

# ALGORITMOS Y PROGRAMACIÓN

---

Clase 4

Programación Orientada a Objetos

# Temario

- POO
  - Características
- Clases
  - Características
  - Variables de instancia
  - Métodos de instancia
  - Constructores
  - Instanciación de objetos
  - Modificadores de acceso
  - Propiedades

# Programación Orientada a Objetos

La Programación Orientada a Objetos es un paradigma de programación.

Cada paradigma:

- \* provee un conjunto de patrones conceptuales que conducen el proceso de diseño y desarrollo de un programa.

- \* brinda un conjunto de herramientas diferentes para comprender, analizar y representar problemas y expresar la solución en términos de una computadora.

# Programación Orientada a Objetos

En POO se programa por '**simulación**', se '**personifican**' *los objetos físicos*, dándoles las características y la funcionalidad que ellos tienen en el mundo real.

Decimos entonces que un programa en este paradigma:

*es un conjunto de OBJETOS que interactúan entre sí enviándose mensajes a través de los cuales solicitan la ejecución de una operación para llevar a cabo una tarea determinada.*

# ¿Qué es un objeto?

- Uno puede mirar a su alrededor y ver muchos objetos del mundo real: un perro, un escritorio, un televisor, una bicicleta etc. Los **sustantivos** son un buen punto de partida para **determinar los objetos** de un sistema.
- Cada objeto presenta dos características:
  - **Estado:** dado por el valor de sus atributos o cualidades.
  - **Comportamiento:** dado por su funcionalidad o las acciones que lleva a cabo.



Por ejemplo: un perro es un objeto que tiene **atributos**: nombre, color, raza.

**comportamiento**: ladrar, correr, jugar, etc.

# Estado y comportamiento de un objeto

## Estado

**Describe las características, cualidades, atributos o propiedades de un objeto.**

**Un alumno tiene nombre y apellido, dni y legajo.**

**Un producto tiene nombre, código, marca y precio.**

## Comportamiento

**Define el conjunto de operaciones que puede llevar a cabo un objeto (su funcionalidad)**

**Se puede consultar el nombre y apellido de un alumno, ver su dni, modificar su legajo, etc.**

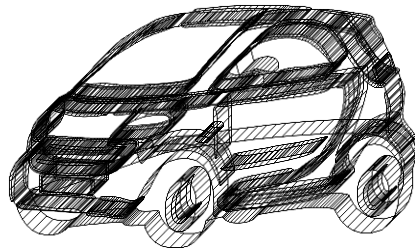
**Se puede guardar el nombre, la marca, el código y el precio de un producto, modificar su precio, etc..**

# ¿Qué es un objeto?

- Formalmente un objeto es *una instancia* de *una clase*.
- Todas las instancias de una clase tienen los mismos atributos y el mismo comportamiento.
- Pero cada instancia **tiene sus propios valores** para cada estado

Clase Auto

Estado de un auto:  
color, marca, tamaño,  
precio



Color  
verde  
Tamaño  
pequeño



Color  
amarillo  
Tamaño  
mediano



Color rojo  
Tamaño  
grande



# Cómo interactúan los objetos?

Los objetos interactúan enviándose *mensajes* y *responden* ejecutando *un método* que contiene las acciones correspondientes al mensaje recibido.

En un mensaje, siempre hay **un emisor** que es el objeto que lo envía y **un receptor que** es el objeto que lo recibe.

Formato del mensaje:    OR.mensaje(argumentos)

El objeto receptor tiene la **responsabilidad** de responder al mensaje recibido ejecutando el método asociado.

La interpretación del mensaje, es decir, el método usado para responder al mensaje es determinado por el receptor y podría variar dependiendo de quién lo recibe.



# Cómo interactúan los objetos?

Definición de un problema: Supongamos que Pablo quiere hacer un pedido de helado para que se lo entreguen en su domicilio.

Pablo, llama a la heladería. Lo atiende la telefonista, Silvina.  
Pablo solicita el pedido, indicándole la cantidad, gustos que desea y el domicilio donde debe enviarse el pedido.

**Pablo se comunicó con la telefonista, y le envió un mensaje con el requerimiento.**

Telefonista,  
agente  
apropiado

**Silvina, la telefonista tiene la responsabilidad de satisfacer el requerimiento.**



Mensaje con el requerimiento

**Pablo no necesita saber cómo la telefonista resolverá el problema.**

**La telefonista usará algún método para satisfacer el requerimiento.**

# Cómo interactúan los objetos?

La resolución del problema, puede requerir la ayuda de otros individuos.



Cada **objeto** cumple un rol: ejecuta una acción, que es usada por otros miembros de la comunidad.

El **emisor** envía un mensaje al **receptor**, junto con los argumentos necesarios para llevar a cabo el requerimiento.

El **receptor** es el objeto a quien se le envía el mensaje.

El receptor en respuesta al mensaje ejecutará un **método** para satisfacer el requerimiento.

Un método es la implementación de un mensaje.

# ¿Qué es una clase?

Una **clase** es un molde a partir de la cual se **crean objetos o instancias** con las mismas características y comportamiento.



**Roco** es una instancia de la clase Perro

**Ana, Abril y Belén** son instancias de la clase Alumno



**Silvia** es una instancia de la clase Cliente

El **auto** de Andrés es una instancia de la clase Auto



El **Banco** donde pago mis cuentas es una instancia de la clase Banco

**Agustina** es una instancia de la clase Empleado



# Clases e instancias.

- Todo objeto es una instancia de una clase, es decir pertenece a una clase determinada (Clasificación).
- Dicha clasificación se hace en base a comportamiento y atributos comunes.
- Una clase es el repositorio de los atributos y el comportamiento común de todos los objetos que pertenecen a dicha clase.



# Abstracción: base de la POO

- Se *abstraen* los objetos del mundo real con sus *características esenciales y su funcionalidad* y se los *encapsula y oculta* en una clase.
- Una clase se define a través de:
  - una *estructura interna*, donde se almacenarán los datos, representada por las *variables de instancia*, que describen los atributos de los objetos
  - de un conjunto de *funciones*, llamadas *métodos*, que definen el comportamiento.
- El estado de un objeto ***sólo se puede cambiar*** mediante los métodos, o sea, a través de las operaciones definidas como su comportamiento.

# Clases en .Net

.NET está organizado en clases: la BCL es un repositorio de *clases* y toda la funcionalidad que nos provee .NET está implementada en *clases*.

