

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Ayudantía 01: C# parte 2

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

8 de Agosto, 2025

Consola

Consola

La clase `Console` dentro del namespace `System` nos permite interactuar con la consola.

Consola

La clase `Console` dentro del namespace `System` nos permite interactuar con la consola.

Escribir:

- `Console.Write(..)`: Muestra el texto en consola.
- `Console.WriteLine(..)`: Muestra el texto en consola y agrega un salto de línea.

Leer:

- `Console.ReadLine()`: Retorna un `string` con la línea escrita en consola.

Consola

La clase `Console` dentro del namespace `System` nos permite interactuar con la consola.

Escribir:

- `Console.Write(..)`: Muestra el texto en consola.
- `Console.WriteLine(..)`: Muestra el texto en consola y agrega un salto de línea.

Leer:

- `Console.ReadLine()`: Retorna un `string` con la línea escrita en consola.

Para convertir un `string` en otro tipo de dato se usa la clase `Convert`.

```
2 string str = Console.ReadLine();
3 int intVar = Convert.ToInt32(str);
4 double doubleVar = Convert.ToDouble(str);
```

Listas y Diccionarios

Listas

El namespace `System.Collections` incluye las clases `List` y `Dictionary`. En ambos casos se **debe definir el tipo de dato** sobre el que operan.

Listas

El namespace `System.Collections` incluye las clases `List` y `Dictionary`. En ambos casos se **debe definir el tipo de dato** sobre el que operan.

`List<tipo>` funciona prácticamente igual que una lista en python, pero solo puede contener datos del mismo tipo (que se define al crear la lista).

```
1  // Creamos una lista que solo puede contener strings
2  List<string> listaDeStrings = new List<string>();
3
4
5
6
7
8
9
10
11
```

Listas

El namespace `System.Collections` incluye las clases `List` y `Dictionary`. En ambos casos se **debe definir el tipo de dato** sobre el que operan.

`List<tipo>` funciona prácticamente igual que una lista en python, pero solo puede contener datos del mismo tipo (que se define al crear la lista).

```
1  // Creamos una lista que solo puede contener strings
2  List<string> listaDeStrings = new List<string>();
3  listaDeStrings.Add("hola");
4  listaDeStrings.Add("auto");
5  listaDeStrings.Add("mañana");
6
7
8
9
10
11
```

Listas

El namespace `System.Collections` incluye las clases `List` y `Dictionary`. En ambos casos se **debe definir el tipo de dato** sobre el que operan.

`List<tipo>` funciona prácticamente igual que una lista en python, pero solo puede contener datos del mismo tipo (que se define al crear la lista).

```
1  // Creamos una lista que solo puede contener strings
2  List<string> listaDeStrings = new List<string>();
3  listaDeStrings.Add("hola");
4  listaDeStrings.Add("auto");
5  listaDeStrings.Add("mañana");
6  Console.WriteLine(listaDeStrings[1]);     // muestra "auto"
7  Console.WriteLine(listaDeStrings.Count); // largo de la lista
8  listaDeStrings.Remove("auto");
9  Console.WriteLine(listaDeStrings[1]);     // muestra "mañana"
10
11
```

Listas

El namespace `System.Collections` incluye las clases `List` y `Dictionary`. En ambos casos se **debe definir el tipo de dato** sobre el que operan.

`List<tipo>` funciona prácticamente igual que una lista en python, pero solo puede contener datos del mismo tipo (que se define al crear la lista).

```
1  // Creamos una lista que solo puede contener strings
2  List<string> listaDeStrings = new List<string>();
3  listaDeStrings.Add("hola");
4  listaDeStrings.Add("auto");
5  listaDeStrings.Add("mañana");
6  Console.WriteLine(listaDeStrings[1]);     // muestra "auto"
7  Console.WriteLine(listaDeStrings.Count); // largo de la lista
8  listaDeStrings.Remove("auto");
9  Console.WriteLine(listaDeStrings[1]);     // muestra "mañana"
10 // Pero NO puedes agregar elementos que no sean del tipo de la lista
11 listaDeStrings.Add(3.2); // -> ERROR!
```

Listas

Para recorrer una lista podemos usar un `for`

```
List<int> listaDeInts = new List<int>();
listaDeInts.Add(93);
listaDeInts.Add(2);
listaDeInts.Add(4);
for (int i = 0; i < listaDeInts.Count; i++)
{
    int num = listaDeInts[i];
    Console.WriteLine($"Este numero es {num}");
}
```

Listas

... o un foreach:

```
List<int> listaDeInts = new List<int>();
listaDeInts.Add(93);
listaDeInts.Add(2);
listaDeInts.Add(4);
foreach(int num in listaDeInts)
    Console.WriteLine($"Este numero es {num}");
```

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

```
1  // En este caso, las llaves son string y los elementos son ints
2  Dictionary<string, int> dict = new Dictionary<string, int>();
3
4
5
6
7
8
9
10
11
12
13
14
```

