



DECODED

CODE

FULL STACK DEVELOPER BOOTCAMP

Desarrollamos
{ talento }

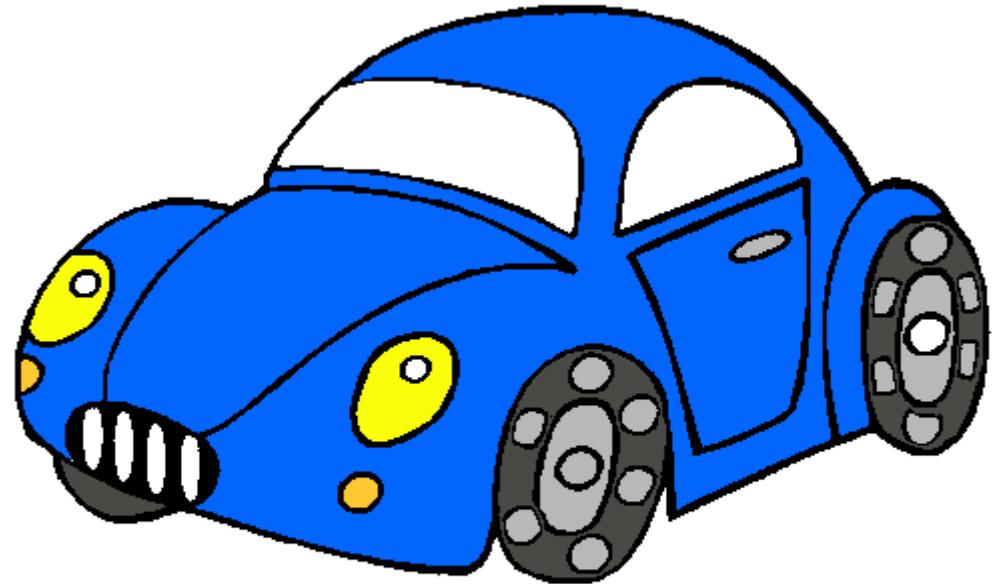
CSharp

Programación orientada a objetos.

Introducción



La programación orientada a objetos es paradigma de programación que cambia la forma en la que entendemos los lenguajes clásicos que utilizaban el paradigma de la programación funcional. Intentar pensar en términos de objetos es muy similar a como lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche, el coche sería un objeto que tendría una serie de características como pueden ser el color, el número de ruedas, el potencia, el modelo o el color y además dispone de una serie de funcionalidades como la puesta en marcha, parar, acelerar, etc.



Clases

Las clases son declaraciones de objetos, o la forma en que abstreamos los objetos, en definitiva la clase es la plantilla por la cual creamos los objetos. Cuando programamos un objeto y definimos sus características y funcionalidades en realidad lo que estamos haciendo es programar un clase.

```
[definicion] class [Nombre de la clase] {  
    //Declaraciones de los métodos y los atributos.  
}
```

Clases

Atributos.

Atributos del objeto.

Los atributos o propiedades son las características de los objetos, por ejemplo en el caso del coche, el color, la velocidad punta, etc. Cuando definimos un atributo normalmente especificamos su nombre y su tipo, en definitiva son variables donde almacenamos datos relacionados con los objetos.

```
//[Ámbito visibilidad] [Tipo Variable] [Nombre variable];  
private int Edad;
```

También disponemos de constantes utilizando la palabra reservada **final**

```
//[Ámbito visibilidad] final [Tipo Variable] [Nombre variable];  
private final int Edad;
```

Clases

Atributos.

Atributos de la clase.

También hay que tener en cuenta que podemos disponer de atributos de clase que no necesariamente se debe instanciar en un objeto, para ello utilizamos la palabra reservada **static**

```
//[Ámbito de visibilidad] static [Tipo Variable] [Nombre variable];  
private static int Edad = 28;
```

También disponemos de constantes utilizando la palabra reservada **final**

```
//[Ámbito visibilidad] final [Tipo Variable] [Nombre variable];  
private static final int Edad = 28;
```

Clases



Métodos.

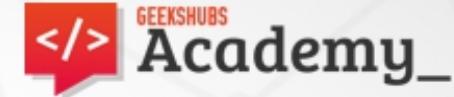
Métodos de objetos.

Son las funcionalidades asociadas a los objetos. Cuando estamos programando las clases los llamamo métodos, la forma de definirlos y las características ya las vimos en el módulo anterior.

```
[Ámbito de visibilidad] [TipoDato/void] [nombre del método]
([parámetros){
    declaración de variables.
    sentencias 1;
    sentencias_n;
}

//Ejemplo
public class Calculadora{
    public int suma(int a, int b)
    {
        return a+ b;
    }
}
```

Clases



Métodos.

Métodos de clase.

Al igual que los atributos los métodos de clase son aquellas que se definen pero no necesita instanciar la clase en un objeto para acceder a ella, utilizando la palabra reservada **static**

```
[Ámbito de visibilidad] static [TipoDato/void] [nombre del método]
([parámetros){
    declaración de variables.
    sentencias 1;
    sentencias_n;
}

//Ejemplo
public class Calculadora{
    public static int suma(int a, int b)
    {
        return a+ b;
    }
}
```

Clases

Instancias.

La instancia de una clase es el objeto que se crea en memoria, en c# todos los objetos utilizados en un programa han de pertenecer a una clase predefinida en C# o declarada por el usuario y referenciada por una variable que almacena su dirección de memoria. En general el acceso a los atributos se realiza mediante el operador .

```
cliente.edad = 25;
```

En el caso de los métodos también se utiliza dicho operador .

```
cliente.setNombre("Javi");
```

Clases

Instancias.

Ejemplo:

```
public class Usuario{  
    private String usuario;  
  
    public void setUsuario(String usuario)  
    {  
        this.usuario = usuario;  
    }  
    public string getUsuario()  
    {  
        return this.usuario;  
    }  
}
```

Clases

Instancias.

Instanciamos la clase Usuario en un nuevo objeto.

```
public class Prueba{  
    public static void main (String []args){  
        Usuario u;  
        u = new Usuario();  
        Usuario u2 = new Usuario();  
        u.setUsuario("Pepe");  
        Console.WriteLine(u2.getUsuario());//Que devuelve?  
    }  
}
```

Clases

Instancias.