Program.cs

```
using System;
namespace Ejercicio1
{
    class Program
    {
        public static void Main(string[] args)
        {
            ...
        }
        public static void HacerAlgo()
        {
            ...
        }
        public static void NoHacerNada()
        {
            ....
        }
    }
}
```

Por defecto los proyectos utilizan la función Main de la clase Program como punto de entrada para la ejecución de cualquier programa.

Todo el código ejecutable tiene que estar dentro de funciones pertenecientes a alguna clase.

# Creación de Clases en C#

- Sintaxis de definición de clases

```
<tipo> class <nombreClase>
{
    <definición de los datos miembro>
    <definición de los funciones miembro>
}
```

## Donde:

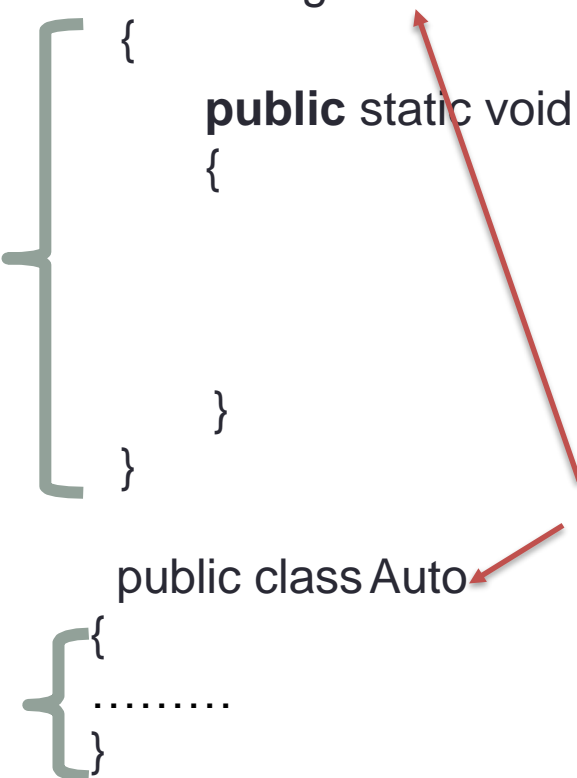
- **datos miembro**, son las variables de instancia
- **funciones miembro**, son los métodos
- **tipo**: especifica si es pública o interna

# Creación de Clases en C#

1ra forma: En este ejemplo la nueva clase está definida en el proyecto que la usa: se escribe en el mismo archivo debajo de la clase Program, dentro del mismo namespace.

```
namespace Ejercicio1
{
    class Program
    {
        public static void Main(string[] args)
        {
        }
    }

    public class Auto
    {
        .....
    }
}
```

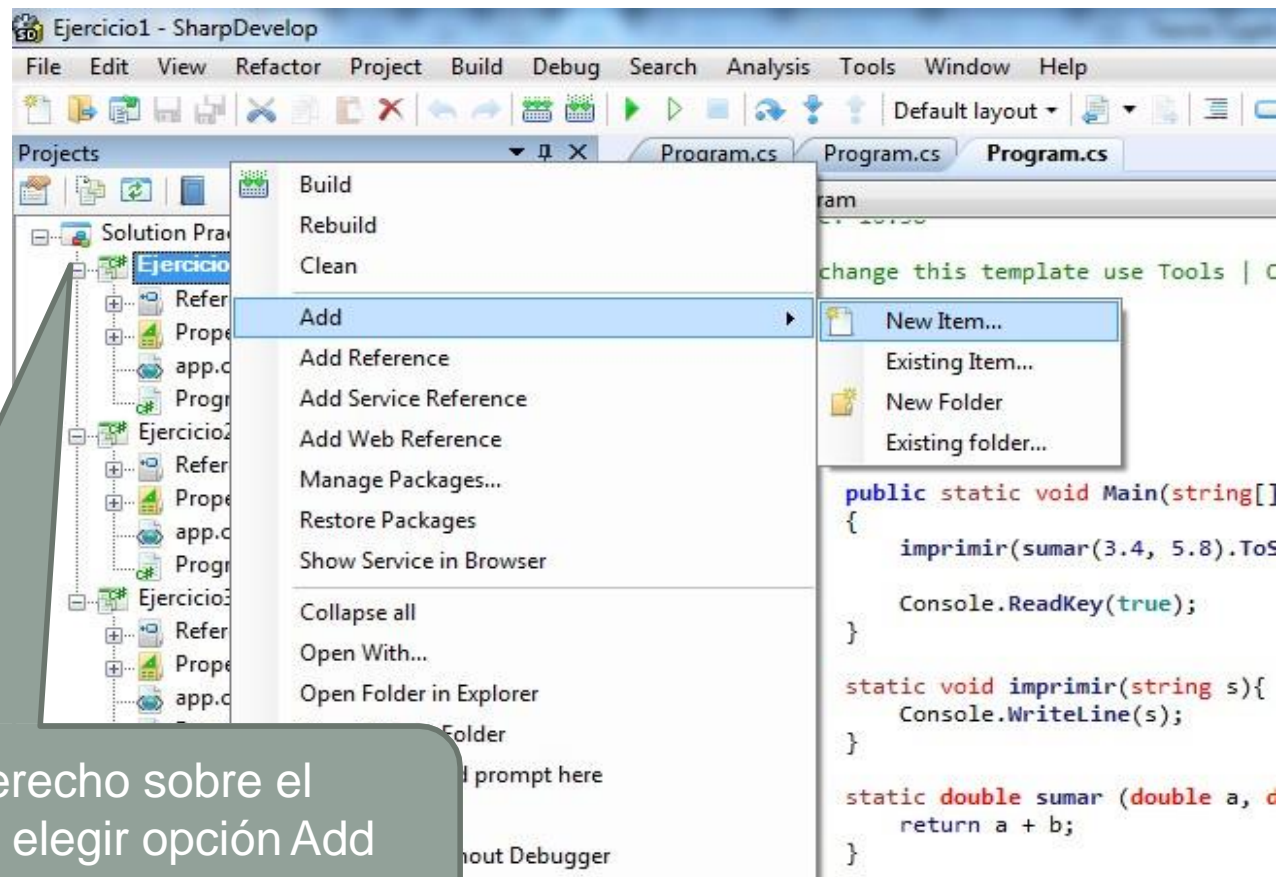


Recordar que cada clase debe tener su propio bloque de código encerrado entre llaves, en este caso, dentro del mismo namespace.



