Revisión de C#

2021-2022

Escuela Politécnica Superior

Universidad de Alicante

OFICIAL Máster Web

Contenidos

- Introducción
- Tipos de datos
- Variables
- Parámetros
- Operadores
- Sentencias de control
- Clases
- Interfaces
- Arrays
- Colecciones
- Excepciones
- Async, await
- Threads

Introducción

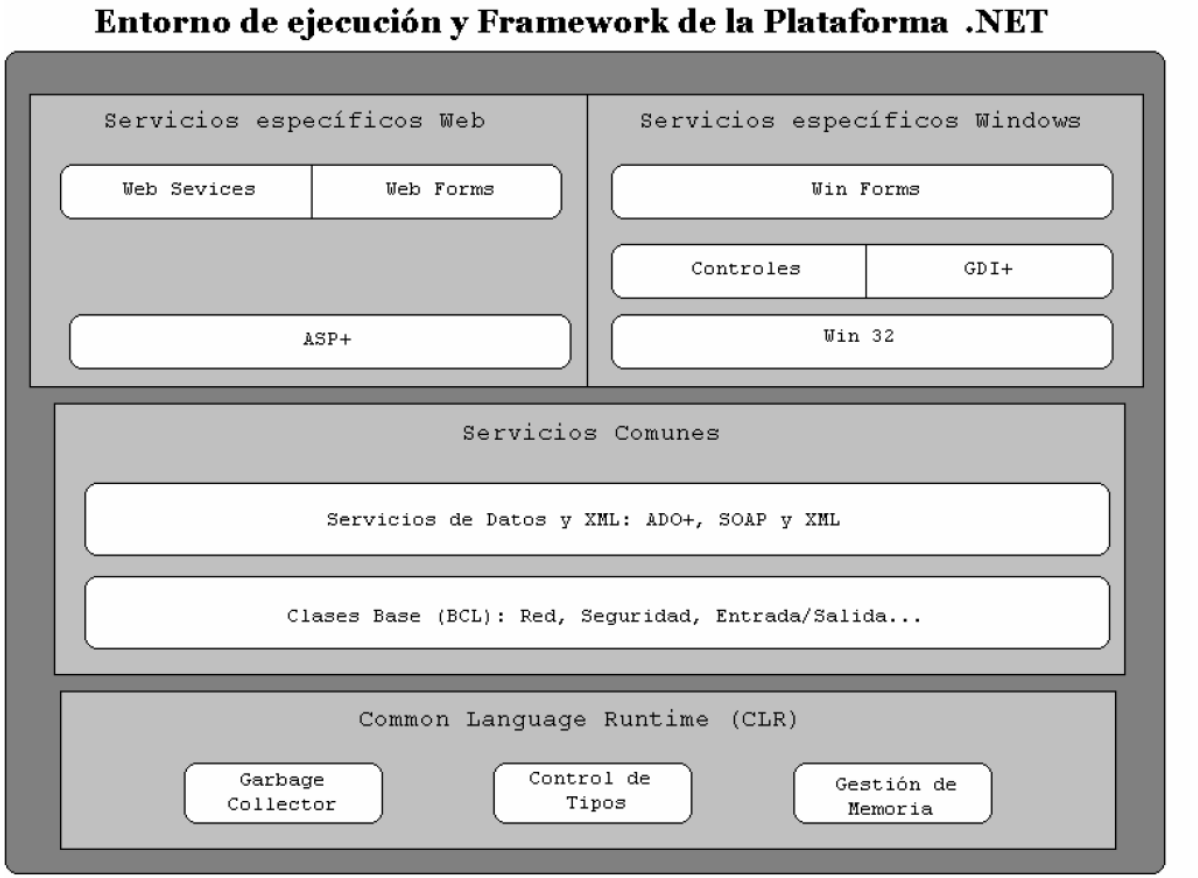
- Lenguaje Orientado a Objetos
- Diseñado para el CLR (Common Language Runtime) de la plataforma .NET
- Se compila en un lenguaje intermedio llamado IL (Intermediate Language):
  - No es binario dependiente de la plataforma.
  - Se ejecuta sobre una máquina virtual que provee el CLR