En el ejemplo anterior vemos 2 formas de instanciar los objetos, en los dos casos tanto como u y u2 tienen como misión almacenar la dirección de memoria a lo componentes antes de ser instanciados o referenciados. En ese momento la referencia o puntero que se ha creado almacena una dirección de memoria nula ***null***. Una vez creamos la ***instancia*** mediante la palabra reservada ***new*** en el caso anterior nos creamos dos nuevas instancias de la clase usuario es decir dos nuevos objetos y por tanto disponen de los métodos y atributos descritos en la clase.

[Referencia a la clase][nombreobjeto]= new [Referencia a la Clase]();

Clases

Instancias.

¿Pero que ocurre si en el caso anterior asignamos a u2 a u?

```
u2 = u;
```

Lo que puede ocurrir es que u2 se quede sin referencia, pero no nos debe preocupar ya que C# que elimina automáticamente las instancias cuando detecta que no se van a usar más, mediante la recolección de basuras. También hay que tener en cuenta que podemos instanciar tantos objetos como el sistema donde estemos ejecutando el programa nos permita asignar.

Constructores

Un constructor es un elemento de una clase cuyo identificador coincide con el de la clase correspondiente y que tiene por objetivo obligar a controlar como se inicializa una instancia de una determinada clase. Como características podriamos citar.

- Se llama igual que la clase.
- No devuelve nada, no hace falta referenciar **void**
- Pueden existir varios constructores utilizando la sobrecarga de métodos, tal y como vimos en el apartado anterior.
- De todos los que se han creado en un método solo se ejecutará uno en función de los parámetros referenciados.

Constructores

Todas las clases disponen de un constructor por defecto, si queremos modificarlo debemos asignar el mismo identificados que la clase por ejemplo:

```
public class Vehiculo {  
    private String color;  
    private int ruedas;  
  
    public Vehiculo()  
    {  
        this.color= "rojo";  
        this.ruedas= 4;  
    }  
    public Vehiculo(String color, int ruedas)  
    {  
        this.color=color;  
        this.ruedas = ruedas;  
    }  
    public String toString(){  
        return "Vehiculo de color: "+ this.color + " número de  
ruedas: " + this.ruedas;
```



Constructores

Ahora instanciamos:

```
public class Prueba{  
    public static void main (String []args){  
        Vehiculo v1 = new Vehiculo();  
        Console.WriteLine(v1.toString());//Que devuelve?  
        Vehiculo v2 = new Vehiculo("Azúl",5);  
        Console.WriteLine(v2.toString());//Que devuelve?  
    }  
}
```

Tal como podemos ver en el momento de instanciar los objetos establecemos con los parámetros que constructor utilizar.

Argumento Variables.

Los **params** los cual nos permit una lista de argumentos de longitud variable. Se puede utilizar en la definición de un método, y se utiliza cuando el mismo puede tomar una cantidad variable de argumentos, y no es fija.

```
[Visibilidad] [void/tipo] [nombre del método] (params[tipo]... v);
```

Params Variables.

Ejemplo

```
class VarArgs{  
    static void vaTest(params string[] values){  
        Console.WriteLine("Número de args: "+values.length);  
        Console.WriteLine("Contiene: ");  
        for (int i=0; i<values.length;i++)  
            Console.WriteLine(" arg "+i+": "+values[i]);  
    }  
    public static void main(String[] args) {  
        vaTest(10); //1 arg  
        vaTest(1,2,3); //3arg  
        vaTest(); //sin arg  
    }  
}
```

Argumento Variables.

También nos permite la sobrecarga de métodos que disponga ***params**. La única condición es que los parámetros este por delante de la definición de **params**

```
public void miMetodo(int a, int b, double c, params int[] args)
```

Visibilidad o modificador de acceso.

Los modificadores de acceso nos ayudan a restringir el alcance de una clase, constructor, variable, método y miembro de datos, normalmente se consideran tres, pero realmente son 4.

- Default- no hace falta indicarlo.
- Private.
- Protected.
- Public.

Tabla de Modificadores de Acceso en Java

Modificador/Acceso	Clase	Paquete	Subclase	Todos
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
default	Sí	Sí	No	No
private	Sí	No	No	No

Visibilidad o modificador de acceso.

Default

Es el modificador por defecto, y no require definirlo. Los miembros de datos, clases o métodos que utilicen **default** sólo serán accesibles dentro del mismo paquete.

```
class HolaMundo{
    void saludo(){
        Console.WriteLine("Hola Mundo");
    }
}
public class Demo{
    public static void main(String args[])
    {
        HolaMundo holaMundo = new HolaMundo();
        holaMundo.saludo(); // Veremos que nos devuelve un error ya que no
es accesible.
    }
}
```

Visibilidad o modificador de acceso.

Private



Los métodos o miembros de los datos declarados como privados utilizando la palabra reservada ***private*** sólo son accesibles dentro de la clase que lo declaran. Como características:

- Cualquier otra clase del mismo paquete no podrá acceder a estos miembros.
- Las clases e interfaces no se puede declarar como privadas.

```
class HolaMundo{  
    private void saludo(){  
        Console.WriteLine("Hola Mundo");  
    }  
}  
public class Demo{  
    public static void main(String args[]){  
        HolaMundo holaMundo = new HolaMundo();  
        holaMundo.saludo(); // Veremos que nos devuelve un error ya que  
no es accesible.  
    }  
}
```

Visibilidad o modificador de acceso.

Protected

Los miembros o datos declarados como **protected** son accesibles dentro del mismo paquete o subclases que heredan de la clase padre donde se define en diferentes paquetes.

```
class HolaMundo{  
    private void saludar(){  
        Console.WriteLine("Hola Mundo");  
    }  
}  
public class Demo :HolaMundo {  
    public static void main(String args[]){  
        Demo demo = new Demo();  
        demo.saludar(); // Mostrará Hola Mundo  
    }  
}
```

Visibilidad o modificador de acceso.



