Tutorial C#

www.profmatiasgarcia.com.ar

El Lenguaje C#

- El último en una línea de evolución de los lenguajes derivados de C, que incluye C++ y Java
- Creada con .NET en mente, por tanto es el lenguaje ideal para el desarrollo en .NET
- C# introduce varias mejoras sobre C++ en las áreas de seguridad de datos, versionamiento, eventos y recolección de basura
- C# provee acceso a SO y COM APIs y soporta el modo unsafe que permite el uso de punteros como en C
- Más simple que C++ pero tan poderoso y flexible
- Totalmente orientado a objetos
  - Seguridad en tipos
  - Polimorfismo
  - Sobrecarga de operadores y métodos
  - Herencia

Ventajas de C#

- Seguridad de tipos
- Orientado a Objetos
  - Objetos <- Tipos básicos de datos

```
int Counter = 14;
Console.WriteLine( Counter.ToString() );
```

- Administración automática de memoria
- Sintaxis simplificada: operadores de referencia y namespaces:
  ., ->, :: sustituidos por ->
- XML: ¿Comentarios? ¡Documentación!
- COMponentes y componentes .NET
- Integración con otros lenguajes .NET
- Independencia de plataforma
  - .NET
  - Liberación de la especificación
  - MONO: Emulación en ambientes Linux

Estructura de un programa C# I

- Un programa en C# contiene:
  - Uno o más ficheros los cuales contienen:
    - Uno o más espacios de nombres que contienen:
      - Tipos de datos: clases, estructuras, interfaces, enumeraciones y delegates
- Si no se declara un namespace se asume el global por defecto
- Un programa ejecutable ha de contener obligatoriamente una función Main
  - `static void Main()`
  - `static int Main()`
  - `static void Main(string[] args)`
  - `static int Main(string[] args)`
- Para acceder a un tipo podemos usar un camino absoluto: `System.Console.WriteLine(...);` o: `using System; ...; Console.WriteLine(...);`

Estructura de un programa C# II

```
namespace N1 {
  class C1 {
    // ...
  }
  struct S1 {
    // ...
  }
  interface I1 {
    // ...
  }
  delegate int D1();
  enum E1 {
    // ...
  }
}
```

Identificadores

- Se usan para dar nombres a elementos de un programa como variables, constantes y métodos
- Consta de caracteres alfanuméricos y '_', es sensible a mayúsculas y minúsculas, debe comenzar con letra o '_'
- Si se quiere usar un identificador como palabra clave hay que usar como prefijo el carácter '@':
  - `Object @this; //` prevenir conflicto con "this"

Palabras reservadas

abstract	class	event	if	new	readonly	struct	unsafe
as	const	explicit	implicit	null	ref	switch	ushort
base	continue	extern	in	object	return	this	using
bool	decimal	false	int	operator	sbyte	throw	virtual
break	default	finally	interface	out	sealed	true	volatile
byte	delegate	fixed	internal	override	short	try	void
case	do	float	is	params	sizeof	typeof	while
catch	double	for	lock	private	stackalloc	uint	
char	else	foreach	long	protected	static	ulong	
checked	enum	goto	namespace	public	string	unchecked	

Tipos de datos

Tipo de C#	Tipo en la plataforma .NET	¿Signo?	Bytes empleados	Posibles valores
sbyte	System.Sbyte	Si	1	-128 a 127
short	System.Int16	Si	2	-32768 a 32767
int	System.Int32	Si	4	-2147483648 a 2147483647
long	System.Int64	Si	8	-9223372036854775808 a 9223372036854775807
byte	System.Byte	No	1	0 a 255
ushort	System.Uint16	No	2	0 a 65535
uint	System.UInt32	No	4	0 a 4294967295
ulong	System.Uint64	No	8	0 a 18446744073709551615
float	System.Single	Si	4	Aproximadamente ±1.5 x 10-45 a ±3.4 x 1038 con 7 dígitos significativos
double	System.Double	Si	8	Aproximadamente ±5.0 x 10-324 a ±1.7 x 10308 con 15 o 16 dígitos significativos
decimal	System.Decimal	Si	12	Aproximadamente ±1.0 x 10-28 a ±7.9 x 1028 con 28 o 29 dígitos significativos
char	System.Char	-	2	Cualquier carácter de Unicode (16 bit)
bool	System.Boolean	-	1 / 2	true o false

Tipos de datos - conversión

- La conversión entre tipos de datos se puede hacer de forma explícita utilizando una conversión de tipos (cast); en algunos casos, se permiten conversiones implícitas.
- Una conversión de tipos invoca de forma explícita al operador de conversión de un tipo a otro.
- La palabra clave explicit declara un operador de conversión de tipos definido por el usuario que se debe invocar con una conversión de tipos.
- Se puede realizar una conversión comprobada en C# con la palabra clave checked delante del operador de conversión explícita.
- Tipo_de_Dato.Parse(string) convierte un valor numérico ingresado como string al tipo de dato especificado. Ej int32.Parse(num) siendo num = ”145”.
- Dato.ToString() convierte un dato de cualquier tipo a una cadena de caracteres. Tengo num = 145 como int, con num.ToString() paso a cadena.

