APUNTES C#

José Juan Urrutia Milán

Reseñas

- Curso C#: https://www.youtube.com/playlist?list=PLU8oAlHdN5BmpIQGDSHo5e1r4ZYWQ8m4B
- Todas las clases y paquetes de C#: https://docs.microsoft.com/es-es/dotnet/standard/class-libraries
- API C#: https://docs.microsoft.com/es-es/dotnet/api/
- Expresiones regulares de C#: https://docs.microsoft.com/es-es/dotnet/standard/base-types/regular-expression-language-quick-reference

Siglas/Vocabulario

- **IDE:** Entorno de Desarrollo Integrado.
- **API:** Interfaz de programación de aplicaciones (donde vienen explicados la mayoría de paquetes, clases y métodos de un lenguaje de programación)
- **BBDD:** Bases de datos.

Leyenda

Cualquier abreviatura o referencia será subrayada.
Cualquier ejemplo será escrito en **negrita.**
Cualquier palabra de la que se pueda prescindir irá escrita en cursiva.
Cualquier abreviatura viene explicada a continuación:

Abreviaturas/Referencias:

- 123: Hace referencia a cualquier número.
- nombre: Hace referencia a cualquier palabra/cadena de caracteres.
- a: Hace referencia a cualquier caracter.
- cosa: Hace referencia a cualquier número/palabra/cadena.
- Tipo_var o **…** : Hace referencia a cualquier tipo de variable primitiva o de tipo String.
- nombre_var: Hace referencia a cualquier nombre que se le puede dar a una variable.
- código: Hace referencia a cualquier instrucción. (Se usará para indicar dónde se podrá inscribir código.)
- variable: Hace referencia a cualquier variable.
- condición: Hace referencia a cualquier condición. Entiéndase por condición, una afirmación que devuelve un true o un false. Ej: (variable == 123). *Una variable del tipo boolean puede ser usada como una condición.

Índice

Introducción

Curso destinado al aprendizaje del lenguaje C#.
Es el lenguaje principal en Windows y parte principal en las herramientas Visual Studio y .NET.
Este lenguaje engloba las características más provechosas de otros.
Permite crear aplicaciones de escritorio, web, móviles y videojuegos.
Su sintaxis es parecida a C, C++ o Java.
C# es 100% orientado a objetos.
Durante el curso, usaremos Visual Studio como IDE y la versión 8 de C#.
Descarga del IDE en vídeo 1.
Uso principal del IDE en vídeo 2.
Toda aplicación en C# debe estar formada por al menos una clase.
C# es case sensitive, distingue entre mayúsculas y minúsculas.
Las sentencias en C# terminan con un “;” , salvo que la sentencia vaya seguida de corchetes o paréntesis.
No puede haber dos clases que se llamen de la misma forma a no ser que estén en diferentes **namespace** .

Capítulo I: Conceptos básicos

Título I: Comienzo

Como hemos visto, anteriormente, todo programa debe tener al menos una clase que esté dentro de un **namespace**. Dentro de esta clase, podremos nombrar al método **static void Main(string[] args){}**, el cual se ejecuta al abrir el programa. Este es el equivalente al método **main()** de Java.
Dentro del mismo namespace no podemos repetir nombres de clases o de métodos, pero en diferentes namespace sí.

namespace

Podríamos decir que un namespace es nuestro proyecto. En general, se le suele dar el mismo nombre que el al proyecto.
El namespace contiene a todas las clases, por lo que sí aquí declaramos una constante (o tipo enumerado), podremos acceder a él desde cualquier clase.
Este funciona a modo de paquete, ya que cuando importamos paquetes realmente estamos importando **namespace**.

Hola mundo

Para nuestro primer hola mundo, debemos importar el paquete System (ver **importar paquetes**). Posteriormente, crearemos nuestro namespace, al que le daremos un nombre. Dentro de este creamos nuestra clase y dentro de esta, el método **Main()** . Dentro de este haremos mención al método estático **WriteLine(string msg)** de la clase **Console**, al que le indicaremos una cadena de texto (ver tipos de variables).

```
using System;

namespace App
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hola mundo");
        }
    }
}
```

*Consultar Título III: Interpolación de strings

Atajo Visual Studio Ejecutar apps

Para ejecutar aplicaciones en Visual Studio mediante atajos de teclado:
Ctrl + S para guardar.
Ctrl + F5 para ejecutar.

Importar paquetes

Podemos importar paquetes con la ayuda de la palabra reservada **using**, seguida del nombre del paquete a importar:
using System;
De esta forma, podemos usar clases de otros paquetes como si fueran de nuestro programa.

También podemos hacer uso de clases de otros paquetes sin necesidad de importarlo, aunque en ese caso deberemos hacer mención al paquete de la clase, aunque es más recomendable importar el paquete:
System.Console.WriteLine("Hola mundo");

Importar métodos estáticos

Para importar todos los métodos estáticos que pertenezcan a una clase (y así dejar de tener que indicar el nombre de la clase delante),

indicamos la palabra **static** delante de la clase perteneciente a su paquete que queremos importar:

using static System.Math;

double raiz = Sqrt(8); *//Esto dentro del Main*

*Esto no se recomienda ya que puede llegar a causar confusión.

Comentarios

Para comentar una línea entera, indicaremos dos barras delante de esta: "//".

Para comentar varias líneas, indicaremos una barra y asterisco al principio "/*" y un asterisco y barra al final "*/":

//Esto es un comentario de una línea

/*Así se pueden

comentar

varias líneas*/

Existe otro tipo de comentarios, dedicados a decirle al programador por dónde se quedó programando el último día.

Para ello, indicamos un comentario de un línea que contenga al principio: "TODO:"

// TODO: Seguir mañana por aquí

Esto nos es de gran ayuda ya que en Visual Studio podemos abrir la pestaña Ver>Lista de tareas donde nos indica todos los comentarios de este estilo, la línea en la que se encuentran y si pinchamos nos lleva a esos comentarios.

Introducción de datos por teclado

Para introducir datos a través de la consola, debemos usar el método estático **ReadLine()** de la clase **Console**, el cual nos devuelve un **string**, por lo que en muchas ocasiones tendremos que usar castings (consultar título II, conversión de texto a número)**:**

string entrada = Console.ReadLine();

Compilar

Podemos compilar el programa para comprobar que este no tenga errores de ningún tipo (salvo errores en tiempo de ejecución).
Para ello, Compilar>Compilar solución.

Título II: Variables

El tipo de variable en C# depende del contenedor, similar a Java, no del contenido (como Python).
*Aunque podemos establecer una variable a partir del contenido, aunque este contenido no podrá variar de tipo (ver **declaración implícita de variable**).

Tipos de variables

En C#, sólo existen 3 tipos primitivos: enteros, reales y booleanos, aunque hay diferentes variantes de estos (también se pueden incluir dentro de tipos primitivos los char):

Tipo	Descripción	Tamaño (bits)
int	Números enteros	32
long	Números enteros muy grandes	64
float	Números decimales	32
double	Números decimales con parte decimal larga	64
decimal	Números decimales con parte decimal muy larga	128
string	Cadena de caracteres	16 / carácter
char	Un único carácter	16
bool	Booleanos	8

*Si indicamos una variable como n.0, será un **double**, no un **float**:
Console.WriteLine(5.0);
En este caso, está imprimiendo un double.