Public

El modificador de acceso público tiene el alcance más amplio entre todos los demás modificadores de acceso, todas las clases, métodos o miembros de datos que se declaran como públicos, son accesibles desde cualquier lugar del programa. No hay restricciones en el alcance de los miembros de datos públicos.

```
class HolaMundo{  
    public void saludo(){  
        System.out.println("Hola Mundo");  
    }  
}  
public class Demo{  
    public static void main(String args[]){  
        HolaMundo holaMundo = new HolaMundo();  
        holaMundo.saludo(); // Veremos que nos muestra Hola Mundo;  
    }  
}
```

This

Normalmente cuando llamamos a un método dispone de un indicador con la referencia al objeto invocado. Para ello se utiliza la palabra reservada **this**.

This

Utilizando This.



Las ventajas de utilizar this, en la definición o referencia de las variables por ejemplo es que nos permite que un nombre de un parámetro o una variable local sea el mismo que el nombre de una variable de instancia, por ejemplo:

```
class Potencia {  
    double b;  
    int e;  
    double valor;  
    Potencia(double base, int exp){  
        this.b=base;  
        this.e=exp;  
        this.valor=1;  
        if (exp==0) return;  
        for ( ; exp>0; exp--) this.valor = this.valor * base;  
    }  
    double get_potencia(){  
        return this.valor;  
    }  
}
```


This

Utilizando this().

Podemos invocar desde una clase al constructor actual mediante **this()**

```
public class Calculadora{  
    int a,b;  
    Calculadora()  
{  
        this(3,5);  
        Console.WriteLine("Dentro del constructor predeterminado  
\n");  
    }  
    Calculadora(int a, int b)  
{  
        this.a=a;  
        this.b=b;  
        Console.WriteLine("Dentro del constructor parametrizado.  
\n");  
    }  
    public static void main( String[] args)  
{  
        Calculadora cal = new Calculadora();  
    }  
}
```

This

Utilizando this.



También podemos utilizarlo para devolver la instancia de una clase actual.

```
class Test
{
    int a;
    int b;
    Test()
    {
        a = 10;
        b = 20;
    }
    Test get()
    {
        return this;
    }
    void display()
    {
        Console.WriteLine("a = " + a + " b = " + b);
    }
    public static void main(String[] args)
```

```
{  
    Test object = new Test();  
    object.get().display();  
}  
}
```

This

Utilizando this.

Otro uso de **this** es la invocación de métodos en la misma clase.

```
class Hola{  
    void saluda()  
    {  
        this.show;  
    }  
    void show()  
    {  
        Console.WriteLine("Hola");  
    }  
    public static void main (String args[])  
    {  
        Hola hola = new Hola();  
        hola.saluda();  
    }  
}
```

This

Utilizando this.



Pasando this como argumento en la llamada constructor.

```
Class A
{
    B obj;
    A(B obj)
    {
        this.obj = obj;
        obj.display();
    }
}
class B
{
    int x = 5;
    B()
    {
        A obj = new A(this);
    }
    void display()
    {
        Console.WriteLine("Valor de x en Class B: " + x);
    }
}
```

```
public static void main(String[] args) {  
    B obj = new B();  
}
```

Clases anidadas internas e internas locales.

En C# podemos disponer clases internas, clases anidadas o clases separadas en diferentes paquetes.

- Nested Classes / Clases anidadas.
- Inner Classes / Clases internas.
- Local Inner Classes / Clases locales internas.

Clases anidadas internas e internas locales.

Nested Classes

Las clases anidadas te permiten agrupar lógicamente clases que sólo se utilizan en un lugar, por tanto se aumenta el uso de la **encapsulación**. Además dispone de diferentes características.

- Una clase anidada no existe fuera de su clase adjunto. Por tanto el alcance de la misma es su clase externa.
- Una clase anidada también es miembro de su clase externa.
- Como miembro de una clase externa, se puede declarar con cualquier modificador de acceso.
- Una clase anidada tiene acceso a los miembros, incluidos los miembros privados de la clase externa. Sin embargo la clase externa no tiene acceso a los miembros de la clase anidada.

Clases anidadas internas e internas locales.

Nested Classes

- Existen dos tipo de clases anidadas:
- Static nested class, utilizan el modificador **static**
- Inner class aquellos que no utilizan el modificador **static**.

```
class ClaseExterior
{
    ...
    class [static] ClaseAnidada
    {
        ...
    }
}
```

Clases anidadas internas e internas locales.

Static Nested Classes

Al igual que en otros casos que utilizamos la palabra reservada **static** , una clases anidada estática no pude hacer referencia directamente a variables de instancia o modos definidos en su clase, sólo pude usarlo a traves de una referencia al objeto.

Clases anidadas internas e internas locales.

Static Nested Classes

Ejemplo:

```
class ClaseExterna
{
    static int a=10;
    int b =20;
    private static int c =30;
    static class ClaseAnidadaStatic{
        void mostrar(){
            Console.WriteLine("externo a: "+ a );
            Console.WriteLine("externo c: "+ c);
            Console.WriteLine("externo b =" + b);
        }
    }
}
```

Clases anidadas internas e internas locales.

Static Nested Classes

Ejemplo:

```
class ClaseAnidadaStaticDemo {  
    public static void main(String[] args) {  
        ClaseExterna.ClaseAnidadaStatic objetoAnidado= new  
        ClaseExterna.ClaseAnidadaStatic();  
        objetoAnidado.mostrar();  
    }  
}
```

Clases anidadas internas e internas locales.

Inner Classes

Una clase interna tiene acceso a todas las variables y métodos de su clase externa y puede referirse a ellos directamente de la misma manera que lo hacen otros miembros no estáticos de la clase externa.

```
clase ClaseExterior
{
    ...
    clase ClaseInterna
    {
        ...
    }
}
```

Clases anidadas internas e internas locales.



Inner Classes

Ejemplo:

```
class ClaseExterna {
    int nums[];
    ClaseExterna(int n[]){
        nums=n;
    }
    void analizar(){
        Interna in0b=new Interna();
        Console.WriteLine("Promedio: "+in0b.promedio());
    }
    class Interna{
        int promedio(){
            int a=0;
            for (int i=0; i<nums.length;i++)
                a+=nums[i];
            return a/nums.length;
        }
    }
}
```



{}

Clases anidadas internas e internas locales.

Inner Classes

Ejemplo:

```
class ClaseAnidada {  
    public static void main(String[] args) {  
        int x[]={3,2,1,5,6,7,8,9};  
        ClaseExterna extob= new ClaseExterna(x);  
        extob.analizar();  
    }  
}
```

Clases anidadas internas e internas locales.