Conversión Implícita

```
int num = 2147483647;
long bigNum = num;
```

Conversión Explícita

```
double x = 1234.7;
int a;
a = (int)x;
```

System.Convert

```
String texto = “3,1416“;
Double nro =
System.Convert.ToDouble(texto);
Bool dato =
System.Convert.ToBoolean(nro);
```

Variables I

- Una variable en C# representa la localización en memoria donde una instancia de un tipo es guardada
- Es simplemente una capa encima del sistema de tipos independiente del lenguaje de .NET (CTS)
- Recordar distinción entre tipos por valor y por referencia
  - Tipos por valor son tipos simples como 'int', 'long' y 'char'
  - Los objetos, strings y arrays son ejemplos de tipos por referencia

Variables II

- Las variables por valor pueden declararse e iniciarse:

```
bool bln = true;
byte byt1 = 22;
char ch1='x', ch2='\u0066'; // Unicode para 'a'
decimal dec1 = 1.23M;
double dbl1=1.23, dbl2=1.23D;
short sh = 22;
int i = 22;
long lng1 =22, lng2 =22L; // 'L' long
sbyte sb = 22;
float f=1.23F;
ushort us1=22;
uint ui1=22, ui2=22U; // 'U' unsigned
ulong ul1 =22, ul2=22U, ul3=22L, ul4=2UL;
```

Variables III

- Los valores por referencia son creados con la palabra clave new:

```
object o = new System.Object();
```

- Strings se pueden inicializar directamente:

```
string s = "Hola"; // usan caracteres Unicode de 2 cars
```

- C# soporta secuencias de escape como en C:

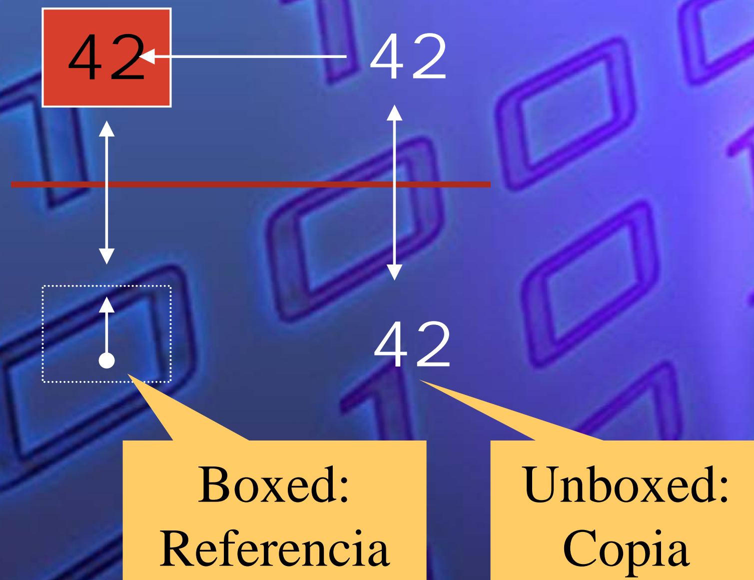
```
string s1 = "Hola\n"; // salto de línea
string s2 = "Hola\tque\ttal"; // tabulador
```

- Como las sentencias de escape comienzan con '\', para escribir este carácter hay que doblarlo, o usar '@':

```
string s3 = "c:\\WINNT";
string s4 = @"C:\WINNT";
```

Boxing y Unboxing

- Tipos por valor pueden ser "boxed" y "unboxed"
- Boxing permite que los tipos por valor se traten por referencia, o sea, que un tipo por valor sea convertido a un tipo objeto y viceversa.
- Basado en el tipo **object** implicito de todos los tipos.
- Ponga el valor en una caja y referénciela



En **Boxing** un valor se refiere a la conversión implícita de cualquier tipo de valor al tipo objeto. Cuando un tipo de valor es boxed se asigna espacio a una instancia de objeto y el valor del value type es copiado al nuevo objeto.

Boxing y Unboxing

```
using System;
class App{
      public static void Main(){
       int iEdad = 33;
       object oNumero = iEdad; //Box
       int iNumero = (int)oNumero; //Unbox
       //cast necesario porque oNumero podría contener cualquier
tipo de objeto } }
```

Al asignar el valor de la variable entera iEdad a una variable objeto se realiza internamente una operación boxing, donde el valor de la variable iEdad es copiado al objeto oNumero, entonces las variables entera y objeto existen en la pila pero los valores de los objetos residen en el área o espacio asignado, lo que implica que los valores son independientes y no hay una liga entre ellos.

Unboxing es un mecanismo de una operación explícita, por lo que es necesario indicar al compilador que tipo de valor deseamos extraer de un objeto, al realizar la operación Unboxing C# checa que el value type que se requiere este almacenado en la instancia del objeto, si la verificación es exitosa el valor es Unboxing.

