Programación de Aplicaciones Visuales I

Introducción a C#

Objetivo

- Presentar una introducción al lenguaje de programación C#. Se describirá la estructura general de un programa, sus características más importantes, sintaxis, tipos de datos, operadores, expresiones, estructuras de control, etc.

Temario

- Estructura de un programa C#.
- Características de la sintaxis.
- Tipos de datos simples.
- Estructuras de control.
- Manejo de excepciones.
- Tablas.
- POO en C#.
- Generics

Temario

- Estructura de un programa C#
  - La clase.
  - El método Main().
  - Namespace utilizados por la aplicación.
  - Namespace de la aplicación.
  - Namespace System y la clase Console.
  - Algo de código, “Hola Mundo” vía consola.

La clase

- Una aplicación C# es una colección de clases, estructuras y tipos.
- Una clase contiene una colección de datos y métodos para manipular los mismos.
- Sintaxis.

```
class nombre
{
        ...
}
```

- Una aplicación C# puede abarcar más de un archivo. Una clase también, para ello se utiliza “partial class”.

El método Main()

- En C# todas las aplicaciones para consola y winforms deben tener un punto de inicio, que es el método Main().
- Se debe declarar como.

```
class nombre
{
        static void Main()
        {
                ...
        }
}
```

- Puede devolver void o int.

Namespace utilizados por nuestra aplicación

- Son el conjunto de clases, funciones y tipos de datos que nuestra aplicación puede utilizar.
- .NET Framework ofrece muchas clases útiles provistas a través de la Base Clase Library.
- Para indicar que se utilizará un determinado namespace se utiliza la palabra reservada “using”.
- Podemos hacer uso de los namespaces provistos en la plataforma (BCL), creados por nosotros, desarrollados por terceras partes.

Namespace de la aplicación

- Es una forma lógica de agrupar las clases, funciones y tipos de datos de nuestra aplicación.
- Se declaran con la palabra reservada “namespace”.
- Dentro de un namespace no es posible declarar dos clases con el mismo nombre.
- Puede agrupar uno o archivos de código de nuestra aplicación.
- Se pueden declarar en forma anidada.

Namespace System y la clase Console

- Los namespace están organizados de forma jerárquica. System es la raíz del namespace más importante provisto por la plataforma.
- Dentro de él podemos encontrar la clase Console.
- Console brinda toda la funcionalidad para crear aplicaciones cuya interfaz es una consola del SO.
- Algunos métodos: WriteLine, ReadLine, Clear, Beep.

“Hola Mundo” vía consola

```
using System;

namespace HolaMundo
{
    class Holamundo
    {
        static void Main()
        {
            System.Console.WriteLine("¡Hola Mundo!");
        }
    }
}
```

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
  - Separación y agrupación de instrucciones. El “;” y las “{}”.
  - Mayúsculas y minúsculas (**Case Sensitivity)**.
  - Comentarios.

Separación y agrupación de instrucciones. El “;” y las “{}”.

- El punto y coma “;” se utiliza para separar instrucciones.

```
objeto.propiedad = “Valor”;

objeto.propiedad =
“Valor”;
```

- Las llaves se utilizan para agrupar instrucciones dentro de un mismo bloque.

```
class nombre{
    static void Main()
    {
        Instrucción1;
        Instrucción2;
    }
}
```

Mayúsculas y minúsculas (Case Sensitivity)

- Todas las palabras reservadas (using, namespace, public, class, if, for, etc.) se escriben en minúsculas.
- El compilador de C# puede distinguir entre dos variables declaradas con igual nombre pero con al menos una letra que difiera en mayúscula – minúscula.

Comentarios

- Comentarios en una sola línea.

```
// Obtener el nombre del usuario
Console.WriteLine("¿Cómo se llama? ");
```

- Comentarios que abarcan más de una línea.

```
/* Encontrar la  mayor raíz
de la ecuación cuadrática */
x = (…);
```

- Comentarios para generadores de documentación.

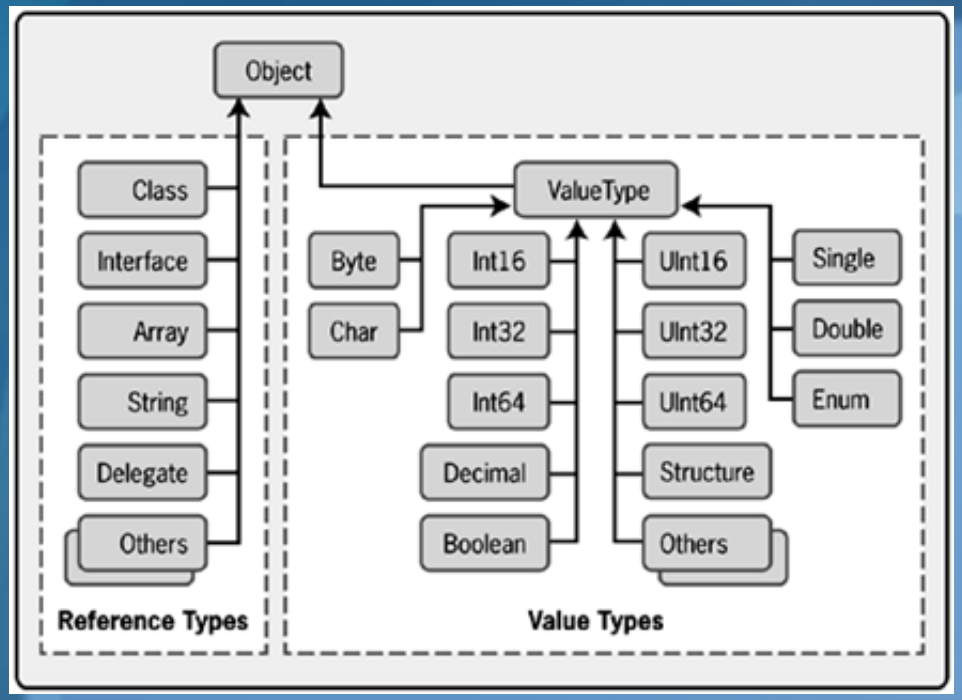
```
/// <summary>
/// Devuelve la suma dos números enteros
/// </summary>
/// <param name="a">Numero a</param>
/// <param name="b">Numero B</param>
public int sumar(int a, int b)
```

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
- Utilizar tipos de datos simples.
  - El sistema de tipos comunes (CTS).
  - La memoria y los tipos de datos.
  - Nombres de variables.
  - Los tipos de datos predefinidos.
  - Creación de tipos de datos.
  - Conversión de tipos de datos.