Inner Classes

Existen ciertas diferencias entre clases estaticas e internas.

- Las clases anidadas estáticas no tienen acceso directo a otro miembros, no pueden hacer referencia directamente a los miembros no estáticos de su clase adjunta.
- Las clases internas, tiene acceso a todos los miembros d su clase externa y pueden referirse a ellos directamente de la misma manera que otros miembros no estáticos de la clase exterior.

Clases anidadas internas e internas locales.

Local Inner Classes

Las clases interna locales tiene acceso a los atributos de la clase que lo incluye pero tiene que ser de tipo final .

Clases anidadas internas e internas locales.

Local Inner Classes

Ejemplo:

```
class MyOuter
{
    private int x = 10;
    public void createLocalInnerClass()
    {
        int y = 20;
        final int z = 30;
        class LocalInner
        {
            public void accessOuter()
            {
                Console.WriteLine(x);
                Console.WriteLine(y); //Error: No puede accder..
                Console.WriteLine(z);
            }
        }
    }
}
```

Clases anidadas internas e internas locales.

Local Inner Classes

Ejemplo:

```
    LocalInner li = new LocalInner();
    li.accessOuter();
}
public class MethodLocalInnerClassDemo
{
    public static void main(String[] args)
    {
        MyOuter mo = new MyOuter();
        mo.createLocalInnerClass();
    }
}
```

Recursividad en CSharp

C# nos permite la recurrencia o recursión en la programación de los métodos, este consiste en que un método se llame así mismo. En la recursividad las nuevas variables y parámetros locales se almacenan en el stack, y el código de método se ejecuta con estas nuevas variables desde el principio. En ningún caso se crean un nuevo objeto.

Recursividad en CSharp

Ejemplo de uso el calculo de un número factorial.

```
public class PruebaFactorialR {
    public static void main (String [] args){
        for (int i=1; i<=20; i++)
            System.out.println("Factorial de " + i + " = " +
factorialR(i));
    }
}

public static long factorialR (int n) {
    if (n==0)
        return 1;
    else
        return n * factorialR(n-1);
}
```

Recursividad en CSharp

Hay que tener en cuenta que demasiadas llamadas recursiva nos podrían causar un **debordamiento de pila o stack overflow**. Se produce cuando en algún caso, la ejecución del método definido de forma recursiva implica la llamada del método de forma infinita.

Herencia en CSharp

Es uno de los pilares fundamentales de la POO, en sí es una propiedad que permite la declaración de nuevas clases a partir de otras ya existentes. En C# una clase que se hereda se denomina **superclase**. La clase que hereda se llama **subclase**, esta clase hereda todos los atributos y métodos existentes en la superclase y nos permite añadir nuevo métodos concepto de **polimorfismo**

Herencia en CSharp

Esto nos da una serie de ventajas respecto a una programación imperativa.

- **Reutilizacion del código** En aquellos casos donde se necesita crear una clase que, además de otros propios , deba incluir los métodos definidos en otra, la herencia evitar tener que reescribir todos esos métodos de la nueva clase.
- **Mantenibilidad** Dado que con herencia las subclases pueden ser modificadas sin que afecte a la superclase la mantenibilidad de las aplicaciones es elevada.

Herencia en C

Para indicar a una clase que hereda de otra se utiliza el símbolo : en la cabecera de la declaración de la subclase. Definición:

```
public class ClaseB : ClaseA {  
    // Declaracion de atributos y metodos especificos de ClaseB  
    // y/o redeclaracion de componentes heredados  
}
```

Herencia en C

Ejemplo:

```
class DosDimensiones{
    private double base;
    private double altura;
    double getBase(){return base;}
    double getAltura(){return altura;}
    void setBase(double b){base=b;}
    void setAltura (double h){altura=h;}
    void mostrarDimension(){
        Console.WriteLine("La base y altura es: "+base+" y "+altura);
    }
}
```

Herencia en C

Ejemplo:

```
class Triangulo :DosDimensiones{
    String estilo;
    double area(){
        return getBase()*getAltura()/2;
    }
    void mostrarEstilo(){
        Console.WriteLine("Triangulo es: "+estilo);
    }
}
```

Herencia en C



Ejemplo:

```
class Lados3{
    public static void main(String[] args) {
        Triangulo t1=new Triangulo();
        Triangulo t2=new Triangulo();
        t1.setBase(4.0);
        t1.setAltura(4.0);
        t1.estilo="Estilo 1";
        t2.setBase(8.0);
        t2.setAltura(12.0);
        t2.estilo="Estilo 2";
        Console.WriteLine("Información para T1: ");
        t1.mostrarEstilo();
        t1.mostrarDimension();
        Console.WriteLine("Su área es: "+t1.area());
        Console.WriteLine("Información para T2: ");
        t2.mostrarEstilo();
        t2.mostrarDimension();
        Console.WriteLine("Su área es: "+t2.area());
    }
}
```

Herencia en C

Tipos de Herencia.

En C# podríamos considerar diversos tipos de herencia:

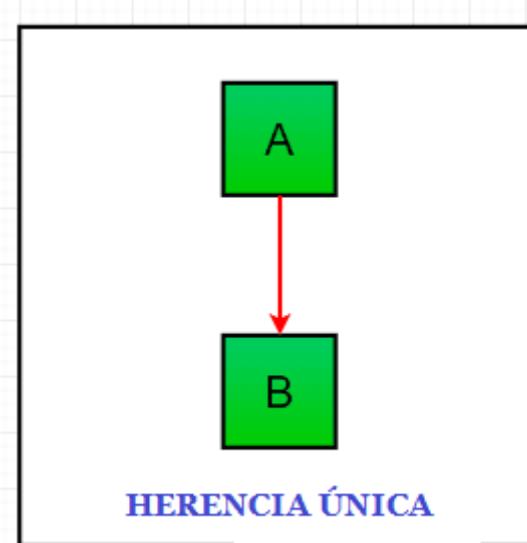
- Herencia única.
- Herencia multinivel.
- Herencia jerárquica.
- Herencia múltiple.
- Herencia híbrida.

Herencia en C

Tipos de Herencia.