Constantes

- C# provee el modificador const que se puede usar para crear constantes de programas:

```
const int min = 1;
const int max = 100;
Const int range = max – min;
```

Arrays I

- Los arrays son tipos por referencia y sus índices comienzan en 0:

```
string[] a;
```

- Desarrollados a partir de la clase System.Array
- El tipo de datos viene dado por string[], el nombre del array es una referencia al array
- Para crear espacio para los elementos usar:

```
string[] a = new string[100]
```

- Los arrays se pueden inicializar directamente:

```
string[] a1 = {"gato", "perro", "caballo"}
int[] a2 = {1, 2, 3};
```

- Puede haber arrays multidimensionales y arrays de arrays:

```
string[,] ar = {{"perro", "conejo"}, {"gato", "caballo"}}
int[][] matrix;
object[] ar  = {3, "cat", 2.45}; // Los elementos de un array mismo tipo
string animal=(string)ar[1];
```

Arrays II

- Arreglos de una dimensión

```
int[] i = new int[100];        // Arreglo de 100 enteros
int[] i = new int[2];  //Llenado del arreglo
i[0] = 1;
i[1] = 2;
int[] i = new int[] {1,2};     // Declaración y asignación
int[] i = {1,2};
```

- Arreglos rectangulares

```
// Declaración de una matriz de 2x3
int[,] squareArray = new int[2,3];
//Declaración y asignación de una matriz 2x3
int[,] squareArray = {{1, 2, 3}, {4, 5, 6}};
//Algunos métodos inherentes de la clase: System.Array
squareArray.GetLowerBound(1); // Nivel inferior
squareArray.GetUpperBound(1); // Nivel superior
squareArray.GetLength(0);     // Tamaño del arreglo
```

Expresiones y Operadores I

- Las expresiones en C# son muy similares a las de C y C++
- Operadores aritméticos:
  - + , Suma unaria , +a
  - - , Resta unaria, -a
  - ++, Incremento, ++a o a++
  - –, Decremento, –a o a–
  - +, Suma, a+b
  - -, Resta, a-b
  - *, Multiplicación, a*b
  - /, División, a/b
  - %, Resto, a%b

Expresiones y Operadores II

- Operadores relacionales:
  - == , Igualdad , a==b
  - != , Inigualdad, a!=b
  - <, Menor que, a<b
  - <=, Menor o igual, a<=b
  - >, Mayor que, a>b
  - >=, Mayor que o Igual a, a>=b
- Los programadores de Visual Basic deben notar que se usa ‘==‘ en vez de ‘=‘ para igualdad, ‘!=‘ para desigualdad en vez de ‘<>’ e ‘=‘ para asignación

Expresiones y Operadores III

- Operadores relacionales:
  - ! , Negación , !a
  - & , And binario, a&b
  - |, Or binario, a|b
  - ^, Or exclusivo, a^b
  - ~, Negación binaria, ~a
  - &&, And lógico, a&&b
  - ||, Or lógico, a||b
- Operadores de manipulación de bits:
  - int i1=32;
  - int i2=i1<<2; // i2==128
  - int i3=i1>>3; // i3==4

Expresiones y Operadores IV

- Operadores de asignación (para a==3 y b==7):
  - = , a=b , 7
  - += , a+=b, 10
  - -=, a-=b, -4
  - *=, a*=b, 21
  - /=, a/=b, 0
  - %=, a%=b, 3
  - |=, a|=b, 7
  - >>=, a>>=b, 0
  - <<=, a<<=b, 384

Expresiones y Operadores V

- Otros operadores:
  - min=a<b ? a:b; // if a<b min=a else min=b;
  - . → para acceso a miembros, e.j. args.Length
  - () → para conversión de tipos
  - [] → como índice de arrays, punteros, propiedades y atributos
  - new → para crear nuevos objetos
  - typeof → para obtener el tipo de un objeto
  - is → para comparar el tipo de un objeto en runtime
  - sizeof → para obtener el tamaño de un tipo en bytes
  - * → para obtener la variable a la que apunta un puntero
  - ->, p->m es lo mismo que (*).m
  - & → devuelve la dirección de un operando

Operadores

Categoría	Operadores
Aritméticos	+ - * / %
Lógicos	& | ^ ! ~ && || true false
Concatenación	+
Incremento/Decremento	++ –
Corrimiento	<< >>
Relacionales	== != < > <= >=
Asignación	= += -= *= /= %= &= |= ^= <<= >>=
Acceso a miembros	.
Índices	[]
Cast	()
Condicional	?:
Concatenación y remoción	+ -
Creación de objetos	new
Información de tipos	is sizeof typeof
Desreferencia y dirección	* -> [] &

www.profmatiasgarcia.com.ar

Sobrecarga de operadores

- La mayor parte pueden ser redefinidos
  - Aritméticos, relacionales, condicionales, y lógicos
- No permitido para
  - Operadores de asignación
  - Operadores especiales (sizeof, new, is, typeof)
- Ejemplos:

```
Public static Total operator +(int cantidad, Total t)
  {
    t.total += cantidad;
  }

public static bool operator == (Value a, Value b)
  {
    return a.Int == b.Int
  }
```