**Las cadenas de texto siempre se indican entre comillas dobles, al igual que en Java:
Console.WriteLine("Hola");
***Las variables char siempre se indican entre comillas simples, al igual que en Java:
char letra = 'A';
string texto = "Hola";
****Al inicializar una variable de tipo float, debemos especificar un sufijo F (al igual que en Java, pero con f mayúscula):
float gramos = 3.56F;
Esto no pasa con double, ya que si no indicamos nada, C# interpretará que es un double como hemos visto en (*).
*****En una cadena de texto, indicamos un salto de línea con "\n", al igual que en Java:
string texto = "Esto es una línea\nEsto es otra\nY esto otra";

Declaración de variables

Para declarar una variable, indicamos el tipo de dato que almacenará, seguido del nombre de la variable:
int edad;
string nombre;
*Podemos declarar múltiples variables en la misma fila siempre y cuando sean del mismo tipo:
int edad1, edad2, edad3;
string nombre, apellido, municipio, pais, comunidad;

Inicialización de variables

Para inicializar una variable, si esta no está creada, indicamos el tipo de dato, el nombre, un espacio y el valor que almacenará.
Si la variable ya está declarada, podemos indicar sólo el nombre de la variable, el igual y el valor (la sintaxis es idéntica a la de Java):
int edad;
edad = 5;

int año = 1999;

*C# nos permite inicializar múltiples variables a la vez con el mismo valor:
int edad1, edad2, edad3, edad4;
edad1 = edad2 = edad3 = edad4 = 23;

Valor por defecto

A veces, no se inicializan variables y estas adoptan su valor por defecto. Este es:
- En variables numéricas: 0. (0.0, 0.0F ...)
- En string: **null**.
- En char: **null**.
- En bool: false.
- En objetos: **null**.

Constantes

Podemos crear variables cuyo valor no podrá cambiar una vez inicializadas, si las declaramos constantes. Para ello, debemos crear una variable de forma normal pero antes del tipo de variable debemos especificar la palabra reservada **const** .
*Las constantes deben declararse e inicializarse en la misma línea, no se pueden declarar y luego inicializar.
-Convención: por convención, el nombre de las constantes debe ir en mayúsculas:
const double PI = 3.1415;
Todas las constantes se consideran campos estáticos aunque no se le indique.

Declaración implícita de variable

Podemos crear una variable cuyo tipo se establecerá dependiendo del contenido, aunque esta variable no podrá cambiar de tipo nunca más:
var edad = 27; *// es lo mismo que: int edad = 27;*

*Lo que no se puede hacer es cambiar ahora el tipo de la variable, como se puede hacer en Python:
edad = "Hola"; *//Esto está MAL , es un fallo.*
**Tampoco está permitido declarar la variable y luego inicializarla, se deben de hacer las dos cosas a la vez.

Castings

Podemos realizar conversiones de valores almacenados en unas variables a otras variables de diferente tipo mediante castings.
Estos se realizan indicando el tipo de variable al que se quiere convertir delante de la variable que se quiere convertir:
A estos castings se les llama conversión explícita.

```
double temperatura = 34.5;
int temp = (int) temperatura;
```

*Cuando convertimos de real a entero, la parte decimal se trunca, se pierde.

También podemos realizar conversiones implícitas, aunque esto sólo se nos permite entre tipos de variables compatibles entre sí, como de int a long o de float a double:

```
int habitantes = 1000000;
double ciudadanos = habitantes;
```

Conversión de texto a número

Podemos convertir texto a número con la ayuda del método **Parse(string value)**, ya que esto no es posible con un casting:

```
string numero = "36";
int num;
num = int.Parse(numero);        //o Int32.parse(numero);
```

Ejecutamos el método **Parse()** sobre el tipo de dato al que queremos convertir el texto.

Generar número aleatorio

Random nombre = new Random();
int nombre2 = nombre.Next(int limite1, int limite2);

El número aleatorio estará entre limite1 y limite2.

Título III: Operadores

Aritméticos

-Suma/Concatenación: +
-Aumento: ++
-Incremento en: +=5
-Resta: -
-Decremento: --
-Decremento en: -=5
-Multiplicación: *
-División: /
-Resto: %
Idéntico a Java.
**Si en C# operamos con valores enteros, el resultado será un entero, pero si operamos con valores decimales, el resultado será un decimal.
***La concatenación funciona igual a la de Java.

Comparación

-Igual a: ==
-Menor: <
-Menor o igual: <=
-Mayor: >
-Mayor o igual: >=
-Diferente: !=

Lógicos

-AND: &&
-(*) NOT: !

-OR: ||

*Las variables booleanas se pueden invertir:
bool mayor = false;
if(!mayor) Console.WriteLine(“Eres menor”);

Nota a la hora de incrementar o decrementar

Si estamos imprimiendo o pasando cadenas de texto a la vez que aumentamos o reducimos una variable, debemos indicar el operador antes de la variable:
int edad = 17;
Console.WriteLine($“Tengo {edad++} años”); *//Tengo 17 años*
Console.WriteLine(edad) *//18*
edad = 17;
Console.WriteLine($“Tengo {++edad} años”); *//Tengo 18 años*

Comparación de cadenas de texto

Los datos **string** al no ser un dato primitivo, si comparamos dos cadenas con “==” puede darnos error aunque esté bien hecho, por lo que es recomendable el uso del método **String.Compare(string n1, string n2, bool ignoreCase),** donde introducimos las dos cadenas y si va a ignorar o no mayúsculas y minúsculas (Compare es igual al equals de Java):

```
string nombre = “Juan”;
if(String.Compare(nombre, “juan”, true))
        Console.WriteLine(“Sí”);
```

Interpolación de strings

La interpolación de strings es una forma de concatenar strings con variables de forma diferente a la del uso del “+”.
Para ello, indicamos el símbolo del dólar “$” y a continuación abrimos las comillas dobles y declaramos nuestra cadena de texto.

La diferencia es que las variables las nombraremos dentro de esta cadena, dentro de unos corchetes:
Console.WriteLine(“Tengo ” + edad + “ años y ” + meses + “ meses”); *//Concatenación normal*
Console.WriteLine($”Tengo {edad} años y {meses} meses”);
// ^ Interpolación de strings

*Existe otra forma de concatenar strings con valores numéricos dentro de un **WriteLine**, pasando varios parámetros y haciendo referencia a ellos dentro del string:
Consolo.WriteLine(“Yo tengo {0} años y {1} meses”, edad, meses);

Título IV: Clase Math

La clase Math nos ofrece métodos para realizar ciertas operaciones.

(static) Pow(double base, double exponente)

Eleva el primer parámetro al segundo parámetro y devuelve el resultado.

(static) Sqrt(double x)

Devuelve (**double**) la raíz cuadrada del número especificado.

(static) Round(double x)

Devuelve (int) el redondeo de un **double**.

Título V: Métodos

Un método puede recibir múltiples parámetros pero sólo puede devolver uno o ninguno (tipo de variable = **void**)
Esta es la sintaxis de un método o función:

```
tipoDevuelto nombre(parámetros){
    código;
}
```

Dentro de **parámetros**, debemos especificar el tipo de variable y el nombre de las variables. Los parámetros se separan entre ellos por una coma.
La instrucción de devuelta de parámetros se hace con un **return** Si el método devuelve void, no se podrá especificar ningún **return** .
*Si un método no es estático, no podremos acceder a él sin un objeto, por lo que si lo hacemos estático podremos acceder a él desde la misma clase sin necesidad de hacer mención al nombre de la clase. (La sintaxis es idéntica a la de Java).
-Convención: Si el nombre del método sólo tiene una palabra, en C# suelen empezar por mayúscula (esto no es igual en otros lenguajes).

Expression-bodied / Operador lambda

Si el método que vamos a construir sólo contiene una línea en su interior y esta es una instrucción **return**, no es necesario indicar los corchetes (realizamos un expression-bodied):
*El operador lambda (=>) equivale a escribir **return** .

```
/*static int Suma(int n1, int n2)
{
        return n1+n2;
}*/
static int Suma(int n1, int n2) => n1+n2;
```

Sobrecarga de métodos

Podemos construir diferentes métodos con el mismo nombre, siempre y cuando reciban diferentes tipos de parámetros o diferentes números de parámetros.

```
int Suma(int n1, int n2) => n1+n2;
double Suma (double n1, int n2) => n1+n2;
int Suma(int n1, int n2, int n3) => n1+n2+n3;
```

El compilador selecciona el método adecuado.

Podemos usar parámetros opcionales en métodos para no tener que realizar una sobrecarga, pero en algunas ocasiones no podremos hacer esto.