El sistema de tipos comunes (CTS)

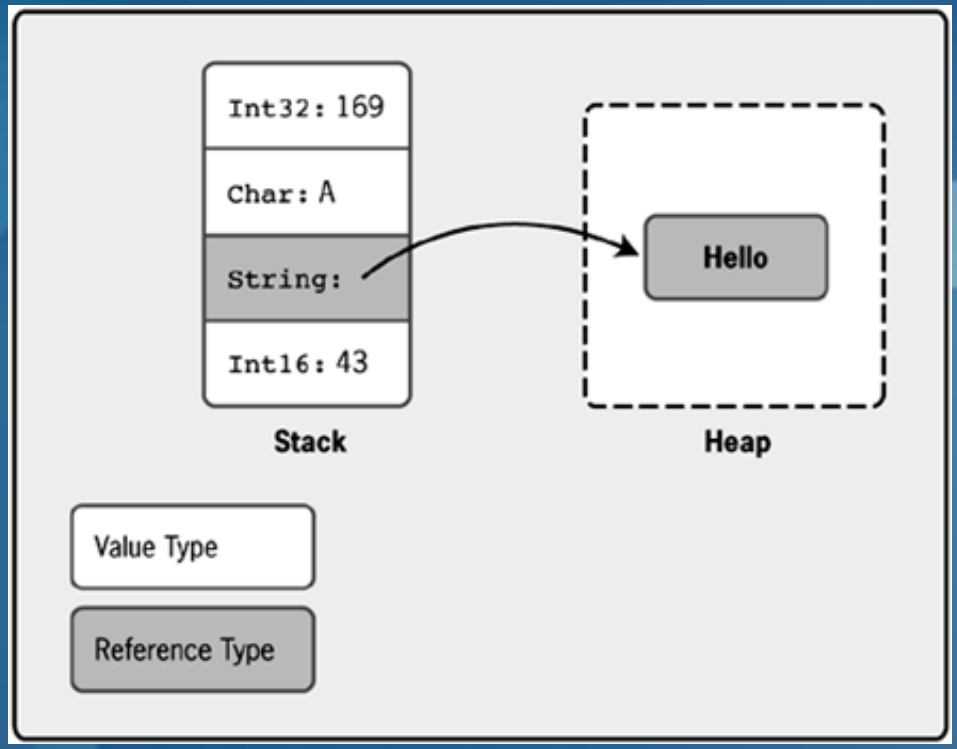
- Define un conjunto común de “tipos” de datos orientados a objetos.
- Todo tipo hereda directa o indirectamente del tipo System.Object.
- Define tipos de VALOR y de REFERENCIA.



La memoria y los tipos de datos

- El CLR administra dos segmentos de memoria: **Stack (Pila)** y **Heap (Montón).**
- El **Stack** es liberado automáticamente y el **Heap** es administrado por el **GC** (**Garbage Collector**).

- Los tipos **VALOR** se almacenan en el Stack.
- Los tipos **REFERENCIA** se almacenan en el Heap.



Tipos de datos

Tipo	Descripción	Bytes	Rango de valores	Alias
SByte	Bytes con signo	1	[-128 | 127]	sbyte
Byte	Bytes sin signo	1	[0 | 255]	byte
Int16	Enteros cortos con signo	2	[-32.768 | 32.767]	short
UInt16	Enteros cortos sin signo	2	[0 | 65.535]	ushort
Int32	Enteros normales con signo	4	[-2.147.483.648 | 2.147.483.647]	int
UInt32	Enteros normales sin signo	4	[0 | 4.294.967.295]	uint
Int64	Enteros largos con signo	8	[-9.223.372.036.854.775.808 | 9.223.372.036.854.775.807]	long
UInt64	Enteros largos sin signo	8	[0 | 18.446.744.073.709.551.615]	ulong
Single	Reales con 7 dígitos de presición	4	[1,5 * 10 -45 | 3,4 * 10 38]	float
Double	Reales con 15-16 dígitos de presición	8	[5,0 * 10 -324 | 1,7 * 10 308]	double
Decimal	Reales con 28-29 dígitos de presición	16	[1,0 * 10 -28 | 7,9 * 10 28]	decimal
Boolean	Valores lógicos	1	[true | false]	bool
Char	Valores Unicode	2	[0 | 65.535]	char
String	Cadenas de caracteres	Variable	Limitado por la memoria	string
Object	Cualquier objeto	Variable	Limitado por la memoria	object

Reglas y recomendaciones para nombrar variables

- Reglas
  - Use letras, el signo de subrayado y dígitos.
- Recomendaciones
  - Evite poner todas las letras en mayúsculas.
  - Evite empezar con un signo de subrayado.
  - Evite el uso de abreviaturas.
  - Use PascalCasing para nombres con varias palabras.

Palabras clave de C#

- Las palabras clave son identificadores reservados.

```
abstract, base, bool, default, if, finally
```

- No utilice palabras clave como nombres de variables.
  - Produce errores en tiempo de compilación

- Procure no usar palabras clave cambiando mayúsculas y minúsculas.

```
int INT;  // Mal estilo
```

Los tipos de datos predefinidos

- Declaración de variables locales.
- Asignación de valores a variables.
- Asignación compuesta.
- Operadores comunes.
- Incremento y decremento.
- Precedencia de operadores.

Declaración de variables locales

- Se declaran indicando el tipo de dato y nombre de variable.

```
int objetoCuenta;
```

- Es posible declarar múltiples variables en una declaración.

```
int objetoCuenta, empleadoNúmero;
```

```
int objetoCuenta,
    empleadoNúmero;
```

Asignación de valores a variables

- Asignar valores a variables ya declaradas.

```
int empleadoNumero;
empleadoNumero = 23;
```

- Inicializar una variable cuando se declara.

```
int empleadoNumero = 23;
```

- También es posible inicializar valores de caracteres.

```
char inicialNombre = 'J';
```

Asignación compuesta

- Es muy habitual sumar un valor a una variable.

```
itemCount = itemCount + 40;
```

- Se puede usar una expresión más práctica.

```
itemCount += 40;
```

- Esta abreviatura es válida para todos los operadores aritméticos.

```
itemCount -= 24;
```

Operadores comunes

Operadores comunes	Ejemplo
Operadores de igualdad	== !=
Operadores relacionales	< > <= >= is
Operadores condicionales	&& || ?:
Operador de incremento	++
Operador de decremento	- -
Operadores aritméticos	+ - * / %
Operadores de asignación	= *= /= %= += -= <<= >>= &= ^= |=

Incremento y decremento

- Es muy habitual cambiar un valor en una unidad.

```
objetoCuenta += 1;
objetoCuenta -= 1;
```

- Se puede usar una expresión más práctica.

```
objetoCuenta++;
objetoCuenta--;
```

- Existen dos formas de esta abreviatura.

```
++objetoCuenta;
--objetoCuenta;
```

Precedencia de operadores

- Precedencia y asociatividad de operadores
  - Todos los operadores binarios, salvo los de asignación, son asociativos por la izquierda.
  - Los operadores de asignación y el operador condicional son asociativos por la derecha.