Sentencias I

- Las sentencias pueden ocupar más de una línea y deben terminarse con un ;
- Grupos de sentencias se pueden agrupar en bloques con { y }
- E.j:

```
int i, j;
// un sentencia simple
i=1;
// un bloque de sentencias
{
  j=2;
  i=i+j;
}
```

Sentencias II

- if, sirve para saltar en base a una condición:

```
if (i<5) // una sentencia sólo parte if
    System.Console.WriteLine("i < 5");

if (i<5) // una sentencia con parte if y else
  System.Console.WriteLine("i<5");
else
  System.Console.WriteLine("i>=5);
if (i<5) { // bloque de sentencias con parte if y else
  System.Console.WriteLine("i<5");
  System.Console.WriteLine("i es más pequeño");
} else {
  System.Console.WriteLine("i>=5");
  System.Console.WriteLine("i NO es más pequeño");
}
```

Sentencias III

- do, sirve para repeticiones de sentencia que se ejecutan al menos una vez:

```
int i=1;
do
    System.Console.WriteLine(i++);
While (i<=5);
```

- while, para repeticiones de 0 a N veces

```
int i=1;
While (i<=5)
    System.Console.WriteLine(i++);
```

Sentencias IV

- for, se usa cuando el número de repeticiones se conoce a priori

```
for (int i=1; i<=5; i++)
    System.Console.WriteLine(i); // Visualiza digitos 1 a 5
```

  - La primera expresión es de inicialización, declara un entero
  - La segunda la condición del bucle
  - La tercera es la expresión del iterador
- Un bucle infinito se podría codificar como:

```
for (;;) {
    // bucle infinito
    ...
}
```

- Se pueden insertar múltiples expresiones:

```
for (int i=1, j=2; i<=5; i++, j+=2) {
    System.Console.WriteLine("i=" + i + ", j=" + j);
}
```

Sentencias V

- continue, se usa para saltar el resto de la iteración y comenzar la siguiente

```
for (int i=1; i<=5; i++) {
   if (i==3)
     continue;
   System.Console.WriteLine(i);
}
```

- break se usa para salir de un bucle:

```
for (int i=1; i<=5; i++) {
   if (i==3)
     break;
   System.Console.WriteLine(i);
}
```

Sentencias VI

- Switch sirve para seleccionar entre múltiples opciones posibles

```
uint i=2;
switch(i) {
   case 0:
     goto case 2; // para permitir varios casos usar goto
   case 1:
     goto case 2;
   case 2:
     System.Console.WriteLine("i<3");
     break; // el uso de goto o break es obligatorio
   case 3:
     System.Console.WriteLine("i==3"),
     break;
   default:
     System.Console.WriteLine("i>3");
     break;
}
```

Sentencias VII

- **foreach representa un interesante caso para iterar sobre un array:**

```
int[] arr = {2, 4, 6, 8};
foreach (int i in arr) // Visualizará 2, 4, 6 y 8
    System.Console.WriteLine(i);
```

- **return termina la ejecución del método actual y devuelve control al invocador**

```
class Add { // devuelve resultado de sumar 2 enteros
    public static void Main() {
      System.Console.WriteLine("2+3=" + add(2,3));
    }
    private static int add(int i, int j) {
      return i+j;
    }
}
```

Sentencias VIII

- goto transfiere control a una instrucción que contiene una etiqueta

```
using System;
public class EjemploGoto {
  public static void Main(string [] args) {
    if (args.Length == 0) {
      Console.WriteLine("No se han pasado args");
      goto end;
    }
    Console.WriteLine("El primer argumento es " + args[0]);
    end: return;
  }
}
```

