

# METODOLOGÍAS DE PROGRAMACIÓN I

---

Repaso del paradigma de la programación orientada a objetos implementado en C#

- Objetos
- Clases
- Jerarquías de clases

# Clases en C#

## Sintaxis de definición de clases

```
public class <nombreClase>
{
    <miembros>
}
```

- Los **miembros** de una clase son las variables y funciones de los que van a disponer todas las instancias de la misma

# Clases en C#

```
public class Persona {  
    string nombre, apellido;  
    int edad;  
  
    public Persona(string n, string a, int e){  
        nombre = n;  
        apellido = a;  
        edad = e;  
    }  
  
    public void saludar(){  
        Console.WriteLine("Buen día !!!");  
    }  
  
    public string getNombreYApellido(){  
        return nombre + " " + apellido;  
    }  
}
```

# Clases en C#

```
public class Persona {  
    string nombre, apellido;  
    int edad;
```

Estado.  
Se definen como  
variables

```
        public Persona(string n, string a, int e){  
            nombre = n;  
            apellido = a;  
            edad = e;  
        }
```

```
        public void saludar(){  
            Console.WriteLine("Buen día !!!");  
        }
```

```
        public string getNombreYApellido(){  
            return nombre + " " + apellido;  
        }
```

```
    }
```

# Clases en C#

```
public class Persona {  
    string nombre, apellido;  
    int edad;  
  
    public Persona(string n, string a, int e){  
        nombre = n;  
        apellido = a;  
        edad = e;  
    }
```

```
        public void saludar(){  
            Console.WriteLine("Buen día !!!");  
        }  
  
        public string getNombreYApellido(){  
            return nombre + " " + apellido;  
        }  
    }
```

Comportamiento.  
Se definen como  
funciones

# Clases en C#

```
public class Persona {  
    string nombre, apellido;  
    int edad;
```

```
    public Persona(string n, string a, int e){  
        nombre = n;  
        apellido = a;  
        edad = e;  
    }
```

```
    public void saludar(){  
        Console.WriteLine("Buen día !!!");  
    }
```

```
    public string getNombreYApellido(){  
        return nombre + " " + apellido;  
    }
```

```
}
```

Constructor.  
Se define como  
una función sin  
tipo de retorno y  
con el mismo  
nombre que la  
clase

# Instanciando objetos

- Para poder usar objetos hay que ***instanciar*** una clase.
- Para crear una instancia se utiliza el operador **new** y se debe especificar la clase del objeto a crear, por ejemplo:  
    per1 = **new** Persona();
- Se pueden crear tantas instancias de una clase como se necesite.

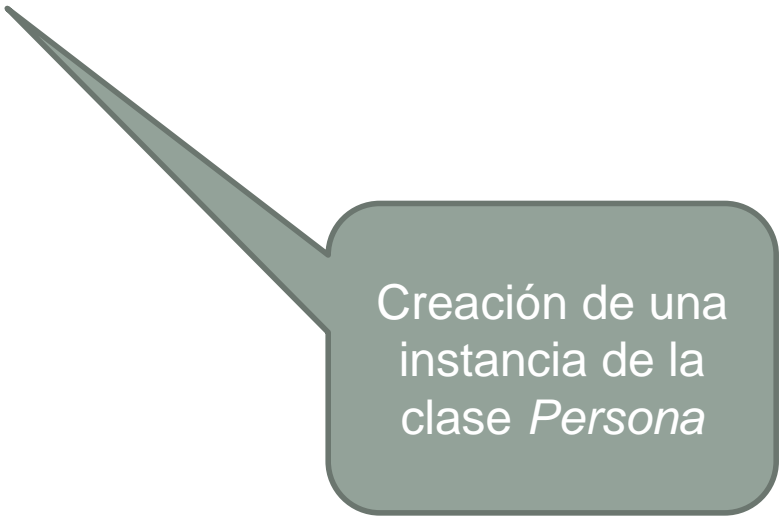
# Clases en C#

```
public class Program {  
    public static void Main(string[] args) {  
        Persona p = new Persona("Ronnie", "Dio", 67);  
        p.saludar();  
    }  
}
```



# Clases en C#

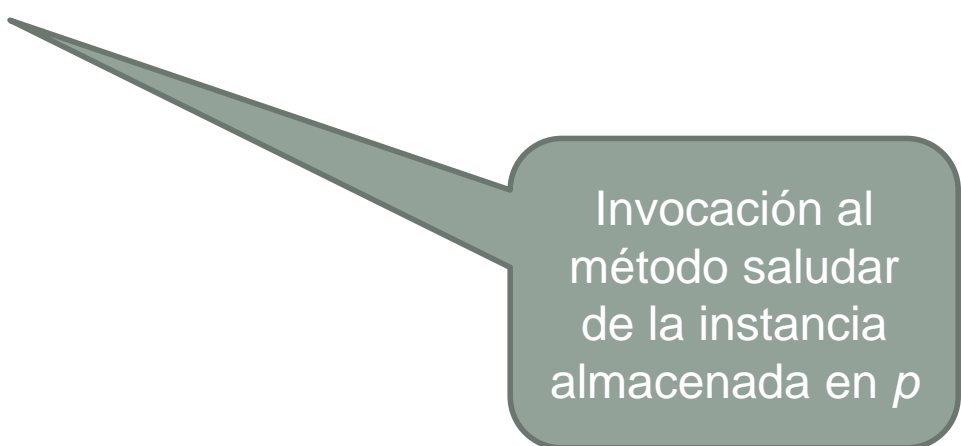
```
public class Program {  
    public static void Main(string[] args) {  
        Persona p = new Persona("Ronnie", "Dio", 67);  
        p.saludar();  
    }  
}
```



