Class vs Object

A blueprint for a house design is like a class description. All the houses built from that blueprint are objects of that class. A given house is an instance.

https://alfredjava.wordpress.com/2008/07/08/class-vs-object-vs-instance/