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

```
// En este caso, las llaves son string y los elementos son ints
Dictionary<string, int> dict = new Dictionary<string, int>();
dict["siete"] = 7;
dict["cuatro"] = 4;
```

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

```
// En este caso, las llaves son string y los elementos son ints
Dictionary<string, int> dict = new Dictionary<string, int>();
dict["siete"] = 7;
dict["cuatro"] = 4;
Console.WriteLine(dict["cuatro"]);  // Muestra 4
```

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

```
// En este caso, las llaves son string y los elementos son ints
Dictionary<string, int> dict = new Dictionary<string, int>();
dict["siete"] = 7;
dict["cuatro"] = 4;
Console.WriteLine(dict["cuatro"]);  // Muestra 4
// Recorrer las llaves
foreach (string llave in dict.Keys)
    Console.WriteLine(llave);
```

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

```
1  // En este caso, las llaves son string y los elementos son ints
2  Dictionary<string, int> dict = new Dictionary<string, int>();
3  dict["siete"] = 7;
4  dict["cuatro"] = 4;
5  Console.WriteLine(dict["cuatro"]);  // Muestra 4
6  // Recorrer las llaves
7  foreach (string llave in dict.Keys)
8      Console.WriteLine(llave);
9  // Recorrer los valores
10 foreach (int valor in dict.Values)
11     Console.WriteLine(valor);
12
13
14
```

Diccionarios

Los `Dictionary<tipo llave, tipo elemento>` también funcionan prácticamente igual que en python, pero tienen tipos fijos de llaves y elementos.

```
// En este caso, las llaves son string y los elementos son ints
Dictionary<string, int> dict = new Dictionary<string, int>();
dict["siete"] = 7;
dict["cuatro"] = 4;
Console.WriteLine(dict["cuatro"]);  // Muestra 4
// Recorrer las llaves
foreach (string llave in dict.Keys)
    Console.WriteLine(llave);
// Recorrer los valores
foreach (int valor in dict.Values)
    Console.WriteLine(valor);
// Recorrer ambos
foreach (var e in dict)
    Console.WriteLine($"{e.Key} -> {e.Value}");
```

Manejo de Archivos

Manejo de Archivos

El namespace `System.IO` contiene todo lo que podrías necesitar para manejar archivos.

Manejo de Archivos

El namespace `System.IO` contiene todo lo que podrías necesitar para manejar archivos.

La carpeta donde inicia tu programa es donde se encuentra el binario que se genera al compilar el código $\approx$ `bin/Debug/net8.0/`. Digamos que contiene lo siguiente:

- `pictures/Lenna.png`
- `imgs`

Manejo de Archivos

El namespace `System.IO` contiene todo lo que podrías necesitar para manejar archivos.

La carpeta donde inicia tu programa es donde se encuentra el binario que se genera al compilar el código $\approx$ `bin/Debug/net8.0/`. Digamos que contiene lo siguiente:

- `pictures/Lenna.png`
- `imgs`

Ejemplo:

```
string file = "pictures/Lenna.png";
if (File.Exists(file))
{
    File.Copy(file, "imgs/Lenna.png");
    File.Delete(file);
    File.Move("imgs/Lenna.png", file);
}
```

¿Qué hace el siguiente código?

Manejo de Archivos

El namespace `System.IO` contiene todo lo que podrías necesitar para manejar archivos.

La carpeta donde inicia tu programa es donde se encuentra el binario que se genera al compilar el código $\approx$ `bin/Debug/net8.0/`. Digamos que contiene lo siguiente:

- `pictures/Lenna.png`
- `imgs`

Ejemplo:

```
string file = "pictures/Lenna.png";
if (File.Exists(file))
{
    File.Copy(file, "imgs/Lenna.png");
    File.Delete(file);
    File.Move("imgs/Lenna.png", file);
}
```

¡Este código no funciona en Windows!

Manejo de Archivos

Las rutas en Mac y Linux se separan con /:

```
2 string file = "pictures/Lenna.png";
```

Manejo de Archivos

Las rutas en Mac y Linux se separan con /:

```
string file = "pictures/Lenna.png";
```

Pero las rutas en Windows se separan con \.

```
string file = "pictures\\Lenna.png";
if (File.Exists(file))
{
    File.Copy(file, "imgs\\Lenna.png");
    File.Delete(file);
    File.Move("imgs\\Lenna.png", file);
}
```

Manejo de Archivos

Para que el código funcione en cualquier SO hay que usar `Path.Combine(...)`

```
1 string file = Path.Combine("pictures", "Lenna.png");
2 if (File.Exists(file))
3 {
4     File.Copy(file, Path.Combine("imgs", "Lenna.png"));
5     File.Delete(file);
6     File.Move(Path.Combine("imgs", "Lenna.png"), file);
7 }
```