Creación de una  
instancia de la  
clase *Persona*

# Clases en C#

```
public class Program {  
    public static void Main(string[] args) {  
        Persona p = new Persona("Ronnie", "Dio", 67);  
        p.saludar();  
    }  
}
```



Invocación al  
método saludar  
de la instancia  
almacenada en *p*

# Sobrecarga de métodos

- Una clase puede tener más de un método con el mismo nombre siempre que sus firmas sean diferentes.
- La firma de un método consiste en:
  - El nombre
  - El número de parámetros
  - El tipo y el orden de los parámetros
  - Los modificadores de los parámetros
- El tipo de retorno no es parte de la firma.
- Los nombres de los parámetros tampoco son parte de la firma.

# Sobrecarga de métodos

Agreguemos dos sobrecargas al método "saludar" de la clase *Persona*.

```
public void saludar(){  
    Console.WriteLine ("Buen día !!!");  
}
```

```
public void saludar(string nombre){  
    Console.WriteLine("Buen día " + nombre+ " !!!");  
}
```

```
public void saludar(Persona p){  
    Console.WriteLine("Buen día " +  
        p.getNombreYApellido() + " !!!");  
}
```

# Clases en C#

```
public class Program {  
  
    public static void Main(string[] args) {  
  
        Persona p1 = new Persona("Ronnie", "Dio", 67);  
        Persona p2 = new Persona("Tony", "Iommi", 69);  
  
        p1.saludar();  
        p1.saludar("Vinny Appice");  
        p1.saludar(p2);  
    }  
  
}
```

# Clases en C#

```
public class Program {
```

```
    public static void Main(string[] args) {
```

```
        Persona p1 = new Persona("Ronnie", "Dio", 67);
```

```
        Persona p2 = new Persona("Tony", "Iommi", 69);
```

```
        p1.saludar();  
        p1.saludar("Vinny Appice");  
        p1.saludar(p2);
```

```
    }
```

```
}
```

Invocación a las  
tres sobrecargas  
del método  
saludar

# Constructor por defecto

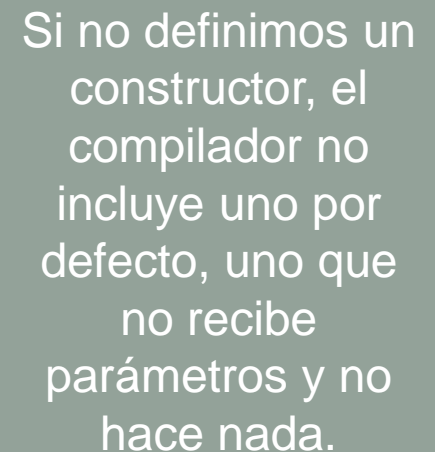
En caso de no definir un constructor para la clase el compilador creará uno por defecto:

```
<nombreClase>()  
{  
}
```

# Constructores

```
public class Persona {  
    public void saludar(){  
        Concole.WriteLine("Buen día !!!");  
    }  
}
```

```
public static void Main(String[] args) {  
    Persona p = new Persona();  
    p.saludar();  
}
```



Si no definimos un constructor, el compilador no incluye uno por defecto, uno que no recibe parámetros y no hace nada.



# Constructores

```
public class Persona {  
    public Persona(string n, string a, int e){ }  
    public void saludar(){  
        Concole.WriteLine("Buen día !!!");  
    }  
}
```

Al definir un constructor no-nulo, el compilador no incluye el compilador por defecto

```
public static void Main(string[] args) {  
    Persona p = new Persona();  
    p.saludar();  
}
```

Esta línea ahora da error

# Constructores

```
public class Persona {  
    public Persona() { }  
    public Persona(string n, string a, int e) {  
    public void saludar() {  
        Console.WriteLine("Buen día !!!");  
    }  
}
```

Si queremos seguir construyendo instancias sin pasar parámetros, tenemos que declarar explícitamente el constructor que no recibe parámetros

```
public static void Main(string[] args) {  
    Persona p = new Persona();  
    p.saludar();  
}
```

Esta línea ahora no da error

# Constructores

- Al igual que los métodos, los constructores permiten sobrecarga.
- Se puede definir más de un constructor, siempre que sus firmas sean diferentes.

