

## C# desde 1.0 hasta 4.0

Brian J. Cardiff - Manas s.r.l.  
<mailto://bcardiff@manas.com.ar>  
<http://weblogs.manas.com.ar/bcardiff>

17 de junio de 2009

# Agenda

- 1 Introducción
- 2 C# 1.0
- 3 C# 2.0
- 4 C# 3.0
- 5 C# 4.0

# Historia

- C# 1.0 - 2002 Lenguaje representativo del CLR.
- C# 2.0 - 2005 Generics. Delegates. Nullable. Partial Classes. ...
- C# 3.0 - 2008 LINQ. Expresiones Lambda. Inicializadores. ...
- C# 4.0 - 2010 Lenguajes Dinámicos. Varianza en Generics. Optional, Named parameters. ...

# História

- C# 1.0 - 2002 Lenguaje representativo del CLR.
- C# 2.0 - 2005 Generics. Delegates. Nullable. Partial Classes. ...
- C# 3.0 - 2008 LINQ. Expresiones Lambda. Inicializadores. ...
- C# 4.0 - 2010 Lenguajes Dinámicos. Varianza en Generics. Optional, Named parameters. ...

## Extensiones

**Cw** attempts to make data stores (such as databases and XML documents) accessible with the same ease and type safety as traditional types like strings and arrays. [...] also includes new constructs to support concurrent programming; [...]

**Spec#** is a programming language with specification language features [...] contracts, object invariants, preconditions and post-conditions.

# C# 1.0

2002

# Características generales

- Orientado a objetos
- Tipado fuerte
- clases, interfaces, clases abstractas
- Herencia simple de clases
- Herencia múltiple de interfaces
- Visibilidad `public`, `private`, `protected`, `internal`
- Organización en namespace y assembly.

# Declaración de clases

```
public class Person {  
    private string name;  
  
    public Person(string name) {  
        this.name = name;  
    }  
  
    public string GetName() {  
        return this.name;  
    }  
  
    public void SetName(string name) {  
        this.name = name;  
    }  
}
```

# Properties

## Uso

```
Person a = new Person ();  
a.Name = "Juan_Perez";  
Console.WriteLine(a.Name);
```

## Definición

```
public class Person  
{  
    private string name;  
    public string Name  
    {  
        get { return this.name; }  
        set { this.name = value; }  
    }  
}
```

# Interfaces

## Definición

```
public interface INameable
{
    string Name { get; set; }
}
```

## Uso

```
public class Person : INameable
{
    public string Name
    {
        get { /* ... */ }
        set { /* ... */ }
    }
}
```

# Herencia de clases

```
public class Person : INameable { /* ... */ }

public class Employee : Person {
    public virtual decimal ComputeSalary() {
        return 0.0;
    }
}

public class FixedSalaryEmployee : Employee {
    /* ... */
    public override decimal ComputeSalary() {
        /* ... */
    }
}
```

- También hay clases abstractas, con métodos abstractos.

# Eventos

```
public class App {  
    public void Init() {  
        WaterTank tank = new WaterTank(1, 9, 10);  
        tank.LevelHigh += new EventHandler(tank_LevelHigh);  
        tank.LevelLow += new EventHandler(tank_LevelLow);  
        // tank.Provide(2);  
        // tank.Consume(1);  
    }  
  
    void tank_LevelHigh(object sender, EventArgs e) {  
        // tanque casi lleno  
    }  
  
    void tank_LevelLow(object sender, EventArgs e) {  
        // tanque casi vacio  
    }  
}
```

## Eventos (Cont.)

```
public class WaterTank {
    public event EventHandler LevelLow, LevelHigh;
    private double lowLevel, highLevel, capacity;
    private double level;
    public WaterTank(double lowLevel, double highLevel,
        double capacity) { /* ... */ }

    public void Consume(double amount) { /*buggy. why?*/
        level = Math.Max(level - amount, 0);
        if (level <= lowLevel && level + amount > lowLevel)
            OnLevelLow();
    }

    private void OnLevelLow() {
        if (LevelLow != null)
            LevelLow(this, EventArgs.Empty);
    }
    // ...
}
```

# Delegates “punteros a funciones”

## Syntaxis

```
delegate TResult Name(TParam1 p1, TParam2 p2, ...);
```

## Ejemplo

```
class A {  
    public delegate void D(string x, int y);  
  
    void Foo() {  
        D unD = new D(Bar);  
        unD("un_string", 2);  
    }  
  
    void Bar(string x, int y) {  
        // ...  
    }  
}
```