Creación de tipos de datos

- Enumeraciones
- Estructuras

Enumeraciones

- Definición de una enumeración.

```
enum Color { Rojo, Verde, Azul }
```

- Uso de una enumeración.

```
Color colorPaleta = Color.Rojo;
```

- Visualización de una variable de enumeración.

```
Console.WriteLine("{0}", colorPaletta); // Muestra Rojo
```

Estructuras

- Definición de una estructura.

```
public struct Empleado
{
    public string pilaNombre;
    public int age;
}
```

- Uso de una estructura.

```
Empleado empresaEmpleado;
empresaEmpleado.pilaNombre = "Juan";
empresaEmpleado.age = 23;
```

Conversión de tipos de datos

- Conversión implícita de tipos de datos.
- Conversión explícita de tipos de datos .

Conversión implícita de tipos de datos

- Conversión de int a long

```
using System;
class Test
{
    static void Main()
    {
        int intValor = 123;
        long longValor = intValor;
        Console.WriteLine("(long) {0} = {1}", intValor, longValor);
    }
}
```

- Las conversiones implícitas no pueden fallar
  - Se puede perder precisión, pero no magnitud.

Conversión explícita de tipos de datos

- Para hacer conversiones explícitas se usa una expresión de cast (molde):

```
using System;
class Test
{
    static void Main( )
    {
        long longValor = Int64.MaxValue;
        int intValor = (int) longValor;
         Console.WriteLine("(int) {0} = {1}", longValor, intValor);
    }
}
```

Boxing y Unboxing

- Boxing: se utiliza cuando se convierte tipo valor a tipo referencia.

```
int a= 20;
object b = a;
```

- Unboxing: cuando se convierte tipo referencia a tipo valor

```
int a = 20;
object b = a;
int c = (int)b;
```

Conversión de Datos

- Las clases de datos tienen métodos para convertir objetos a su tipo.
  - Clase.Parse(String) Convierte un string a la clase destino. Si no tiene el formato correcto da error.
  - Clase.TryParse(String, out Objeto Clase): La función convierte el string en un objeto del tipo de la clase y lo devuelve como parametro Out. Si convierte, la funcion devuelve true y si no lo puede convertir, devuelve false.

Conversión de Datos

- Las clases de datos tienen métodos para convertir objetos a su tipo.
  - Convert.ToTipoObjeto: Convert tiene un listado de “To” (ToString, ToInt16, ToBoolean, etc) que permite convertir un string u otro tipo de dato a un tipo específico.
  - Objeto = (TipoObjeto) Objeto: Castear un objeto para que sea de otro tipo

Nullables types (C# 2.0)

- Permite que un tipo de dato valor tenga un valor “null”
- Se define de la siguiente manera:
  int? x = 125;
  (notación frecuentemente utilizada)
  o tambien como:
  Nullable<int> i;
- Para determinar si posee un valor:
  if (x.HasValue) {...}
  O
  if (x != null) {...}
- **Se puede usar el operador ?? para asignar un valor por default que va a ser aplicado cuando el valor es “null”**
  int? x = null; int y = x ?? -1;

Funcionalidad de Tipos de Datos

- Caracteres (Strings)
  - ToUpper - Mayúscula
  - ToLower - Minúscula
  - Trim – Remueve espacios en blanco
  - Substring – Cadena parcial de una cadena
  - IndexOf – Busqueda en la cadena
  - Length - Longitud
  - CompareTo – Compara por mayor o menor

Funcionalidad de Tipos de Datos

- Fechas (Datetime)
  - Datetime.Now: Fecha y Hora del sistema
  - Datetime.Today: Fecha del sistema
  - Datetime.Parse: Convierte un texto a fecha. Si no tiene el formato, da error.
  - Objeto.ToString(Formato): Convierte a texto
  - Objeto.Add(TimeSpan): Suma un TimeSpan a la fecha del objeto.
  - Objeto.AddDays(dias): Existen Add de todas las unidades de tiempo.
  - Objeto.Substract: Resta una Fecha o un TimeSpan a otra fecha.

Funcionalidad de Tipos de Datos

- Fracción de Tiempo (TimeSpan)
  - Almacena cantidad de años, meses, dias, horas, minutos, segundos y fracciones.
  - Se pueden sumar a fechas y es lo que se obtiene al restar 2 fechas.
  - Objeto.Days/TotalDays: Devuelve los dias sin y con francción respectivamente.

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
- Utilizar tipos de datos simples.
- Estructuras de control.
  - Introducción a las instrucciones.
  - Uso de instrucciones condicionales.
  - Uso de instrucciones iterativas.
  - Uso de instrucciones de salto.

Introducción a las instrucciones

- Bloques de instrucciones

Se usan llaves para delimitar bloques.

```
{
   // code
}
```

Un bloque y su bloque padre no pueden tener una variable con el mismo nombre.

```
{
   int i;
   ...
   {
      int i;
      ...
   }
}
```

Bloques hermanos pueden tener variables con el mismo nombre.

```
{
   int i;
   ...
}
...
{
   int i;
   ...
}
```

Introducción a las instrucciones

- Tipos de instrucciones
  - Instrucciones condicionales: if y switch.
  - Instrucciones de iteración: while, do, for y foreach.
  - Instrucciones de salto: break y continue.