Introducción. C# VS C/C++

- Gestión de memoria automática
- No utiliza punteros (código inseguro)
- Cambios en algunos operadores
- Ciertas palabras clave se utilizan de modo distinto (p.e. extern y static)
- El manejo de errores se hace mediante excepciones
- No se utilizan ficheros de cabecera .h
- Diferente mecanismo de herencia (no hay herencia múltiple).
- Diferencias en algunas sentencias

Plataforma .NET

- Plataforma para el desarrollo de aplicaciones en general (no sólo de escritorio para Windows)



Tipos de datos

- Todos los tipos de datos derivan de la clase base `System.Object`
- Dos categorías de tipos
  - Tipos Valor:
    - Tipos simples: `char`, `int`, `float`,…
    - Enumeraciones: tipo `enum`
    - Estructuras: tipo `struct`
  - Tipos Referencia:
    - Clases: `class` (incluyendo `string` y `object`)
    - Interfaces: `interface`
    - Tipo `delegate`
    - Tipo `array`
- Punteros: sólo disponibles en el 'código inseguro'

Tipos de datos. Tipos valor

- Categorías principales:
  - Tipo simple
  - Tipo estructura
  - Tipo enumeración.

- Una variable de tipo valor no puede ser **`null`**
- Heredan de la clase `System.Object`
  - p.e. es correcto:

```
7.ToString()
int i = int.MaxValue
```

Tipos de datos. Tipos valor

- Tipos simples:
  - Tipo entero
    - Con signo: `sbyte`, `short`, `int`, `long`
    - Sin signo: `byte`, `ushort`, `uint`, `ulong`
  - Tipo coma flotante: `float` y `double`
  - Tipo `char`
  - Tipo `bool`
  - Tipo `decimal`

Tipos de datos. Tipos valor

- Tipos `enum`:
  - Permite utilizar un grupo de constantes a través de un nombre asociado
  - Las constantes pueden ser del tipo: `byte`, `short`, `int` o `long`
  - Declaración:

```
enum Colores{ Rojo = 2, Verde = 3, Azul = 4}
```

  - Si no se especifica valor, lo asigna el compilador:

```
enum Colores{ Rojo=20, Verde=30, Azul} (Azul=31)
```

  - Uso: `int x = (int)Colores.Verde;`

Tipos de datos. Tipos valor

- Tipos `struct`:
  - Similar a class, pero de tipo valor
    - Puede contener constantes, métodos…
  - Interesante para crear 'objetos
  - ligeros' Ejemplo:

```
struct Punto{ public int x,y;
                public Punto(int x, int y){
                    this.x = x;
                    this.y = y;
                    }
              }
punto p = new Punto( 7, 9);
int coordenadaX = p.x;
int coordenadaY = p.y;
```

Tipos referencia

- Admite el valor `null` → no apunta a ningún objeto.
- Implican la creación de:
  - Una variable de referencia en la pila
  - Un objeto con los datos a los que apunta
- Tipos referencia:
  - Tipo clase: `class`, `object` y `string`
  - Tipo `interface`
  - Tipo `array`
  - Tipo `delegate`

Tipos referencia

- `class`
  - Estructura de datos que puede contener:
    - Miembros de datos
    - Funciones miembro
    - Otros tipos anidados
  - Soporta herencia simple.

- `object`
  - Alias de `System.Object`
  - Admite cualquier valor: `object var = 20;`

Tipos referencia

- `string`
  - Se permite crear y utilizar directamente a partir de literales: `string unacadena = "Hola";`
  - Algunas características:
    - Si puede concatenar con '+'
    - Operador `[]` para acceder a los caracteres.
    - Operadores `==` y `!=` comparan **valores**
  - Algunos métodos interesantes:
    - `int IndexOf(char unCaracter)`
    - `int Length`
    - `string Replace(string viejo, string nuevo)`

Tipos referencia

- `interface`
  - Similar a una clase o estructura pero **todos sus miembros son abstractos** → no están definidos, no tienen código asociado.
  - Serán implementados por una clase que herede de la interface.
  - Son los únicos que admiten herencia múltiple.

