Hada T0
MonoDevelop y C#

Un IDE libre y de código abierto para C# y otros lenguajes de programación.

Objetivos del tema

- Conocer Monodevelop y saber usarlo
- Conocer y saber usar las herramientas de la línea de órdenes que emplea Monodevelop.
- Conocer las direcciones web asociadas a esta aplicación para ampliar nuestro conocimiento sobre ella.

¿Qué es MonoDevelop?

- Se trata de un IDE ligero multiplataforma (*Linux, Windows y Mac OS*).
- Permite escribir código en diversos lenguajes de programación (C#, F# (1 y 2), C/C++, etc…).
- Dispone de completado de código, plantillas de código y ocultación de bloques de código.
- Tiene depurador a nivel de código fuente así como diseñador de interfaces de aplicaciones de escritorio integrados.
- Es de código abierto y está escrito en C#.
- Permite abrir y trabajar con *soluciones* creadas con VisualStudio en Windows, tanto de escritorio como de ASP.NET.

Aspectos relevantes

- Características más importantes de este IDE.
- Cómo crear una solución.
  - Las *soluciones* son agrupaciones de *proyectos*.
  - Permite abrir y trabajar con archivos de *soluciones* creados con VisualStudio. Estos ficheros están en el formato conocido como MSBuild.

Xamarin Studio

- Se trata de un IDE basado en MonoDevelop.
- Dispone de una serie de características adicionales que permiten desarrollar con este IDE proyectos para iOS y Android.
- En febrero de 2016 Microsoft compró Xamarin.

¿Es necesario MonoDevelop?

- Para proyectos *grandes* es aconsejable, para proyectos sencillos... no es imprescindible.
- Las herramientas para el terminal (CLI) permiten automatizar tareas (clean, build, test, etc…).
- Los desarrolladores de MonoDevelop lo saben y nos proporcionan dichas herramientas para usar desde el intérprete de órdenes:
  - **`mcs`**: Es el compilador de `C#`, `man mcs`.
  - **`mono`**: Es la máquina virtual de `.NET`, `man mono`.
  - **`mdtool`**: Es la herramienta que permite ejecutar *sub-componentes* de MonoDevelop desde la línea de órdenes: `man mdtool`.
  - **`xbuild`**: Es la herramienta que permite desde la línea de órdenes construir proyectos en formato `MSBuild` (creados con `VisualStudio` o `MonoDevelop`), `man xbuild`.
  - **`csharp`**: Es un *REPL* para `C#` (con auto-completado), `man csharp`.
  - **`al`**: Es el enlazador del proyecto mono, `man al`.

Breve introducción a C# I

- Es un lenguaje de la familia de C, muy parecido a C++ y a Java.
- Es orientado objetos.
- Dispone de tipos de valor que admiten valores NULL, enumeraciones, delegados, expresiones lambda y acceso directo a memoria, que no se encuentran en Java.
- C# admite métodos y tipos genéricos, que proporcionan mayor rendimiento y seguridad de tipos.
- C# admite los conceptos de encapsulación, herencia y polimorfismo.
- Todas las variables y métodos, incluido el método **Main** que es el punto de entrada de la aplicación, se encapsulan dentro de definiciones de clase.
- Una clase puede heredar directamente de una sola clase primaria, pero puede implementar cualquier número de interfaces.

Breve introducción a C# II

- Los métodos que reemplazan a los métodos virtuales en una clase primaria requieren la palabra clave `override` como medio para evitar redefiniciones accidentales.
- En C#, una struct es como una clase sencilla; es un tipo asignado en la pila que puede implementar interfaces pero que no admite la herencia.
- También dispone de *propiedades*, que actúan como descriptores de acceso para variables miembro privadas.
- La E/S se hace a través de métodos de la clase `Console`, p.e.: `Console.WriteLine("Hello World!");`
- Los ejemplos de código que verás a continuación pertenecen a msdn.

