Introducción a C#

Compact Framework .Net

- El Compact Framework es una versión reducida de .Net para ordenadores de sobremesa.
- Incluye un subconjunto compatible de las clases base de la versión completa y añade algunas específicas para los dispositivos móviles. La librería de clases de CF contiene menos del 50% de las clases de .Net Framework.
- Cuando .Net CF tiene una clase con el mismo nombre que .Net Framework, la clase se diseña para que sea semánticamente compatible con la versión completa.
- Las técnicas de uso de las clases y métodos que aparecen en ambos entornos son idénticas.

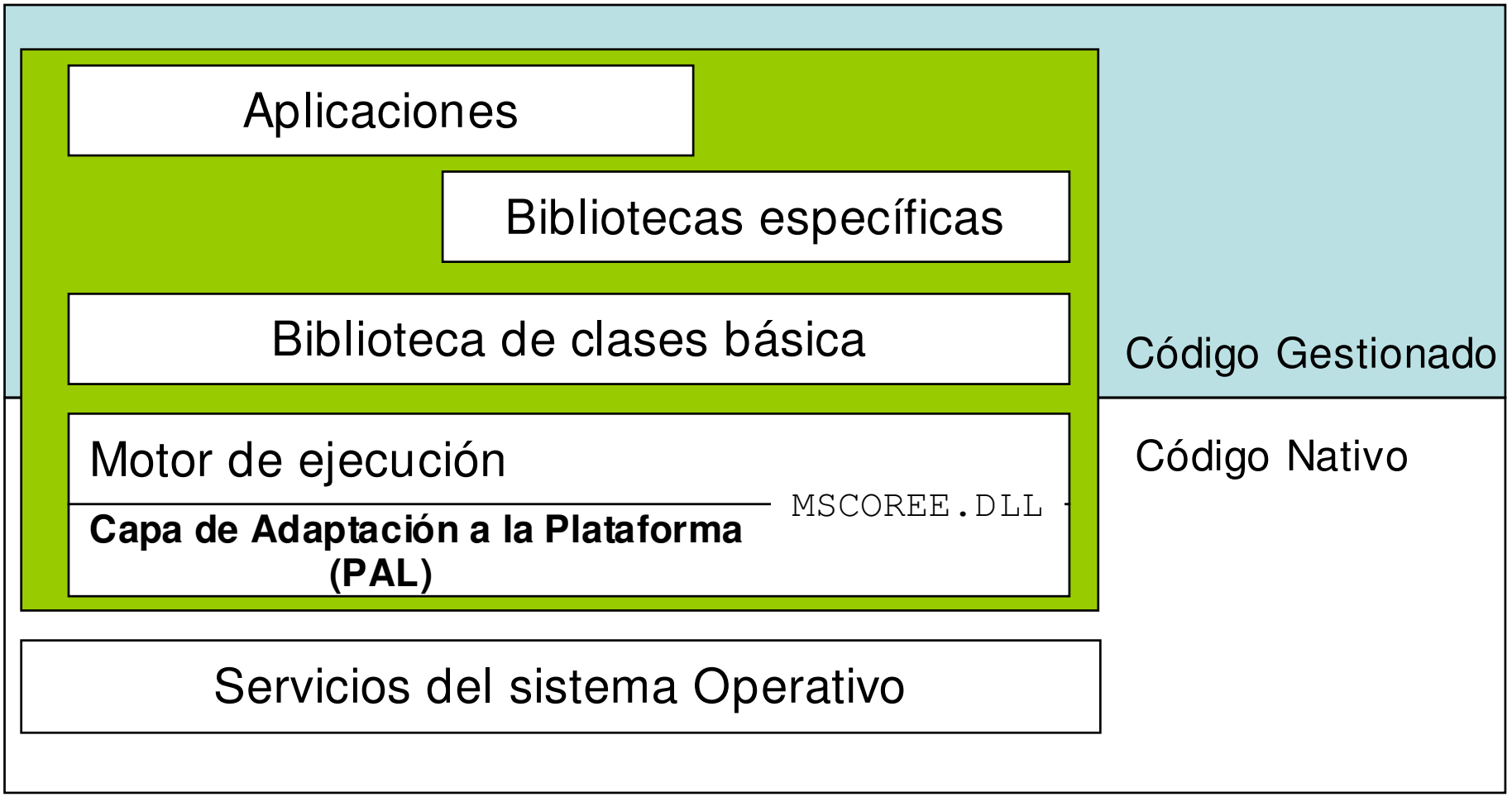
Compact Framework .Net

- El .Net Compact Framework descansa sobre tres capas de tecnología:
  1. El CLR o "*Common Language Runtime*".
     - Es la plataforma para la que se diseña el código gestionado. Es una abstracción de una CPU.
     - Se programa en *Lenguaje Intermedio* (**IL**).
  2. El compilador en tiempo de ejecución o **JIT**.
     - Compila el Lenguaje Intermedio a código máquina de la CPU real. Dos JIT disponibles.
       1. **IJIT**.- Está disponible para todas las CPU que soportan .Net CF. Rápido en compilar pero poco óptimo.
       2. **SJIT.-** Sólo para los procesadores ARM. Muy optimizado para aprovechar la arquitectura. Lento al compilar pero muy eficiente en ejecución.
  3. El sistema operativo Windows CE.
     - Controla el hardware, suministra rutinas de gestión de memoria y un cargador de programas.

Compilación JIT

- Para evitar un exceso de compilación JIT, ambos compiladores compilan método a método.
- Únicamente se compilan los métodos que se ejecutan.
- El código nativo compilado se mantiene en memoria hasta:
  - Que termina la ejecución de la aplicación. Si se vuelve a ejecutar es necesario recompilar de nuevo.
  - El CLR necesita memoria para compilar otro método. En ese caso elimina el método más antiguo.

El entorno de ejecución



Generalidades C#

Es un lenguaje diseñado para ejecutarse sobre la plataforma .Net.

Orientado a objetos. Similar a Java, C++ y C.

Permite el desarrollo de aplicación gráficas sin la necesidad de utilizar herramientas externas.

Tipos de Datos

- Todos los tipos de datos son objetos.
- Los tipos intrínsecos (enteros, caracteres, etc) funcionan como si No lo fuesen ➔ mejora de rendimiento.
- Los tipos básicos se pueden *empaquetar* en un objeto.

```
int Num=5; Object RNum;