La instrucción IF

- Sintaxis

```
if (expresión-booleana)
  primera-instrucción-incrustada;
else
{
  segunda-instrucción-incrustada;
  tercera-instrucción-incrustada;
}
```

- El bloque else es opcional.
- Si existen dos o más de una instrucciones en un bloque, estas deben estar agrupadas en {}.
- No hay conversión implícita de int a bool

```
int x;
...
if (x) ...      // Debe ser if (x != 0) en C#
if (x = 0) ... // Debe ser if (x == 0) en C#
```

Instrucciones if en cascada

```
enum Palo {Treboles,
           Corazones, Diamantes,
Picas}
Palo cartas = Palo.Corazones;
string color = “”;
if (cartas == Palo.Treboles)
{
    color = “Negro”;
}
else
{
    if (cartas == Palo.Corazones)
    {
        color = “Rojo";
    }
    else
    {
        if (palo == Palo.Diamantes)
        {
            color = "Rojo";
        }
        else
        {
            color = “Negro";
        }
    }
}
```

```
enum Palo {Treboles,
           Corazones, Diamantes,
Picas}
Palo cartas = Palo.Corazones;
string color = “”;
if (cartas == Palo.Treboles)
{
    color = “Negro”;
}
else if (cartas == Palo.Corazones)
{
    color = “Rojo";
}
else if (palo == Palo.Diamantes)
{
    color = "Rojo";
}
else
{
    color = “Negro";
}
```

La instrucción switch

- Las instrucciones switch se usan en bloques de varios casos.
- Se usan instrucciones break para evitar caídas en cascada (fall through).

```
switch (palo)
{
    case Palo.Treboles :
        color = "Negro";
        break;
    case Palo.Picas :
        color = "Negro";
        break;
    case Palo.Corazones :
         color = "Rojo";
        break;
    case Palo.Diamantes :
        color = "Rojo";
        break;
    default:
        color = "ERROR";
        break;
}
```

```
switch (palo)
{
    case Palo.Treboles :
    case Palo.Picas :
        color = "Negro";
        break;
    case Palo.Corazones :
    case Palo.Diamantes :
        color = "Rojo";
        break;
    default:
        color = "ERROR";
        break;
}
```

La instrucción while

- Ejecuta instrucciones en función de un valor booleano.
- Evalúa la expresión booleana al principio del bucle.
- Ejecuta las instrucciones mientras el valor booleano sea true.

```
int i = 0;

while (i < 10)
{
    Console.WriteLine(i);
    i++;
}
```

```
0
1
2
3
4
5
6
7
8
9
```

La instrucción do

- Ejecuta instrucciones en función de un valor booleano.
- Evalúa la expresión booleana al final del bucle.
- Ejecuta las instrucciones mientras el valor booleano sea true.

```
int i = 0;

do
{
    Console.WriteLine(i);
    i++;
} while (i < 10);
```

```
0
1
2
3
4
5
6
7
8
9
```

La instrucción for

- La información de actualización está al principio del bucle.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

```
0
1
2
3
4
5
6
7
8
9
```

- Las variables de un bloque for sólo son válidas en el bloque.

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
Console.WriteLine(i); // Error: i está fuera de ámbito
```

- Un bucle for puede iterar varios valores.

```
for (int i = 0, j = 0; ... ; i++, j++)
```

La instrucción foreach

- Elige el tipo y el nombre de la variable de iteración.
- Ejecuta instrucciones incrustadas para cada elemento de la clase collection.

```
ArrayList numeros = new ArrayList( );

for (int i = 0; i < 10; i++ )
{
    numeros.Add(i);
}

foreach (int number in numeros)
{
    Console.WriteLine(numero);
}
```

```
0
1
2
3
4
5
6
7
8
9
```

Las instrucciones break and continue

- La instrucción break abandona la instrucción switch, while, do, for o foreach más próxima.
- La instrucción continue salta a la siguiente iteración de una instrucción while, do, for, foreach.

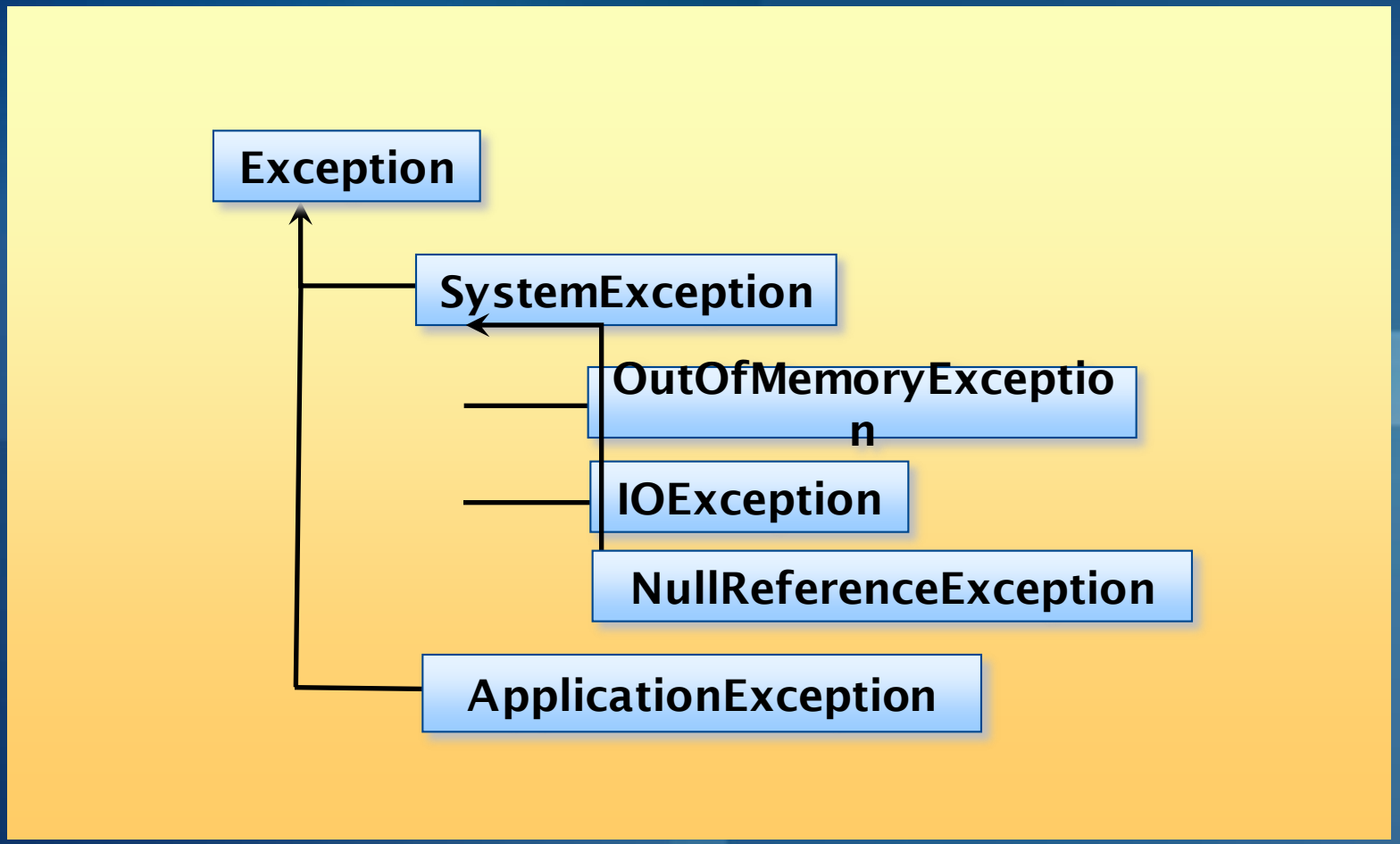
```
int i = 0;
while (true)
{
    Console.WriteLine(i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
- Utilizar tipos de datos simples.
- Estructuras de control.
- Manejo de excepciones.
  - Objetos excepción.
  - Uso de bloques try-catch.
  - Bloques catch múltiples.
  - La cláusula finally.
  - Lanzamiento de excepciones.
  - Normas para el tratamiento de excepciones.

Objetos Excepción

- En .NET Framework se han definido una serie de clases de excepción.



Uso de bloques try-catch

- Solución orientada a objetos para el tratamiento de errores
  - Poner el código normal en un bloque **try.**
  - Tratar las excepciones en un bloque catch aparte.
  - Los bloques try-catch se pueden anidar.

```
try
{
    Console.WriteLine("Escriba un número");
    int i = int.Parse(Console.ReadLine());
}
catch (OverflowException capturada)
{
    Console.WriteLine(capturada);
}
```

Lógica del programa

Tratamiento de errores

Bloques catch múltiples

- Cada bloque catch captura una clase de excepción.
- Un bloque try puede tener un bloque catch general.
- Un bloque try no puede capturar una clase derivada de una clase capturada en un bloque catch anterior.

```
try
{
    Console.WriteLine(“Escriba el primer número");
    int i = int.Parse(Console.ReadLine());
    Console.WriteLine("Escriba el segundo número");
    int j = int.Parse(Console.ReadLine());
    int k = i / j;
}
catch (OverflowException capturada) {…}
catch (DivideByZeroException capturada) {…}
```

La cláusula finally

- Las instrucciones de un bloque finally se ejecutan siempre.
- Normalmente se utilizan para liberar recursos.

```
Monitor.Enter(x);
try
{
    ...
}
finally
{
    Monitor.Exit(x);
}
```

Bloques catch opcionales

La instrucción throw

- Lanza una excepción apropiada.
- Asigna a la excepción un mensaje significativo.

```
throw expression;
```

```
if (minuto < 1 || minuto >= 60)
{
    throw new InvalidTimeException(minuto + " no es un minuto válido");
    // !! Instrucciones no ejecutadas !!
}
```

Normas para el tratamiento de excepciones

- Lanzamiento
  - Evitar excepciones para casos normales o esperados.
  - Nunca crear ni lanzar objetos de la clase Exception, en el caso más general utilizar SystemException.
  - Incluir una cadena de descripción en un objeto Exception.
  - Lanzar objetos de la clase más específica posible.
- Captura
  - Ordenar los bloques catch de lo específico a lo general.
  - No permitir que se generen excepciones sin tratar en Main.

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
- Utilizar tipos de datos simpes.
- Estructuras de control.
- Manejo de excepciones.
- Arrays.
  - ArrayList.

Listas ArrayList

- Es una lista de elementos indexada
- Puede contener cualquier tipo de dato
- Necesita el Namespace System.Collections
- Se declara
  - ArrayList myLista = new ArrayList()
- Propiedades
  - Count
  - Items

Listas ArrayList

- Métodos
  - Add
  - Clear
  - IndexOf
  - Insert
  - Remove
  - RemoveAt
  - Sort

```
ArrayList myList = new ArrayList();
int i;

for (i = 0; i < 5; i++)
        myList.Add(i+1);

for (i = 0; i < myList.Count; i++)
   {
   Console.WriteLine("Indice [{0}]: {1}", i, myList[i].ToString());
   }
 myList.Clear();
```

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
- Utilizar tipos de datos simples.
- Estructuras de control.
- Manejo de excepciones.
- Arrays.
- POO en C#.
  - Clases.
  - Herencia.

Clases

- Definición e instancia de una clase.
- Constructor por defecto.
- Constructores múltiples.
- Modificadores de acceso
- Datos miembros de una clase.
- Datos miembros de tipo y datos miembros de objeto.
- Métodos.
- Propiedades.

Definición e instancia de una clase

- Las clases se definen con la palabra reservada class.
- Dentro de ellas se definen datos y métodos.
- Las instancias de una clase se obtienen utilizando el operador new.

```
class CuentaBancaria
{
    private decimal saldo;

    public void Depositar(decimal monto)
    {
        saldo += monto;
    }
}
```

```
class Test
{
    static void Main()
    {
        CuentaBancaria cuenta1;
        cuenta1 = new CuentaBancaria();
        CuentaBancaria cuenta2 = new CuentaBancaria();
        cuenta1.Depositar(1000M);
    }
}
```

Constructor por defecto

- Características de un constructor por defecto
  - Acceso público.
  - Mismo nombre que la clase.
  - No tiene tipo de retorno (ni siquiera void).
  - No recibe ningún argumento.
  - Inicializa todos los campos a cero, false o null.
  - Si no se define el compilador de C# lo hace por nosotros.
- Sintaxis del constructor

```
class CuentaBancaria
{
    public CuentaBancaria()
    {}
}
```

Constructores múltiples

- Es posible definir múltiples constructores.
- La firma de los constructores debe ser distinta.
- Múltiples constructores permiten inicializar objetos de manera diferente.

```
class Alumno
{
}

class Alumno
{
    public Alumno(){.....}
}

class Alumno
{
    public Alumno()
    {.....}
    public Alumno(int edad, string nombre)
    {....}
}
```

El compilador declara el constructor por defecto.

El programador declara el constructor por defecto.

El programador declara un constructor vacío y otro con parámetros.

Modificadores de acceso

Modificador	Significado
public	Acceso no restringido.
protected	Acceso limitado a la propia clase y sus derivadas.
internal	Acceso limitado al Assembly donde está declarada la clase. Este es el modificador por defecto.
protected internal	Acceso limitado a los tipos derivados de este siempre que estén en el mismo Assembly.
private	Acceso restringido a la misma clase.

Modificadores de acceso

	Para tipos		Para miembros	
	Valor por defecto	**Valores posibles**	**Valor por defecto**	**Valores posibles**
class	internal	public Internal private	private	public protected internal protected internal private
struct	internal	public internal	private	public internal private
enum	public		public	

- La accesibilidad de un miembro es establecida por la accesibilidad declarada del miembro combinado con el accesibilidad del tipo que la contiene.

Datos miembros de una clase

- Dato común a todos los objetos de una determinada clase.
- Se declaran de la siguiente manera:

```
<tipoCampo> <nombreCampo>;
```

```
class Alumno
{
    private int Edad;
    string Nombre="Ninguno";
    public Alumno(int edad)
    {
        this.Edad = edad;
        Nombre = "Juan Perez";
    }
    public static void Main()
    {
        Alumno A=new Alumno(23);
    }
}
```

Por defecto los campos son privados a la clase.

Se puede inicializar en el momento de declarar.

La palabra reservada this permite hacer referencia al propio objeto (instancia).

Se utiliza en aplicaciones Win32, como punto de entrada de ejecución de la aplicación.

Datos miembros de tipo y datos miembros de objeto

- Si la definición de un miembro va precedida de la palabra static, este va a pertenecer a la clase (Miembros de tipo), de lo contrario pertenecerá al objeto (Miembro de objeto).
- Para acceder a un miembro del objeto se utiliza la notación <identificadorObjeto>.<miembro> sino, para miembros de tipo <clase>.<campo>.

```
class Cuenta
{
    public static decimal interes = 
12;
    public decimal saldo;
}
```

```
Class Test
{
    public static void Main()
    {

Console.WriteLine(Cuenta.interes);
        Cuenta C = new Cuenta();
        C.saldo = 5000;
    }
}
```

Propiedades

- Las propiedades permiten controlar la lectura y escritura de los datos miembro.
- Permiten lograr un buen nivel de encapsulación.
- Se pueden definir propiedades de solo lectura o solo escritura definiendo solo el get o el set.
- Value es un parámetro de entrada del mismo tipo que la propiedad que se usa en el bloque set.

```
class Ejemplo
{
    private intX, intY;
    public int X
    {
        set {intX = value;}
        get {return intX;}
    }
    public int Y
    {
        get {return intY;}

    }
}
```

```
    Ejemplo ejeA;
    ejeA = new Ejemplo();

    ejeA.X = 5;
    Console.WriteLine("Valor {0}", ejeA.X);

    ejeA.Y = 10;
```

Error: la propiedad Y es de solo lectura.

Propiedades (Parte II)

- C# 2.0 permite definir diferentes modificadores de visibilidad para los bloques **get** y **set**.

```
class A
{
    string miPropiedad;ç
    public string MiPropiedad
    {
        get { return miPropiedad;}
        protected set { miPropiedad = value;}
    }
}
```

- Se puede configurar la visibilidad del bloque **get** o del bloque **set** de una cierta propiedad, pero no se puede cambiar la de ambos.

Métodos

- Uso de métodos
- Uso de parámetros
- Uso de métodos sobrecargados.
  - Declaración de métodos sobrecargados.
  - Signaturas de métodos sobrecargados.
  - Uso de métodos sobrecargados.

Uso de métodos

- Definición de métodos - sintaxis.
- Llamada a métodos.
- Devolución de valores (return).
- Variables locales.
- Métodos estáticos.

Definición de métodos

- Es un miembro de la clase que lleva a cabo una acción o calcula un valor.
- Tiene un nombre y contiene un bloque de código.
- Todos los métodos pertenecen a una clase.

```
class Alumno
{
    int NotaParcial;

    void EstablecerNota(int notaParcial)
    {
        NotaParcial = notaParcial;
    }

    float Promedio(int n1, int n2)
    {
        return (float)(n1+n2)/2;
    }
}
```

Definición de métodos - sintaxis

- Todo método debe devolver algún valor, si no devuelve nada se indica void. Si devuelve algo se debe indicar con la instrucción return <objeto> que debe coincidir con <tipoDevuelto>.

```
<tipoDevuelto> <nombreMétodo> (<parámetros>)
{
    <instrucciones>
}
```

Llamadas a métodos

- Una vez definido un método, se puede:
  - Llamar a un método desde dentro de la misma clase.
    - Se usa el nombre del método seguido de una lista de parámetros entre paréntesis.
  - Llamar a un método que está en una clase diferente
    - Hay que indicar al compilador cuál es la clase o instancia que contiene el método que se desea llamar.
    - El método llamado debe tener un modificador de acceso que permita la llamada.
  - Usar llamadas anidadas
    - Unos métodos pueden hacer llamadas a otros, que a su vez pueden llamar a otros métodos, y así sucesivamente.

Llamadas a métodos (Parte II)

- Si es un método de objeto

  <objeto>.<nombreMetodo>(<valoresParametros>)
- Si se invoca desde la misma clase a la que pertenece:

  <nombreMetodo>(<valoresParametros>)
- Si es un método de tipo (static)

  <tipo>.<nombreMetodo>(<valoresParametros>)

Devolución de valores (return)

- El método se debe declarar con un tipo que no sea void.
- Se añade una instrucción return con una expresión:
  - Fija el valor de retorno.
  - Se devuelve al llamador.
- Los métodos que no son void deben devolver un valor:

```
static int DosMasDos( )
{
    int a,b;
    a = 2;
    b = 2;
    return a + b;
}
```

```
int x;
x = DosMasDos( );
Console.WriteLine(x);
```

Variables locales

- Variables locales:
  - Se crean cuando comienza el método.
  - Son privadas para el método.
  - Se destruyen a la salida.
- Variables compartidas
  - Para compartir se utilizan variables de clase.
- Conflictos de ámbito
  - El compilador no avisa si hay conflictos entre nombres locales y de clase.

