P/Invoke. Utilizando el código nativo desde .Net Compact Framework

Código Nativo en .Net CF

- Las aplicaciones desarrolladas con Visual Studio se compilan a *Intermediate Language* (IL) y son aplicaciones gestionadas que se compilan a código nativo JIT.
- La ejecución de código gestionado tiene una serie de ventajas como puede ser la gestión automatizada de la memoria, el control de tipos, la herramienta de desarrollo, etc…
- A pesar de eso, el código intermedio tiene una serie de limitaciones:
  - El código gestionado no se ejecuta tan rápido como el código nativo. Las aplicaciones que tienen subsistemas computacionalmente intensos se beneficiarían de su que estuvieran escritos en código nativo.
  - Hay mucho código en DLLs antiguas escrito en código nativo que ofrecen funcionalidad difícil o imposible de dar en código gestionado.
  - La interacción con componentes COM sólo se puede hacer desde código nativo.
  - La API del sistema operativo Windows CE se encuentra en una serie de llamadas a funciones dentro de DLLs. Para invocar el sistema operativo es necesario ejecutar código nativo.

Código Nativo en .Net CF. Requisitos

- Se puede ejecutar cualquier código contenido en una DLLs sin más que esté codificado para la plataforma en que se invoque.
- Si la función a ejecutar está escrita en C++ soportando el polimorfismo, es necesario comprobar el nombre asignado dentro de la DLL.
- Las funciones del sistema operativo no tienen problemas con sus nombres.
- Antes de invocar una función de código nativo es necesario declarar su nombre y los parámetros que recibe y devuelve. Se usa la cláusula DllImport.
- Si algo va mal se lanza una excepción. Las más comunes:

Excepción	Causa
`MissingMethodEception`	1) La DLL no se ha podido encontrar. 2) La DLL se ha encontrado pero no contiene la función solicitada.
`NotSupportedException`	La aplicación gestionada intenta pasar por valor un elemento mayor de 32 bits.

Paso de parámetros

- Se pueden pasar parámetros desde el código gestionado al código nativo y retornar valores desde código nativo a código gestionado.
- Los tipos fundamentales se pueden pasar por valor o por dirección (referencia).
- Si se pasan por referencia, el código nativo puede modificar los objetos recibidos.
- En el .Net Compact Framework sólo los objetos que son de 32 bits o menores pueden ser pasados por valor. Objetos mayores tienen que ser pasados por referencia.
- Ejemplo:

```
//Declaración
[DllImport("miprog.dll")]
Private static extern void FuncConInts(int Entra, ref int Sal)

//Uso
FuncConInts(a, ref dato);
```

Paso de parámetros. Ejemplos

- Enteros con signo (`int`) o sin signo (`uint`):
  - Tamaño 32 bits. Se pueden pasar por valor o por referencia.
- Reales (`float`):
  - Tamaño 64 bits. Sólo se pueden pasar por referencia.
- Enteros largos con signo (`long`) y sin signo (`ulong`):
  - Tamaño 64 bits. Sólo se pueden pasar por referencia.
- Enteros cortos con signo (`short`) y sin signo (`ushort`):
  - Tamaño 16 bits. Se pueden pasar por valor o referencia.
- Caracteres (`char`), bytes (`byte`) y booleanos (`bool`):
  - Tamaño 8 bits. Se pasan por valor o por referencia.
- Tipo `DWORD`. Es un tipo muy común en código nativo de Windows CE. Es compatible con el `int`.
- Tipo `IntPtr`. Este tipo corresponde con un manejador (*handle*) del sistema operativo Windows CE.
  - Se puede pasar por valor.
  - Internamente es un puntero a un entero. Si el código nativo cambia el valor, el cambio se refleja en el código gestionado.
  - Para pasar el valor `null` se puede pasar `IntPtr.Zero`.
  - Para extraer su valor numérico se puede usar `IntPtr.ToInt32` ó `IntPtr.ToInt64`.

Paso de parámetros. Cadenas

- Cadenas estáticas (`string`) :
  - Las cadenas de caracteres se pueden pasar como parámetros.
  - En el código nativo aparecen como punteros a caracteres “anchos” (*wide*), esto es aparecen como `WCHAR *` en lenguaje nativo.
  - Son de sólo lectura en el código nativo. Los cambios hechos en código nativo no aparecen en código gestionado.
  - No conviene modificar las cadenas en código nativo porque pueden obtenerse errores de acceso ilegal a la memoria.
- Cadenas modificables (`System.Text.StringBuilder`):
  - La clase `StringBuilder` se puede pasar por valor.
  - En el código nativo aparecen como punteros a caracteres “anchos” (*wide*), esto es aparecen como `WCHAR *` en lenguaje nativo.
  - El código nativo puede modificar la cadena y los cambios se reflejan en el `StringBuilder` del código gestionado.
  - Para obtener la cadena desde código gestionado simplemente se llama a `StringBuilder.ToString()`.
  - Es recomendable crear el `StringBuilder` con una reserva de espacio antes de pasarlo a código nativo.
  - El código nativo debería de conocer el espacio reservado para evitar sobrepasar la memoria disponible.
  - Regla general: La memoria se debería de asignar en el código gestionado. La memoria asignada en código nativo debe ser liberada desde el código nativo sino quedarán huecos.

Paso de parámetros. Estructuras

- Se pueden pasar automáticamente estructuras *simples* entre código gestionado y código nativo.
- Todas las estructuras se pasan por referencia:
  - El código nativo puede modificar el contenido de las estructuras y sus cambios se verán en el código gestionado .
- Una estructura *simple* puede contener cualquier cantidad de tipos de datos simples pero NO puede contener cadenas u otras estructuras.

```
public struct Pasable {
        public int un_int;
        public char un_char;
        public Int32 un_DWORD;
        public bool un_booleano;
}
public struct NoPasable {
        public int un_int;
        public Int32 un_DWORD;
        public Pasable una_struct;
}
```

Invocando el Sistema Operativo

- **Para llamar a una función del sistema operativo Microsoft Windows CE tenemos que seguir los mismos pasos que hasta ahora:**
  1. **Determinar cuál es la función que se quiere invocar.**
  2. **Determinar en qué DLL se encuentra.**
  3. **Declarar en código manejado una llamada a esa función con los parámetros adecuados.**
  4. **Realizar la llamada desde código gestionado.**

```
[DllImport("coredll.dll")]
        public static extern
            void GlobalMemoryStatus(ref MEMORYSTATUS lpBuffer);

        public struct MEMORYSTATUS
        {
            public int dwLength;
            public int dwMemoryLoad;
            public int dwTotalPhys;
            public int AvailPhys;
            public int dwTotalPageFile;
            public int dwAvailPageFile;
            public int dwTotalVirtual;
            public int dwAvailVirtual;
        };
```

```
MEMORYSTATUS ms = new MEMORYSTATUS();
ms.dwLength = Marshal.SizeOf(ms);
GlobalMemoryStatus(ref ms);
```