Herencia única.

La **herencia única** es aquella que las subclases heredan las características de solo una superclase.

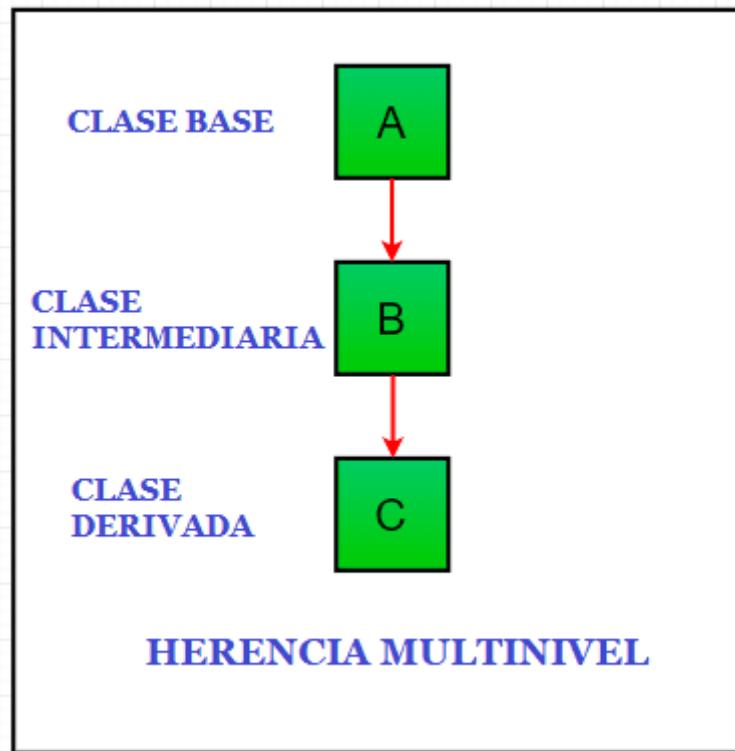


Herencia en C

Tipos de Herencia.

Herencia multinivel.

La **herencia multinivel**, es donde una clase derivada herdará una clase base y además, la clase derivada también auctará como la clase base de otra clase.

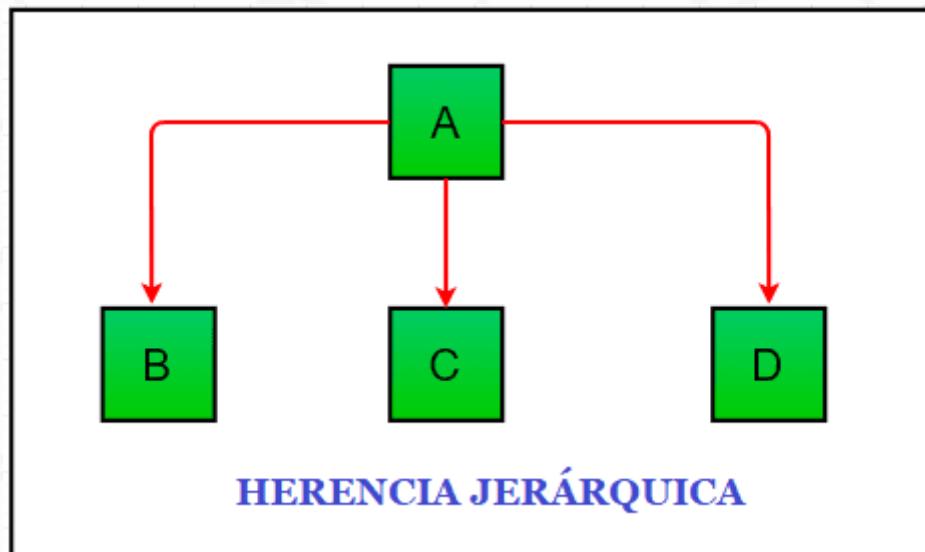


Herencia en C

Tipos de Herencia.

Herencia jerárquica.

La **herencia jerárquica**, una clase sirve como una superclase, para más de una subclase.

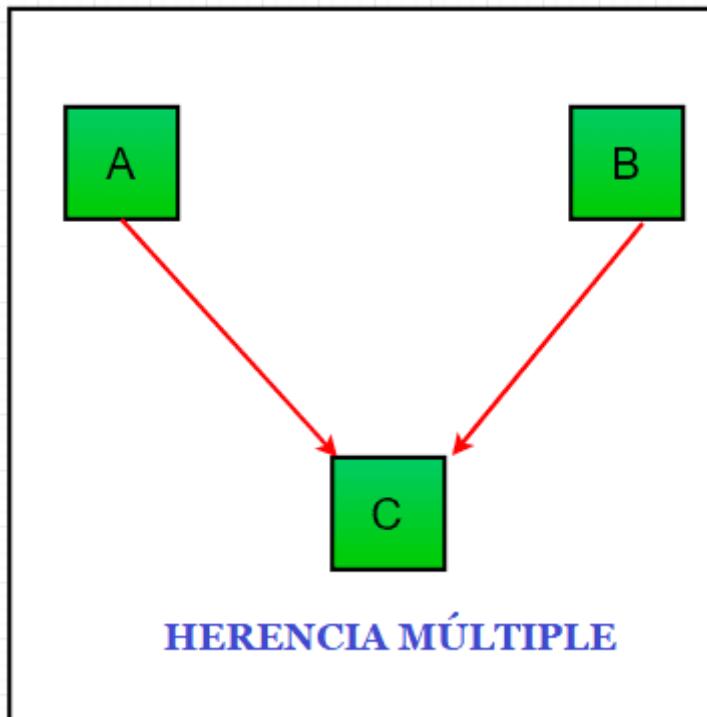


Herencia en C

Tipos de Herencia.

Herencia Múltiple.