# Miembros

- En POO hay dos tipos de miembros (variables y métodos)
  - De instancia
  - De clase
- En C# si queremos declarar miembros de clase usamos la palabra reservada `static`.

```
public static void UnMetodoDeClase() { }
```

- Si queremos que los miembros sean de instancia no ponemos nada

```
public void UnMetodoDeInstancia() { }
```

# Miembros de instancia

- Los miembros de instancia, ya sean variables o métodos son utilizados cuando se trabaja con instancias

```
p1.saludar();
```

```
p1.saludar("Vinny Appice");
```

```
p1.saludar(p2);
```

```
Console.WriteLine(p2.getNombreYApellido());
```

# Miembros de clase

- Los miembros de clase no pertenecen a ninguna instancia, pertenecen a la clase
- Se declaran con la palabra reservada **static**
- Todas las instancias comparten los mismos miembros de clase
- La referencia a un miembro de clase se hace mediante el nombre de clase

`<clase> . <miembro>`

# Modificadores de acceso

- C# nos provee mecanismos para asegurar el encapsulamiento mediante los modificadores de acceso a campos
- Los miembros de las clases pueden declararse como:
  - Públicos
  - Privados
  - Protegidos

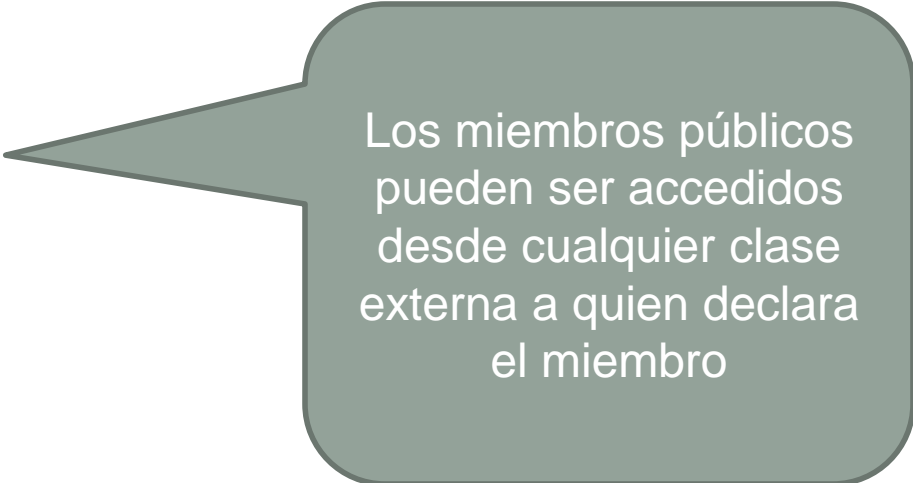
# Modificadores de acceso

- Miembros públicos  
**public** int variable;
- Miembros privados  
**private** int variable;
- Miembros protegidos  
**protected** int variable;



# Modificadores de acceso

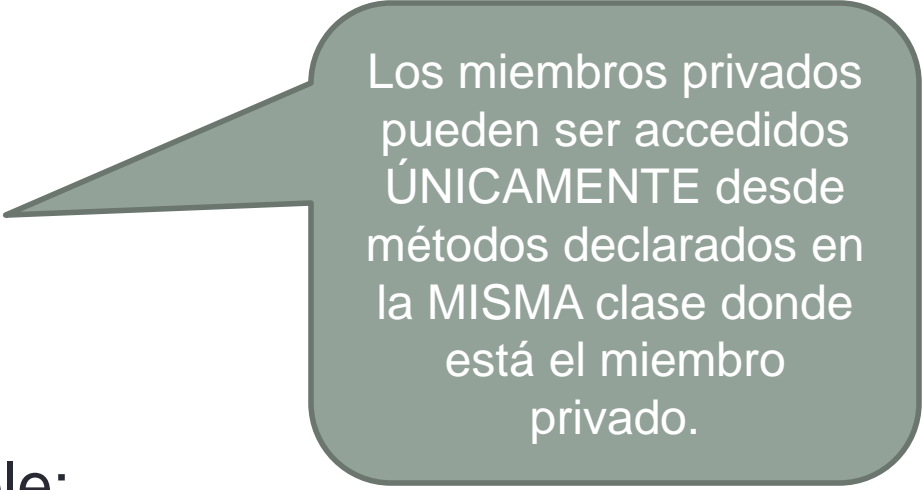
- Miembros públicos  
**public** int variable;
- Miembros privados  
**private** int variable;
- Miembros protegidos  
**protected** int variable;



Los miembros públicos pueden ser accedidos desde cualquier clase externa a quien declara el miembro

# Modificadores de acceso

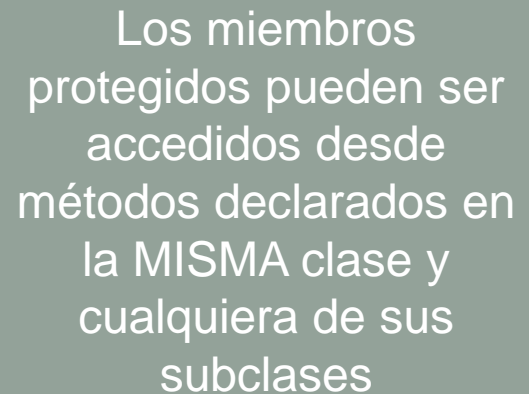
- Miembros públicos  
**public** int variable;
- Miembros privados  
**private** int variable;
- Miembros protegidos  
**protected** int variable;



Los miembros privados pueden ser accedidos ÚNICAMENTE desde métodos declarados en la MISMA clase donde está el miembro privado.