Tipos referencia

- **interface**

```
interface IVolar{  void Volar();
    void Despegar();  void
    Aterrizar();
    }

class Avion:IVolar{  public void
      Volar(){
        // Código que implementa el método
        }
    public void Despegar{
        // Código que implementa el método
        }
    public void Aterrizar{
        // Código que implementa el método
        }
    }
```

Tipos referencia

- **`array`**
  - Conjunto ordenado de datos a los que se accede a través de índices.
  - Todas las variables son del mismo tipo.
  - Declaración:

```
int[]  unArray;
int[   unArray = new int[4];
]
       unArray = {2,5,12,56}
int[           ;
```

  - Propiedad `Length`
  - Más de una dimensión:

```
string [3,2] matrizString;

int [,] otroArray = {3,5,7},{6,9,1},{4,5,7}};
```

Tipos referencia

- `delegate`
  - Similar a un puntero a función de C++
  - Estructura de datos que referencia a:
    - Un método estático
    - Un método de instancia de un objeto

Tipos referencia

- delegate

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}


// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");

public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

```
MethodWithCallback(1, 2, handler);
```

```
The number is: 3
```

Variables

- Zonas de memoria donde se almacenan datos
- Definición:

```
Tipo NombreVariable;
```

- Si es un tipo referencia:

```
TipoRef nombreRef; // crea la ref
nombreRef = new TipoRef(parámetros);
```

Parámetros

- Parámetros valor.
  - Una copia del original.

```
public static long Añadir(long i, long j) {...}

ClaseSumadora.Añadir(20,50);
```

- Parámetros referencia (`ref`).

```
static void F(ref int x, ref int y){...}

F(ref i, ref j);
```

- Parámetros de salida (`out`).
  - Referencia a una instancia ya existente **No tiene por**
  - **qué estar inicializada**

```
static void Salida(out int x){ x = 5 }

salida(out i);
```

Operadores

- Tipos:
  - Unarios: un solo operando
  - Binarios: dos operandos
  - Ternarios: tres operandos (sólo existe `?:`)

- Sobrecarga:
  - Implementación diferente a la predefinida.
  - Un operador sobrecargado tiene preferencia sobre el original.
  - Si se sobrecarga `+` → es también una sobrecarga de `+=`

Operadores

- Operadores de información de tipo:
  - **`typeof`**:
    - Para obtener el objeto System.Type correspondiente a un tipo: `Type typeof(tipo)`
  - **`is`**:
    - Para chequear, en tiempo de ejecución, si el tipo de una expresión es compatible con un tipo de objeto dado.

      ```
      expresion is tipo
      ```

    - Devuelve True/False
  - **`as`**:
    - Para realizar conversiones entre dos tipos compatibles

      ```
      expresion as tipo
      ```

    - Si no tiene éxito, devuelve `null`

Operadores

- Sobrecarga de operadores
  - El método debe ser público y `static`. Se
  - debe indicar qué tipo devuelve.
  - Se nombra con la palabra `operator` seguida del operador que se va a sobrecargar.
  - Entre paréntesis, se indica el o los operandos, especificando sus tipos, como argumentos.

```
public static tipoRetorno operator+ (parámetros)
```

Instrucciones de control

- Instrucciones de control de flujo:
  - **if**:

```
if( expresión_booleana ){ ... }
else { ... }
```

    - La condición sólo puede ser de tipo booleana
  - **switch**:

```
switch( opcion ){
   case valor1:  // bloque1
                 break;
   case valor2:  // bloque2
                 break;
   ...
   default: // bloque default
}
```

    - No puede omitirse el break de los case
    - Permite evaluar: int, string, char y enum

Instrucciones de control

- Instrucciones de control de flujo:
  - **while**:

```
while( expresión_booleana ){ ... }
```

  - **do-while**:

```
do { ... } while( expresión_booleana );
```

  - **for**:

```
for( inicializador; condicion; iterador) {
   // Código
   }
```

  - **foreach**:

```
foreach( tipo identificador in expresion ){
   // Código
   }
```

Instrucciones de control

- Sentencias de salto:
  - **`break;`**
    - Permite terminar una secuencia de sentencias (salir del bucle.
  - **`continue;`**
    - Para dejar de procesar el resto del código de un bucle, para esa iteración, pero sin salir de él.

```
for( int i=1; i<=10; i++) {
    if( i<9 )    continue;
    Console.WriteLine(i);
    }
```