- throw sirve para lanzar una excepción

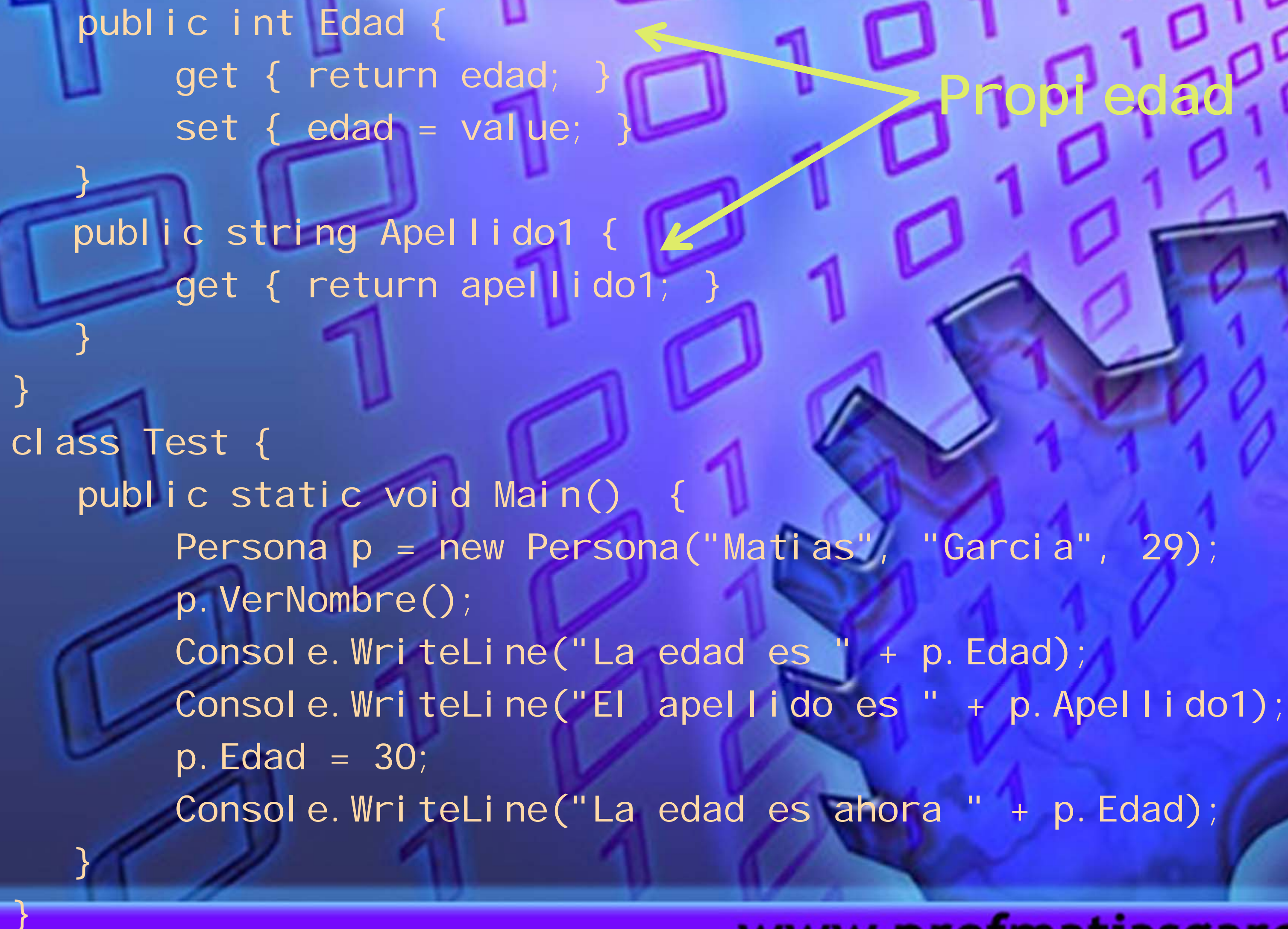
```
if (val > max) throw new Exception("Valor excede máximo");
```

Clases y estructuras I

- El bloque principal de construcción de bloques en C# es class

```
using System;
class Persona {
   // campos
   string apellido1, nombre; // private string apellido1, nombre;
   int edad;
   // constructor
   public Persona(string apellido1, string nombre, int edad) {
        this.apellido1 = apellido1;
        this.nombre = nombre;
        this.edad = edad;
   }
   // método
   public void VerNombre() {
        Console.WriteLine(this.nombre + " " + this.apellido1);
   }
```

Clases y estructuras II



```
    public int Edad {
        get { return edad; }
        set { edad = value; }
    }
    public string Apellido1 {
        get { return apellido1; }
    }
}
class Test {
    public static void Main()  {
        Persona p = new Persona("Matias", "Garcia", 29);
        p.VerNombre();
        Console.WriteLine("La edad es " + p.Edad);
        Console.WriteLine("El apellido es " + p.Apellido1);
        p.Edad = 30;
        Console.WriteLine("La edad es ahora " + p.Edad);
    }
}
```

Propiedad

Clases y estructuras III

- Definimos una clase usando la palabra reservada class seguida del nombre de la clase y el cuerpo de la clase entre { y }
- Los miembros que una clase puede contener son:
  - Campos
  - Métodos
  - Propiedades
- El constructor (Persona()) tiene el mismo nombre que la clase y no devuelve ningún valor
  - La palabra clave this se usa para referenciar a la instancia de la clase actual
- C# soporta punteros cuando funciona en modo inseguro, se usa la palabra clave unsafe para preceder a este tipo de datos

Clases y estructuras IV

- C# provee propiedades para recuperar (get) y modificar (set) campos de una clase:

```
public int Height {
    get { return height; }
    set {
        if (value > 0)
            height = value;
        else
            throw new ArgumentOutOfRangeException("Altura debe ser
1 o mayor"); }
}
```

  - Dentro de un atributo set C# provee la variable value
- Tanto los campos, como atributos y métodos pueden ir acompañados de modificadores (public). Si se omite un modificador entonces los miembros son privados
  - La convención es hacer que miembros privados empiecen por minúscula, mientras que todos los demás lo hagan por mayúscula

Clases y estructuras V

- Un miembro static puede ser accedido sin crear una instancia de una clase (se suelen usar para guardar valores globales)

```
class Persona {
    public static int MinimumAge = 18;
    …
}
int age = Persona.MinimumAge;  // accedemos a MinimumAge usando nombre
    clase
```

- Las clases se pueden anidar:

```
class C1 {
    int i, j;   string s;
    void m() {  // …   }
    class c2 {
        // …
    }
}
```