# Modificadores de acceso

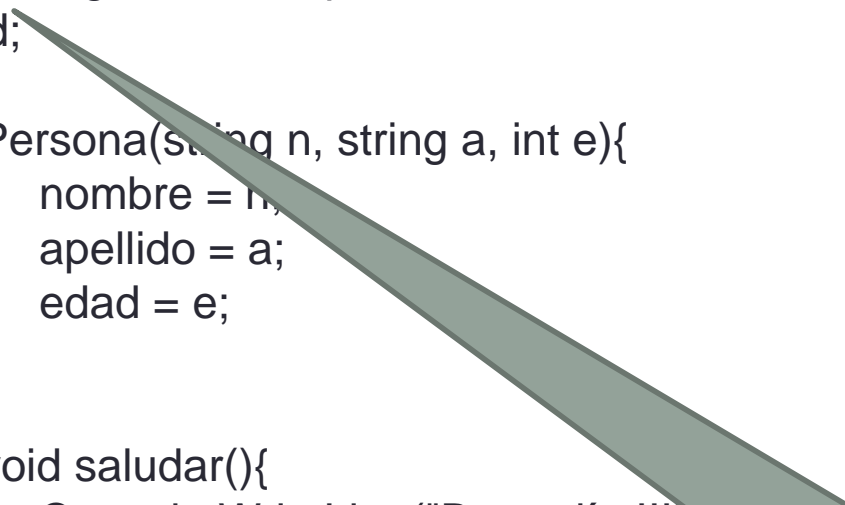
- Miembros públicos  
**public** int variable;
- Miembros privados  
**private** int variable;
- Miembros protegidos  
**protected** int variable;



Los miembros protegidos pueden ser accedidos desde métodos declarados en la MISMA clase y cualquiera de sus subclases

# Modificadores de acceso

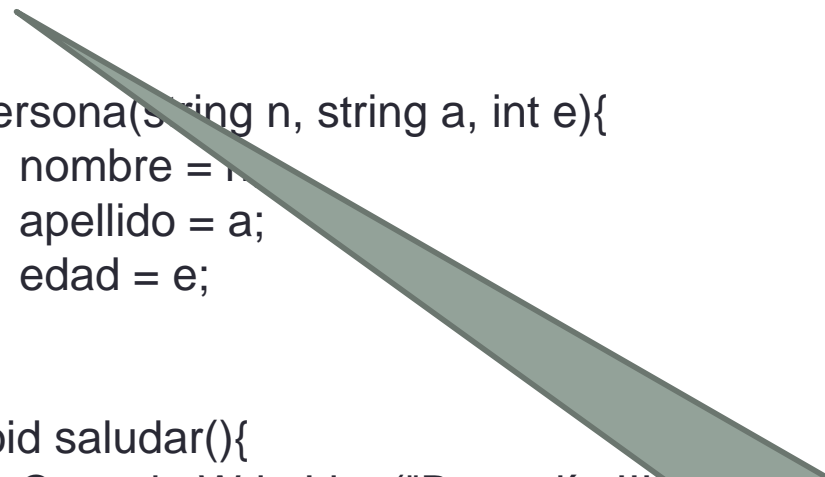
```
public class Persona {  
    private string nombre, apellido;  
    int edad;  
  
    public Persona(string n, string a, int e){  
        nombre = n;  
        apellido = a;  
        edad = e;  
    }  
  
    public void saludar(){  
        Console.WriteLine("Buen día");  
    }  
  
    public string getNombreYApellido(){  
        return nombre + " " + apellido;  
    }  
}
```



El estado siempre lo vamos a declarar como privado o protegido.

# Modificadores de acceso

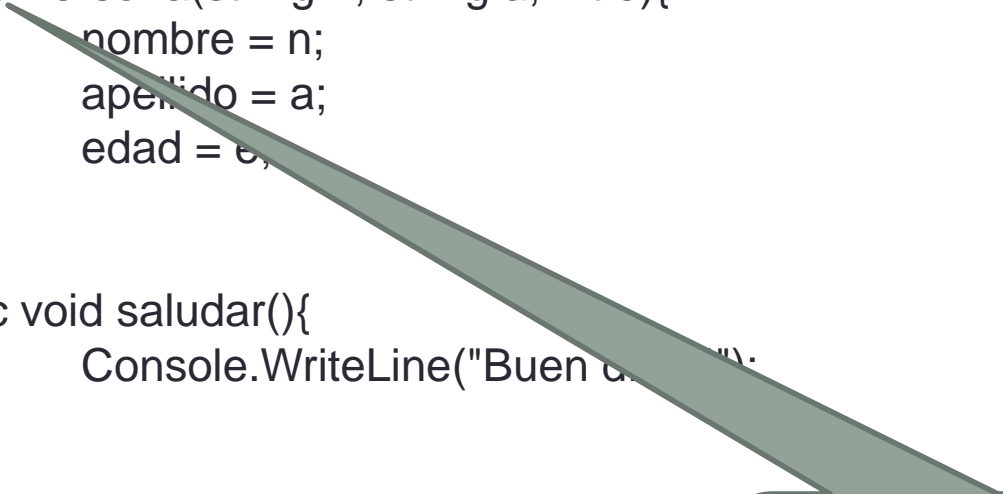
```
public class Persona {  
    private string nombre, apellido;  
    int edad;  
  
    public Persona(string n, string a, int e){  
        nombre = n;  
        apellido = a;  
        edad = e;  
    }  
  
    public void saludar(){  
        Console.WriteLine("Buen día");  
    }  
  
    public string getNombreYApellido(){  
        return nombre + " " + apellido;  
    }  
}
```



Si no se agrega un  
modificador se asume  
privado

# Modificadores de acceso

```
public class Persona {  
    private string nombre, apellido;  
    int edad;  
  
    public Persona(string n, string a, int e){  
        nombre = n;  
        apellido = a;  
        edad = e;  
    }  
  
    public void saludar(){  
        Console.WriteLine("Buen día.");  
    }  
  
    public string getNombreYApellido(){  
        return nombre + " " + apellido;  
    }  
}
```



En el caso de los constructores y métodos los declararemos privados, protegidos o públicos según su función

# Herencia de clases

- Las subclases se declaran usando el operador ":"

```
public class MiPrimerSubClase : MiPrimeraClase  
{  
  
}
```

# Clases abstractas

```
abstract public class Vehiculo {  
    private int cant;  
  
    abstract protected void acelerar(double vel);  
    abstract protected void girar(double angulo);  
    abstract protected void frenar();  
  
    public int cantidadDePasajeros() {  
        return cant;  
    }  
}
```



# Clases abstractas

```
abstract public class Vehiculo {  
    private int cant;
```

```
    abstract protected void acelerar(double vel);
```

```
    abstract protected void girar(double angulo);
```

```
    abstract protected void frenar;
```

```
    public int cantidadDePasajeros()  
    {  
        return cant;  
    }
```

```
}
```

```
}
```

El modificador **abstract** determina que la clase es abstracta.

Una clase abstracta **NO** se puede instanciar.

# Clases abstractas

```
abstract public class Vehiculo {  
    private int cant;
```

```
    abstract protected void acelerar(double vel);
```


```
    abstract protected void girar(double angulo);
```

```
    abstract protected void frenar();
```

```
    public int cantidadDeVehiculos() {  
        return cant;
```

```
    }
```

```
}
```



La clase Vehiculo "dice" que todo vehículo **debe saber** acelerar, girar y frenar, pero "no dice" **como hacerlo**. Por eso los métodos se declaran como abstractos (NO tienen implementación)

# Clases abstractas

```
abstract public class Vehiculo {  
    private int cant;
```

```
    abstract protected void acelerar(double vel);
```

```
    abstract protected void girar(double angulo);
```

```
    abstract protected void frenar();
```

```
    public int cantidadDePasajeros()  
    {  
        return cant;  
    }
```

```
}
```

```
}
```

Notar que los métodos abstractos NO tienen bloque de código, ni siquiera vacío.

# Clases abstractas

```
abstract public class Vehiculo {  
    private int cant;
```

```
    abstract protected void acelerar(double vel);
```

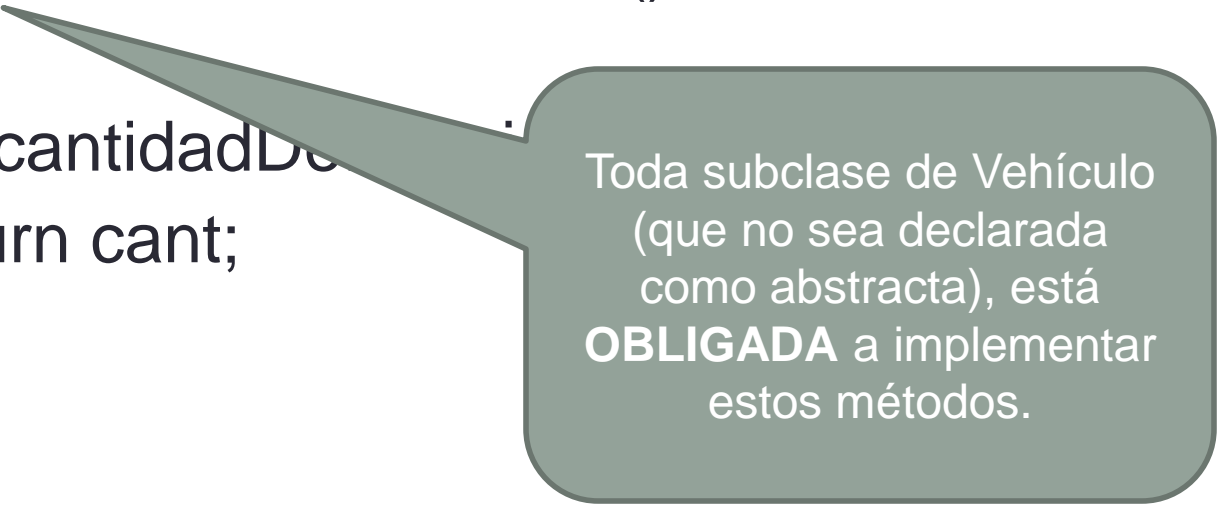
```
    abstract protected void girar(double angulo);
```

```
    abstract protected void frenar();
```

```
    public int cantidadDe  
        return cant;
```

```
}
```

```
}
```



Toda subclase de Vehículo  
(que no sea declarada  
como abstracta), está  
**OBLIGADA** a implementar  
estos métodos.