Instrucciones de control

- Sentencias de salto:
  - **goto**
    - Realiza un salto a la sentencia etiquetada:
      - `goto` etiqueta;
      - `goto case` expresión_constante;
      - `goto default;`

```
goto Bilbao;
// código inaccesible
Bilbao: // otras sentencias
```

  - **return**
    - Para volver de manera explícita desde un método.
    - Puede devolver un valor (de cualquier tipo) al punto de llamada del método.

```
return expresion;
```

Clases

- La programación Orientada a Objetos permite unir elementos simples en objetos formados por:
  - Datos: campos o variables miembro
  - Funciones: métodos

- La definición de los objetos se realiza mediante la clase.

- Los objetos son las variables concretas de una determinada clase → instancias.

Clases

- Ejemplo:

```
public class Rectangulo{
  int x;
  int y;
  private double ancho;
  private double alto;

  public Rectangulo(double w, double h){
    x = 0;
    y = 0;
    ancho = w;
    alto = h;
  }
  public CalcularArea(){
    return ancho * alto;
  }
}
```

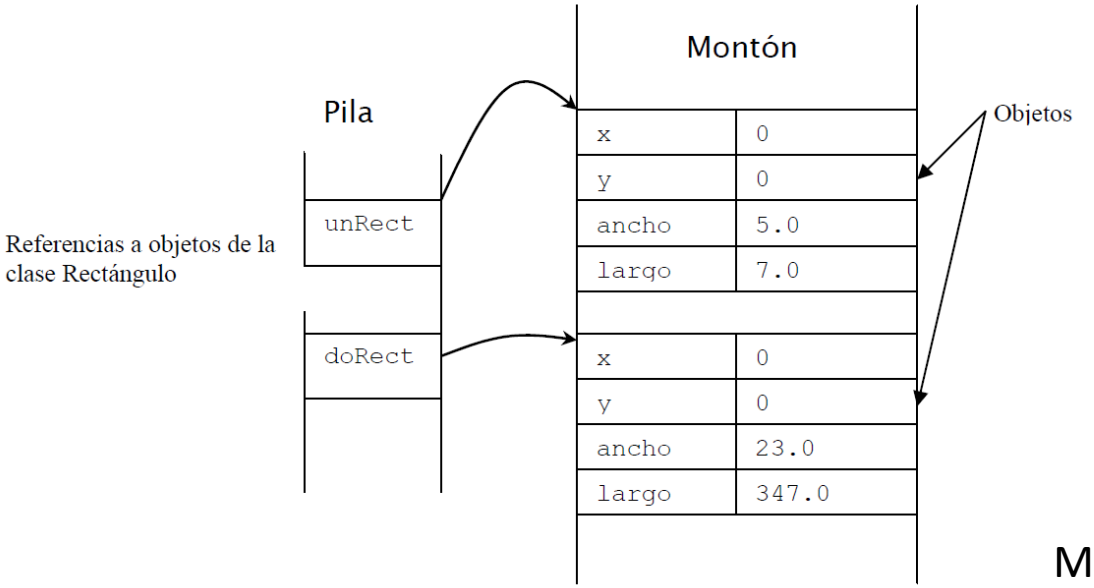
Clases

- Al crear un objeto o instancia de una clase:
  1. Se crea una variable de tipo referencia que almacene la referencia al objeto:

```
Rectangulo unRect;
```

  2. Se crea un objeto de la clase y se asigna a la referencia anterior:

```
unRect = new Rectangulo(5,7);
```



Montón = Heap

Clases

- Forma general:

```
[atributos][modificadores] class identificador [:clase-base]
{
    // Cuerpo de la clase
    // Declaraciones de los miembros de la clase
}
```

- Sólo se permite herencia simple.
- Modificadores:
  - De control de acceso: `public`, `protected`, `private` `internal` (visibilidad assembly/paquete)
  - Otros: `new`, `abstract`, `sealed` (no se puede heredar)
- Referencia a `this` → Referencia al propio objeto dentro de la propia clase.