Clases y estructuras VI

- Se pueden crear clases ligeras usando una estructura (struct)
  - Las estructuras son tipos por valor y se guardan en la pila
  - No se permite la herencia
  - Más eficientes, no requieren referencias y no hay garbage collection

```
struct Point {
    public int X, Y;
    public Point(int x, int y) {
        X = x;
        Y = y;
    }
    void MoveBy(int dX, int dY)
       { X+=dX; Y+=dY; }
  }
  ...
}
```

Herencia I

- C# nos permite diseñar una clase que usa herencia para extender otra ya existente
- C# como Java sólo soporta herencia simple
- Ejemplo herencia simple:

```
using System;
class Persona {
    protected string nombre, apellido1;
    public Persona(string nombre, string apellido1) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
    }
    public void Saludar() {
        Console.WriteLine("¡Hola " + this.nombre + " " + this.apellido1
    + "!");
    }
}
```

Herencia II

```
class Hombre: Persona {
    public Hombre(string nombre, string apellido1): base(nombre,
    apellido1) {}
    public new void Saludar() {
        Console.WriteLine("¡Hola " + this.apellido1 + "!");
    }
}
class Test {
    public static void Main() {
        Persona p = new Persona("Matias", "Garcia");
        p.Saludar(); // Visualizar ¡Hola Matias Garcia!
        Hombre h = new Hombre("Juan", "Perez");
        h.Saludar(); // Visualizar ¡Hola Perez!
    }
}
```

Herencia III

- La herencia representa una relación is-a
- Al cambiar la accesibilidad de los campos nombre y apellido a protected, ahora son accesibles en las clases que los derivan

```
class Hombre: Persona {
    …
}
```

- La única diferencia entre Hombre y Persona es la implementación del método Saludar.
- La palabra clave base se usa para referenciar al objeto padre
- La palabra clave new indica al compilador que queremos sobre-escribir el método Saludar del padre (Persona).

Métodos virtuales I

- Una de las ventajas de la herencia que podemos usar una clase base para referenciar a una instancia de una clase derivada

```
Persona p = new Hombre("Matias", "Garcia");
p.Saludar(); // problema, invoca a Saludar de Persona y no de
    Hombre
```

- La solución a esto son métodos virtuales a través de las palabras claves virtual (a usarse en el método de la clase padre) y override (a usarse en el método de la clase hijo)

Métodos virtuales II

```
using System;
class Persona {
    protected string nombre, apellido1;
    public Persona(string nombre, string apellido1) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
    }
    public virtual void Saludar() {
        Console.WriteLine("¡Hola " + this.nombre + " " +
    this.apellido1 + "!");
    }
}
```

Métodos virtuales III

```
class Hombre: Persona {
    public Hombre(string nombre, string apellido1): base(nombre,
    apellido1) {}
    public override void Saludar() {
        Console.WriteLine("¡Hola " + this.apellido1 + "!");
    }
}
class Test {
    public static void Main() {
        Persona p1 = new Hombre("Matias", "Garcia");
        p1.Saludar(); // Visualizar ¡Hola Garcia!
        Persona p2 = new Persona("Juan", "Perez");
        p2.Saludar(); // Visualizar ¡Hola Juan Perez!
    }
}
```

Clases abstractas I

- Supongamos que estamos interesados en modelar hombres y mujeres, pero no personas perse → No queremos permitir la creación de objetos Persona directamente, pero queremos permitir la creación de objetos Hombre y Mujer directamente.
- Usamos abstract delante de Persona para evitar que esta clase se pueda instanciar
- Usamos abstract en el método Saludar de Persona para indicar que las clases que heredan deben sobreescribirlo

Clases abstractas II

```
using System;
abstract class Persona {
   protected string nombre, apellido1;
   public Persona(string nombre, string apellido1) {
      this.nombre = nombre;
      this.apellido1 = apellido1;
   }
   abstract public void Saludar();
}
```

Clases abstractas III

```
class Hombre: Persona {
    public Hombre(string nombre, string apellido1): base(nombre,
    apellido1) {}
    public override void Saludar() {
        Console.WriteLine("¡Hola Señor " + this.apellido1 + "!");
    }
}
class Mujer: Persona {
    public Mujer(string nombre, string apellido1): base(nombre,
    apellido1) {}
    public override void Saludar() {
        Console.WriteLine("¡Hola Señorita " + this.apellido1 + "!");
    }
}
```

Clases abstractas IV

```
class Test {
    public static void Main() {
        Hombre h = new Hombre("Matias", "Garcia");
        h.Saludar(); // Visualizar ¡Hola Señor Garcia!
        Mujer m = new Mujer("Angie", "Citterio");
        m.Saludar(); // Visualizar ¡Hola Señorita Artaza!
    }
}
```

- **Para deshabilitar la herencia se puede marcar una clase como sealed, resultando en un error de compilación si se intenta derivar de ella**

```
sealed class Persona {
    …
}
```

Métodos I

- Los métodos aceptan parámetros y devuelven un resultado

```
int Add(int x, int y) {
    return x+y;
}
```