Hola Mundo en C#

```
// A "Hello World!" program in C#

class Hello {

   static void Main() {

      System.Console.WriteLine("Hello World!");

   }

}
```

- El método Main puede tener estas signaturas:
  - `static void Main()`
  - `static int Main()`
  - `static void Main(string[] args)`
  - `static int Main(string[] args).`

Matrices en C#

```
class TestArraysClass {
    static void Main() {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = {
           { 1, 2, 3 },
           { 4, 5, 6 }
        };
        ...
```

```
        ....
        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array
        // in the jagged array structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Cadenas en C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

//Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual
Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual
Studio 9.0";
// Use System.String if you prefer.
System.String greeting = "Hello World!";
```

```
// In local variables (i.e. within a
// method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Clases y Estructuras en C#

```
namespace ProgrammingGuide {
    // Class definition.
    public class MyCustomClass {
        // Class members:
        // Property.
        public int Number { get; set; }
        // Method.
        public int Multiply(int num) { return num * Number; }
        // Instance Constructor.
        public MyCustomClass() { Number = 0; }
    }
    // Another class definition. This one contains the Main method, the entry point for the program.
    class Program {
        static void Main(string[] args) {
            // Create an object of type MyCustomClass.
            MyCustomClass myClass = new MyCustomClass();
            // Set the value of a public property.
            myClass.Number = 27;
            // Call a public method.
            int result = myClass.Multiply(4);
        }
    }
}
```

Interfaces en C#

```
// Admiten genericidad
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

```
// O no
interface ICarComparable
{
    bool Equals(Car obj);
}
```

Genéricos en C#

```
// Declare the generic class.
public class GenericList<T> { void Add(T input) { } }
class TestGenericList {
   private class ExampleClass { }
   static void Main() {
      // Declare a list of type int.
      GenericList<int> list1 = new GenericList<int>();
      // Declare a list of type string.
      GenericList<string> list2 = new GenericList<string>();
      // Declare a list of type ExampleClass.
      GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
   }
}
```

Delegados en C#

- Son punteros a funciones.
- Tienen una sintaxis muy sencilla.
- Se suelen emplear con expresiones lambda.

```
public delegate int PerformCalculation(int x, int y);
```

Expresiones λ en C#

- Una expresión lambda es una función anónima que se puede usar para crear tipos delegados.
- Al utilizar expresiones lambda, se pueden escribir funciones locales que se pueden pasar como argumentos o devolverse como valor de llamadas de función.

```
delegate int del(int i);
static void Main(string[] args) {
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Eventos en C# I

- Cuando ocurre algo *interesante*, los *eventos* permiten que un objeto pueda notificarlo a otros objetos.
- La clase que envía (o genera) el evento recibe el nombre de *publicador* y las clases que reciben (o controlan) el evento se denominan *suscriptores*.
- El *publicador* determina el momento en el que se genera un evento; los *suscriptores* determinan la acción que se lleva a cabo en respuesta al evento.
- Un evento puede tener varios suscriptores. Un suscriptor puede controlar varios eventos de varios publicadores.
- Nunca se generan eventos que no tienen suscriptores.

Eventos en C# II

- Los eventos se suelen usar para indicar acciones del usuario, como los clicks de los botones o las selecciones de menú en las interfaces gráficas de usuario.
- Cuando un evento tiene varios suscriptores, los controladores de eventos se invocan síncronamente cuando éste se genera.
- En la biblioteca de clases .NET Framework, los eventos se basan en el delegado EventHandler y en la clase base EventArgs.

Espacios de nombres en C# I

- Los espacios de nombres propios pueden ayudar a controlar el ámbito de clases y nombres de métodos en proyectos de programación grandes.
- Se declaran mediante el uso de la palabra clave namespace.

```
namespace SampleNamespace {
        class SampleClass {
            public void SampleMethod() {
                System.Console.WriteLine("SampleMethod inside SampleNamespace");
            }
        }
    }
```

Espacios de nombres en C# II

- Los espacios de nombres tienen las propiedades siguientes:
  - Organizan proyectos de código de gran tamaño.
  - El operador “.” delimita los espacios de nombres.
  - La directiva “`using`” hace que no sea necesario especificar el nombre del espacio de nombres para cada símbolo del mismo:
  - El espacio de nombres **global** es el espacio de nombres "raíz", por tanto `global::System` siempre hará referencia al espacio de nombres `System` de `.NET Framework`.
- Alias de espacio de nombres: `using Co = Company.Proj.Nested;`
- Sintaxis sencilla para nombres anidados: `namespace N1.N2 { class C3 /* N1.N2.C3 */ {} }`

Colecciones en C# I

- Las colecciones de elementos en C# forman parte de su biblioteca estándar.
- Existen varios tipos de colecciones, las más versátiles emplean genericidad (List<T>, Stack<T>, Queue<T>, etc…).
- Puedes encontrar más información sobre ellas aquí.