Clases

- Constantes y miembros de sólo lectura
  - Modificador '`const`':
    - Valores constantes asociados a la clase
    - No pueden ser modificados a lo largo del programa
    - Es obligatoria su inicialización en la clase

```
class Matematicas{
public const double PI = 3,141592;
}
```

  - Modificador '`readonly`':
    - Permite la inicialización después de la inicialización.

```
class usuario{
   public readonly string password;
   public Personal(string palabra){
      password = palabra;
   }
}
```

Clases

- Métodos
  - Funciones o bloques de código que los objetos de esa clase pueden ejecutar.
  - Forma general:

```
[modificadores] tipoDeRetorno
nombreDelMetodo([parámetros]){
    // cuerpo del método
}
```

  - Si no devuelve nada, se indica mediante la palabra reservada `void`
  - Los métodos de un objeto se invocan con el operador punto "."

Clases

- Miembros estáticos (`static`)
  - Miembros (variables o métodos) de una clase a los que puede accederse **sin** haber creado una **instancia** de la propia clase.
    - `NombreDeLaClase.metodoStatic();`
    - `NombreDeLaClase.campoStatic;`
  - Los campos estáticos de una clase son **variables globales compartidas** por todos los objetos de esa clase.
  - **Restricciones** de los métodos `static`:
    - Sólo pueden llamar a métodos `static`.
    - Sólo pueden acceder a campos `static`.
    - No pueden referirse a '`this`'

Clases

- Control de acceso
  - Protege a los miembros de una clase de modificaciones no deseadas → **Encapsulación**.
  - El modo de acceso al miembro de la clase → **modificadores** de visibilidad.
  - El **interfaz** de una clase está constituido por el conjunto de métodos públicos de una clase.
  - Un miembro que no tiene modificador de acceso → `private` por defecto.
  - Los **objetos** de una clase → acceso a miembros públicos.
  - Los **miembros** de una clase → acceso a miembros públicos y privados.

Clases

- Campos de lectura/escritura
  - Acceso `get` (proceso de lectura)
    - Contiene el código que se ha de ejecutar en la consulta o lectura de la propiedad.
    - Debe terminar siempre con return o throw.
  - Acceso `set` (proceso de escritura)
    - Contiene el código que se ha de ejecutar al escribir en la propiedad.
    - El parámetro value es el valor que se le da a la propiedad.

```
public string Nombre{  get {return nombre;}

   set { nombre = value;}

}

string n = unaPersona.Nombre; // gracias a get
unaPersona.Nombre = "Andreu"; // gracias a set
```

Clases

- Constructor
  - Permite inicializar un objeto de una clase en el momento de su creación
  - Un constructor debe:
    - Tener el mismo nombre que la clase
    - No devolver nada (ni `void`)

- Destructor
  - Contiene el código que se ha de ejecutar cuando se destruye el objeto.

```
[atributos] ~nombreDeLaClase() {

    // código

}
```

  - El destructor es llamado por el recolector de basura cuando el objeto va a ser destruido.

Clases

- Clase abstracta (`abstract`)
  - Aquella que contiene uno o más miembros abstractos
  - Indica que no se pueden crear instancias u objetos de esa clase
    → no está totalmente implementada.
  - Sí pueden crearse objetos de clases derivadas de una clase abstracta.
  - Se utiliza para definir el interfaz que deben heredar todas las clases que hereden de ella.
  - Todas las clases que derivan de ella deben implementar de forma obligatoria todos los métodos abstractos de la clase base.

```
abstract class MiClaseBase{
    // código
}
class MiClaseDerivada: MiClaseBase{
// código
}
```

Herencia

- Mecanismo para definir una nueva clase (clase derivada o subclase) a partir de otra existente (clase base o superclase).
- La clase derivada:
  - Mismos miembros que la clase base.
  - Añade los suyos propios.
  - Puede redefinir los heredados.
- Jerarquía de clases
- Ventajas de la herencia:
  - Reutilización de código.
  - Fomenta el polimorfismo de referencias.

Herencia

- Control de acceso a miembros de la clase base
  - 'public' en la clase base:
    - Pasa a ser un miembro 'public' en la clase derivada.
    - Se puede acceder desde la derivada y desde el exterior.
  - 'private' en la clase base:
    - Pertenece a la clase derivada pero no es accesible desde el código de la clase derivada.
  - 'protected' en la clase base:
    - Accesible desde el código de la clase derivada
    - No accesible desde el exterior.