Rnum=Num;                                  //empaquetado
Num=(int) Rnum;                            //desempaquetado
Console.Writeline(10.GetType());  //empaquetado
```

Tipos Intrínsecos

`byte`	8 bits sin signo	`float`	Real 32 bits
`sbyte`	8 bits con signo	`double`	Real 64 bits
`short`	16 bits con signo	`decimal`	Real coma fija
`ushort`	16 bits sin signo	`bool`	Booleano
`int`	32 bits con signo	`char`	Carácter UNICODE
`uint`	32 bits sin signo	`string`	Cadena caracteres
`long`	64 bits con signo	`object`	Referencia a cualquier objeto
`ulong`	64 bits sin signo		

Declaración de variables

- Como en C:

```
int var;
```

- Enumeraciones

```
enum Color {Rojo, Blanco, Verde};
queColor= Color.Blanco;
```

- Estructuras

```
struct prueba {
   public prueba (int a, float b) //Constructor
   {
      x=a;  y=b;
   }
   public int x;                  //Miembros
   private float y;
}
```

Vectores y Matrices

- Declaración:

```
int [] Diasmes = new int[12];
```

- No hay arreglo hasta que no se declara su tamaño con **new**.
- Varias dimensiones:

```
float [,] Valores= new float[23,56];
```

- Métodos de la clase **Array**:
  - **Rank** (dimensión)
  - **GetLength** (número de elementos)
  - **Clone** (Crea arreglo idéntico)
  - **Copy** (Copia elementos de un arreglo a otro)
  - **Sort** (Ordena)
  - **Reverse** (invierte el orden de los elementos)
  - **IndexOf** (Devuelve el índice del elemento de cierto valor)

Cadenas de Caracteres

- Los caracteres en .Net son Unicode (no ASCII).
- Se pueden declarar caracteres sueltos (**`char`**) o cadenas de caracteres (**`string`**).
- La clase **`string`** tiene multitud de métodos que nos hacen más fácil su manejo (conocer su longitud, extraer una parte, reemplazar, copiar, rellenar por la izda/dcha, cortar, extraer subcadenas, etc).
- El uso intensivo de cadenas es muy costoso: se crea un objeto nuevo cada vez que se modifica. En estos casos es mejor usar **`StringBuilder`**.
- **`StringBuilder`** prepara un bloque de memoria y no mueve la cadena de él hasta que agota la capacidad.
- La clase **`StringBuilder`** tiene métodos parecidos a los de la clase **`string`**

Constantes literales

- En .Net se pueden usar valores literales dentro del código. El tipo asociado se identifica como muestra la tabla.
- También se pueden utilizar identificadores para representar valores constantes. Ejemplo:

```
const short valor = 234;
const byte otro = 87;
```

Sufijo	Tipo	Ejemplo
	`char`	'P'
U	`uint`	34U
L	`long`	34L
D	`double`	3.1416D
F	`float`	3.1416F
M	`decimal`	34M

Estructuras de Control

- Estructuras condicionales:

```
if ( ) {    } else { }
```

- Condiciones múltiples:

```
switch () {
    case :
        break;
    case:
        break;
    default:
}
```

- Bucles:

```
for ( inicialización; condición de parada; actualización) {  }
while (condicion) {   }
do {  } while (condición);
foreach ( elemento in arreglo) { }
```

- Excepciones:

```
try {
//Código a proteger }
catch (excepcion) {
// código control }
finally  {  //código finalización }
```

Funciones y métodos

- Todas las funciones tienen que pertenecer obligatoriamente a una clase. Todos son métodos.
- Por defecto, los parámetros se pasan por valor.
- Para pasar una referencia, se hace en la cabecera

```
public int metodo (ref int dato,…)
```

y en la llamada

```
metodo( ref valor, …)
```

- Paso y retorno de un arreglo:

```
public int [] metodo (params int [] elementos,…)
```

Lista variable de elementos

- Se puede pasar una lista variable de argumentos:

```
public void metodo (params object [] elementos,…)
```

Cualquier tipo de variable

- Por defecto los métodos son privados.

Programación orientada a objetos

- Todos los elementos del lenguaje tienen que estar en una clase.
- Todos los elementos tienen que pertenecer a un ámbito con nombre (*namespace*).
- Definición:

  `namespace` Nombre { //Definiciones }

- Cualquier cosa en la definición pertenece al ámbito. Un ámbito puede contener otros ámbitos, clases, etc.

```
namespace alto {
    namespace medio {
        class holamundo {....
        }
    }
}
```

- La clase `holamundo` tiene el nombre cualificado: `alto.medio.holamundo`.

Ámbitos con nombre

- Para crear una clase de un ámbito:

```
alto.medio.holamundo miholamundo =
                    new alto.medio.holamundo();
```

- Se pueden evitar poner todo el nombre usando la cláusula **`using`**:

```
using alto.medio;
```

- La cláusula **`using`** sólo se puede usar con ámbitos con nombre, no con clases.
- Podemos crear alias al usar using:

```
using AM = alto.medio;
AM.holamundo miholamundo = new AM.holamundo();
```

Trabajar con clases de objetos

- Se definen con la palabra reservada **`class`**.
- Siempre están incluidas dentro de un ámbito con nombre.
- Las clases y sus miembros tienen una visibilidad fuera del ámbito que depende de los siguientes modificadores:
  - Public .- La clase o miembro es visible en todos los ámbitos. Las clases pueden usarse desde ámbitos externos y los miembros son accesibles por objetos externos.
  - Protected.- Sólo es aplicable a los miembros, no a las clases. La visibilidad se reduce a la clase y a sus derivadas.
  - Private.- Una clase privada sólo se puede usar en el interior del ámbito en que se encuentra incluida. Los miembros privados sólo se pueden usar desde dentro de la clase dónde se hayan definido.
  - Internal.- Similar a public. El elemento es visible desde fuera de su ámbito pero no desde fuera del ensamblado.
  - Protected internal.- Los identificadores pueden usarse en el ensamblado en que se definieron, así como en clases derivadas a pesar de que se encuentren en otros ensamblados.

Herencia y derivación

- Una clase puede heredar de otra:

```
public class ClaseBase {... }
public class ClaseDerivada : ClaseBase { ... }
```

- La clase derivada hereda todos los miembros de la clase base.
- Se puede evitar la herencia con **`sealed`** delante de **`class`**.
- Se pueden definir clases abstractas, que son clases que no tendrán funcionalidad propia. Son base para otras:

```
public abstract class Base {
    protected int x, y, valor;
    public void metodo1() {
    }
}
public class Derivada : Base {
    public new void metodo1() {
    }
}
```

Clase abstracta. Sólo se puede usar para derivar otras

Indica que ocultará el método de la clase base.

Trabajar con clases de objetos

- El/los constructor/es de la clase se llaman como ella. Puede tener cualquier número siempre que la lista de parámetros sea distinta.
- Un único destructor. Sin retorno y sin parámetros. Nombre: `~nombreClase()`
- El constructor se ejecuta al invocar **`new`**. El destructor se invoca automáticamente cuando no hay referencias sobre el objeto.
- Las clases tienen dos tipos de miembros: instancia (asociado al objeto creado a partir de la clase) y compartidos.
- Un miembro compartido lo es por todos los objetos de la clase. Se declaran con la cláusula **`static`**.
- Un método estático se puede invocar sin asociar a ningún objeto:

```
nombreObjeto.Metodoinstancia();
nombreClase.Metodoestatico();
```

Trabajar con clases de objetos

- Se admite la sobrecarga de métodos con la condición de que la lista de parámetros sea distinta.
- La clase heredada puede ocultar/redefinir un método de la clase base usando la palabra **new**. Si quiere invocar el método oculto de la clase base, hay que usar el identificador **base**.

```
public class Derivada : ClaseBase {
  public Derivada () : base()
   // llama antes al constructor de la clase base
    { … }
```

- Se soportan los métodos abstractos: no tienen implementación en la clase base y es obligatorio implementarlos en las derivadas.

```
public abstract class ClaseBase {
    public abstract void Metodo();
}
public class ClaseDerivada : ClaseBase {
    public override void Metodo () {
    // implementación
    }
}
```

Redefine el método en la clase derivada

Trabajar con clases de objetos

- Cuando se desea que un método tenga implementación por defecto en la clase base pero tenga las propiedades de los métodos abstractos (polimorfismo), se declara como **`virtual`**.
- No se admite la herencia múltiple. Sólo se puede heredar de una única clase.
- Sí se soportan múltiples interfaces.
- Una interfaz especifica sus miembros, métodos, propiedades, etc. de manera similar a las clases, pero sin implementación:

```
interface MiInterface {
    void MiMetodo();
}
```

- Una clase puede implementar múltiples interfaces:

```
class MiClase : MiInterface1, MiInterface2 {
}
```

La clase debe implementar los métodos de estos interfaces.
Si derivase de una clase, aparecería delante.

Clases de Componentes

- Una ***Propiedad*** es similar a una variable, pero con los métodos de acceso prefijados. Tienen que ser **`set`** y **`get`**:

```
public string Nombre{
    get {
        return nombre;
    }
    set {
        nombre = value;
    }
}
```

Se asume la existencia de una variable con este nombre.

Es el valor a la derecha de la asignación.

- Uso:

```
LaClase.Nombre="Esto es una prueba";
Console.WriteLine("{0}",LaClase.Nombre);
```

Si faltan **`get`** ó **`set`** la propiedad será de sólo lectura o escritura

Clases de Componentes

- Un ***evento*** es una señal generada por un objeto. Un ***gestor de evento*** es un método que se ejecutará automáticamente al recibir esa señal. El ***tipo*** de un evento viene definido por los parámetros que le acompañan.
- En C#, el tipo de un evento se define con un ***delegado***. Un delegado es una declaración en la que se asigna un nombre al tipo y se da la lista de parámetros:

```
public delegate void MiEvento(int valor);
```

- Cualquier clase que quiera generar un evento de este tipo tiene que definir un campo con la palabra **`event`**.

```
public class MiClase {
  public event MiEvento GeneradorEvento;
```

- La invocación de **`GeneradorEvento`** necesita de un parámetro entero que se enviará al gestor que desee recibirlo.
- Las clases que quieran recibir un evento de tipo **`MiEvento`** necesitan construir un gestor de evento apropiado.

Clases de Componentes

- Para asignar el evento a un gestor de evento se utiliza el operador +=.

```
public delegate void MiEvento(int valor); //Declaracion tipo

public class MiClase {
  public event MiEvento GeneradorEvento; //Declaracion evento
            …
    GeneradorEvento(N); //Generación del evento
            …
}
public class OtraClase {
    MiClase gestor = new MiClase();
            // Asignación del gestor de evento
    gestor.GeneradorEvento += new MiEvento(GestorEvento);
            …
    void GestorEvento(int parametro) {
            … // Código a ejecutar al recibir el evento
    }
}
```

Clases de Componentes

- Las propiedades y los eventos permiten crear ***componentes*** reutilizables. Los componentes son objetos creados a partir de una clase.
- Se hacen derivar de la clase base `System.ComponentModel.Component` y se compilan usando la opción de ***biblioteca de clases*** de Visual Studio.
- Para que VS pueda usar el componente es obligatorio incluir un constructor.
- Cuando el componente está creado, aparecerá en el Cuadro de Herramientas de VS, las propiedades aparecerán en la ventana de Propiedades y los eventos en la pestaña de eventos.
- Para crear un objeto simplemente se arrastraría el componente sobre el panel de diseño.