... este es un error típico del proyecto que mata los test cases :/

Manejo de Archivos

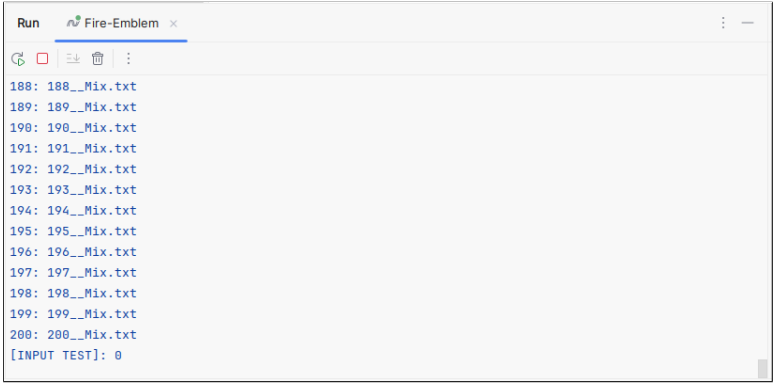
`Directory`: Permite manejar carpetas, crearlas, obtener los archivos en ella, etc.

```
// Creamos una carpeta si no existe
string folder = "imgs";
if(!Directory.Exists(folder))
    Directory.CreateDirectory(folder);

// Leemos los archivos .png en una carpeta y los mostramos en consola
string[] files = Directory.GetFiles("pictures", "*.png");
foreach (string file in files)
    Console.WriteLine(file);
```

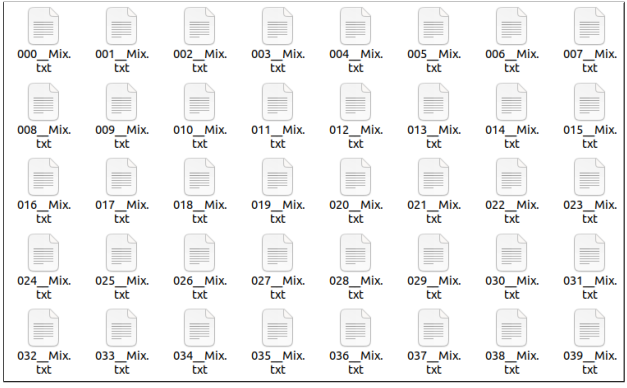
Manejo de Archivos

Otro error típico del proyecto.



Manejo de Archivos

Otro error típico del proyecto.



Manejo de Archivos

Otro error típico del proyecto.

```
1 view.WriteLine("Elige un archivo para cargar los equipos");
2 string[] teamFiles = Directory.GetFiles(teamsFolder, "*.txt");
3 for(int i = 0; i < teamFiles.Length; i++)
4 {
5     string teamFile = Path.GetFileName(options[i]);
6     view.WriteLine($"{i}: {teamFile}");
7 }
```

Manejo de Archivos

Otro error típico del proyecto.



Manejo de Archivos

Otro error típico del proyecto.

```
view.WriteLine("Elige un archivo para cargar los equipos");
string[] teamFiles = Directory.GetFiles(teamsFolder, "*.txt");
Array.Sort(teamFiles);
for(int i = 0; i < teamFiles.Length; i++)
{
    string teamFile = Path.GetFileName(options[i]);
    view.WriteLine($"{i}: {teamFile}");
}
```

Manejo de Archivos

Otro error típico del proyecto.

```
view.WriteLine("Elige un archivo para cargar los equipos");
string[] teamFiles = Directory.GetFiles(teamsFolder, "*.txt");
Array.Sort(teamFiles);
for(int i = 0; i < teamFiles.Length; i++)
{
    string teamFile = Path.GetFileName(options[i]);
    view.WriteLine($"{i}: {teamFile}");
}
```

Ahora el código funciona en cualquier S.O. :)

Lectura y Escritura de Archivos

Lectura y Escritura de Archivos

Podemos leer y escribir archivos de texto usando `File`.

Lectura y Escritura de Archivos

Podemos leer y escribir archivos de texto usando `File`.

Escribir un archivo:

```
1 string fileName = "myFile.txt";
2 string[] lines = new []{"line 1", "line 2", "line 3"};
3 File.WriteAllLines(fileName, lines);
```

El archivo queda guardado en la carpeta `bin/Debug/net8.0` (o equivalente)

Lectura y Escritura de Archivos

Podemos leer y escribir archivos de texto usando `File`.