Métodos estáticos

- Van precedidos de la palabra static
- Métodos que pertenecen a la clase (Tipo), no a la instancia (objeto).
- Las variables que se utiliza dentro del método deben ser privadas del método o estáticas de la clase.

```
int x;
static int y;
static void Incrementa()
{
    x++; //Error x es miembro de objeto
    y=9; //Ok
}
```

Uso de parámetros

- Declaración y llamadas a parámetros.
- Paso por valor.
- Paso por referencia.
- Parámetros de salida.
- Uso de listas de parámetros de longitud variable.
- Normas para el paso de pámetros.

Declaración y llamadas a parámetros

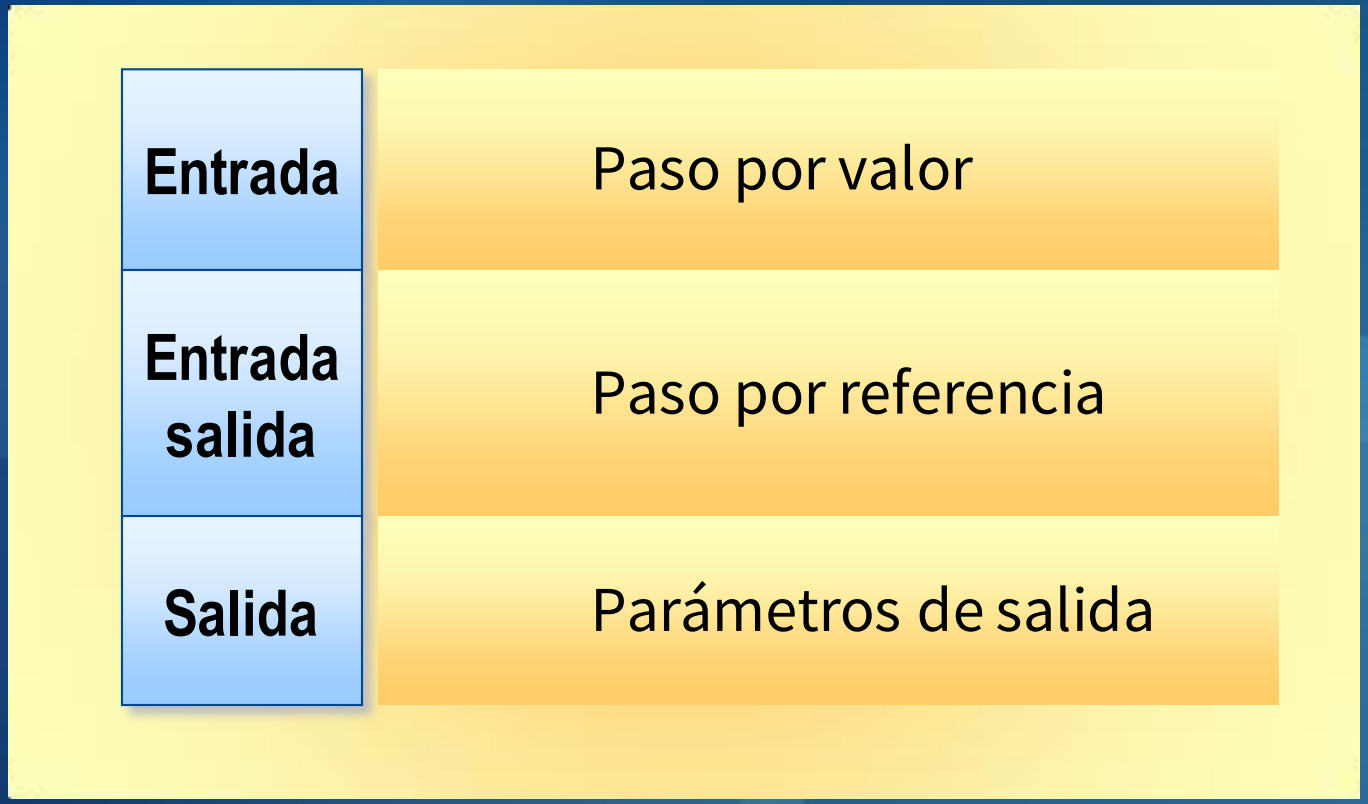
- Declaración de parámetros
  - Se ponen entre paréntesis después del nombre del método.
  - Se definen el tipo y el nombre de cada parámetro.
- Llamadas a métodos con parámetros
  - Un valor para cada parámetro.

```
static void  MethodWithParameters(int n, string y)
{ ... }

MethodWithParameters(2, "Hola, mundo");
```

Mecanismos de paso de parámetros

- Tres maneras de pasar parámetros.



Paso por valor

- Mecanismo predeterminado para el paso de parámetros:
  - Se copia el valor del parámetro.
  - Se puede cambiar la variable dentro del método.
  - No afecta al valor fuera del método.
  - El parámetro debe ser de un tipo igual o compatible.
  - Si el objeto es de tipo valor se pasa una copia.
  - Si el objeto es de tipo referencia se pasa una copia de la referencia del mismo.

```
static void SumaUno(int x)
         {x++; // Incrementar x
         Return;}

static void Main( )
{    int k = 6;
     SumaUno(k);
     Console.WriteLine(k); // Muestra el valor 6, no 7
}
```

Paso por referencia

- ¿Qué son los parámetros referencia?
  - Una referencia a una posición de memoria.
- Uso de parámetros referencia
  - Se usa la palabra clave ref en la declaración y las llamadas al método.
  - Los tipos y valores de variables deben coincidir.
  - Los cambios hechos en el método afectan al llamador.
  - Hay que asignar un valor al parámetro antes de la llamada al método.

Parámetros de salida

- ¿Qué son los parámetros de salida?
  - Pasan valores hacia fuera, pero no hacia dentro.
- Uso de parámetros de salida
  - Como **ref**, pero no se pasan valores al método.
  - Se usa la palabra clave out en la declaración y las llamadas al método.

```
static void OutDemo(out int p)
{
    // ...
}
int n;
OutDemo(out n);
```

Uso de listas de parámetros de longitud variable

- Se usa la palabra clave params.
- Se declara como tabla al final de la lista de parámetros.
- Siempre paso por valor.
- Solo puede tener una sola dimensión.
- Si no se conoce el tipo de dato utilizar object.

```
static long AddList(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = AddList(63,21,84);
}
```

Normas para el paso de parámetros

- Mecanismos
  - El paso por valor es el más habitual.
  - El valor de retorno del método es útil para un solo valor.
  - ref y/o out son útiles para más de un valor de retorno.
  - ref sólo se usa si los datos se pasan en ambos sentidos .
- Eficiencia
  - El paso por valor suele ser el más eficaz .