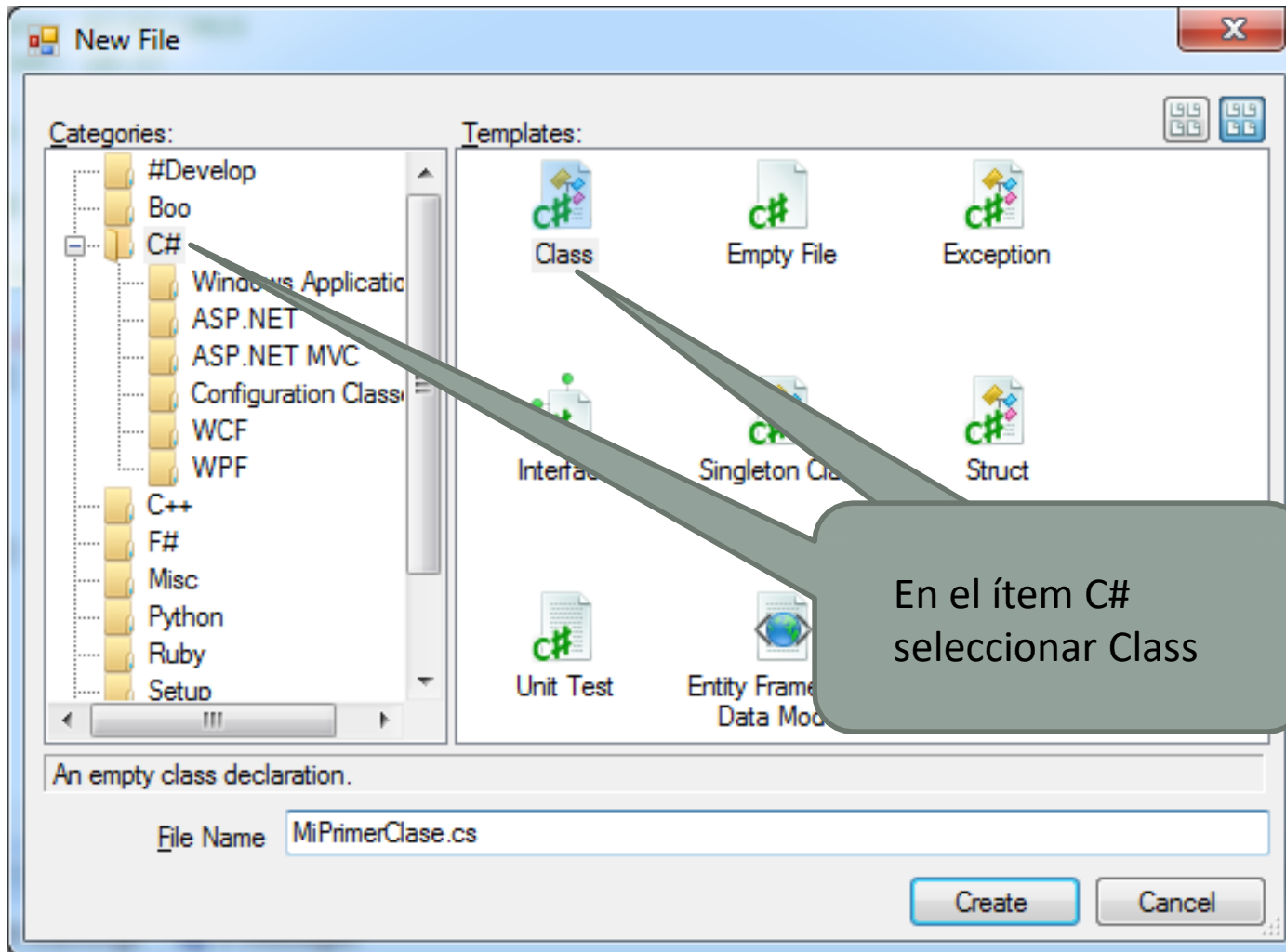
# Creación de Clases en C#

2da forma: En este ejemplo la nueva clase estará definida en otro archivo que se agrega al proyecto, siempre dentro del mismo namespace.