Recibir parámetros opcionales

Nuestros métodos pueden recibir parámetros opcionales. Lo que hacemos es inicializar cierto parámetros opcional para que, en caso de que no se pase este adopte ese valor, pero si este se pasa, adopte el valor pasado por el usuario:

static int Suma(int n1, int n2, int n3=0) => n1+n2+n3;
Suma(5, 6) *//11, n3 = 0*
Suma(5, 6, 11) *//22, n3 = 11*

*Se pueden recibir infinitos parámetros opcionales, aunque estos **deben** ir después de los obligatorios.
**En caso de ambigüedad, de que se pueda llamar a un método o a otro, se llamará al que mejor se adapte a la petición:

Suma(5, 7);
static int Suma(int n1, int n2, int n3=0) => n1+n2+n3;
static int Suma(int n1, int n2) => n1+n2;

En este caso, aunque sea correcto llamar a cualquiera de los dos, se llama al segundo ya que es el que más se adapta a la situación.

Título VI: Condicionales

if

```
if(condición){
        código;
}else if(condición){
        código;
}else
{
        código;
```

}

*El else if y el else son opcionales, se pueden indicar o no.
Se pueden indicar tantos else if como se desee.
**Las llaves de un if, else if o else se pueden eliminar si el código a ejecutar es una única línea (la tercera especificada se ejecutará sí o sí):

```
if(edad>18)
        Console.WriteLine("Eres mayor de edad");
        Console.WriteLine("Esta línea se ejecuta sí o sí");

if(edad>18) Console.WriteLine("Eres mayor);
else Console.WriteLine("Eres menor");
```

***Si se evalúa una variable booleana, no es necesario indicar ningún operador, la condición será true si la variable es true, false si la variable es false:

```
bool mayor = true;
if(mayor)
        Console.WriteLine("Eres mayor");
```

switch

```
switch(variable){
        case valor1:
                código;
                break;
        case valor2:
                código;
                break;
        default:
                código;
                break;
}
```

El default se ejecutará si ningún case se cumple.

Sólo se ejecutará el case que tenga el valor de la variable especificada en el switch.
*Podemos introducir tantos casos como queramos.
**No es obligatorio incluir siempre un default.
***Sólo podemos utilizar switch para evaluar int, char o string.
****Podemos sustituir un break por un return, throw o goto.

Título VII: Bucles

Idénticos a Java (salvo el for each).
*Los bucles pueden prescindir de los corchetes siempre y cuando el código a repetir sea una línea. En este caso, la tercera línea no estará dentro del bucle.

while

```
while(condición){
    código;
}
```

do while

```
do{
    código;
}while(condición)
```

for

```
for (int i=valor, condiciónConi; cambioEni){
    código;
}
```

for each

La principal misión del bucle for each es la de recorrer arrays o elementos iterables desde el principio hasta el final.

```
foreach(tipovar nombre in array){
    código;
```

}

En el bucle for each, un objeto del mismo tipo de variable que el array a recorrer pero de una dimensión menos (si es un array 2D, se usa un array, si es un array, se usa una variable), adopta el valor que hay en cada posición del array por cada vuelta del bucle.
*Si queremos recorrer un array de clases anónimas, indicar **var** como tipo de variable.

ej:

```
int[] edades = {4, 6, 10, 14, 18, 26};
foreach(int e in edades){
    Console.WriteLine(e);
}
```

Título VIII: Excepciones

try catch

```
try{
    código;
}catch(ClaseExcepcion nombre){
    código;
}finally{
    código;
}
```

-Convención: por convención, el nombre de las excepciones suele ser "e" o "ex"
Si el try se ejecuta, el catch no se ejecuta y viceversa.
*El finally se ejecutará siempre, independientemente de que el try se ejecute o que el catch se ejecute.
**No es necesario que todas las estructuras try catch lleven siempre un finally.

***Se pueden añadir tantos catch como se desee para que cada uno capture un error diferente y por consiguiente, ejecute un código diferente.

Message

Podemos imprimir en consola el error exacto de la excepción, esto lo hacemos dentro del catch con el campo de clase **Message** del objeto de la excepción:

```
try{
    código;
}catch(Exception e){
    Console.WriteLine(e.Message);
}
```

Excepciones genéricas

En vez de capturar una excepción concreta, podemos capturar la excepción **Exception** , la cual es el padre de todas las excepciones, por lo que independientemente del tipo de excepción que salte, se capturará. Esta opción no es recomendable, ya que habrá que afinar todo lo posible. Podemos darle un tratamiento especial a una excepción y aún así seguir capturando las demás (en este caso, la genérica siempre DEBE ir al final aunque (ver **filtros**)).

*Si realizamos esto, no es necesario usar los paréntesis del catch ni indicar nada:

```
try{
    código;
}catch(Exception e){
    código;
}
//o
try{
    código;
}catch{
```

código;
}

filtros

Podemos hacer que un catch genérico atrape todas las excepciones menos una/algunas:

```
try{
}catch (Exception e) when (e.GetType()!=typeof(ClaseExcepcion))
{
    código;
}catch (ClaseExcepcion e){código;}
```

checked

En C# , cuando alcanzamos el valor máximo de una variable (por ejemplo, intentamos sumar 20 al valor máximo que puede almacenar un int y lo guardamos en un int), el compilador ignora este desbordamiento y sigue con su ejecución, pasando del número más alto al número más bajo.

Para que esto no suceda y nos lance la excepción pertinente, podemos meter el código dentro de un **checked**:

```
int numero = int.MaxValue + 20;
Console.WriteLine(numero); //-2147183629
```

Para que lance excepción:

```
checked{
    int numero = int.MaxValue + 20; //Lanza OverflowException
    Console.WriteLine(numero);
}
```

//o

```
int numero = checked(int.MaxValue+20);
Console.WriteLine(numero);
```

*Visual Studio tiene una opción para que todo el código que se escriba se almacene automáticamente dentro de un **checked:**

Explorador de soluciones>click dcho sobre proyecto>Propiedades>Compilación>Avanzadas…>Comprobar el desbordamiento y subdesbordamiento aritmético
Checked y unchecked sólo funcionan con **int y **long**.

unchecked

Si tenemos la opción activada vista en (*), **checked**, podemos usar un bloque **unchecked** para que lo de dentro no lance esta excepción.
*Checked y unchecked sólo funcionan con **int** y **long**.

Lanzar excepciones de forma manual / throw

throw new ClaseExcepción();
throw new Exception();

Esto hará que debemos capturar con un try catch el método donde se encuentra esta línea de texto.

Título IX: Arrays

Los arrays se usan igual que las variables pero declarando corchetes a la hora de declarar e inicializarlos al lado del tipo de dato.
Y al lado de su nombre a la hora de usarlo.
Los arrays llevan especificado la cantidad de valores que se almacena en su interior. Esta cantidad no se puede cambiar.
Los arrays empiezan a contar en 0.
La sintaxis es similar a Java.

Declarar array:	**int[] edades;**
Inicializar array:	**edades[] = new int[4];** *//Cantidad valores*
En una línea:	**int[] edades = new int[4];**
Dando valores:	**int[] edades = {8, 9, 12, 14};**
Otra forma:	**int[] edades = new int[4] {8, 9, 12, 14};**
Cambiar valores:	**edades[0] = 9;**

*Si no especificamos ningún valor en un hueco, se indica el valor por defecto del tipo de dato que estemos almacenando.

Arrays implícitos

En estos arrays no especificamos ni el tipo de dato que almacena el array ni la cantidad de datos que este almacena, por lo que es bastante flexible a la hora de trabajar con él:

var datos = new[] {“Juan”, “Ana”, “Marta”};

El tipo de dato es dado por el compilador. Este tipo no puede variar, aunque se pueden especificar datos compatibles como int y double (en este caso, el compilador indica que es de tipo double).

*Se pueden crear arrays de clases anónimas, aunque estos deben ser arrays implícitos.

Length

Length es un campo de clase de array que nos devuelve (int) cuantos elementos tenemos dentro de nuestro array (Interesante con bucle for).