Uso de métodos sobrecargados

- Declaración de métodos sobrecargados.
- Signaturas de métodos sobrecargados.
- Uso de métodos sobrecargados.

Declaración de métodos sobrecargados

- Métodos que comparten un nombre en una clase.
  - Se distinguen examinando la lista de parámetros.

```
class EjemploSobrecarga
{
    static int Suma(int a, int b)
    {
        return a + b;
    }
    static int Suma(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Suma(1,2) + Suma(1,2,3));
    }
}
```

Firma de métodos

- Las firmas de métodos deben ser únicas dentro de una clase.
- Definición de signatura o firma.

Forman la definición de la signatura	No afectan a la signatura
Nombre del método Tipo de parámetro Modificador	Nombre de parámetro Tipo de retorno de método

Uso de métodos sobrecargados

- Conviene usar métodos sobrecargados si:
  - Hay métodos similares que requieren parámetros diferentes.
  - Se quiere añadir funcionalidad al código existente.
- No hay que abusar, ya que:
  - Son difíciles de depurar.
  - Son difíciles de mantener.

Herencia

- Derivación de clases.
  - Extensión de clases base.
  - Acceso a miembros de la clase base.
  - Llamada a constructores de la clase base.
- Métodos virtuales.
  - Definición.
  - Sustitución (override).

Extensión de clases base

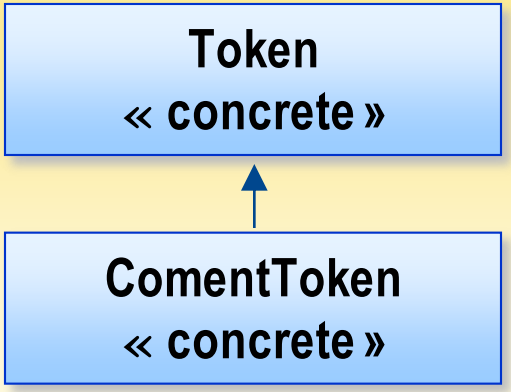
- Sintaxis para derivar una clase desde una clase base.

```
class Token
{
    ...
}
class ComentToken: Token
{
    ...
}
```

Clase derivada

Clase base

2 puntos



- Una clase derivada hereda la mayor parte de los elementos de su clase base.
- Una clase derivada no puede ser más accesible que su clase base.
- Solo se permite la herencia simple.
- La clase padre se denomina *clase base*, y la hija *clase derivada*.

Acceso a miembros de la clase base

```
class Token
{    ...
     protected string name;
}
class ComentToken: Token
{    ...
     public string Name()
     {
          return name;
     }
}
```

```
class Outside
{
     void Fails(Token t)
     {
          ...
          t.name = "S";      //Error
          ...
     }
}
```

- Los miembros heredados con protección están implícitamente protegidos en la clase derivada.
- Los miembros de una clase derivada sólo pueden acceder a sus miembros heredados con protección.
- En una struct no se usa el modificador de acceso protected.

Llamadas a constructores de la clase base

- Las declaraciones de constructores deben usar la palabra base.

```
class Token
{
   protected Token(string name) { ... }
   ...
}
class ComentToken: Token
{
   public ComentToken(string name): base(name) { }
   ...
}
```

- Una clase derivada no puede acceder a un constructor privado de la clase base.
- Se usa la palabra base para habilitar el ámbito del identificador.

Métodos virtuales

- Son útiles cuando se implementa herencia.
- Permite dar una nueva definición al método en las clases hijas.
- El método debe ir precedido de la palabra *virtual*.
- En la clase hija, si se desea sobreescribir el método se debe preceder al método de la palabra override.
- Si se precede de override un método en una clase hija y el método de la clase padre no va precedido de virtual, se produce un error de compilación.
- No se puede definir un método como virtual y override a la vez.
- No se pueden declarar como estáticos.
- No se pueden declarar como privados.

Definición de métodos virtuales

- Sintaxis: Se declara como virtual.

```
class Token
{
    ...
    public int LineNumber()
    { ...
    }
    public virtual string Name()
    { ...
    }
}
```

- Los métodos virtuales son polimórficos.

Sustitución de métodos (override)

- Sólo se sustituyen métodos virtuales heredados idénticos.

```
class Token
{   ...
    public int LineNumber( )   { ... }
    public virtual string Name( ) { ... }
}
class ComentToken: Token
{   ...
    public override int LineNumber( ) { ... }       //Error
    public override string Name( ) { ... }
}
```

- Un método override debe coincidir con su método virtual asociado.
- Se puede sustituir un método override. Un método override es virtual de manera implícita (no se puede declarar explícitamente virtual).
- No se puede declarar explícitamente un override como virtual. No se puede declarar un método override como static o private.

Temario

- Estructura de un programa C#.
- Característica de la sintaxis.
- Utilizar tipos de datos simpes.
- Estructuras de control.
- Manejo de excepciones.
- Arrays.
- POO en C#.
- Generics
  - Generics C# 2.0.
  - Creando y usando Generics.
  - Listas genéricas

Generics (C# 2.0)

- Permite que las clases, estructuras, interfaces, delegados y métodos sean parametrizados por el tipo de datos que van a almacenar y manipular.
- Es muy útil porque provee chequeo de tipos de datos en tiempo de compilación
- Requiere menos conversiones explícitas entre tipos de datos.

¿Porqué generics?

```
public class Stack
{
object[] items;
int count;
public void Push(object item)
{ … }
public object Pop()
{ … }
}
```

```
Stack stack = new Stack();
stack.Push( new Student() );
Student s = (Student)stack.Pop();
```

```
Stack stack = new Stack();
stack.Push( new Student() );
Student s = (Student)stack.Pop();
```

Creando y usando Generics

```
public class Stack<T>
{
T[] items;
int count;
public void Push(T item)
{ … }
public T Pop()
{ … }
}
```

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

```
Stack< Student > stack = new Stack< Student >();
stack.Push(new Student());
Student x = stack.Pop();
```

Listas Genéricas (C# 2.0)

- Es una colección de objetos que posee metodos para agregar, eliminar, buscar, acceder por un índice, etc.
- Es como si fuese un array dinámico.
- Están definidas dentro **System.Collections.Generic**
- **Los métodos mas comunes son Add, Insert, Remove, Item, Clear, Count**

List<T> Ejemplo

```
List<string> dinosaurs = new List<string>();
Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}
Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);
```