Herencia

- Sobrescritura de métodos.
  - Sobrescribir un método o propiedad de la clase base en la derivada.
  - En la clase base → `virtual`

    En la clase derivada → `override`

```
public class ClaseBase{
  public int a, b;
  public virtual void Imprimir(){...} // Muestra a y b
}
public class ClaseDerivada :ClaseBase{
  public int c;
  public override void Imprimir(){...} // Muestra tb c
}
```

  - No es posible cambiar la accesibilidad del método.

    Si se quiere forzar la invocación
  - del método de la clase base → '`base.método()`'.

Herencia

- Sobrescritura de métodos.
  - Modificador `new`
  - Para definir un campo o un método de la clase derivada con el mismo nombre que otro de la clase base.
  - Diferencia con virtual/static: Sólo se ejecuta el método de la clase derivada. (El miembro derivado 'esconde' al de la clase base).

```
public class ClaseBase{

    public int a, b;
    public void Imprimir(){...} // Muestra a y b
}
public class ClaseDerivada :ClaseBase{
    public int c;
    public new void Imprimir(){...} // Muestra solo c
}
```

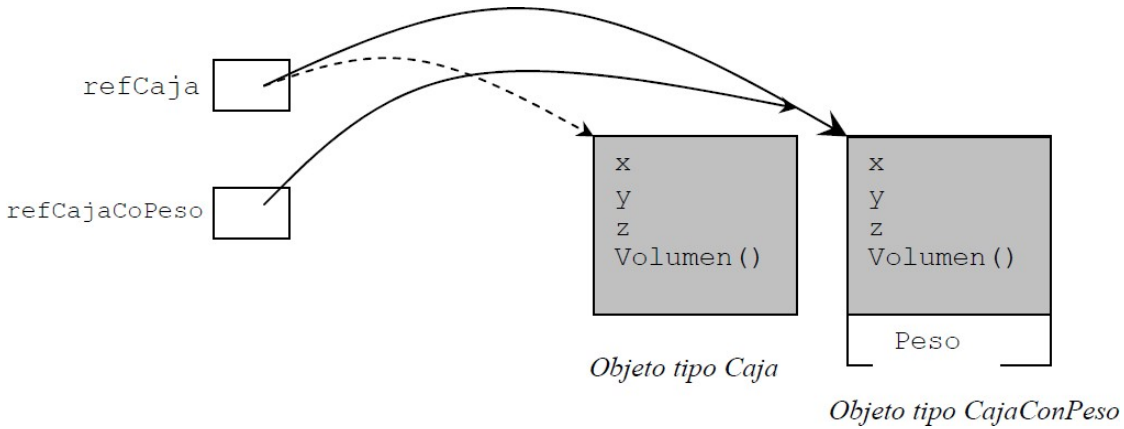
https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/knowing-when-to-use-override-and-new-keywords

Herencia

- Polimorfismo de referencias
  - Se puede asignar a una referencia de una superclase, una referencia de una subclase.

```
Caja refCaja = new Caja(1,2,3);// Clase base
CajaConPeso refCajaConPeso = new
CajaConPeso(3,4,5,6);//Clase derivada
// asigna una referencia a la clase derivada
refCada = refCajaConPeso;
```

  - OJO: `refCaja` no puede acceder a los campos nuevos de la derivada.



Interfaces

- Similar a una clase pero todos su miembros son abstractos → no tienen código
- Cuando una clase implementa una interface, debe implementar TODOS sus métodos.
- Heredar de una clase abstracta → Se hace referencia a lo que se es

VS

Implementar una interface → Se hace referencia a la capacidad de comportarse

Interfaces

- Estructura de una interface

```
[atributos][modificadores]
interface nombreInterface [:interfacesBase] {
    // cuerpo de la interface
}
```

  - Modificadores de acceso: `new`, `public`, `protected`, `internal` y `private` (no: `abstract`, `sealed`)

  - Todos los métodos de la interface son `public`
    → no llevan modificador

Interfaces

```
public interface IImprimible
{
    void Imprimible();
}
public interface IArchivable
{
    void Leer();
    void Escribir();
}
public class Documento :IImprimible, IArchivable
{
    void Leer(){ … }
    void Escribir(){ … }
    void Imprimir(){ … }
    // Otros miembros y código propio de la clase
}
```