- Los parámetros x e y se pasan por valor, se recibe una copia de ellos
- Si queremos modificar dentro de una función un parámetro y que el cambio se refleje en el código de invocación de la función, usaremos ref, tanto en la declaración del método como en su invocación:

```
void incrementar(ref int i, int valor)
    {
    i = valor;
}

void Increment(out int i) {
    i = 3;
}
```

```
…
int i; int valor = 5;
Increment(out i);
// i == 3
incrementar(ref i, valor)
// i pasa de 3 a valer 5
```

Métodos II

- Mientras ref se usa para modificar el valor de una variable, si el método asigna el valor inicial a la variable habrá que usar out:
- Para pasar un numero variable de parámetros se usa params:

```
class Adder {
    int Add(params int[] ints) {
        int sum=0;
        foreach(int i in ints)
            sum+=i;
        return sum;
    }
    public static void Main() {
        Adder a = new Adder();
        int sum = a.Add(1, 2, 3, 4, 5);
        System.Console.WriteLine(sum); // visualiza "15"
    }
}
```

Métodos III

- Métodos de sobrecarga, es decir, métodos con el mismo nombre, con firmas (número y tipo de parámetros) diferentes:

```
class Persona {
    string nombre, apellido1, apellido2;
    public Persona(string nombre, string apellido1) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
    }
    public Persona(string nombre, string apellido1, string
    apellido2) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
        this.apellido2 = apellido2;
    }
}
Persona p1 = new Persona("Matias", “Gonzalez");
Persona p2 = new Persona("Matias", “Gonzalez", "Artaza");
```

Modificadores de acceso

- Los modificadores de acceso controlan la visibilidad de los miembros de una clase
  - `private`, sólo código dentro de la misma clase contenedora tiene acceso a un miembro privado. Es el modo de acceso por defecto.
  - `public` , visible a todos los usuarios de una clase
  - `protected`, miembros accesibles tanto por dentro de la clase como en clases derivadas
  - `internal` , miembros accesibles sólo dentro de un assembly
  - `protected internal` , permite acceso protected e internal
  - `sealed`, no se puede usar como clase base de una jerarquía

Excepciones I

- Modo recomendado para manejar errores excepcionales en C#
- Todas las excepciones definidas en la FCL derivan de System.Exception
  - System.Exception define una serie de propiedades comunes a todas las excepciones:
    - Message: contiene un mensaje de error indicando que ha ido mal
    - StackTrace: detalla los detalles de la llamada que produjo la excepción
- Para distinguir entre excepciones lanzadas por la framework y excepciones generadas por aplicaciones, estas últimas se recomienda deriven de System.ApplicationException

Excepciones II

- ArgumentNullException → **una referencia nula es pasada como argumento**
- ArgumentOutOfRangeException → **nos hemos salido de rango, e.j. entero demasiado grande**
- DividedByZeroException
- IndexOutOfRangeException → **se usa un índice inválidado del array**
- InvalidCastException
- NullReferenceException → **se intenta invocar un método en un objeto que es null**
- OutOfMemoryException

Excepciones III

```
using System;
class Persona {
   string nombre, apellido1;
   int edad;
   public Persona(string nombre, string apellido1, int edad) {
      this.nombre = nombre;
      this.apellido1 = apellido1;
      this.edad = edad;
      if (edad < 18) // La edad debe ser mayor que 18 sino
   excepción
         throw new Exception("ERROR: Persona debajo edad legal");
      this.edad = edad;
   }
}
```

Excepciones IV

```
class Test {
    public static void Main() {
        try {
            Persona p = new Persona("Matias", "Garcia", 12);
        } catch (Exception e) { // capturar excepción lanzada
            Console.WriteLine(e.Message);
        }
}
```

- Se puede incluir un bloque finally también, para asegurarnos que recursos abiertos son cerrados:

```
try {
    ...
} catch {
    ...
} finally {
    ...
}
```

Enumeraciones

- La palabra clave enum puede usarse para definir un conjunto ordenado de tipos integrales:

```
enum Modalidad {
    Treboles=1, Rombos, Corazones, Picas
}
class Enums {
    public static void Main() {
        Modalidad m = Modalidad.Treboles;
        System.Console.WriteLine(m); // Visualiza Treboles
    }
}
```

- Las enumeraciones por defecto empiezan de 0 a no ser que se sobreescriba esto como en el ejemplo

Delegates y Eventos I

- Un delegate es una referencia a un método, similar a un puntero a función en C++, se suelen usar para implementar manejadores de eventos y callbacks:

```
using System;
public delegate void Callback(string name);
public class DelgHola {
   public static void Main(string[] args) {
      string nombre = "Amigo";
      if (args.Length > 0) nombre = args[0];
      Callback cb = new Callback(diHola);
      cb(nombre);
   }
   private static void diHola(string nombre) {
      Console.WriteLine("Hola, {0}", nombre);
   }
}
```

Delegates y Eventos II

- Toda aplicación gráfica usa eventos para notificar cambios (botón pulsado)
- Los eventos son manejados por un manejador de eventos
- Se pueden crear manejadores propios o usar los provistos por FCL (Framework Class Library) como System.EventHandler
- Vamos a crear un reloj usando la clase System.Timers.Timer y el delegate System.Timers.ElapsedEventHandler
- El objeto Reloj creará un objeto System.Timers.Timer y registrará un manejador de evento para el evento Timer.Elapsed

```
ElapsedEventHandler tickHandler = new ElapsedEventHandler(Tic);
Timer.Elapsed += tickHandler;
```

- El operador += se usa para añadir una instancia del delegate al evento, se pueden añadir varios manejadores de eventos porque ElapsedEventHandler es un delegate multicast.