La **herencia múltiple**, una clase puede tener más de una superclase y heredar características de todas las clases principales, sólo lo podemos lograr a través del uso de **interfaces**

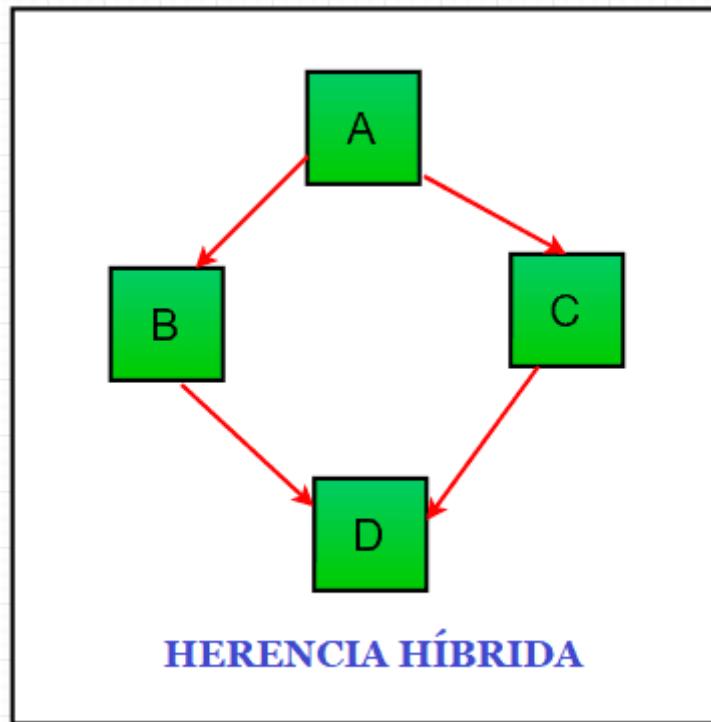


Herencia en C

Tipos de Herencia.

Herencia Híbrida.

La **herencia híbrida**, es una mezcla de dos o más de los tipos de herencia anteriores, sólo puede realizarse mediante el uso de *interfaces*



Herencia en C

Elementos heredados.

Hay que tener en cuenta que las subclases pueden añadir sus propios atributos y métodos pero también sustituir u ocultar los heredados:

- Se puede declarar un nuevo atributo con el mismo identificador que uno heredado quedando ese atributo oculto.
- Se puede declarar un nuevo método de instancia con la misma cabecera que el de la clase ascendiente, lo que supone su sobreescritura.
- Se puede declarar un nuevo método de clase con la misma cabecera que el de la clase ascendiente, lo que hace que éste quede oculto. Por lo tanto los métodos de clase o estáticos **static** no pueden ser redefinidos.
- Un método declarado con el modificador **final** tampoco puede ser redefinido por una clase derivada.

Herencia en C

Elementos heredados.

- Se puede declarar un **constructor** de la subclase que llame al de la superclase de forma implícita o de mediante la palabra reservada **base**
- En general puede accederse a los métodos de la clase ascendiente que han sido redefinidos empleando la palabra reservada **base** delante del identificador del método. Este mecanismo sólo permite acceder al metodo perteneciente a la clase en el nivel inmediatamente superior de la jerarquía de clases.

Base

La palabra reservada **base** es una variable de referencia que se usa para referir a los objetos de la clase padre.

Base

Uso de base con variables

Se usa cuando una subclase y un superclase tiene los mismos miembros de datos.

Base

Uso de base con variables



Ejemplo:

```
class vehicle
{
    int maxSpeed = 120;
}
class Car: Vehicle
{
    int maxSpeed =180;
    void display()
    {
        Console.WriteLine("Velocidad máxima : " + base.maxSpedd);
    }
}
class Text
{
    public static void main (String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```


Base

Uso de super con métodos.

Se usa cuando queremos llamar al método de clase padre. Entonces, cuando una clase padre e hijo tienen los mismos métodos nombrados, entonces para resolver la ambigüedad utilizamos la palabra clave **base**

Uso de base con métodos.

Ejemplo:

```
class Person{
    void message()
    {
        Console.WriteLine("Esta es una clase persona");
    }
}
class Student : Person{
    void message()
    {
        Console.WriteLine("Esta es una clase estudiante");
    }
    void display()
    {
        message();
        super.message();
    }
}
```

Base

Uso de base con métodos.

Ejemplo:

```
class Text
{
    public static void main (String[] args)
    {
        Student s = new Student();
        s.display();
    }
}
```

Base

Uso de base con constructores.

También se usa para poder acceder al constructor de la clase padre. Una cosa más importante es que "base" puede llamar constructores tanto con parámetros como sin parametros dependiendo de la situación.

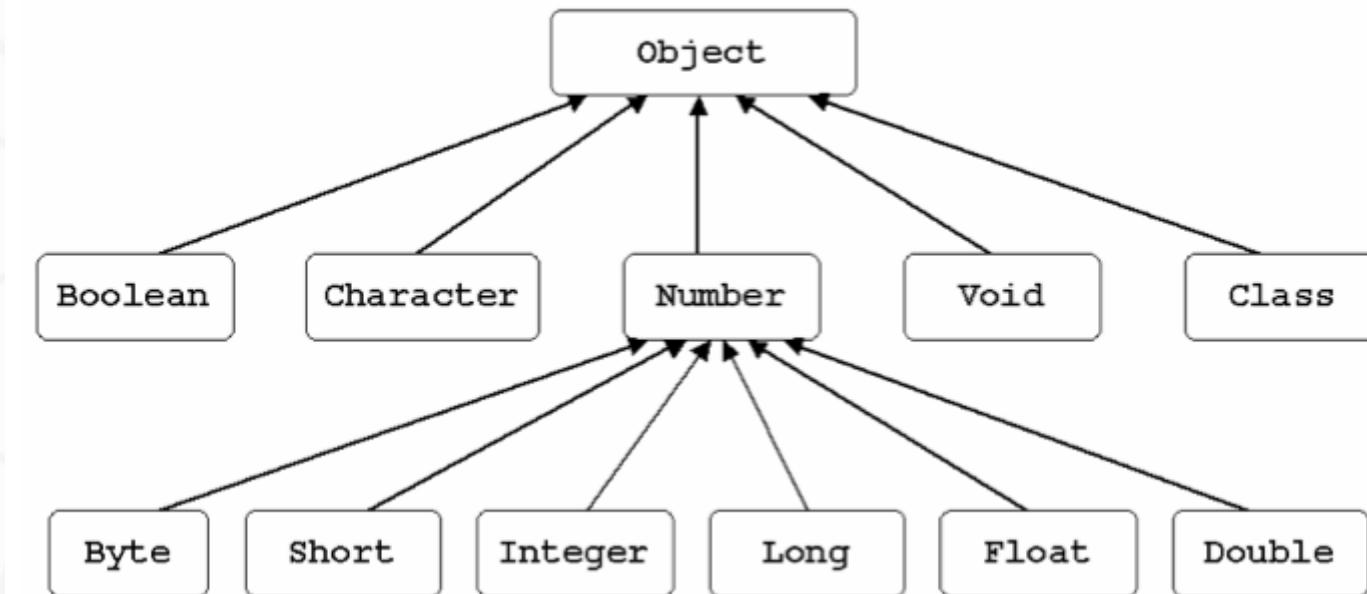
Uso de base con constructores.

Ejemplo:

```
class Person {
    Person(){
        Console.WriteLine("Constructor de la clase Persona");
    }
}
class Student : Person {
    Student(){
        base();
        Console.WriteLine("Constructor de la clase Student");
    }
}
class Test{
    public static void main(String[] args){
        Student s = new Student();
    }
}
```

La clase Object

La clase **Object** es la superclase que derivan todas las clases pero en este caso particular no se considera herencia múltiple.



La clase Object



Como consecuencia de ello todas las clases tiene algunos métodos heradados de la clase **Object**

Tabla de Métodos de la Clase Object en Java

Método	Sintaxis	Propósito
getClass	class _ getClass()	Obtiene la clase de un objeto en tiempo de ejecución
hashCode	int hashCode	Devuelve el código hash asociado con el objeto invocado
equals	boolean equals(Object obj)	Determina si un objeto es igual a otro
clone	Object clone()	Crea un nuevo objeto que es el mismo que el objeto que se está clonando
toString	String toString()	Devuelve una cadena que describe el objeto
notify	void notify()	Reanuda la ejecución de un hilo esperando en el objeto invocado
notifyAll	void notifyAll()	Reanuda la ejecución de todo el hilo esperando en el objeto invocado
wait	void wait(long timeout)	Espera en otro hilo de ejecución
wait	void wait(long timeout,int nanos)	Espera en otro hilo de ejecución
wait()	void wait()	Espera en otro hilo de ejecución
finalize	void finalize()	Determina si un objeto es reciclado (obsoleto por JDK9)

La clase Object

toString()

Proporciona la representación String de un Objeto y se usa para convertir un objeto a Cadena (String) El método predeterminado **toString()** para la clase Object devuele una cadena que consiste en el nombre de la clase de la cual el objeto es una instancia. El carácter arroba '@' y la representación hexadecimal sin signo del código hash del objeto.

La clase Object

toString()

```
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

La clase Object

GethashCode()

Por cada objeto, se genera un número único que es hashcode, éste método devuelve un valor hash que se utiliza para buscar objetos en una colección. c# utiliza el método GethashCode al guardar objetos en estructuras de datos relacionadas con el hashing como HashSet, HashMap, Hashtable, etc.

La clase Object

GetHashCode()

```
class ObjectA{  
    static int a=100;  
    int b;  
    ObjectA(){  
        b=a;  
        a++;  
    }  
    @override  
    public int GetHashCode(){  
        return b;  
    }  
    public static void main(String args[]){  
        ObjectA objectA = new ObjectA();  
        System.out.println(objectA);  
        System.out.println(objectA.toString());  
    }  
}
```

La clase Object

referenceEquals (Object obj)

Compara el objeto dado con el objeto "this". Da una forma genérica de comparar objetos para la igualdad.

La clase Object

referenceEquals()

Devuelve la clase de objeto y se utiliza para obtener la clase en tiempo de ejecución real de objeto. También se puede usar para obtener metadatos de esta clase. El objeto Class devuelto es el objeto que está bloqueado por métodos estáticos sincronizados de la clase representada. Como es final, no lo anulamos.

La clase Object

finalize()

Este método se llama justo antes que un objeto sea recolectado por el recolector de basuras de C#. Es invocado por el propio recolector en un objeto cuando el mismo determina que no hay más referencias al objeto. Debemos anular el método **finalize()** para eliminar los recursos del sistema.

La clase Object

MemberwiseClone()

Devuelve un nuevo objeto que es igual que el objeto que se llama el método.

Overriding

Es una característica que permite que una subclase o clase secundaria proporcione una implementación específica de un método que ya está provisto por una de sus superclases. Cuando un método en una subclase tiene el mismo nombre, los mismos parámetros o firma y el mismo tipo de devolución(o subtipo) que un método en superclase, se dice que el método de la subclase anula al método en la superclase.

Overriding

En un método.

Cuando se llama a un método anulado dentro de una subclase, siempre se referirá a la versión de ese método definida por la subclase. Para ello se utiliza la palabra reservada ***Override***

Overriding

En un método.



Ejemplo:

```
class Parent{
    void show(){
        Console.WriteLine("Método Show() del padre");
    }
}
class Child : Parent{

    override void show(){
        Console.WriteLine("Método Show() del hijo");
    }
}
class Main{
    public static void main(String[] args){
        Parent obj1 = new Parent();
        obj1.show();
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Overriding

La anulación de métodos forma base de uno de los conceptos más potentes de c#, ya que nos permite el **polimorfismo**

Overriding

Ejemplo:

```
class Sup{
    void desde(){
        Console.WriteLine("desde() Sup");
    }
}
class Sub1:Sup{
    void desde(){
        Console.WriteLine("desde() Sub1");
    }
}
class Sub2:Sup{
    void desde(){
        Console.WriteLine("desde() Sub2");
    }
}
```

Overriding

Ejemplo:

```
class Polimorfismo {  
    public static void main(String[] args) {  
        Sup superob=new Sup();  
        Sub1 subob1=new Sub1();  
        Sub2 subob2=new Sub2();  
        Sup referencia;  
        referencia=superob;  
        referencia.desde();  
        referencia=subob1;  
        referencia.desde();  
        referencia=subob2;  
        referencia.desde();  
    }  
}
```

Clases Abstractas

Una clase abstracta es una clase de la que no se pueden crear instancias. Su utilidad consiste en permitir que otras clases derive de ella. De esta forma, proporciona un modelo de referencia a seguir a la vez que una serie de meodos de tuilidad general. Se utiliza la palabra reservada **abstract**