# Sobre-escritura de métodos

```
public class Auto : Vehiculo {  
  
    override protected void acelerar(double vel) {  
    }  
  
    override protected void girar(double angulo){  
    }  
  
    override protected void f  
    }  
}
```

La clase *Auto* DEBE  
implementar los métodos  
abstractos de *Vehiculo*

# Sobre-escritura de métodos

```
public class Auto : Vehiculo {
```

```
    override protected void acelerar(double vel) {  
    }
```

```
    override protected void girar(double angulo){  
    }
```

```
    override protected void f  
    }
```

Para sobre-escribir un método, sea abstracto o no, se debe anteponer a la declaración del método la directiva **override**

```
}
```

# Sobre-escritura de méto

```
public class Taxi : Auto {
```

En algunos casos, la sobre-escritura de un método necesita ejecutar el método de la superclase. Para ello se utiliza el comando `base`.

```
    override protected void girar(double angulo){  
        base.girar(angulo);  
        Console.WriteLine("Soy taxista, no  
                            necesito usar la luz de giro");  
    }  
}
```

# Sobre-escritura de método

```
public class Taxi : Auto {
```

```
    override protected void girar(double angulo){  
        Console.WriteLine("Soy taxista, no  
        necesito usar la luz de giro");  
        base.girar(angulo);  
    }  
}
```

El método girar de la superclase es un método más. Puede ser llamado en cualquier momento, incluso puede ser llamado más de una vez.



# Constructores en jerarquías

```
public class Musico : Persona {  
  
}
```

```
public static void Main(string[] args) {  
    Persona p = new Musico("Frank", "Zappa", 35);  
    p.saludar();  
}
```

Los constructores no se heredan de manera directa. A pesar de que en la clase *Persona* tenemos un constructor que recibe un int, como la clase *Musico* no lo tiene, esta línea da error.

# Constructores en jerarquías

```
public class Musico : Persona {  
    public Musico(string n, string a, int e) : base(n, a, e) {  
    }  
}
```

Estamos obligados a definir el constructor que recibe dos strings y un int en la clase *Musico*

```
public static void Main(string[] args) {  
    Persona p = new Musico("Frank", "Zappa", 35);  
    p.saludar();  
}
```

# Constructores en jerarquías

```
public class Musico : Persona {  
    public Musico(string n, string a, int e) : base (n, a, e){  
    }  
}
```

```
public static void Main(string[] args) {  
    Persona p = new Musico("Frank", "  
    p.saludar();  
}
```

Desde el constructor si se puede invocar al constructor de la superclase. Se hace con la palabra reservada **base** e indicando todos los parámetros que necesite el constructor heredado.

# Constructores en jerarquías

```
public class Musico : Persona {  
    public Musico(string n, string a, int e) : base (n, a, e){  
  
    }  
}
```

```
public static void Main(string[] args) {  
    Persona p = new Musico();  
    p.saludar();  
}
```

Esta línea ahora da error  
¿Por qué?

# Constructores en jerarquías

```
public class Musico : Persona {  
    public Musico(string n, string a, int e) : base (n, a, e){  
  
    }  
}
```

```
public static void Main(string[] args)
```

```
    Persona p = new Musico();  
    p.saludar();
```

```
}
```

Porque ahora la clase *Musico* no define un constructor sin parámetros (a pesar de que *Persona* si lo tiene)

# Constructores en jerarquías

```
public class Musico : Persona {  
    public Musico(string n, string a, int e) : base (n, a, e){  
  
    }  
    public Musico() { }  
}
```

Si queremos construir instancias sin pasar parámetros entonces estamos obligados a definir explícitamente el constructor sin parámetros en la clase *Musico*

```
public static void Main(string[] args) {  
    Persona p = new Musico();  
    p.saludar();  
}
```

# Interfaces

- Una interface es un "tipo de dato". Es como declarar una clase abstracta pero sin ser superclase de una jerarquía.
- Se declaran con la palabra reservada *interface*.

```
public interface Hablable {  
  
}
```

# Interfaces

- Todo método declarado en una interface es abstracto y público.
- La interface dice que métodos se deben implementar.

```
public interface Amigable {  
    void jugar();  
}
```



# Interfaces

- Todo método declarado en una interface es abstracto y público.
- La interface dice que métodos se deben implementar.

```
public interface Amigable {  
    void jugar();  
}
```

Notar que no es necesario usar los modificadores public ni abstract en la declaración de las funciones

# Interfaces

- Todo método declarado en una interface es abstracto y público.
- La interface dice que métodos se deben implementar.

```
public interface Amigable {  
    void jugar();  
}
```

*Amigable* es un tipo de objeto más, por lo tanto se puede usar como variable, como parámetro o como retorno de una función

# Interfaces

- Las clases puede implementar cero, una o más interfaces.