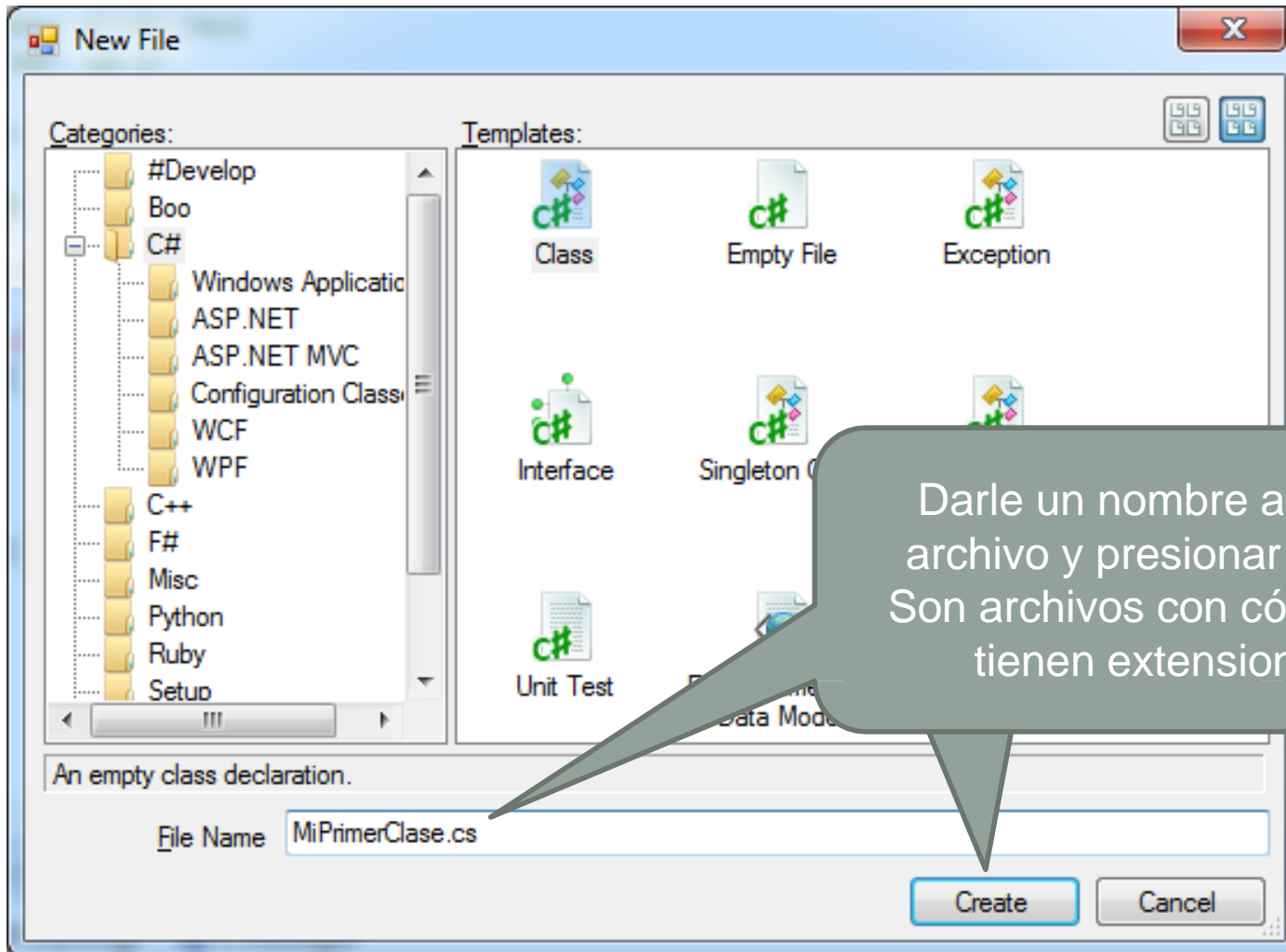


Click derecho sobre el proyecto y elegir opción Add  
→ New Item...