```
public abstract class IdClase...
```

Una clase abstracta puede compoenerse de varios arributos y métodos pero debe tener, al menos, un etodo abstracto. Los métodos abstractos no se implementan en el código de la clase abstracta pero las clases descendientes de ésta han de implementarlos o volver a declararlos como abstractos.

Clases Abstractas.

Ejemplo:

```
public abstract class FiguraGeometrica {  
    // Declaracion de atributos  
    private String nombre;  
  
    // Declaracion de metodos  
    abstract public double area();  
    public figuraGeometrica (String nombreFigura ) {  
        this.nombre = nombreFigura;  
    }  
  
    final public boolean mayorQue (FiguraGeometrica otra) {  
        return this.area()>otra.area();  
    }  
  
    final public String toString()  
        return this.nombre + " con area " + this.area();  
}
```

Clases Abstractas.

Ejemplo (continuación):

```
public class Rectangulo : FiguraGeometrica {  
    private double base;  
    private double altura;  
  
    public Rectangulo (double largo, double ancho) {  
        base("Rectangulo");  
        this.base = largo;  
        this.altura = ancho;  
    }  
  
    public double area () {  
        return this.base * this.altura;  
    }  
}
```

Clases Abstractas.

Ejemplo (continuación):

```
public class pruebaRectangulo {  
    public static void main (String [] args ) {  
        Rectangulo r1;  
        r1 = new Rectangulo(12.5, 23.7);  
        Console.WriteLine("Area de r1 = " + r1.area());  
  
        Rectangulo r2 = new Rectangulo(8.6, 33.1);  
        Console.WriteLine("Area de r2 = " + r2.toString());  
  
        if (r1.mayorQue(r2))  
            Console.WriteLine("El rectangulo de mayor area es r1");  
        else  
            Console.WriteLine("El rectangulo de mayor area es r2");  
    }  
}
```

Clases y métodos finales.

Clases finales.

Una clase declarada con la palabra reservada ***sealed*** no puede tener clases descendientes. Por ejemplo, la clase predefinida de C# **Math** está declarada como ***sealed***.

Por tanto podemos utilizarlo también para evitar la herencia.

Clases y métodos finales.



Métodos finales.

Cuando se declara un método de tipo **sealed** , éste método no puede ser anulado.

```
class A
{
    sealed void m1()
    {
        Console.WriteLine("Este es un método final.");
    }
}
class B:A
{
    void m1()
    {
        // COMPILEACIÓN-ERROR!
        Console.WriteLine("Illegal!");
    }
}
```

Interfaces

Las interfaces son similares a una clase pero emplea la palabra reservada ***interface*** en lugar de ***class*** y no incluye ni la declaración de variables de instancia ni la implementación del cuerpo de los métodos.

Interfaces



Implementando nuestra primera interfaz

Para poder implementar nuestra interfaz primero la declaramos:

```
public interface IModificacion {  
    void incremento(int a);  
}  
  
public class Acumulador:IModificacion{  
    private int valor;  
    public Acumulador (int i){  
        this.valor = i;  
    }  
    public int getValor(){  
        return this.valor;  
    }  
    public void incremento( int a){  
        this.valor +=a;  
    }  
}
```

Interfaces

Implementando nuestra primera interfaz

Tal como podemos observar para utilizarlo en nuestra clase debemos utilizar el símbolo :; a la vez podemos observar que los métodos que se declaran en una interfaz deben ser públicos. A la vez las interfaces nos permite crear lo que se denomina ***variables de referencia del interfaz***. Dicha variables puede refererise desde cualquier objeto que implemente la interfaz.

Interfaces

Implementando nuestra primera interfaz

También disponemos de las variables en interfaces, estas son implicitamente públicas, staticas y finales, es decir se convierten en constantes, por ejemplo:

```
interface Constante{  
    int MIN=0;  
    int MAX=10;  
    double PI=3.1416;
```

Interfaces

Jerarquía entre interfaces.

La jerarquía entre interfaces permite la herencia simple y múltiple. Es decir tanto la declaración de una clase, como la de una interfaz pueden incluir la implementación de otras interfaces.

```
public class Prueba :interface1, interface2, ....interfacen{  
    //Cuerpo de la clase.  
}
```

Interfaces

Utilización de la interfaz como un tipo de dato.

Cuando declaramos una interfaz implicitamente estamos creando un nuevo tipo de referencia, por ejemplo:

```
public interface Comparable{  
    public int esMayor(Comparable otro);  
}
```

Interfaces



Las interfaces pueden ser extendidas

Una interfaz puede heredar otra mediante el uso de la palabra clave extendida : su funcionamiento es identico a la herencia de clases.

```
interface A{
    void metodo1();
    void metodo2();
}
interface B :A{
    void metodo3();
}
class Miclase:B{
    public void metodo1(){
        Console.WriteLine("Implementacion de método1");
    }
    public void metodo2(){
        Console.WriteLine("Implementacion de método2");
    }
    public void metodo3(){
        Console.WriteLine("Implementacion de método3");
    }
}
```


NameSpace

Los ***NameSpace** es un conjunto de clases e interfaces relacinadas entre sí. En C# las clase e interfaces que lo conforman se estructuran en varios paquetes organizados por funciones o tareas. Por ejemplo, las clases fundamentales entán en **System.**, las clases para operaciones matemáticas están **Math**

NameSpace

Como definir los NameSpace.

Para crear un package, sólo se dee incluir la palabra reservada **NameSpace** y el nombre que le asignamos en la parte superior del archivo fuente de C#. Las clases declaradas dentro de este archivo pertenecerán al paquete especificado. Como un paquete define un espacio de nobmres, los nombres de las calses que colocas en el archivo forman parte del espacio de nombres de ese paquete.

```
NameSpace usuarios;
```

NameSpace

Acceso a los componentes de un NameSpace

Sólo serán accesibles aquellos componentes que sean públicos, para ello lo podemos realizar de dos formas distintas:

- Referenciarlo mediante su identificador:

```
r1= new geometria.Rectangulo();
```

- O incorporar el componente del paquete mediante la sentencia **Using** al principio de larchivo fuente:

```
Using geometria.Rectangulo;  
///  
Rectangulo r1 = new Rectangulo();
```