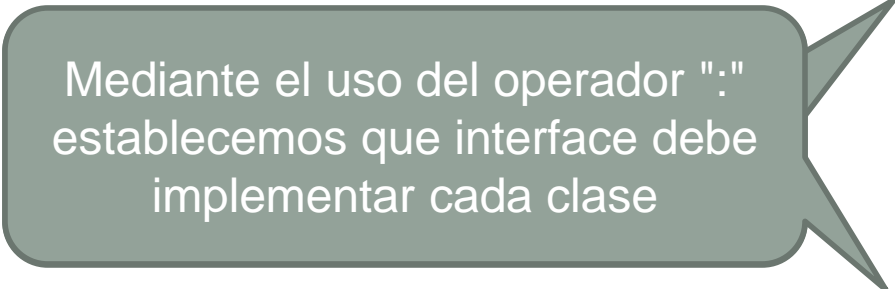
```
public class Persona : Amigable {  
  
}
```

```
public class Perro : Amigable {  
  
}
```

# Interfaces

- Las clases puede implementar cero, una o más interfaces.

```
public class Persona : Amigable {
```



Mediante el uso del operador ":"  
establecemos que interface debe  
implementar cada clase

```
public class Perro : Amigable {
```

```
}
```

# Interfaces

- Las clases puede implementar cero, una o más interfaces.

```
public class Persona : Amigable {
```

Las clases *Persona* y *Perro* están obligadas a implementar todos los métodos declarados en *Amigable*

```
public class Perro : Amigable {
```

```
}
```

# Interfaces

- Se pueden declarar variables de tipo interface, pero solo se pueden asignar instancias de clases que la implementen.

```
public void metodo(){  
    Amigable c1, c2, c3;  
  
    c1 = new Musico();  
    c2 = new Perro();  
  
    c3 = new Amigable();  
}
```

# Interfaces

- Se pueden declarar variables de tipo interface, pero solo se pueden asignar instancias de clases que la implementen.

```
public void metodo(){  
    Amigable c1, c2, c3;  
  
    c1 = new Musico();  
    c2 = new Perro();  
  
    c3 = new Amigable();  
}
```



Declaramos tres variables de tipo *Amigable*.

# Interfaces

- Se pueden declarar variables de tipo interface y se pueden asignar instancias de clases que implementen.

```
public void metodo(){  
    Amigable c1, c2, c3;  
  
    c1 = new Musico();  
    c2 = new Perro();  
  
    c3 = new Amigable();  
}
```

A estas variables le podemos asignar instancias de la clase *Persona*, *Perro* o cualquier subclase de ellas, ya que estas clases implementan la interface *Amigable*.



# Interfaces

- Se pueden declarar variables de tipo interface, pero solo se pueden asignar instancias de clases que la implementen.

```
public void metodo(){  
    Amigable c1, c2, c3;  
  
    c1 = new Musico();  
    c2 = new Perro();  
  
    c3 = new Amigable();  
}
```

Esta operación es incorrecta, las interfaces no se pueden instanciar.

# List<T>

- Usaremos la clase List como estructura de colecciones de elementos dinámicas.
- Para poder usar un List primero hay que instanciarlo indicando entre corchetes angulares el tipo de objetos que almacenarán.

```
using System.Collections.Generic;
```

```
List<Musico> m = new List<Musico>();
```

```
List<Persona> p = new List<Persona>();
```

```
List<Amigable> h = new List<Amigable>();
```

# List<T>

- Usaremos la clase List como estructura de colecciones de elementos dinámicas.
- Para poder usar un List primero hay que instar entre corchetes angulares el tipo de objetos que

Crea una colección que solo admite instancias de la clase *Musico*.

```
using System.Collections.Generic;
```

```
List<Musico> m = new List<Musico>();
```

```
List<Persona> p = new List<Persona>();
```

```
List<Amigable> h = new List<Amigable>();
```

# List<T>

- Usaremos la clase List como estructura de colecciones de elementos dinámicas.
- Para poder usar un List primero hay que instanciarlo entre corchetes angulares el tipo de objetos que contendrá.

```
using System.Collections.Generic;
```

```
List<Musico> m = new List<Musico>();
```

```
List<Persona> p = new List<Persona>();
```

```
List<Amigable> h = new List<Amigable>();
```

Crea una colección que solo admite instancias de la clase *Persona* y cualquiera de sus subclases.

# List<T>

- Usaremos la clase List como estructura de colecciones de elementos dinámicas.
- Para poder usar un List primero hay que instanciarlo entre corchetes angulares el tipo de objetos que contendrá.

```
using System.Collections.Generic;
```

```
List<Musico> m = new List<Musico>();
```

```
List<Persona> p = new List<Persona>();
```

```
List<Amigable> h = new List<Amigable>();
```

Crea una colección que solo admite instancias de cualquier clase que implemente la interface *Habla*.

# List<T>

- Al instanciar un List indicando el tipo de objetos que almacenarán permite controlar que no se agregue cualquier objeto y evita la necesidad de realizar un casting al momento de obtener los elementos que almacena.

```
List<Persona> a = new List<Persona>();  
Persona p = new Musico();  
a.add(p);  
a.add(new Musico());  
  
a[1].saludar();
```

Explícitamente  
decimos que el List  
solo va a contener  
Personas

# List<T>

- Al instanciar un List indicando el tipo de objetos que almacenarán permite controlar que no se agregue cualquier objeto y evita la necesidad de realizar un casting al momento de obtener los elementos que almacena.

```
List<Persona> a = new List<Persona>();  
Persona p = new Musico();  
a.add(p);  
a.add(new Musico());  
  
a[1].saludar();
```

Al List solo le podremos agregar Personas. Si agregamos elementos de otro tipo el compilador protestará.

# List<T>

- Al instanciar un List indicando el tipo de objetos que almacenarán permite controlar que no se agregue cualquier objeto y evita la necesidad de realizar un casting al momento de obtener los elementos que almacena.

```
List<Persona> a = new List<Persona>();  
Persona p = new Musico();  
a.add(p);  
a.add(new Musico());  
  
a[1].saludar();
```

El compilador ya sabe que el List *a* solo tiene Personas, no hace falta hacer el casteo de los elementos



# List<T>

- Los List permiten realizar muchas operaciones sobre ellos.
  - Add(T Valor). Añade el objeto representado por Valor (de tipo T).
  - Remove(T Valor). Elimina de la colección el elemento Valor.
  - RemoveAt(int indice). Elimina de la colección el elemento que se encuentra en la posición indice.
  - IndexOf(T Valor). Devuelve la posición donde se encuentra el elemento Valor o -1 si no se encuentra dentro de la colección.
- [ int indice ]. Los elementos de una List se acceden indicando el índice entre corchetes. Ej: list[0] = null;

# List<T>

- Los elementos de los List pueden ser recorridos con las estructuras de control for

```
List<Amigable> a = new List<Amigable>();
```

```
for(int x=0; x < a.Count; x++)  
    a[x].jugar();
```

```
foreach(Amigable h : a)  
    h.jugar();
```

# List<T>

- Los elementos de los List pueden ser recorridos con las estructuras de control for

```
List<Amigable> a = new List<Amigable>
```

```
for(int x=0; x < a.Count; x++)  
    a[x].jugar();
```

```
foreach(Amigable h : a)  
    h.jugar();
```

La propiedad Count devuelve la cantidad de elementos que tiene un List

# List<T>

- Los elementos de los List pueden ser recorridos con las estructuras de control for

```
List<Amigable> a = new List<Amigable>
```

```
for(int x=0; x < a.Count; x++)  
    a[x].jugar();
```

```
foreach(Amigable h : a)  
    h.jugar();
```

Los elementos se acceden mediante corchetes.  
El primer elemento se encuentra en la posición cero.

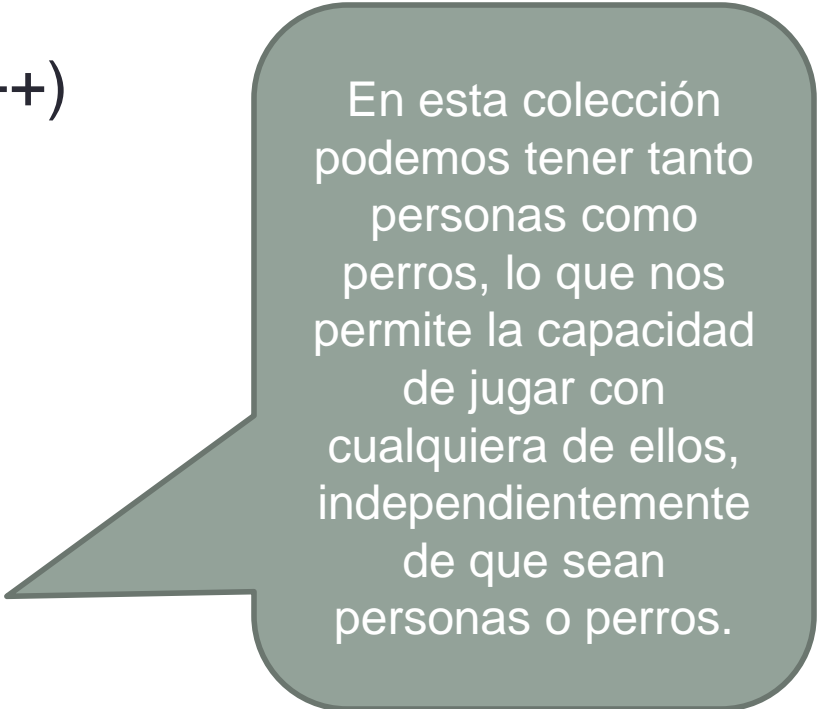
# List<T>

- Los elementos de los List pueden ser recorridos con las estructuras de control for

```
List<Amigable> a = new List<Amigable>();
```

```
for(int x=0; x < a.Count; x++)  
    a[x].jugar();
```

```
foreach(Amigable h : a)  
    h.jugar();
```



En esta colección podemos tener tanto personas como perros, lo que nos permite la capacidad de jugar con cualquiera de ellos, independientemente de que sean personas o perros.