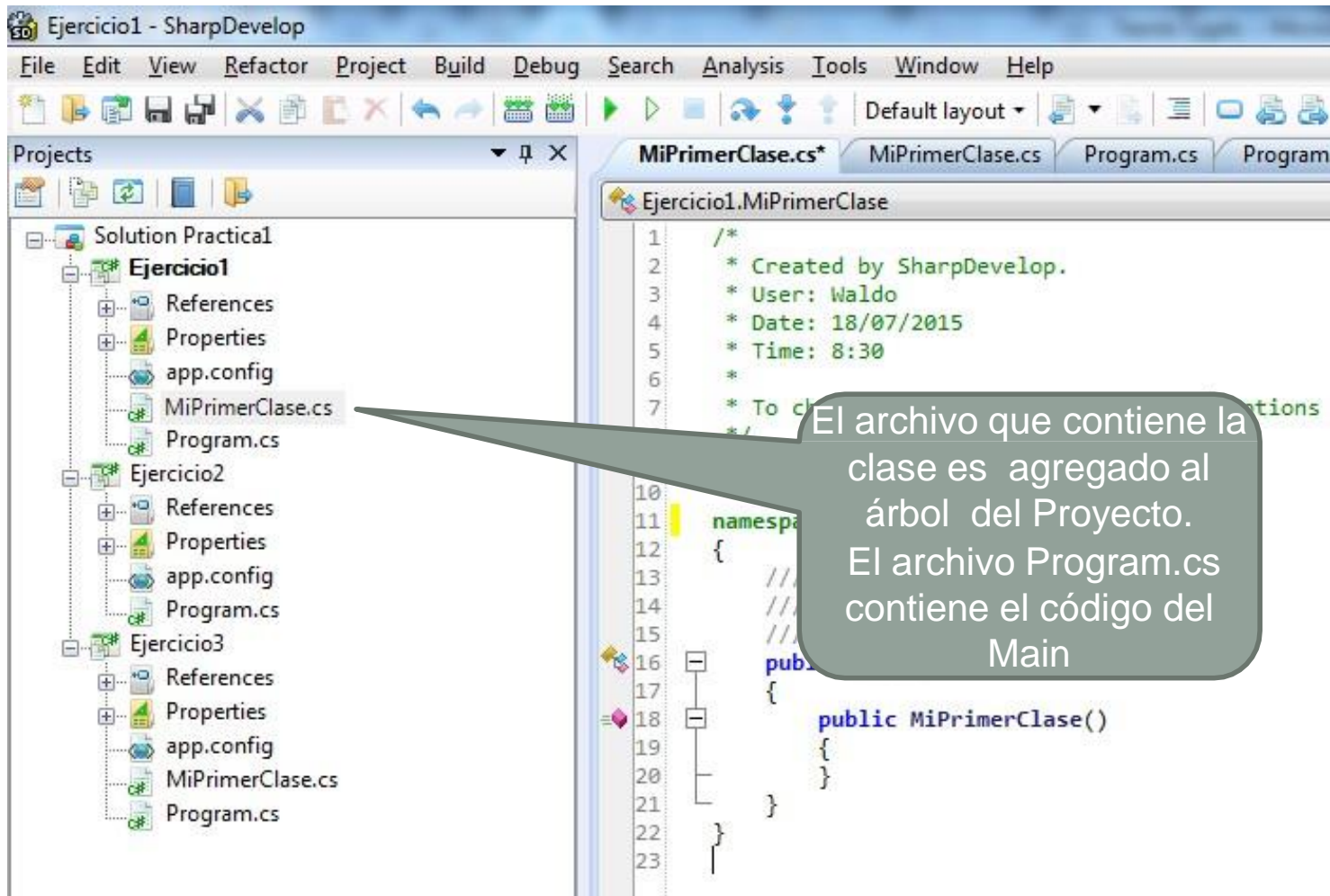
# Agregando una nueva clase al proyecto



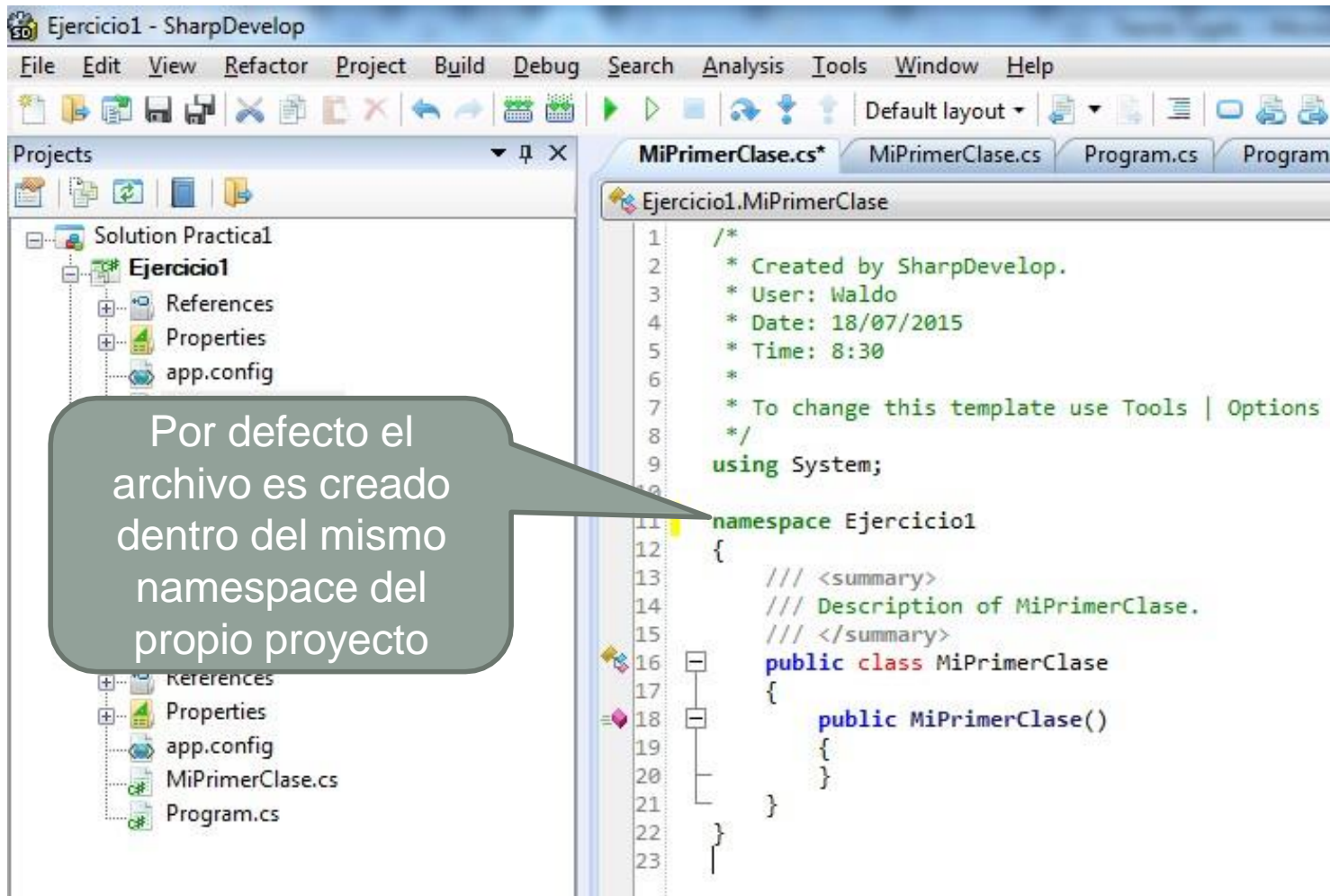
# Agregando una nueva clase al proyecto



# Agregando una nueva clase al proyecto



# Agregando una nueva clase al proyecto



# Usando la nueva clase desde el Main

```
namespace Ejercicio1
{
    class Program
    {
        public static void Main(string[] args,
        {

            MiPrimerClase m1;
            .....

            Console.ReadKey(true);

        }
    }
}
```

Declara un objeto  
de la clase  
MiPrimerClase

Al estar la clase dentro del  
mismo namespace que el  
proyecto, el Sistema  
Operativo reconoce su uso.

# Usando la nueva clase desde el Main

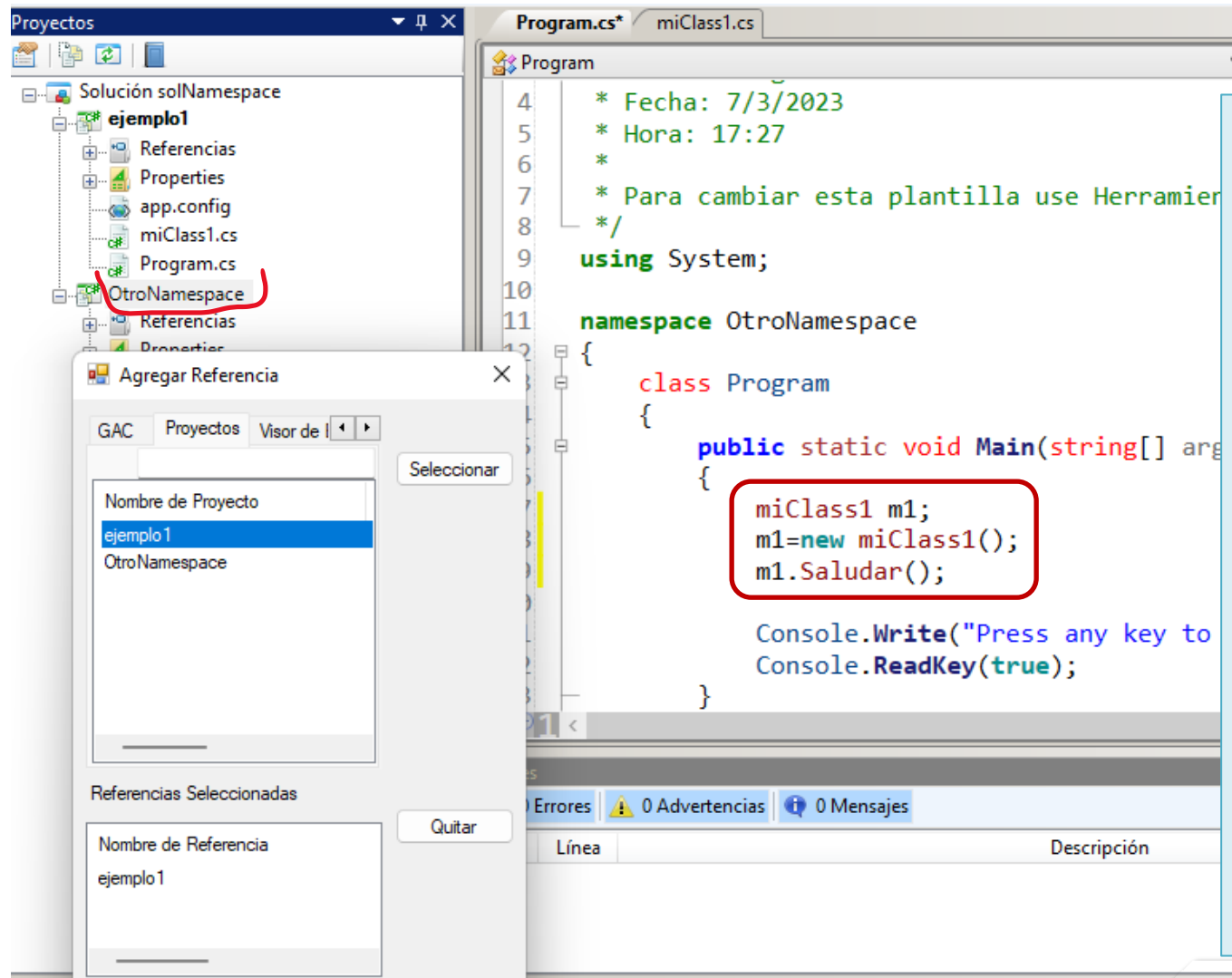
```
namespace ejemplo1
{
    public class MiPrimerClase
    {
        ....
        ....
    }
}
```

```
namespace OtroNamespace
{
    class Program
    {
        public static void Main(string[]
args)
        {
            MiPrimerClase m1;
            .....
            Console.ReadKey(true);
        }
    }
}
```

Si la clase está creada en un namespace distinto al del proyecto donde va a ser utilizada, hay que habilitar su uso y luego incluir el namespace con la cláusula using.