Interfaces

- Herencia de interfaces

```
interface Icontrol
{
   void Paint();
}
interface ITextBox :Icontrol
{
   void SelText(string text);
}
class TextBox :ITextBox
{
   public void Paint(){ ...}
   public void SelText(string texto){ ...}
}
```

Interfaces

- Polimorfismo de referencias
  - No se puede crear objetos de tipo interface:

```
~~Iimprimible refImprimible = new Iimprimible();~~
```

  - Pero sí se puede crear una referencia de tipo interface para que apunte a cualquier objeto de un tipo que implementa dicha interface:

```
Iimprimible refImprimible;

Documento unDocumento = new Documento("texto");

refInterface = (Iimprimible)unDocumento;
```

    O bien:

```
Iimprimible refImprimible =

            (Iimprimible)new Documento("texto");
```

Arrays

- Estructura de datos que contiene variables (elementos) a los que se accede a través de índices.
- Todos los elementos son del **mismo tipo**.
- Un array puede tener más de una dimensión.
- Declaración: `tipoDeDatos[] nombreDelArray;`
  - Tras la declaración el array apunta a `null`
- Inicializaciones flexibles:

```
string[] unArray = new string[3];
         unArray[0] = "Andreu";
         unArray[1] = "Gerard";
         unArray[2] = "Jordi";
string[]  unArray  =  {"Andreu",  "Gerard",  "Jordi"};
string[] unArray = new string[3]{"Uno", "Dos", "Tres"};
```

Arrays

- Array de varias dimensiones
  - Array multidimensional o matriz rectangular
    - Todas sus filas tiene el mismo número de columnas
    - Declaración: `string[,] arrayMulti;`
    - Instanciación:
      - `string[,] arrayMulti = new string[5,4];`
      - `string[,] par = new string[2,2] {{"1x1","1x2"},{"2x1","2x2"}};`
    - Acceso:
      - `par[1,1] = "nuevo 2x2";`

Arrays

- Array de varias dimensiones
  - Array de arrays
    - Cada fila tiene un número de columnas que puede variar.
    - Declaración: `int[][] arrayArrays;`
    - Instanciación:
      - `int[][] arrayArrays = new int[5][];`
      - `int[][] numeros = new int[2][] {{2,3,4},{5,6,7,8}};`
    - Acceso:
      - `numeros[1][1] = 9;`
  - Se pueden mezclar arrays multidimensionales y arrays de arrays: `int [][,,][,] numeros;`

Arrays

- Recorrer el array
  - Bucle for

```
int[] unArray = {4,5,6,-1,0};
for(int i=0; i<unArray.Length; i++)
        {  system.Console.WriteLine(unArray[i
        ]);
    }
```

  - Foreach

```
int[] unArray = {4,5,6,-1,0};
foreach(int i in unArray)
    {  system.Console.WriteLine(
    i);
    }
```

Colecciones

- Limitación de los arrays → tamaño fijo.
- Una colección se utiliza para trabajar con:
  - Listas
  - Conjuntos ordenados de objetos
- Proporcionan métodos básicos para acceder a los elementos → corchetes como los arrays []
- Interfaces que implementan:
  - IEnumerable: para recorrer la colección con un foreach.
  - ICollection: Obtener el número de elementos de la colección y copiar los elementos a un array tradicional.
  - IList: Proporciona la lista de los elementos de la colección.
  - IDictionary: Proporciona una lista de elementos de la colección, accesibles a través de un valor en un índice.