Escribir un archivo:

```
string fileName = "myFile.txt";
string[] lines = new []{"line 1", "line 2", "line 3"};
File.WriteAllLines(fileName, lines);
```

El archivo queda guardado en la carpeta `bin/Debug/net8.0` (o equivalente)

Leer un archivo:

```
string fileName = "myFile.txt";
string[] lines = File.ReadAllLines(fileName);
```

En este caso, leemos el archivo ubicado en `bin/Debug/net8.0/myFile.txt`

Serialización en JSON

Serialización

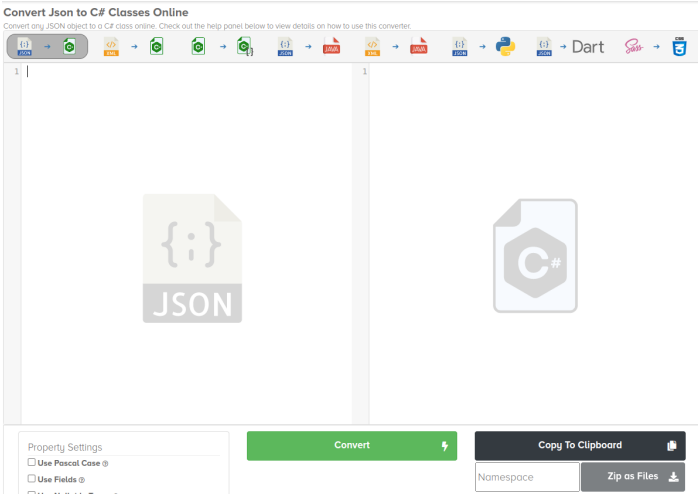
Digamos que te dan un JSON y te piden que lo importes en C#:

```
[
    {
        "Name":"Alexis",
        "LastName":"Sanchez",
        "Age":33
    },
    {
        "Name":"Arturo",
        "LastName":"Vidal",
        "Age":35
    }
]
```

Hay muchas formas de hacer esto (ver: `learn.microsoft.com`), pero acá les mostraré la forma más simple de todas.

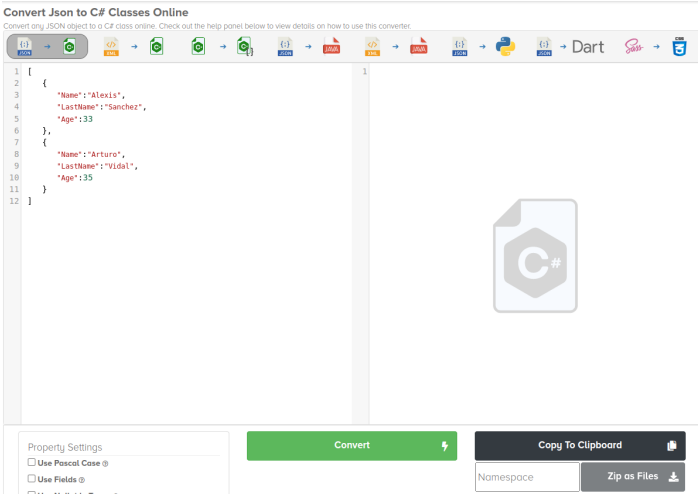
Serialización

Ir a `https://json2csharp.com/`



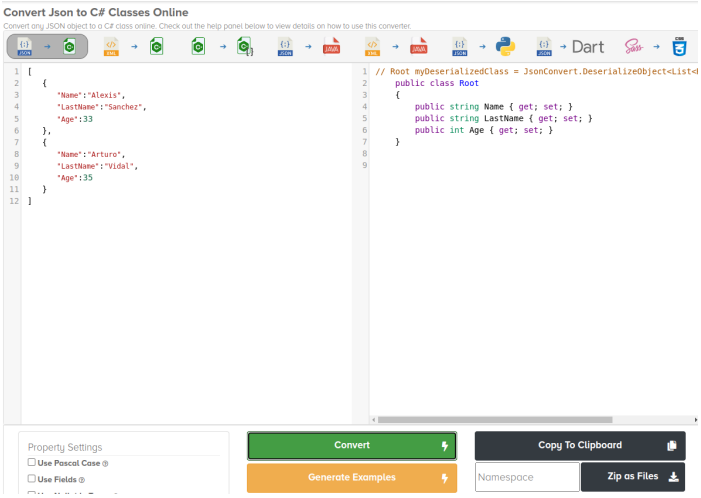
Serialización

Ir a `https://json2csharp.com/`



Serialización

Ir a `https://json2csharp.com/`



Serialización

La clase generada es una clase auxiliar cuya única responsabilidad es guardar la información del JSON leído (de hecho tiene atributos públicos por lo mismo).

```
1 public class PlayerInfo
2 {
3     public string Name { get; set; }
4     public string LastName { get; set; }
5     public int Age { get; set; }
6 }
```

Serialización

La clase generada es una clase auxiliar cuya única responsabilidad es guardar la información del JSON leído (de hecho tiene atributos públicos por lo mismo).

```
public class PlayerInfo
{
    public string Name { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

Para leer el JSON usamos el namespace: `System.Text.Json`

```
using System.Text.Json;

string myJson = File.ReadAllText("data.json");
var players = JsonSerializer.Deserialize<List<PlayerInfo>>(myJson);
foreach(var player in players)
  Console.WriteLine($"{player.Name} {player.LastName}: {player.Age}");
```

Serialización

La clase generada es una clase auxiliar cuya única responsabilidad es guardar la información del JSON leído (de hecho tiene atributos públicos por lo mismo).