# Usando la nueva clase desde el Main



Para habilitar su uso: hay que ir al árbol de proyectos y hacer click derecho sobre el proyecto de destino donde será usada la clase, luego seleccionar "Agregar referencia". En la ventana nueva ir a la solapa "Proyectos" y elegir el proyecto que contiene la clase y clicar en Seleccionar. Finalizar dando click en el botón ok.



# Usando la nueva clase desde el Main

```
using ejemplo1
```

Luego solo falta incluir el espacio de nombres en el proyecto.

```
namespace OtroNamespace
{
    class Program
    {
        public static void Main(string[] args)
        {
            MiPrimerClase m1;
            .....

            Console.ReadKey(true);
        }
    }
}
```

# Definición de las variables de instancia

- Las variables de instancia se definen dentro de la clase con la siguiente sintaxis:

```
<modificadores> <tipoVarInstancia><nombreVarInstancia>;
```

Ejemplo:

```
public class Auto
{
    private string marca;
    private int motor;
}
```

El **modificador de acceso** determina el nivel de visibilidad de la variable de instancia. Si es **private** significa que no puede ser utilizado fuera de la clase en forma directa.

# Definición de los métodos de instancia

Los métodos se definen dentro de la clase con la siguiente sintaxis:

```
<modificadores><tipoResultado> <nombreMétodo> (<parametros>)  
{  
    <instrucciones>  
}
```

Los **métodos de instancia** permiten modificar y/o consultar los datos almacenados en las variables de instancia de un objeto desde cualquier lugar externo a la clase.

Dentro de los métodos pueden usarse directamente todas las variables de instancia de la clase.

# Definición de los métodos de instancia

- Ejemplo: Agregando el método `imprimir()` en la clase `Auto`, permite mostrar el valor de sus variables de instancia fuera de la clase.

```
public class Auto {  
    private string marca;  
    private int modelo;  
    public void imprimir(){  
        Console.WriteLine("Marca " + marca + " modelo " + modelo);  
    }  
}
```

Se utilizan las variables de instancia `marca` y `modelo` dentro del método `imprimir`

Cada vez que declaramos un método de instancia le agregamos el **modificador de acceso public**. De esta forma permitimos que sea invocado desde cualquier lado.

# Modificadores de acceso

- C# nos provee mecanismos para asegurar el **ocultamiento** mediante los modificadores de acceso que son palabras claves que se anteponen a la variable de instancia o al método y permiten especificar su nivel de accesibilidad o visibilidad.
- Los modificadores pueden ser:
  - Public
  - Private
  - Internal
  - Protected

# Modificadores de acceso

- **Miembros públicos**  
**public** int variable;

Pueden ser accedidos desde cualquier clase externa a quien declara el miembro.

- **Miembros privados**  
**private** int variable;

Pueden ser accedidos **ÚNICAMENTE** desde métodos declarados en la **MISMA** clase donde está el miembro privado.

- **Miembros internos**  
**internal** int variable;

Sólo tendrán acceso al miembro o tipo definido como internal dentro del mismo ensamblado/archivo.

- **Miembros protegidos**  
**protected** int variable;

Sólo la clase en la que se ha utilizado el modificador y sus clases derivadas tendrán acceso al miembro o tipo definido como protected.

Lo mismo vale para los métodos

# Instanciando objetos

Una vez definida la clase e implantada en el ambiente de desarrollo se pueden **crear o instanciar** objetos de dicha clase.

Para crear una instancia se utiliza el operador **new** y se debe especificar la clase del objeto a crear.

Si suponemos definida en el ambiente la clase Auto, hacemos:

**Auto a1;** declaración del objeto a1 de clase Auto  
**a1=new Auto();** y creamos un objeto de dicha clase.

Se pueden crear tantas instancias de una clase como se necesite.

# Instanciando objetos

```
namespace Ejercicio1
```

```
{
```

```
    class Program
```

```
    {
```

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

```
        {
```

```
            Auto a1;
```

```
            Auto a2;
```

```
            a1 = new Auto();
```

```
            a2 = new Auto();
```

```
        }
```

```
    }
```

```
    class Auto
```

```
    {
```

```
    }
```

```
}
```

Declaración de  
dos variables  
de tipo Auto:  
a1 y a2

Creación de dos  
instancias de la  
clase Auto: una  
almacenada en la  
variable a1 y otra  
en a2



# Invocando métodos

```
public static void Main(string[] args)
{
    Auto a1;
    a1 = new Auto();
    .....
    .....
    a1.imprimir();
    a2.imprimir();
}
```

Se le envía al objeto a1 el mensaje imprimir. El mensaje es la solicitud o invocación para que ejecute el método imprimir asociado.

Como el método es público se lo puede invocar desde cualquier unidad de programa.

Lo mismo ocurre con el objeto a2

# Sobrecarga de métodos en C#

La firma de un método está compuesta por:

- El nombre
- La cantidad de parámetros
- El tipo y el orden de los parámetros
- El tipo de pasaje de los parámetros
- El tipo de resultado NO es parte de la firma. Los nombres de los parámetros tampoco son parte de la firma.

```
<modificadores> <tipoResultado> <nombreMétodo> (<parametros>)  
{  
    <instrucciones>  
}
```

Una clase puede tener más de un método con el mismo nombre siempre que sus firmas sean diferentes. En este caso se produce la sobrecarga.