Colecciones

- **ArrayList**
  - Permite elementos de tipos de datos diferentes, pero al acceder a ellos se debe hacer un cast del tipo correspondiente
  - Penalización en el 'boxing' y 'unboxing'.
- **List**
  - Todos los elementos son del mismo tipo
- Ambos tipos crecen automáticamente al añadir nuevos elementos, con el método 'add'.
- Algunos métodos: Clear, Foreach, Contains, Insert, Sort.

```
ArrayList l  = new ArrayList();
l.add(5); l.add("hola");
List<string> l2 = new List<string>();
l2.add("hola");
```

Colecciones

- **Dictionary**
- Un tipo de colección cuyos elementos son pares clave/valor.
- Se declaran especificando el tipo de las claves y de los valores:

```
Dictionary<string, int> D= new Dictionary<string, int>();
```

- Algunos métodos para gestionar este tipo de dato:
  - Count
  - Item
  - Keys, Values
  - Add
  - Remove
  - Clear
  - Contains, ContainsKey, ContainsValue

Excepciones

- Ofrecen un modo estructurado, uniforme y seguro de tipos para el manejo de situaciones de error.
- Diferencias con C++:
  - Las excepciones son instancias de tipos clase derivados de la clase `System.Exception`.
  - Un bloque `finally` puede ser utilizado para escribir código que se ejecute tanto si se da una excepción como si no.
  - Las excepciones de sistema, como desbordamiento, división por cero y de referencias nulas tienen clases de error bien definidas.

Excepciones

- Clase `System.Exception`. Propiedades:
  - **`Message`**: Describe la causa de la excepción.
  - **`InnerException`**: Excepción interna de la excepción (si la excepción se relanzó como respuesta a otra) o `null`.
- Lanzamiento de excepciones. Dos modos:
  - Mediante la sentencia `throw`:
    - Lanza la excepción de modo condicional o inmediata.
    - En ningún caso devuelve el control a la sentencia siguiente al `throw`.

```
if(s==null) throw new ArgumentNullException();
```

  - Durante la ejecución de sentencias y expresiones, si se dan situaciones en las que la operación no pueda completarse de modo normal.
    - Ejemplo, una división por cero: `System.DivideByZeroException`

Excepciones

- Manejo de excepciones:

```
try
{
    string s = null;
    x.MiFun(s);
}
catch(ArgumentNullException e)
{
    Console.WriteLine("Primera excepción");
}
catch(Exception e)
{
    Console.WriteLine("Segunda exception");
}
finally
{
    // Se ejecuta haya excepción o no
}
```

Excepciones

- Clases de excepción más comunes:
  - `System.OutOfMemoryException`
    - Cuando falla un intento de reserva de memoria mediante el operador `new`.
  - `System.NullReferenceException`
    - Cuando se pretende acceder a un supuesto objeto mediante una referencia a `null`
  - `System.IndexOutOfRangeException`
    - Cuando se intenta acceder a un array mediante un índice menor que cero o mayor que el límite del array.
  - `System.DivideByZeroException`
    - Cuando se intenta dividir un valor de tipo integral por cero.
  - `System.OverflowException`
    - Cuando una operación aritmética causa un desbordamiento en un contexto `checked`

Async, await

- Async: para declarar una función que contiene una porción de código que implica una operación asíncrona
- Await: para indicar que la operación o invocación se ejecutará asíncronamente.
- Task, Task<tipo>: Define una operación asíncrona y su resultado.

Async, await

- Ejemplo:

```
async Task<string> get_a_web(string url) {
   HttpClient cli = new HttpClient();
   Task<string> contents = await
client.GetStringAsync(url);
   return contents;
}
```

- Esta función obtiene el contenido de una url, de forma asíncrona, de forma que su invocador recupera el control mientras la operación tiene lugar.

Threads

- Permiten crear hilos o líneas de ejecución diferentes al hilo principal de nuestro programa
- Se usan para delegar tareas (normalmente pesadas o costosas) y así descargar la ejecución del hilo principal, permitiendo que el programa pueda seguir realizando sus operaciones normales.
- Se debe incluir la librería System.Threading.
- Se utiliza la clase Thread, la cual tiene entre otras:
  - Propiedades: CurrentContext, CurrentThread, IsAlive, IsBackground, Name, Priority, ThreadState.
  - Métodos: Start, Abort, Join, Sleep.

Threads

- Ejemplo sencillo:

```
public static void hiloHijo(...) {
   //hacer alguna cosa
}

static void Main(string[] args) {
   ThreadStart hijoref = new ThreadStart(hiloHijo);
   Thread ThreadHijo = new Thread(hijoref);
   ThreadHijo.Start();
   Thread.Sleep(2000);
   ThreadHijo.Abort();
}
```