```
public class PlayerInfo
{
    public string Name { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

Guardar un JSON es igual de fácil.

```
using System.Text.Json;

string fileName = "MyData.json";
string jsonString = JsonSerializer.Serialize(players);
File.WriteAllText(fileName, jsonString);
```

Modificadores Adicionales

Modificadores adicionales: static

Normalmente, para usar los métodos de una clase tenemos que crear un objeto de ella.

```
2 Random rnd = new Random();
3 int rndInt = rnd.Next();
```

Modificadores adicionales: static

Normalmente, para usar los métodos de una clase tenemos que crear un objeto de ella.

```
2 Random rnd = new Random();
3 int rndInt = rnd.Next();
```

Pero `Math` y `Console` nos dejan llamar a sus métodos sin crear un objeto.

```
2 double num = Math.Pow(2, 5);
3 Console.WriteLine(num);
```

¿Cómo lo hacen? 😱

Modificadores adicionales: static

Normalmente, para usar los métodos de una clase tenemos que crear un objeto de ella.

```
2 Random rnd = new Random();
3 int rndInt = rnd.Next();
```

Pero `Math` y `Console` nos dejan llamar a sus métodos sin crear un objeto.

```
2 double num = Math.Pow(2, 5);
3 Console.WriteLine(num);
```

¿Cómo lo hacen? 😱

C# nos permite definir métodos y atributos que están asociados a la clase misma y no a sus objetos.

Modificadores adicionales: static

Normalmente, para usar los métodos de una clase tenemos que crear un objeto de ella.

```
2 Random rnd = new Random();
3 int rndInt = rnd.Next();
```

Pero `Math` y `Console` nos dejan llamar a sus métodos sin crear un objeto.

```
2 double num = Math.Pow(2, 5);
3 Console.WriteLine(num);
```

¿Cómo lo hacen? 😱

C# nos permite definir métodos y atributos que están asociados a la clase misma y no a sus objetos. ¿Qué significa esto?

Modificadores adicionales: static

Partamos por los atributos estáticos.

Modificadores adicionales: static

Partamos por los atributos estáticos. Si definimos un atributo como `static` entonces ese atributo tendrá el mismo valor para **todos** los objetos de la clase.

Modificadores adicionales: static

Partamos por los atributos estáticos. Si definimos un atributo como `static` entonces ese atributo tendrá el mismo valor para **todos** los objetos de la clase.

```
2 public class A {
3     public static int StcAtt = 0;
4     public void AddOne() { StcAtt++; }
5     public void Print() {
6         Console.WriteLine("El atributo estático es " + StcAtt);
7     }
8 }
```

Modificadores adicionales: static

Partamos por los atributos estáticos. Si definimos un atributo como `static` entonces ese atributo tendrá el mismo valor para **todos** los objetos de la clase.

```
public class A {
    public static int StcAtt = 0;
    public void AddOne() { StcAtt++; }
    public void Print() {
        Console.WriteLine("El atributo estático es " + StcAtt);
    }
}
```

```
Console.WriteLine(A.StcAtt); // 0
A.StcAtt++;
Console.WriteLine(A.StcAtt); // 1
A a = new A();
a.AddOne(); // Sumamos 1 al valor estático desde objeto 'a'
A b = new A();
b.Print(); // "El atributo estático es 2"
```

Modificadores adicionales: static

También podemos definir métodos estáticos dentro de clases.

```
2 public class A {
3     private static int _stcAtt = 0;
4     public static void AddOne() { _stcAtt++; }
5     public static void Print() {
6         Console.WriteLine("El atributo estático es " + _stcAtt);
7     }
8 }
```

Modificadores adicionales: static

También podemos definir métodos estáticos dentro de clases.

```
2 public class A {
3     private static int _stcAtt = 0;
4     public static void AddOne() { _stcAtt++; }
5     public static void Print() {
6         Console.WriteLine("El atributo estático es " + _stcAtt);
7     }
8 }
```

```
2 A.Print(); // El atributo estático es 0
3 A.AddOne();
4 A.Print(); // El atributo estático es 1
```

Modificadores adicionales: static

También podemos definir métodos estáticos dentro de clases.

```
public class A {
    private static int _stcAtt = 0;
    public static void AddOne() { _stcAtt++; }
    public static void Print() {
        Console.WriteLine("El atributo estático es " + _stcAtt);
    }
}
```

```
A.Print(); // El atributo estático es 0
A.AddOne();
A.Print(); // El atributo estático es 1
```

Observaciones:

- Los métodos estáticos solo pueden acceder a variables y métodos estáticos.
- Pero un método **no** estático puede acceder sin problemas a todos los elementos estáticos de la clase.

Modificadores adicionales: static

Importante: No usen atributos estáticos en el proyecto.

Modificadores adicionales: static

Importante: No usen atributos estáticos en el proyecto.







... entre otros.

Modificadores adicionales: const

Otros modificadores que resultan útiles son `const` y `readonly`. Ambos limitan los cambios que se puede hacer sobre el valor de un atributo o variable.

Modificadores adicionales: const

Otros modificadores que resultan útiles son `const` y `readonly`. Ambos limitan los cambios que se puede hacer sobre el valor de un atributo o variable.

`const`

Indica que el valor de la variable/atributo es **constante** durante toda la ejecución del programa. Este valor debe ser asignado al momento de declarar la variable. Además, la variable debe ser un tipo por valor, un `string` o `null`.

```
2 const double pi = 3.1415;
3 // podemos usar la variable
4 double r = 1;
5 double area = pi*r*r;
6 // pero no podemos cambiar su valor
7 pi = 3.141592; // ERROR!
```

Modificadores adicionales: const

Otros modificadores que resultan útiles son `const` y `readonly`. Ambos limitan los cambios que se puede hacer sobre el valor de un atributo o variable.

`readonly`

Indica que el valor de la atributo **no puede cambiar una vez inicializado**. En este caso, el atributo puede ser declaro y darle un valor más adelante. Pero una vez que dicho valor es asignado no se podrá cambiar.

```
2  class Person {
3      private static int _rutActual = 0;
4      // definimos el rut como readonly porque no queremos
5      // permitir que se cambie el rut de las personas.
6      private readonly int _rut;
7      public Person() {
8          _rutActual++;
9          _rut = _rutActual; }
10 }
```

Punteros en C#

Punteros en C#

C# esconde el manejo de punteros mediante definir dos tipos de datos:

Punteros en C#

C# esconde el manejo de punteros mediante definir dos tipos de datos:

- **Tipos por valor**:
  - Tipos básicos: `int`, `float`, `double`, `short`, ...
  - Estructuras: `struct`.
  - Enumeraciones: `enum`.
- **Tipos por referencia**:
  - Clases (predefinidas y creadas por nosotros).
  - Arreglos.
  - strings.

Punteros en C#

C# esconde el manejo de punteros mediante definir dos tipos de datos:

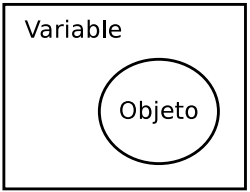
- **Tipos por valor**:
  - Tipos básicos: `int`, `float`, `double`, `short`, ...
  - Estructuras: `struct`.
  - Enumeraciones: `enum`.
- **Tipos por referencia**:
  - Clases (predefinidas y creadas por nosotros).
  - Arreglos.
  - strings.

La diferencia radica en el contenido de la variable declarada!

- **Value**: La variable contiene el valor mismo del objeto.
- **Reference**: La variable contiene una referencia al objeto (i.e., un puntero).

Tipos por valor

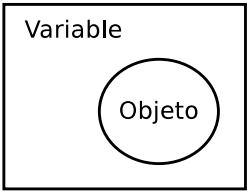
Por valor: La variable *contiene* al objeto.



Ejemplos: `int`, `struct`, `enum` y `Tuples`.

Tipos por valor

Por valor: La variable *contiene* al objeto.



Ejemplos: `int`, `struct`, `enum` y `Tuples`.

Consecuencia: `b = a`, copia objeto `a` en `b`.

Tipos por valor

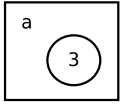
Ejemplo 1:

```
2 int a = 3;
3 int b = a;
4 b += 2;
5 Console.WriteLine("a=" + a + " y b=" + b + ".");
```

Tipos por valor

Ejemplo 1:

```
2 int a = 3;
3 int b = a;
4 b += 2;
5 Console.WriteLine("a=" + a + " y b=" + b + ".");
```

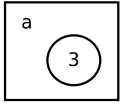


(a) `int a = 3;`

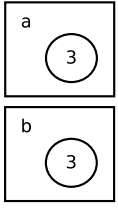
Tipos por valor

Ejemplo 1:

```
2 int a = 3;
3 int b = a;
4 b += 2;
5 Console.WriteLine("a=" + a + " y b=" + b + ".");
```



(a) `int a = 3;`

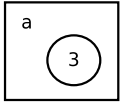


(b) `int b = a;`

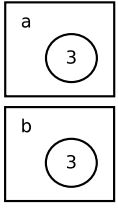
Tipos por valor

Ejemplo 1:

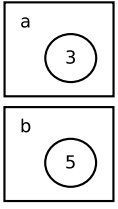
```
int a = 3;
int b = a;
b += 2;
Console.WriteLine("a=" + a + " y b=" + b + ".");
```



(a) `int a = 3;`



(b) `int b = a;`



(c) `b += 2`

Tipos por valor

Ejemplo 2:

```
2 int a = 3;
3 f(a);
4 Console.WriteLine(a);
5
6 void f(int b) {
7     b += 2;
8 }
```

¿Qué imprime este programa?

Tipos por valor

Ejemplo 2:

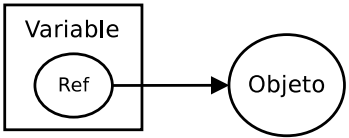
```
2 int a = 3;
3 f(a);
4 Console.WriteLine(a);
5
6 void f(int b) {
7     b += 2;
8 }
```

¿Qué imprime este programa?

Observación: Al dar "a" como parámetro implícitamente se hace un `b = a`.

Tipos por referencia

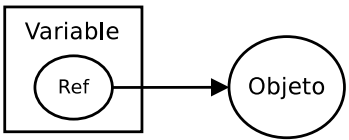
Por referencia: La variable *contiene* una referencia al objeto.



Ejemplos: `string`, arreglos y clases.

Tipos por referencia

Por referencia: La variable *contiene* una referencia al objeto.



Ejemplos: `string`, arreglos y clases.

Consecuencia: `b = a`, copia la referencia al objeto `a` en `b`.

Tipos por referencia

```
2  class Persona {
3      private string n;
4      public Persona(string n) {
5          this.n = n;
6      }
7      public void CambiarNombre(string nombre) {
8          n = nombre;
9      }
10     public void MostrarNombre() {
11         Console.WriteLine(n);
12     }
13 }
```

Tipos por referencia

Ejemplo 1:

```
2 Persona juan = new Persona("juan");
3 Persona pedro = juan;
4 pedro.CambiarNombre("pedro");
5 juan.MostrarNombre();
```

¿Qué muestra `juan.MostrarNombre()`?

Tipos por referencia

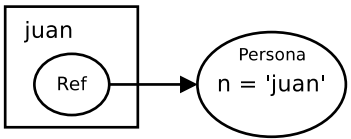
Ejemplo 1:

```
2 Persona juan = new Persona("juan");
3 Persona pedro = juan;
4 pedro.CambiarNombre("pedro");
5 juan.MostrarNombre();
```

¿Qué muestra `juan.MostrarNombre()`?

Ejecución del código:

`Persona juan = new Persona("juan")`



Tipos por referencia

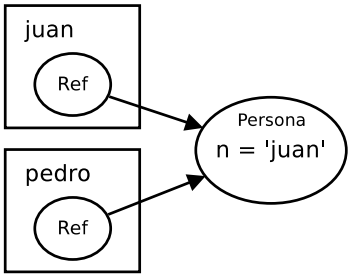
Ejemplo 1:

```
2 Persona juan = new Persona("juan");
3 Persona pedro = juan;
4 pedro.CambiarNombre("pedro");
5 juan.MostrarNombre();
```

¿Qué muestra `juan.MostrarNombre()`?

Ejecución del código:

`Persona pedro = juan`



Tipos por referencia

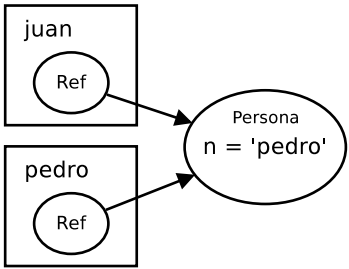
Ejemplo 1:

```
2 Persona juan = new Persona("juan");
3 Persona pedro = juan;
4 pedro.CambiarNombre("pedro");
5 juan.MostrarNombre();
```

¿Qué muestra `juan.MostrarNombre()`?

Ejecución del código:

`pedro.CambiarNombre("pedro")`



Tipos por referencia

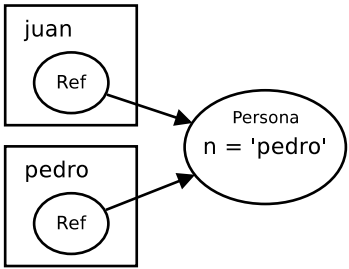
Ejemplo 1:

```
2 Persona juan = new Persona("juan");
3 Persona pedro = juan;
4 pedro.CambiarNombre("pedro");
5 juan.MostrarNombre();
```

¿Qué muestra `juan.MostrarNombre()`?

Ejecución del código:

`pedro.CambiarNombre("pedro")`



Por este motivo, `juan.MostrarNombre()` muestra `"pedro"`.

Tipos por referencia

Ejemplo 2:

```
2 Persona juan = new Persona("juan");
3 YoTeBautizoPedro(juan);
4 juan.MostrarNombre();
5
6 void YoTeBautizoPedro(Persona b) {
7     b.CambiarNombre("Pedro");
8 }
```

¿Qué imprime `juan.MostrarNombre()`?

Tipos por referencia

Ejemplo 2:

```
2 Persona juan = new Persona("juan");
3 YoTeBautizoPedro(juan);
4 juan.MostrarNombre();
5
6 void YoTeBautizoPedro(Persona b) {
7     b.CambiarNombre("Pedro");
8 }
```

¿Qué imprime `juan.MostrarNombre()`?

Observación: Al dar `juan` como parámetro implícitamente se hace un `b = juan`.

Output Arguments

The out keyword

También existe el keyword `out` que fuerza a nuestros argumentos a entregarse como referencia es vez de valor.

```
2 public class OutClass
3 {
4     public static void IsEven(int number, out bool result)
5         => result = (number % 2 == 0);
6 }
```

The out keyword

También existe el keyword out que fuerza a nuestros argumentos a entregarse como referencia es vez de valor.

```
2 public class OutClass
3 {
4     public static void IsEven(int number, out bool result)
5         => result = (number % 2 == 0);
6 }
```

out nos permite sobreescibir un valor desde el argumento.

```
2 bool isEven = true;
3 OutClass.IsEven(5, out isEven);
4 Console.WriteLine(isEven); //False
```

Output Arguments

Técnicamente, este método también tiene un output argument:

```
2 public static void EliminateDuplicationFrom(List<int> collection)
```

Pero tiene un nombre que hace obvio para el lector que está pasando

Output Arguments

Técnicamente, este método también tiene un output argument:

```
2 public static void EliminateDuplicationFrom(List<int> collection)
```

Pero tiene un nombre que hace obvio para el lector que está pasando

Para tu entrega:

- Evita utilizar la keyword `out`
- Usa funciones con muy buenos nombres si te ves obligado a modificar un tipo por referencia

Estructuras

Estructuras

Las estructuras son clases que se fuerzan a ser de **tipo por valor**.

Estructuras

Las estructuras son clases que se fuerzan a ser de **tipo por valor**.

```
2  class CPoint {
3      private int _x;
4      private int _y;
5      public CPoint(int x, int y) {
6          _x = x;
7          _y = y;
8      }
9      public void Square() {
10         _x = _x * _x;
11         _y = _y * _y;
12     }
13     public void Print() { Console.WriteLine($"({_x},{_y})"); }
14 }
```

Estructuras

Las estructuras son clases que se fuerzan a ser de **tipo por valor**.

```
2  struct SPoint {
3      private int _x;
4      private int _y;
5      public SPoint(int x, int y) {
6          _x = x;
7          _y = y;
8      }
9      public void Square() {
10         _x = _x * _x;
11         _y = _y * _y;
12     }
13     public void Print() { Console.WriteLine($"({_x},{_y})"); }
14 }
```

Estructuras

La única diferencia entre `SPoint` y `CPoint` es que `SPoint` es un `struct` y `CPoint` es una `class`. Pero esto hace que sus comportamientos sean distintos:

Estructuras

La única diferencia entre `SPoint` y `CPoint` es que `SPoint` es un `struct` y `CPoint` es una `class`. Pero esto hace que sus comportamientos sean distintos:

1) Las comparaciones entre `structs` son a nivel de sus atributos.

```
SPoint sp1 = new SPoint(3,4);
SPoint sp2 = new SPoint(3,4);
Console.WriteLine(sp1.Equals(sp2)); // True

CPoint cp1 = new CPoint(3,4);
CPoint cp2 = new CPoint(3,4);
Console.WriteLine(cp1.Equals(cp2)); // False
```

Estructuras

La única diferencia entre `SPoint` y `CPoint` es que `SPoint` es un `struct` y `CPoint` es una `class`. Pero esto hace que sus comportamientos sean distintos:

2) Las `structs` son tipos por valor por lo que asignaciones crean copias del `struct`.

```
2  SPoint sp1 = new SPoint(3,4);
3  SPoint sp2 = sp1;
4  sp1.Square();
5  sp1.Print(); // (9,16)
6  sp2.Print(); // (3,4)
7
8  CPoint cp1 = new CPoint(3,4);
9  CPoint cp2 = cp1;
10 cp1.Square();
11 cp1.Print(); // (9,16)
12 cp2.Print(); // (9,16)
```

Estructuras

Para que todo esto funcione – que podamos definir *clases por valor* – los `struct` deben cumplir un par de condiciones:

Estructuras

Para que todo esto funcione – que podamos definir *clases por valor* – los `struct` deben cumplir un par de condiciones:

- Al igual que las clases, los estructuras pueden tener atributos y métodos.
- Al momento de ser creadas, **todos** sus atributos se les deben asignar algún valor.

Estructuras

Para que todo esto funcione – que podamos definir *clases por valor* – los `struct` deben cumplir un par de condiciones:

- Al igual que las clases, los estructuras pueden tener atributos y métodos.
- Al momento de ser creadas, **todos** sus atributos se les deben asignar algún valor.

Normalmente, los `structs` se usan para crear tipos cuyo rol principal es almacenar datos (con poco o sin comportamiento asociado a ellos).