Título X: Lectura de ficheros externos / streams

(dentro de un try catch)

1. Creamos una instancia de la clase **StreamReader** (paquete System.IO) y le pasamos al método constructor un string que contenga la ruta del archivo, indicando antes de las primeras comillas dobles una “@”.
2. Mientras haya líneas para leer, (el método **RealLine()** sobre nuestro objeto sea diferente de **null**), igualamos el método **RealLine()** a una variable **string**, la cual podemos ir almacenando en otra para almacenar todo el archivo.
3. Cerramos el stream (con **close())** que abrimos en el punto 1 para salvar recursos. Esto lo podemos automatizar con la ayuda de un destructor (consultar capítulo III, título IV).

```
using System.IO;

StreamReader stream = null;
string linea;
try{
        stream = new StreamReader(@"ruta");
        while((linea = stream.ReadLine()) != null){
                Console.WriteLine(linea);
        }
}catch{
        código;
}finally{
        stream.close();
}
```

Título XI: Modularización

Podemos dividir nuestra aplicación en varios ficheros para mayor comodidad a la hora de programar.
Para ello, tenemos dos opciones:
1. Click dcho sobre el proyecto>Agregar>Clase…
2. Proyecto>Agregar clase
Posteriormente, indicar que vamos a crear una nueva clase y el nombre de la clase.
Este nuevo fichero tiene el mismo **namespace**.
Así, tenemos dos ficheros independientes pero en funcionalidad, es como si fueran el mismo.

Capítulo II: POO

Título I: Creación de clases

La creación e instanciación de clases es idéntica a la de Java.
Dentro de nuestro namespece, creamos nuestra clase:

class nombre{
    código;
}

Dentro de código, incluiremos el/los métodos constructores, los campos de clase y los métodos de la clase.
-Convención: En C#, los campos de clase y métodos que sean **public** , deben comenzar por letra mayúscula. Los que no lo sean, minúscula.

Campos de clase

Los campos de clase son variables pertenecientes a una clase y por lo tanto, a todos sus objetos. Los campos de clase se declaran igual que las variables normales, aunque esto se suele hacer al principio (o final) de una clase.
Los campos de clase se suelen inicializar en el método constructor.
A los campos de clase se le pueden indicar modificadores de acceso (**public**, **private, protected**, **default**).

tipovar nombrevar;

Método/s constructor/es

Los métodos constructores o constructores son métodos especiales que se ejecutan automáticamente al inicializar los objetos. La principal misión de estos métodos es la de inicializar los campos de clase.
Estos pueden o no recibir parámetros y presentan sobrecarga de constructores, por lo que podemos encontrar varios constructores en la misma clase (consultar Capítulo I, título V, sobrecarga de métodos).

Un constructor se define con el modificador **public** y el nombre de la clase:
public nombre(parámetros){código;}
En C# , siempre habrá un con
*Si se recibe por parámetros una variable con el mismo nombre que un campo de clase, para hacer referencia al campo de clase se deberá indicar **this.** delante del nombre:

```
int largo;
public Coche(int largo){
        this.largo = largo;
}
```

**Si no se especifica ningún constructor dentro de la clase, es como si para C# existiese un constructor default en el que instancia todas las variables a su valor por defecto.

Métodos de clase

Los métodos de clase son funciones que incluimos dentro de una clase y estas funciones indicarán las funcionalidades de los objetos. Los métodos de clase normalmente manipulan los campos de clase.
A los métodos de clase se le pueden indicar modificadores de acceso (**public**, **private, protected**, **default**).

```
tipoDevuelta nombreMetodo(parámetros){
        código;
}
```

Los métodos pueden ser SETTERS si reciben un único parámetro para dar valor a un campo de clase.
O GETTERS, si devuelven en valor de un campo de clase.
Estos presentan sobrecarga de métodos (consultar Capítulo I, título V, sobrecarga de métodos).
*Si se recibe por parámetros una variable con el mismo nombre que un campo de clase, para hacer referencia al campo de clase se deberá indicar **this.** delante del nombre:

```
int largo;
```

```
public void setLargo(int largo){
        this.largo = largo;
}
```

Instanciar / ejemplarizar clases / objetos

(En otra clase)
Hay diferentes formas de instanciar una clase, la más común es:
NombreClase nombreObjeto = new NombreClase(parámetros);
*Una clase puede o no recibir parámetros (depende del/de los métodos constructores).

Dividir clase en trozos

Para ello, primero debemos pensar qué queremos poner en cada trozo. Una vez decidido, debemos indicar la palabra **partial** delante de **class**, y posteriormente cerrar la llave de este trozo dentro de la clase. Para el segundo trozo, vuelve a declarar el mismo nombre con **partial class**:

```
partial class nombre{
        código;
}
código;
partial class nombre{
        código;
}
```

Dividir una clase en dos es exactamente igual que si tuvieras uno solo.

Título II: static

Los campos de clase no estáticos se clonan en nuevos objetos cada vez que se inicializa un nuevo objeto. Por otra parte, los campos estáticos no se clonan, sino que siempre son los mismos, haciendo que este campo de clase pertenezca a la clase, por lo que ningún objeto puede actuar sobre este campo de clase estático.

Para acceder a campos de clase o métodos estáticos, debemos indicar primero el nombre de la clase seguido de un punto y su nombre.
No podemos acceder a campos o métodos estáticos desde un objeto.
Todas las constantes se consideran estáticas sin necesidad de indicarlo.

Título III: Clases anónimas

Las clases anónimas nos permiten instanciar objetos pertenecientes a clases a la vez que creamos sus clases:
var persona = new {Nombre = "Juan", edad = 20};
Console.WriteLine($"Nombre: {persona.Nombre}, edad: {persona.edad}");
El tipo de los campos de clase se da en tiempo de ejecución.

Si creamos dos objetos con los mismos campos de clase, el mismo tipo y el mismo orden, el compilador identifica que ambos objetos pertenecen a la misma clase.
*Se pueden crear arrays de clases anónimas, aunque estos deben ser arrays implícitos.

Requisitos

- Las clases anónimas sólo pueden contener campos públicos.
- Todos los campos de la clase deben estar inicializados.
- Los campos no pueden ser **static** .
- No se pueden definir métodos.

Título IV: Herencia

C# No permite herencia múltiple.
Al hacer que una clase herede de otra, estamos haciendo que esta clase hijo incorpore, además de sus propios métodos y campos, los métodos y campos de la clase padre (además de una posibilidad de

sobreescribir estos métodos y añadir peculiaridades al método constructor (con la ayuda de **base)**).
En C#, todas las clases heredan de **Object**, a modo de superclase cósmica. Esto hace que todas las clases hereden 4 métodos.

Sintaxis

Para indicar que una clase hereda de otra, debemos indicar dos puntos después del nombre de la clase hijo y posteriormente indicar la clase padre:

```
class Coche : Vehiculo{
        código;
}
```

:base()

Si la clase hijo no tiene método constructor, llama al método constructor de la clase padre (con la instrucción **:base()**).
Si creamos un constructor en la clase hijo, este deja de llamar al constructor de la clase padre.
Es útil en algunos casos crear un constructor en la clase hijo que añada alguna peculiaridad (de la clase) y, posteriormente llamamos al constructor de la clase padre para que haga su trabajo.
*Si la clase padre recibe algún parámetro en su método constructor, la clase hijo **debe** tener un método constructor con la instrucción **: base()** con el que le pase este parámetro, aunque el método constructor esté vacío.
**:base() se especifica después de los parámetros del método constructor.

```
class Vehiculo{
        public boll conducir;
        public Vehiculo(bool conducir){
                this.conducir = conducir;
        }
```

```
}

class Coche:Vehiculo{
        public int ruedas;
        public Coche(int ruedas, bool conducir) : base(conducir){
                this.ruedas = ruedas;
        }
}

static void Main(string[] args){
        Coche c = new Coche(4, true);
        Console.WriteLine($"{c.ruedas}, {c.conducir}"); //4, true
}
```

*Es mala práctica indicar **public** en campos de clase, sustituir por GETTERS.

Principio de sustitución

Se puede sustituir un objeto de un tipo por otro objeto de otro tipo siempre y cuando sea una clase superior la que almacene a una clase inferior.
Al usar el principio de sustitución, los campos y métodos nuevos propios de la clase hijo son ignorados.
Los métodos sobreescritos no se ignoran.
*Lo de la dcha tiene que ser SIEMPRE lo de la izqda.

```
class Mamifero{}
class Caballo : Mamifero{}
Mamifero a = new Caballo();
Caballo a = new Mamifero();    //ESTO ES INCORRECTO
```

Este principio nos permite almacenar objetos diferentes del mismo array siempre y cuando el array sea de la clase padre.

Polimorfismo

Esta es la capacidad de algunas clases de comportarse de diferente forma dependiendo del contexto.

new

Si en la clase hijo desarrollamos un método con el mismo nombre, que recibe los mismo parámetros (mismo tipo) y devuelve el mismo tipo de variable, C# oculta el método de la clase padre y usa el método de la clase hijo (*). Por el contrario, si este método no es del todo idéntico (en lo que se refiere a nombre, parámetros y return), estaremos realizando una sobrecarga de métodos.
*Esto lo podemos dejar así (todo funcionará correctamente), pero si de verdad queremos confirmar lo que estamos haciendo (cambiar el método completamente en la clase hijo), tenemos que escribir la palabra **new** delante del modificador de acceso del método.

virtual y override

Si agregamos esta palabra a algún método (se añade después del modificador de acceso), las clases hijo deberían (no obligatorio) sobreescribir este método de la clase padre, para que su comportamiento cambie en cada una de las clases hijo.
Esto indica que este método es susceptible de ser sobreescribido.
Para cambiar este método en la clase hijo, debemos indicar **override** en la declaración del método (después del modificador de acceso).

Desde estos métodos sobreescritos, podemos usar la palabra **base** seguida de un punto y a continuación hacer mención a el método **virtual** de la clase padre, para poder ahorrarnos algo de trabajo en caso de alguna similitud.

***virtual** + **override** crean una modificación del método anterior mientras que **new** crea un método nuevo totalmente diferente.
****No se puede sobreescribir (**override**) un método que no sea **virtual** (a no ser que sea un método ya sobrescrito que se vuelva a sobreescribir).

Título V: Modificadores de acceso

public

Podemos acceder a este método/campo desde cualquier parte o clase de nuestro programa.
No es recomendable crear campos de clase **public** (a no ser que sean constantes estáticas).

private

Sólo podemos acceder a estos campos/métodos desde dentro de la clase. No podemos acceder desde hijos.
Es recomendable que todos los campos de clase sean **private** .

protected

Sólo podemos acceder a estos campos/métodos desde dentro de la clase o desde dentro de clases que hereden de esta (no desde objetos).

Título VI: Método ToString

El método ToString() es un método perteneciente a la clase **Object.** Por consiguiente, todos los objetos heredan este método que es interesante sobreescribir.
public virtual string ToString()
Este método nos ofrece una breve descripción del objeto. Si en algún caso imprimimos un objeto cuya clase sobreescribe (con **override**) el método ToString, se imprimirá en consola el return del método ToString:
class Cosa{

```
        public override string ToString(){
            return "Esto es una cosa";
        }
}
//En el main:
Console.WriteLine(new Cosa()); //Esto es una cosa
//Si no sobreescribiese el método ToString:
Console.WriteLine(new Cosa()); //Object.Cosa
```

Título VII: Interfaces

Una interfaz es un conjunto de directrices que debe cumplir una clase.
Una clase puede implementar (heredar) de varias interfaces.
En el interior de estas interfaces sólo podemos encontrar declaraciones de métodos vacíos (sin código en su interior).
En las interfaces, no se pueden indicar modificadores de acceso.
Una interfaz obliga a una clase a crear métodos idénticos a los que tenga en su interior.
Una interfaz puede contener el número de métodos que se desee.
No se pueden definir en su interior constructores ni destructores.
No se pueden crear clases dentro de interfaces.

Sintaxis

```
interface nombre{
        código;
}
```

-Convenciones: En C#, los nombres de las interfaces suelen empezar por una "I".
ej:

```
interface IToString{
        string ToString();      //Declarar que todas las clases que la
//implementen se vean obligadas a sobreescribir este método
}
```

*En la interfaz se debe indicar el nombre del método, el tipo de parámetro que devuelve y los parámetros que recibe.

Implementar interfaces

Para indicar que una clase implementa una interfaz, se indica del mismo modo que heredar de una clase.

class nombre : interfaz{}
class nombreHijo : nombrePadre, interfaz{}
class nombreHijo : nombrePadre, interfaz1, interfaz2, ...{}

*Primero se indica la clase (si hereda de una clase) y luego las interfaces.

Ambigüedad

Se puede dar el caso de que implementemos dos interfaces que nos obliguen a implementar métodos idénticos y que estos dos métodos tengan que realizar funciones diferentes.
Para resolver esta ambigüedad, debemos prescindir del modificador de acceso e indicar el nombre de la interfaz delante del método.

```
interface I1{int getNum();}
interface I2{int getNum();}
class Clase : I1, I2{
        int I1.getNum(){
                return 3;
        }
        int I2.getNum(){
                return 16;
        }
}
```

Una vez que realizamos esto, ahora nos es imposible acceder a estos métodos desde fuera de esta clase ya que hemos tenido que prescindir del modificador de acceso, por lo que tenemos que hacer uso del principio de sustitución para acceder a este método: Creamos una

instancia de nuestra clase que almacenamos en un objeto de nuestra interfaz.

I1 nombre = new Clase();
Console.WriteLine(nombre.getNum());

Título VIII: Clases abstractas y métodos abstractas

Las clases abstractas son aquellas clases que contienen al menos un método abstracto. Este método se deberá sobreescribir por todas las clases que hereden de esta clase, como si de una interfaz se tratara.

```
abstract class nombre{
        código;
        modificadorAcceso abstract tipoVar nombre();
}
```

Declaramos la clase abstracta con **abstract** y el método lo declaramos tal y como si fuera una interfaz, aunque debemos indicar el modificador de acceso y **abstract** .

Cuando una clase herede de esta, nos vemos obligados a sobreescribir (con **override**) todos los métodos abstractos que esta contenga.

Las clases abstractas nos permiten implementar métodos ya desarrollados por herencia y además nos obliga a implementar nuevos métodos.

Título IX: Clases y métodos selladas / sealed classes

Las clases selladas son aquellas clases de las que no se va a poder heredar.
Los métodos sellados son aquellos métodos que impiden ser sobreescritos en futuras clases hijo.

```
sealed class nombre{
        código;
}

modAcceso sealed TipoVar nombre(){
        código;
}
```

Capítulo III: Conceptos medios

Título I: Properties (Propiedades)

Las propiedades nos permiten facilitar el acceso a campos de clase con modificador **private**, creando unos métodos SETTER y GETTER para un campo de clase y poder acceder a él como si fuera **public**.

Para ello, debemos crear un método que evalúe la condición del SETTER (si es que el SETTER reúne alguna condición a la hora de colocar el valor) y posteriormente creamos la propertie.
La propertie se crea igual que un campo de clase (el nombre de esta propertie se suele indicar con el mismo nombre que la variable que va a manejar y todo en mayúsculas por convención)
Posteriormente, indicamos cómo será el GETTER y el SETTER.

```
private TipoVar nombreVar;          //Campo de clase

private TipoVar evaluaCondicion(TipoVar valor){
        if (condición) return x
        else return y
}                   //Método que evalúa la condición del SETTER

public TipoVar NOMBREVAR{
        get {return this.nombreVar; }          //Propertie
        set {this.nombreVar = evaluaCondicion(value); }
}
```

*No hace falta indicar GETTER y SETTER, se puede crear una propertie de sólo lectura o de sólo escritura (para ello, prescindir de la línea que no haga falta comentandola o eliminandola).
ej más concreto:

```
private double salario;
private double evaluaSalario(double salario){
        if (salario<=100) return 100;
        else return salario
}

public double SALARIO{ç
        get {return this.salario; }
        set {this.salario = evaluaSalario(value); }
}
```

Uso

Una vez que tenemos creada la propertie, podemos usar en otra clase la propertie como si fuera un campo de clase **public** :
(la clase se llama Empleado):

```
Empleado a = new Empleado();
a.SALARIO = 500;
Console.WriteLine(a.SALARIO);     //500
a.SALARIO = 10;
Console.WriteLine(a.SALARIO);     //100
```

(debido a la propiedad del setter)

Expression-bodied / Operador lambda

Para simplificar el código anterior:

```
public double SALARIO{
        get => this.salario;
        set => evaluaSalario(value);
}
```

Título II: Structs (Estructuras)

-Teoría: Existen dos tipos de memoria (Stack y Heap) la primera más rápida que la segunda. La información de la primera a veces se pierde

(variables locales) mientras que la segunda se mantiene durante todo el programa.
A la hora de crear una variable primitiva, esta se almacena en la memoria Stack.
Al crear un objeto de una clase, este se almacena en la memoria Heap a la que se hace una referencia desde la memoria Stack.
Cuando se crea un struct, este se almacena en la memoria Stack como si fuera una variable de tipo primitivo.

Propiedades y creación

Los campos de clase de un struct no pueden variar.
El Struct se suele usar cuando se necesita trabajar con una elevada cantidad de datos en memoria, cuando las instancias no deben cambiar (inmutables), se usa por razones de rendimiento.

Para crear una estructura, sustituimos la palabra **class** por **struct** .

Diferencias con respecto a clases:
-No permite la declaración de un constructor por defecto.
-El compilador no inicia los campos, estos se inician en el constructor.
-No puede haber sobrecarga de constructores.
-No pueden heredar de clases pero sí implementar interfaces.
-Son selladas (sealed), por lo que no se puede heredar de ellas.

Título III: Enum / Tipos enumerados

Los tipos enumerados son un conjunto de constantes con nombre (a modo de array). Sirven para representar y manejar valores constantes.
Los Enum se suelen crear dentro del namespace para que estos se puedan usar en todas las clases.
C# da valores numéricos por defecto a cada valor (0, 1, 2, ...).
Podemos convertir un dato de un Enum a string (adopta el nombre del dato) o a numérico (adopta el valor numérico del dato).

enum nombre {valor1, valor2, ...};

enum Estaciones {Primavera, Verano, Otoño, Invierno};
(Sin comillas, ya que no estamos almacenando Strings, sino valores constantes).

Cambiar valor de dato

enum Tamanos {pequeno=10, mediano=30, grande=50, extra=90};

Uso

Para usar los tipos enumerados, tenemos que crear una especie de "objetos" pertenecientes a este tipo enumerado:

```
enum Estaciones {Primavera, Verano, Otoño, Invierno};
Estaciones calor = Estaciones.Verano;
String str = calor.ToString();
Console.WriteLine(str);    //Verano
```

*Podemos pasar como parámetro un Enum:

```
public void Estacion(Estaciones a){
        Console.WriteLine($"Es {a.ToString()}");
}
Estacion(Estaciones.Verano);
```

Castings

```
enum Tamanos {pequeno=10, mediano=30, grande=50, extra=90};
Tamanos a = Tamanos.pequeno;
string str = a.ToString();          //"pequeno"
int num = (int) a;                  //10
double num2 = (double ) a;          //10.0
```

Valores nulos

Si queremos “inicializar” un “objeto” de un tipo enumerado con **null**, debemos seguir la sintaxis:
NombreEnum? nombre = null;
Estaciones? calor = null;

Título IV: Destructores

Hay momentos después de crear varios objetos en los que la memoria Heap se satura. Para solucionar esto, el Garbage collector (recolector de basura) va mirando los objetos que no se van a usar más y los va eliminando de forma automática. Un método destructor se encarga de ejecutar un código cuando se destruye un recurso de la memoria.

Los destructores nos pueden ser de gran utilidad para cerrar conexiones con BBDD, cerrar streams o eliminar objetos de colecciones.
A pesar de todo esto, no se recomienda del todo el uso de destructores.

Creación

Un método destructor se declara igual que un constructor, sustituyendo el modificador de acceso **public** por un “~”:

```
~nombre(){
        código;
}
```

ej:

```
class AccesoFicheros{                   //Clase
        StreamReader stream
        public AccesoFicheros(){  //Método constructor
                stream = new StreamReader(@”C/”);
                código;
```

```
    }

    ~AccesoFicheros(){          //Método destructor
        stream.close();
    }
}
```

Restricciones

-Los destructores sólo se usan en clases.
-Cada clase sólo puede tener un destructor como máximo.
-Los destructores no se heredan ni sobrecargan.
-Los destructores no se llaman, son invocados automáticamente.
-Los destructores no tienen ni modificadores de acceso ni parámetros.

Título V: Programación genérica

La programación genérica nos puede ser de gran ayuda ya que podemos crear clases comodín que sean capaces de manejar cualquier tipo de objeto, por lo que nos permite la reutilización de código.
Para ello, tenemos que especificar entre "<>" el tipo de clase que vamos a usar.
La programación genérica en C# funciona igual que en Java.

Clases

Para crear una clase genérica, primero debemos indicar después del nombre de la clase entre paréntesis angulares (<>) un genérico (una letra), que por convención suele ser una T:
class nombre<T>{}

Métodos y estructuras genéricas

Posteriormente, indicamos este genérico (T) donde queramos indicar cualquier tipo de dato. Por ejemplo, si queremos que un método reciba un genérico:

public TipoVar nombre(T nombre2){}

Si queremos que un método devuelva un genérico:
public T nombre(){}

También podemos crear variables, constantes y arrays (todo lo que queramos) genéricos.

Uso

Para usar clases genéricas, tenemos que indicar en la especificación de objeto qué clase vamos a usar en esta ocasión:
NombreClase<TipoClase> nombre = new NombreClase<TipoClase>(parámetros);

Restricciones

Podemos restringir el uso de clases en nuestras clases genéricas para que nuestra clase genérica acepte sólo un número limitado de clases.

Para restringir el uso a un cierto número de clases tenemos que hacer que todas las clases implementen una cierta interfaz para luego indicar esta como "filtro" en la clase genérica.
El filtro lo indicamos con la palabra **where** seguido del genérico que implementa la interfaz en cuestión.

class nombreClase<T> where T : NombreInterfaz

De esta forma, ya sólo podemos crear objetos de esta clase genérica en los que especifiquemos como genérico clases que implementan esta interfaz.
Si especificamos en el genérico una clase que no implementa esta interfaz, dará error.

Título VI: Colecciones

Las colecciones son clases que pertenecen al namespace **System.Collection.Generic**
Estas clases nos permiten almacenar elementos.
Las colecciones, a diferencia de los arrays, nos permiten ordenar elementos, añadir más elementos (no hace falta especificar un límite ya que son dinámicas), eliminar elementos, buscar elementos…

Podemos acceder a cualquier posición de las colecciones como si fueran un array (por lo que podemos usar bucles for each):

Tipos de colecciones

Colección	Descripción
List<T>	Parecidos a los array pero con métodos adicionales.
Queue<T>	Un elemento entra y otro sale, el primero es el primero en salir.
Stack<T>	Parecido a las queues, pero primero en entrar y último en salir.
LinkedList<T>	Comportamiento como queue o stack pero con acceso aleatorio.
HashSet<T>	Listas de valores sin ordenar.
Dictionary<Tkey, Tvalue>	Almacena elementos en clave-valor.
SortedList<Tkey, Tvalue>	Igual que dictionary pero ordenados.

List

Cuando eliminamos un elemento en una posición, todos los elementos siguientes se desplazan un hueco hacia arriba (por lo que no es recomendable de usar si vamos a eliminar elementos continuamente).

```
List <int> numeros = new List<int>();
numeros.Add(182);
numeros.Add(464);
Console.WriteLine(numeros[1]);      //464
```

*Existe otra forma de crear listas:
List <int> numeros = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9};

Add(T x)

Agrega el objeto x al final de la lista.

AddRange(T[] x)

Agrega los elementos del array x a la lista.

Clear()

Quita todos los elementos de la lista.

Exists(Predicate<T> x)

Devuelve true si algún elemento del **List** devuelve true con el método al que hace referencia el predicado **x** (Consultar título VIII).

FindAll(Predicate<T> x)

Devuelve un **List** con todos los parámetros de la lista sobre la que se aplica que cumplen la condición de la función especificada por el delegado predicado **x** (Consultar título VIII).

IndexOf(T x)

Busca x en la lista y devuelve el índice en el que se encuentra.

RemoveAt(int x)

Elimina el elemento que se encuentra en la posición x.

Count

Campo de clase que nos devuelve el número de elementos que tenemos almacenados en la lista.

LinkedList

Similar a List, salvo que permite más eficiencia a la hora de borrar ya que internamente, cada elemento está unido a otro por nodos (links), lo que hace que la reestructuración de links sea más eficiente que la reestructuración de elementos en sí.

LinkedList <int> numeros = new LinkedList<int>();

Cada elemento de la LinkedList es un nodo, los cuales podemos manejar con la clase **LinkedListNode** (consultar API).

AddFirst(T x)

Agrega x en la primera posición.

AddLast(T x)

Agrega x en la última posición.

Clear()

Quita todos los elementos de la lista.

Remove(T a)

Quita la primera aparición de a en la LinkedList.

RemoveFirst()

Elimina el primer elemento.

RemoveLast()

Elimina el último elemento.

Count

Campo de clase que nos devuelve el número de elementos que tenemos almacenados en la lista.

Queue

Cola de elementos. Se eliminan en el orden en el que entraron: FIFO (First In First Out).

Queue <int> cola = new Queue<int>();
cola.Enqueue(16);
cola.Enqueue(28);

Enqueue(T a)

Añade el elemento a a la cola.

Dequeue()

Quita el objeto del comienzo del queue (y mueve los demás un paso adelante).

Clear()

Quita todos los elementos de la colección.

ToArray()

Convierte la colección en un array (devuelve un array).

Stack

Pila de elementos (a modo de apilarlos). Se eliminan desde el último: LIFO (Last In First Out).

Stack <int> pila = new Stack<int>();
pila.Enqueue(16);
pila.Enqueue(28);

Push(T a)

Introduce a en la última posición.

Pop()

Elimina el último elemento.

Clear()

Quita todos los elementos de la colección.

Dictionary

Colección que se va llenando de elementos a modo de clave-valor.
Consume más recursos que otras colecciones.
Para acceder a un valor, indicamos entre los corchetes del "array" su key.
*Si se recorre con un bucle foreach, devuelve un objeto clave-valor.
Para recorrerlo de forma correcta:

```
foreach(KeyValuePair<string, string> kvp in diccionario){
        Console.WriteLine("{0} : {1}", kvp.key, kvp.value);
}
```

**Presenta dos formas para agregar elementos.:

```
Dictionary<string, int> edades = new Dictionary<string, int>();
edades.Add("Juan", 16);
edades["María"] = 46;
```

Add(Tkey a, Tvalue b)

Añade al diccionario la key a con el valor b.

Clear()

Quita todos los elementos de la colección.

Tïtulo VII: Delegados

Los delegados son funciones que delegan tareas a otras funciones.
Un delegado es una referencia a un método.
El método al que delega se puede encontrar en otra clase.
Un mismo delegado puede delegar a diferentes métodos de diferentes clases.

delegate TipoVar nombre(parámetros);

*El delegado tiene que ser idéntico (salvo en el nombre y **delegate**) al/a los método/s que delega (los parámetros deben tener el mismo orden y el mismo tipo pero pueden tener distinto orden).

Uso

Para ello, debemos crear una "instancia" del delegado, para luego usarlo:

```
class Main{
	static void Main(string[] args){
		Delegado dele = new Delegado(O1.Saludo);
		dele();
		Delegado dele2 = new Delegado(O1.Despedida);
		dele();
	}
	delegate void Delegado();
}
class O1{
	public string Saludo() => "Hola";
}
class O2{
	public string Despedida() => "Adiós";
}
```

Paso de parámetros

```
class Main{
	static void Main(string[] args){
		Delegado dele = new Delegado(O1.Imprimir);
		dele("Esto es un delegado");
	}
	delegate void Delegado(string msg);
}
```

```
class O1{
	public void Imprimir(string a){
		Console.WriteLine(a);
	}
}
```

Título VIII: Delegados predicados

(Consultar título VII).
Los delegados predicados son aquellos que sólo devuelven datos **bool**. Sirven para filtrar listas (**List**) de valores, comprobando si una condición es cierta para un valor dado.

Predicate <Clase> nombre = new Predicate <Clase>(funcion);

Podemos usar delegados predicados para, por ejemplo, determinar los números pares de una lista:

```
static bool DamePares(int x){
	if (x%2==0) return true;
	else return false;
}

List <int> lista = new List<int>();
lista.AddRange(new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
Predicate <int> delegado = new Predicate<int>(DamePares);
List <int> pares = lista.FindAll(delegado);

foreach(int e in pares){
	Console.WriteLine(e);	//2\n4\n6\n8\n10
}
```

Título IX: Expresiones lambda

Las expresiones lambda son funciones anónimas que se utilizan para realizar tareas sencillas. Se pueden utilizar para crear delegados sencillos.
En las funciones lambda, no es necesario especificar el tipo de parámetros que esta recibe (lo especifica el compilador).
*Si se reciben más de un parámetro, especificar entre paréntesis.
**Si la función lambda tiene más de una línea, debemos especificar corchetes después de "=>" y especificar ";" al final de cada línea.

parámetros => código;

ej:

```
public delegate int Cubo(int num);

Cubo delegado = new Cubo(num => num*num*num);
Console.WriteLine(delegado(8));          //512
```

(*)

```
public delegate int Dele(int n1, int n2);

Dele delegado = new Dele((n1, n2) => n1+n2);
Console.WriteLine(delegado(10, 8));      //18
```

Utilidad real

ej donde se compara si dos strings son iguales con delegados y lambdas:

```
public delegate bool Delegado(string str1, string str2);

Delegado dele = ((str1, str2) => String.Compare(str1, str2));

Console.WriteLine(dele("Juan", "Juan"));      //true
Console.WriteLine(dele("Juan", "Juana"));     //false
```

Título X: Expresiones regulares / Regex

Estas son secuencias de caracteres que funcionan como un patrón de búsqueda (para buscar en cadenas de caracteres letras, números...).
Las expresiones regulares nos permiten reducir el código.

Mayoría de expresiones regulares:
https://docs.microsoft.com/es-es/dotnet/standard/base-types/regular-expression-language-quick-reference
Mayoría de cuantificadores (usar con expresiones regulares):
https://docs.microsoft.com/es-es/dotnet/standard/base-types/quantifiers-in-regular-expressions
Generador de expresiones regulares online:
Buscarlo.
Vídeos (70, 714)

Uso

1. Para usar expresiones regulares, tenemos que crear un objeto **Regex** (del paquete **System.Text.RegularExpressions**) al que le pasamos como parámetro al método constructor la expresión regular que queramos usar (en formato string) (consultar link superior).
2. Ejecutamos el método **Matches(string frase)** sobre nuestro objeto **Regex** (indicando la frase en la que queremos buscar), el cual nos devuelve una colección **MatchCollection**, la cual deberíamos recorrer para ver los resultados de nuestra búsqueda con la expresión regular indicada.

```
Regex miregex = new Regex("[J]");
MatchCollection colec = miregex.Matches("Juan");

if (colec.Count > 0) Console.WriteLine("Hay alguna J");
else Console.WriteLine("No hay ninguna J");
```

*Si vamos a introducir algún carácter raro o de escape (como ‘/’), especificar “@” delante del string.
**Se pueden mezclar varias expresiones regulares concatenándolas.

Condiciones

Podemos crear condiciones dentro de expresiones regulares, como el OR lógico (|).

Capítulo IV: Interfaces gráficas

WPF es una API perteneciente a .net para crear interfaces de usuario gráficas bajo windows.
Para manejarlas, contamos con una interfaz en la que cambiar los componentes.
Trabajaremos con un documento xaml .

Creación de proyecto

Crear un nuevo proyecto: Aplicación de WPF (.NET Framework).
Desplegar herramientas: Ver>Cuadro de herramientas.

Dentro del código xaml podemos cambiar ciertas propiedades, como el título en **Title**, el ancho o el largo (**Heigth**, **Width**).

Modos de trabajo

-Podemos trabajar con la vista diseño con la ayuda del panel de herramientas, y el de propiedades.
-Podemos trabajar desde el código xaml.
-Podemos trabajar con código C#.

En C# , el método **MainWindow()** actúa como método **Main** cuando se inicia el programa de interfaz gráfica.

Título I: Elementos básicos

(vídeos 72-76).
Toda ventana se crea heredando de **Window** y ejecutando **InitializaComponent();**

Propiedades

Todos los objetos sobre los que se pueden aplicar las propiedades se llamarán **obj**, aunque puede ser cualquier objeto de interfaz gráfica.

obj.Width = 300;
obj.Height = 300;
obj.Background = Brushes.Color;
*Para añadir texto, crear wrap dentro de **obj** y añadir un **TextBlock**.
obj.Foreground = Brushes.Color;
obj.HorizontalAlignment = "Left" //(Left, Right, Center, Stretch)
obj.VerticalAlignment = "Center"
obj.Click = "funcion"

Ventana emergente

MessageBox.Show(string texto);

Título II: Láminas

Grid

Un grid es una especie de lámina a modo de contenedor que podemos usar para añadir elementos a nuestra ventana. Para ello, creamos un objeto de tipo **Grid** con el constructor default y añadimos el grid al marco con **this.Content = objetoGrid** .
Posteriormente, podemos ir creando objetos y añadirlos al grid con la instrucción **objetoGrid.Children.Add(objeto);**

```
Grid lamina = new Grid();
this.Content = lamina;
Button boton = new Button();
lamina.Children.Add(boton);
```

Podemos dividir un grid en filas y columnas:
obj.GridColumnDefinitions = ColumnDefinition.Width = 300
obj.GridRowDefinitions = RowDefinition.Height = 300
Una instrucción = una fila/columna.
Indicar "" para que ocupe "lo que queda".

**Indicar “auto” para que ocupe lo justo.
Para agregar un objeto en una columna: **obj.Grid.Column = “0”**
Para agregar un objeto en una fila: **obj.Row.Column = “1”**
Para que ocupe varias columnas: **obj.ColumnSpan = “4”;**
Para que ocupe varias filas: **obj.RowSpan = “2”;**

WrapPanel

Podemos crear **WrapPanel** para añadir componentes dentro de un componente, como **TextBlock** .
Para ello, creamos un **WrapPanel** con el constructor default y le añadimos tantos elementos como queramos con
objetoWrap.Children.Add(subobjeto);
Una vez finalizado, agregamos el **WrapPanel** al objeto con
objeto.Content = objetoWrap;

StackPanel

Un **StackPanel** es una lámina que se añade sobre el marco para añadir componentes. Este tiene incorporado por defecto un layout a modo de pila de elementos, uno se coloca debajo de otro y este intenta ocupar todo el ancho de la fila.
Se puede añadir una propiedad **Margin** para dejar x píxeles de espacio por arriba, abajo, izqda y dcha, aunque le podemos dar un ancho y un largo con **Width** y **Heigth**.
A un **StackPanel** se le puede dar nombre con la propiedad **Name** y un evento con **ButtonBase.Click**

ListBox

Listas donde podemos almacenar elementos como strings.

Título III: Eventos

Para añadir eventos, le damos la propiedad **Click** al objeto indicándole una función que luego programaremos como:

private void nombre(object sender, RoutedEventArgs e){}

Tipos de eventos

-Directos: No propaga el evento.
-Burbuja: El evento de un elemento se propaga al elemento principal (la ventana).
-Tunelado: El evento de un elemento se propaga al último elemento (la ventana).

Evento Burbuja

El evento se propaga al elemento principal (la ventana), por lo que si un botón y una lámina tienen eventos y hacemos click en el botón, primero se ejecutará el evento del botón y luego el evento de la lámina.
Propiedades: **Click** (botón), **ButtonBase.Click** (lámina)

Evento Tunelado

El evento se propaga del elemento superior al más inferior, por lo que si un botón y una lámina tienen eventos y hacemos click en el botón, primero se ejecutará el evento de la lámina y luego el del botón.
Propiedades: **Click** (botón), **PreviewMouseLeftButtonDown** (lámi)

Título IV: Dependency properties

Todos los objetos son controles, y cada control tiene una serie de propiedades, las cuales necesitan del sistema de propiedades de WPF (por tema de compatibilidades o el contexto), por lo que se les llama dependency properties, ya que necesitan de este sistema. Estas propiedades hacen cosas bastante vistosas.

Animar control (pasar botón por encima)

Para estas properties, creamos una etiqueta **Button.Style** en xaml dentro de un control y dentro de esta, indicamos una etiqueta **Style**

con la propiedad **TargetType** = control en cuestión. Dentro de esta nueva etiqueta, creamos otra **Style.Triggers**
Creamos otra etiqueta **Trigger** con el atributo **Porperty=”IsMouseOver”** y **Value=”true”**
Dentro de esta etiqueta, podemos hacer lo que queramos (se ejecutará al pasar el ratón por encima del control), como cambiar la fuente del control:

```
<Button Height=”150” Width=”300” Content=”Dale”>
    <Button.Style>
        <Style TargetType=”Button”>
            <Style.Triggers>
                <Trigger Property=”IsMouseOver”
Value=”true”>
                    <Setter Property=”FontSize”
                Value=”25”/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
</Button>
```

Crear dependencyproperty propia

En cualquier punto de una clase, creamos una property especificando el get y el set, los cuales dejaremos en blanco temporalmente.
Posteriormente, registramos nuestra dependencyproperty, para esto, especificamos:
public static readonly DependencyProperty nombre = DependencyProperty.Register(“property”, typeof(dato), typeof(clase), new PropertyMetadata(0));

Posteriormente, especificamos en el get que nos devuelva un casting a entero de **GetValue(nombre)**

Y en el set, especificamos **SetValue(nombre, value);**

ej:

```
public partial class MainWindow : Window
{
        public int MiProperty
        {
                get { return (int)GetValue(dp); }
                set { SetValue(dp, value) }
        }

        public static readonly DependencyProperty dp =
DependencyProperty.Register("MiProperty", typeof(int),
typeof(MainWindow), new ProperyMetadata(0));

}
```

Data binding

El data binding es la capacidad de un control WPF de recibir y enviar información mediante un binding (puente), conectándose con BBDD, objetos u otros controles WPF.

Existen diferentes tipos: OneWay, OneWayToSource, twoWays y oneTime (Fuente-Destino, Destino-Fuente, bidireccional y Fuente-Destino una única vez).

El databinding lo podemos usar para, por ejemplo, vincular un Slider con un cuadro de texto, el cual mostrará los valores del Slider.
Para crear este databinding, abrimos una llave dentro del Text del TextBox donde escribimos: {**Binding ElementName=NombreSlider, Path=Value, Mode=OneWay}**

INotifyPropertyChanged

Podemos crear un INotifyPropertyChanged cuando queramos ejecutar un evento cuando alguna propiedad de un objeto cambie, identificando la propiedad que cambia.

Para usar esto, debemos crear una clase que implemente esta interfaz (**INotifyPropertyChanged**) posteriormente, creamos dentro de la clase este código: **public event PropertyChangedEventHandler PropertyChanged;**
Posteriormente, creamos el método oyente a cualquier acción que se haga sobre el control:

```
private void OnPropertyChanged(string property){
        if (PropertyChanged!=null) PropertyChanged(this,new
PropertyChangedEventArgs(property));
        código;
}
```

*Indicando en **código** el código que queremos que se ejecute cuando alguna propiedad cambie.
Finalmente, creamos los properties para los campos de clase que se verán asociados en los controles, especificando en el setter el evento que hemos creado anteriormente y pasándole un string a modo de identificador.

Para usar la clase que acabamos de crear, después de inicializar la ventana dentro del MainWindow, creamos una instancia de la clase. Posteriormente, ejecutamos: **this.DataContext = nombreobjeto;**

Curso terminado aquí.
(Siguiente: 81)