# Sobrecarga de métodos: ejemplo

Agreguemos en la clase Auto un nuevo método "acelerar" con tres sobrecargas

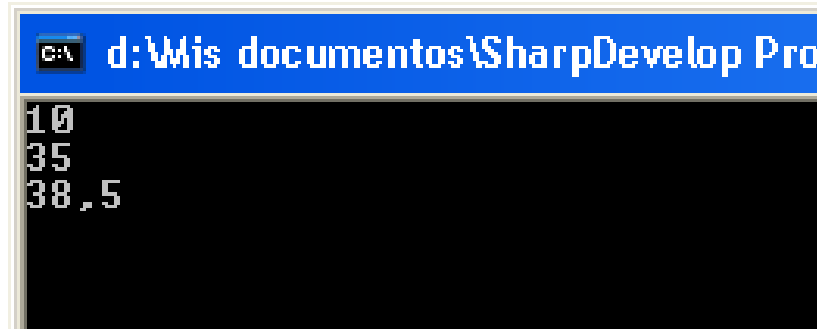
```
private double velocidad = 0;  
public double acelerar() {  
    return velocidad += 10;  
}  
public double acelerar(int valor) {  
    return velocidad += valor;  
}  
public double acelerar(double coeficiente) {  
    return velocidad *= coeficiente;  
}
```

# Sobrecarga de métodos: ejemplo

```
public static void Main(string[] args)
{
    Auto a1 = new Auto();

    Console.WriteLine(a1.acelerar());
    Console.WriteLine(a1.acelerar(25));
    Console.WriteLine(a1.acelerar(1.1));
}
```

Cómo sabe  
cuál método  
acelerar  
debe usar en  
cada caso?  
Por su firma!



The screenshot shows a Windows command prompt window with a blue title bar containing the text "d:\Wis documentos\SharpDevelop Pro". The command prompt itself has a black background with white text. The output of the program is displayed on three lines: "10", "35", and "38,5".

```
d:\Wis documentos\SharpDevelop Pro
10
35
38,5
```

# Constructores

- Un *constructor* definido en una clase es un **método especial** que permite crear una instancia de esa clase, inicializada o no.
- La sintaxis para definir un constructor es similar a la definición de cualquier otro método pero dándole el **mismo nombre que la clase** y **no indicando el tipo de valor de retorno**.


```
<modificadores> <nombreClase>(<parámetros>)  
{  
    <código>  
}
```

Los parámetros se detallan si se desea crear un objeto con sus atributos inicializados; en caso contrario se omiten.

# Constructores

Por ejemplo definimos en la clase **Auto** un constructor para crear un objeto inicializado, es decir que asigne valor a las variables de instancia en el momento de crear el objeto.

```
public class Auto {  
    private string marca;  
    private int modelo;  
    public Auto(string mar, int mod) {  
        marca = mar;  
        modelo = mod;  
    }  
    public void imprimir(){  
        Console.WriteLine("Marca y modelo: {0} {1}", marca, modelo); }  
}
```



**CONSTRUCTOR**

# Constructores

```
public class Auto {  
    private string marca;  
    private int modelo;  
    public Auto(string marca, int modelo)  
    {  
        this.marca = marca;  
        this.modelo = modelo;  
    }  
    public void imprimir(){  
        Console.WriteLine("Marca y modelo: {0} {1}", marca,  
            modelo);  
    }  
}
```

En el constructor utilizamos `this.marca` para diferenciar la variable de instancia `marca`, del parámetro `marca` del método. Idem con `modelo`.

# Uso de constructores

```
public static void Main(string[] args)
{
    Auto a1 = new Auto("Fiat", 2000);
    Auto a2 = new Auto("Ford", 2001);
    Auto a3 = new Auto();
    a1.imprimir();
    a2.imprimir();

    Console.ReadKey(true);
}
```

Los constructores se invocan con el operador new.

Si agregamos esta línea  
¿Qué sucede?



# Constructores por defecto

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

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



```
public Auto()  
{  
}
```

Si nosotros definimos un constructor, el compilador **no incluye** ningún otro constructor.

Por ello **Auto a3=new Auto();** da error de compilación pues el constructor por defecto no existe más.

# Constructores

```
public class Auto
```

```
{
```

```
    private string marca;
```

```
    private int modelo;
```

```
    public Auto(){
```

```
}
```

```
    public Auto(string marca, int modelo){
```

```
        this.marca = marca;
```

```
        this.modelo = modelo;
```

```
}
```

```
    public Auto(string marca, string modelo){
```

```
        this.marca = marca;
```

```
        this.modelo = Int64.Parse(modelo);
```

```
}
```

```
    public void imprimir(){
```

```
        Console.WriteLine("Marca " + marca + " modelo " + modelo); }
```

```
}
```

Se puede definir más de un constructor, ya que C# admite sobrecarga de métodos, siempre que sus firmas sean diferentes.

Constructor por defecto, no recibe parámetros

Constructor que recibe un string y un int

Constructor que recibe dos strings

# Constructores

```
public class Auto
```

```
{
```

```
    private string marca; private int modelo; public  
    Auto(){
```

```
}
```

```
    public Auto(string marca, int modelo){
```

```
        this.marca = marca;
```

```
        this.modelo = modelo;
```

```
}
```

```
    public Auto(string marca, string modelo){
```

```
        this.marca = marca;
```

```
        this.modelo = int.Parse(modelo);
```

```
}
```

```
    public void imprimir(){Console.WriteLine("Marca " + marca + " modelo "  
+ modelo);
```

```
}
```

```
}
```

```
Auto a2 = new Auto("Fiat", 2000);
```

```
Auto a3 = new Auto("Ford", "2010");
```

```
Auto a1 = new Auto();
```

# Puesta en común

Defina una clase llamada "Mascota" con cuatro campos: nombre de la mascota, especie (perro, gato, tortuga, etc.), edad en años y nombre del dueño.

Defina un constructor que reciba solo la especie.

Defina un constructor que reciba el nombre de la mascota y la especie.

Defina un constructor que reciba los cuatro campos.

¿Cómo se llama esta propiedad que permite definir varios constructores con el mismo nombre?

Los 3 constructores son ejemplos de sobrecarga de métodos: igual nombre pero diferente firma.

```
public class Mascota{  
    private string nombre, nombreDelDueño, especie;  
    private int edad;  
  
    public Mascota(string esp){  
        especie = esp;  
    }  
  
    public Mascota(string nom, string esp){  
        nombre = nom;  
        especie = esp;  
    }  
  
    public Mascota(string nom, string esp, string dueño, int e){  
        nombre = nom;  
        nombreDelDueño = dueño;  
        especie = esp;  
        edad = e;  
    }  
}
```

Agregue a la clase "Mascota" un **método de instancia** llamado "hablarConElDueño" que dependiendo de la especie haga su sonido característico (ladrar, maullar, relinchar, etc,)

```
public class Mascota{  
    ...  
  
    public void hablarConElDueño(){  
  
        switch(especie){  
            case "perro": Console.WriteLine("Estoy ladrando"); break;  
            case "gato": Console.WriteLine("Estoy maullando"); break;  
            case "caballo": Console.WriteLine("Estoy relinchando"); break;  
            case "canario": Console.WriteLine("Estoy piando"); break;  
            case "conejo": Console.WriteLine("Estoy chillando"); break;  
            default:  
                Console.WriteLine("Soy un animal que no habla");  
                break;  
        }  
    }  
}
```

Haga un programa que instancie diferentes mascotas, las guarde en un ArrayList y luego recorra la colección haciendo "hablar" a las mascotas



```
ArrayList mascotas = new ArrayList();  
Mascota m;
```

```
m = new Mascota ("Fufu", "conejo");  
mascotas.Add(m);
```

```
m = new Mascota ("Firulai", "gato", "Pedro", 5);  
mascotas.Add(m);
```

```
mascotas.Add(new Mascota ("Fido", "perro"));  
mascotas.Add(new Mascota ("Manuelita", "tortuga", "Elena", 4));  
mascotas.Add(new Mascota ("Dory", "pez"));  
mascotas.Add(new Mascota ("Silver", "caballo", "Llanero", 10));  
mascotas.Add(new Mascota ("Hedwig", "lechuza", "Harry", 5));
```

```
string nom = "Tweety", especie = "canario";  
m = new Mascota (nom, especie);  
mascotas.Add(m);
```

```
foreach(Mascota mm in mascotas)  
    mm.hablarConElDueño();
```

```
Estoy chillando  
Estoy maullando  
Estoy ladrando  
Soy un animal que no habla  
Soy un animal que no habla  
Estoy relinchando  
Soy un animal que no habla  
Estoy piando
```

# Puesta en común Ejercicio 1 – TP 4

**Codifique la clase Hora de tal forma que al ejecutar el siguiente programa de aplicación (Main) se imprima por consola:**

**23 HORAS, 30 MINUTOS Y 15 SEGUNDOS**

```
class Program {  
    public static void Main(string[] args)  
    {  
        Hora h=new Hora(23,30,15);  
        h.imprimir();  
        Console.ReadKey(true);  
    }  
}
```

- Definir la clase Hora en un nuevo archivo dentro del Proyecto
- ¿Qué variables de instancia podemos definir en la clase Hora?
- ¿Cómo será el constructor?

# Propiedades

- Una propiedad es una mezcla entre el concepto de campo y el concepto de método
- ***Es un método especial*** denominado descriptor de acceso, **que proporciona un mecanismo flexible para modificar o consultar el valor de un campo privado**, a la vez que oculta el código de implementación.
- Está compuesta por un:
  - Descriptor de acceso ***get***, se usa para consultar y retornar el valor de un campo privado → (getter)
  - Descriptor de acceso ***set***, se usa para asignar un nuevo valor a un campo privado → (setter)

# Propiedades – Sintaxis

Las propiedades se definen de la siguiente manera:

```
<tipo_propiedad> <nombre_propiedad>
{
    set  /*asignan o setean valor a un atributo*/
    {
        <codigo_escritura>
    }

    get  /*retorna el valor de un atributo*/
    {
        <codigo_lectura>
    }
}
```

# Propiedades

Las propiedades pueden ser:

- **de lectura y escritura** (en ambos casos tienen un descriptor de acceso **get y set**),
- **de solo lectura** (tienen un descriptor de acceso **get**, pero no **set**)
- **o de solo escritura** (tienen un descriptor de acceso **set**, pero no **get**). Las propiedades de solo escritura son poco frecuentes y se suelen usar para restringir el acceso a datos confidenciales.

# Ejemplo de Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno{  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}
```

**Declaración de la variable privada duracionTurno.**

**Declaración de una propiedad pública DuracionTurno que se llama distinto a la variable (en este caso tiene la primer letra en mayúscula).**

```
class Program{  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

**Una propiedad es usada como si fuera una variable.**

# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}  
  
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

Cuando se asigna valor en una propiedad, se ejecuta el bloque set de la propiedad.

# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}
```

```
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

`duracionTurno = value;`

`return duracionTurno;`

Value es el valor que está asignado a la propiedad, en este caso es 30.



# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}
```

**value** solo puede ser usado en el bloque **set**. El valor de value es del mismo tipo que el de la propiedad

```
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}
```

```
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

Cuando consulta el valor de una propiedad, se ejecuta el bloque get de la propiedad

# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}
```

El bloque get siempre debe devolver un valor del mismo tipo que el de la propiedad

```
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}  
  
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

¿Qué sucede con esta  
sentencia?

# Propiedades

```
class Agenda {  
    ...  
    private int duracionTurno;  
  
    public int DuracionTurno {  
        set {  
            duracionTurno = value;  
        }  
        get {  
            return duracionTurno;  
        }  
    }  
    ...  
}  
  
class Program {  
    Agenda a = new Agenda();  
    a.DuracionTurno = 30;  
    Console.WriteLine(a.DuracionTurno);  
    a.duracionTurno = 25;  
}
```

ERROR: La variable de instancia `duracionTurno` es privada, por lo tanto no se puede acceder fuera de la clase.

# Propiedades

- *Las propiedades son un mecanismo muy poderoso para asegurar el ocultamiento:*
  - Una buena práctica de programación es no permitir nunca el acceso directo a variables (declararlas siempre como privadas) sino mediante el uso de propiedades.
- El uso de propiedades permite tener más control sobre las escrituras, lecturas, modificaciones, adaptaciones, etc. del código.

# Puesta en común

Dada la clase Mascota realizada previamente, considere ahora agregar el atributo diagnóstico.

Defina las propiedades get y set para cada atributo.

```
public class Mascota{
    private string nombre, nombreDelDueño, especie, diagnos;
    private int edad;

    /*constructores*/
    public Mascota(){ }

    public Mascota(string nom, string esp, string dueño, int e){
        nombre = nom;
        nombreDelDueño = dueño;
        especie = esp;
        edad = e;
        diagnos ="";
    }
    /*propiedades*/
    public string Nombre{
        set {
            nombre = value;
        }
        get {
            return nombre;
        }
    }
}
```



```
public string NombreDelDueño{  
    set {nombreDelDueño = value; }  
    get {return nombreDelDueño; }  
}
```

```
public string Especie{  
    set {especie = value; }  
    get {return especie;}  
}
```

```
public int Edad{  
    set { edad = value; }  
    get { return edad; }  
}
```

```
public string Diagnostico{  
    set {  diagnos = value; }  
    get {  return diagnos;}  
}  
}
```

# Muchas gracias