Delegates y Eventos III

```
using System.Timers;
Using System.Threading;
public class Reloj {
    public Reloj(int intervalo) {
        this.totalTiempo = 0;
        this.intervalo = intervalo;
        System.Timers.Timer timer = new System.Timers.Timer();
        ElapsedEventHandler tickHandler = new ElapsedEventHandler(Tic);
        timer.Elapsed += tickHandler;
        timer.Interval = intervalo * 1000;
        timer.Enabled = true;
    }
    public void Tic(object source, ElapsedEventArgs e) {
        this.totalTiempo += this.intervalo;
        Console.WriteLine("{0}: {1} segundo tic, tiempo pasado: {2}",
    source, this.intervalo, this.totalTiempo);
    }
    private int intervalo;
    private int totalTiempo;
}
```

Delegates y Eventos IV

```
public class MyReloj {
    public static void Main() {
        Reloj c1 = new Reloj(3);
        Reloj c2 = new Reloj(1);
        Console.WriteLine("Reloj en funcionamiento, usar Ctrl-c
    para acabar");
        while (true) Thread.Sleep(500);
    }
}
```

Interfaces I

- Las interfaces en C# permiten especificar un contrato que una clase que lo implementa ha de cumplir
- Clases o estructuras en C# pueden implementar varias interfaces
- Las interfaces IComparable e IEnumerable se usan frecuentemente en .NET
  - IComparable: interfaz a implementar para definir una comparación específica a un tipo de datos
  - Para poder aplicar un foreach a una colección es necesario que la colección implemente la interfaz `System.Collections.IEnumerator`

Interfaces II

```
using System;
interface ISaludo {
   void Saludar();
}
class Persona: ISaludo {
   protected string nombre, apellido1;
   public Persona(string nombre, string apellido) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
   }
   public void Saludar() {
        Console.WriteLine("Hola " + this.nombre + " " +
   this.apellido1);
   }
   public static void Main() {
        Persona p = new Persona("Matias", "Garcia");
        p.Saludar();
   } }
```

Directivas de preprocesador I

- C# define algunas directivas de procesador de C++:
  - `#define/ #undef` → define/borra un símbolo
  - `#if-#elif-#else-#endif` → evalua un símbolo y actúa de acuerdo a ello
  - `#warning` → advertencia
  - `#error` → genera un error desde una posición del código
  - `#region/#endregion` → usado por Visual Studio.NET para marcar inicio/fin de región
- Como ejemplo vamos a ilustrar como introducir código opcional para propósito de depuración, usando directivas #define e #if.
- En vez de incluir #define DEBUG en el código, pasaremos en tiempo de compilación un parámetro al programa creado:
  - `csc /define:DEBUG preproc.cs`

Directivas de preprocesador II

```
class PreProc {
    public static void Main() {
        System.Console.Write("Hola, ¿cómo te llamas?");
        string nombre = System.Console.ReadLine();
        #if DEBUG
            System.Console.WriteLine("Usuario introducido: " +
    nombre);
        #endif
        if (nombre.Equals(""))
            nombre = "Amigo";
        System.Console.WriteLine("Hola, " + nombre);
    }
}
```

Threading I

- Multithreading crea el efecto de varias tareas ejecutándose simultáneamente
- El módulo System.Threading ofrece amplio suporte para ello.
- Vamos a usar el delegate ThreadStart y la clase Thread para implementar el reloj antes ilustrado

```
using System;
using System.Threading;
public class ThreadReloj {
   public static void Main() {
        // crear delegate del thread
        ThreadStart relojThread = new ThreadStart(empiezaReloj);
        Thread t = new Thread(relojThread);
        Console.WriteLine("empezando el thread …");
        t.Start();
        t.Join();
        Console.WriteLine("thread parado");
   }
```

Threading II

```
private static void empiezaReloj() {
    int timeElapsed = 0;
    Console.WriteLine("ejecutando nuevo thread …");
    for (int i=0; i<5; i++) {
        Thread.Sleep(1000);
        Console.WriteLine("Tic: " + (++timeElapsed));
    }
  }
}
```

Webgrafía & Licencia:

- Textos tomados, corregidos y modificados de diferentes páginas de Internet, tutoriales y documentos.
- Este documento se encuentra bajo Licencia Creative Commons 2.5 Argentina (BY-NC-SA), por la cual se permite su exhibición, distribución, copia y posibilita hacer obras derivadas a partir de la misma, siempre y cuando se cite la autoría del **Prof. Matías E. García** y sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.
- Autor:

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar