Código en C#

Continuación de algunos conceptos y funcionalidades de C#

- **Generics**

Característica del lenguaje para programar sin saber el tipo, pudiendo pasar el tipo por parámetro para determinarlo en el momento que se crean los objetos o se llaman a las funciones sin tener que saber con antelación todas las clases que van a poder usarse.

Métodos genéricos

```
class SomeClass
{
	// This 'T' will be replaced at runtime with an actual type.
	public T GenericMethod<T>(T param)
	{
		return param;
	}
}

// Uso
SomeClass a = new SomeClass();
Console.WriteLine(a.GenericMethod<int>(10));
```

Clases genéricas

```
// 'T' will be replaced with an actual type, as will also
// instances of the type 'T' used in the class.
class GenericClass<T>
{
	T item;

	public void UpdateItem(T newItem)
	{
		item = newItem;
	}
}

// Uso
GenericClass<float> f = new GenericClass<float>();
f.UpdateItem(100);
```

Múltiples elementos genéricos

```
class SomeClass
{
      public T GenericMethod<T, U>(T param, U param2)
      {
            return param;
      }
}

SomeClass a = new SomeClass();
Console.WriteLine(a.GenericMethod<int, float>(10, 100.0f));
```

Constreñimiento de tipos (where)

```
// T debe implementar interfaz IComparable
public T GenericMethod<T>(T param) where T : IComparable
{
      return param;
}

// T debe ser clase Program o derivada
public T GenericMethod<T>(T param) where T : Program
{
      return param;
}

// Las 2 anteriores
public T GenericMethod<T>(T param) where T : Program, IComparable
{
      return param;
}

// T reference type, U value type
class MyClass<T, U>
        where T : class
        where U : struct
{ }

// T debe tener un constructor por defecto (sin parámetros)
// para poder hacer new dentro
public T GenericMethod<T>(T param) where T : new()
{
      param = new T();
      return param;
}
```

Útil en múltiples ocasiones

```
public interface IDamageable<T>
{
      void Damage(T damageTaken);
}

public class Character : IDamageable<float>
{
    //The required method of the IDamageable interface
    public void Damage(float damageTaken)
    {

    }
}
```

- ## Colecciones

Siguiendo con el conocimiento adquirido de generics, las collections son clases que funcionan como una colección de objetos y proveen métodos para su acceso (guardar elementos, removerlos, ordenarlos, etc.)

Para poder utilizarlas es necesario incluir:

```
using System.Collections.Generic;
```

Lista

```
List<int> nums = new List<int>();

nums.Add(1);
nums.Add(256);
nums.Add(30);

nums.Sort();

foreach (int num in nums)
{
      Console.WriteLine(num);
}

nums.Clear();


List<Personaje> personajes = new List<Personaje>();

personajes.Add(new Personaje());
```

También se pueden acceder con [] como si fueran arrays

Para poder hacerse sort es necesario que la clase Personaje implemente la interfaz IComparable:

```
public class Personaje : IComparable<BadGuy>
{
    public int power;

    // This method is required by the IComparable interface
    public int CompareTo(BadGuy other)
    {
        if (other == null)
        {
            // 1 = this is bigger. 0 = equal. -1 = this is smaller
            return 1;
        }

        // Return the difference in power.
        return power - other.power;
    }
}
```

Diccionarios

```
Dictionary<string, int> dict = new Dictionary<string, int>();

dict.Add("Test", 10);

Console.WriteLine( dict["Test"] );

// error
// Console.WriteLine( dict["NotFound"] );

// get value with error check
int value;
if (dict.TryGetValue("NotFound", out value))
{
      Console.WriteLine(value);
}
```

Existen otras colecciones genéricas como pilas, colas, linked lists, etc.
Ver: http://msdn.microsoft.com/en-us/library/System.Collections.Generic.aspx

- **Delegates**

Los delegates funcionan como punteros a funcion en C++.
Permiten hacer llamadas a función sin conocer previamente la función que se ejecutará y son la base para los eventos.

```
static void PrintNum(int num)
{
      Console.WriteLine("Print Num: " + num);
}

static void DoubleNum(int num)
{
      Console.WriteLine("Double Num: " + num * 2);
}

delegate void MyDelegate(int num);

static void Main(string[] args)
{
      MyDelegate myDelegate = DoubleNum;
      myDelegate(10);

      myDelegate = PrintNum;
      myDelegate(10);
}
```

Multidelegate

```
static void doSomeStuff()
{
      Console.WriteLine("1, 2, 3!");
}

static void doSomeOtherStuff()
{
      Console.WriteLine("4, 5, 6!");
}

delegate void MultiDelegate();

static void Main(string[] args)
{
      MultiDelegate myMultiDelegate = null;

      myMultiDelegate += doSomeStuff;
      myMultiDelegate += doSomeOtherStuff;

      if (myMultiDelegate != null)
      {
            myMultiDelegate();
      }
}
```

Sistema de eventos a partir de delegates

```
class EventManager
{
    public delegate void ClickAction();
    public static event ClickAction OnClicked;

    void buttonPressed()
    {
        if (OnClicked != null)
            OnClicked();
    }
}

class TeleportScript
{
    public TeleportScript()
    {
        EventManager.OnClicked += Teleport;
    }

    ~TeleportScript()
    {
        EventManager.OnClicked -= Teleport;
    }

    void Teleport()
    {
        Console.WriteLine("Teleport now");
    }
}
```

- **Indexadores**

Un indexer permite acceder a un objeto como si fuera un array.

Cuando se define un indexer en una clase la clase se maneja como si fuera un array y se puede utilziar el operador [] sobre su instancia para acceder a algún contenido definido dentro de la clase.

```
class Party
{
    const int personajesParty = 3;
    private Personaje[] personajes = new Personaje[personajesParty];

    public Party()
    {
        personajes[0] = new Personaje("Wizard");
        personajes[1] = new Personaje("Ogre");
        personajes[2] = new Personaje("Elf");
    }

    public string this[int index]
    {
        get
        {
            string tmp;

            if (index >= 0 && index < personajesParty)
            {
                tmp = personajes[index].name;
            }
            else
            {
                tmp = "";
            }

            return (tmp);
        }
        set
        {
            if (index >= 0 && index < personajesParty)
            {
                personajes[index].name = value;
            }
        }
    }

    static void Main(string[] args)
    {
        Party playerParty = new Party();

        Console.WriteLine( playerParty[0] );
    }
}
```

- **var**

Las variables declaradas dentro de los métodos pueden tener un tipo implícito determinado por la palabra clave var.

Una variable local con tipo implicito sigue siendo de tipado fuerte como si se haya declarado de un tipo específico, pero el compilador determina el tipo que es la variable en vez de hacerlo nosotros.

```
var a = 10; // implicitly typed
int b = 10; //explicitly typed
```

Puede ser útil en algunos casos en que el código nos quedaría con una declaración muy larga o para un foreach.

```
Dictionary<string, string> datos1 = new Dictionary<string, string>();
var datos2 = new Dictionary<string, string>();

foreach (var dato in datos1)
{
      Console.WriteLine(dato.Value);
}
```