# Exceptions

```
public class IndexOutOfRangeException : Exception
{
    /* ... */
}

public class MyListOfInts
{
    // ...
    public int ElementAt(int index)
    {
        if (0 > index || index >= this.length)
            throw new IndexOutOfRangeException();
        // ...
    }
}
```

# Exceptions (Cont.)

```
MyListOfInts l = new MyListOfInts ();  
try  
{  
    int v = l.ElementAt(10);  
}  
catch (IndexOutOfRangeException e)  
{  
    Console.WriteLine("bad index");  
}
```

## Exceptions (Cont.)

```
try
{
    // código que puede terminar a causa de un
    // IndexOutOfRangeException o
    // NullReferenceException
}
catch (IndexOutOfRangeException e)
{
    // ...
}
catch (NullReferenceException e)
{
    // ...
}
finally
{
    // siempre se ejecuta antes de salir del try
}
```

# Boxing/Unboxing

- Value types: `int`, `structs`, `enums`, ...
- Reference types: `string`, `classes`, ...

```
int foo = 42;  
object bar = foo;  
int foo2 = (int) bar;
```

## out, ref parameters

- out cuando deben ser asignados antes del return.
- ref cuando deben ser de entrada y pueden ser modificados por el método.
- Ayuda del compilador.

```
public bool NextWord(ref string s, out string token) {  
    s = s.TrimStart();  
    int space = s.IndexOf(' ');  
    if (space >= 0) {  
        token = s.Substring(0, space);  
        s = s.Substring(space + 1);  
        return true;  
    } else {  
        token = s;  
        s = string.Empty;  
        return token.Length > 0;  
    }  
}
```

# Otros

- reflection, metadata
- enums
- IDisposable y using
- Garbage Collection
- unsafe
- ...

# C# 2.0

2005

# Generics

- Parámetros de tipos
- A nivel de class, interface, struct o método

```
public class Box<T> {  
    T content;  
    public T Content {  
        get { return content; }  
        set { content = value; }  
    }  
}
```

```
Box<string> b = new Box<string>;  
b.Content = "un_string";
```

```
public interface IFactory {  
    T Build<T>();  
}
```

```
IFactory factory = ...;  
IService service = factory.Build<IService>();
```

# Generics (Cont.)

Restricciones en los generics para que T:

- Implemente cierta interfáz: T : `InterfaceType`
- Que herede de cierta clase: T : `ClassType`
- Que sea un *reference type*: T : **class**
- Que sea un *value type*: T : **struct**
- Que tenga constructor por defecto

```
public T Foo<T>() {  
    // return new T(); //solo si T : new()  
    // return null; //solo si T : class  
    return default(T);  
}
```

```
public void Bar<T>(T a) where T : SomeInterface {  
    Console.WriteLine(a.PropertyOfInterface);  
}
```

# Nullable

- **Nullable**<T>
- cast operators
- T? syntax
- null-coalescing operator: ??

```
public List<Person> Search(string criteria , int? page)
{
    page = page ?? 1;
    // usar page como un int
    // o page.Value
    // ...
}
```

```
Search("b" , null);
Search("b" , 2);
```

# IEnumerable e IEnumerator

```
interface IEnumerator<T> {  
    bool MoveNext();  
    T Current { get; }  
}
```

```
interface IEnumerable<T> {  
    IEnumerator<T> GetEnumerator();  
}
```

foreach

```
IEnumerable<FileInfo> files = ...;  
foreach (FileInfo f in files) {  
    f.Delete();  
}
```

# Colecciones disponibles

- **namespace** System.Collections (no tipadas)
- **namespace** System.Collections.Generic (generics)
- T[], **List**<T>, **Dictionary**<TKey,TValue> ,
- Las interfaces **IList**<T>, **IEnumerable**<T>, **IDictionary**<TKey,TValue> ayudan a desacoplar de los tipos concretos.

# Lazy usando yields

- Un iterador se puede usar para ejecutar operaciones bajo demanda.
- Escribir iteradores es engorroso.
- Que el compilador nos ayude!

```
IEnumerable<int> Range(int from, int to, int by) {  
    for (int i = from; i < to; i += by) {  
        yield return i;  
    }  
    yield break;  
}
```

```
foreach(int e in Range(1, 10, 2)) {  
    Console.WriteLine(e);  
}  
// 1, 3, 5, 7, 9
```

# Anonymous delegates

- Ahora podemos escribir métodos inline
- y usarlos como delegates

```
delegate bool Pred<T>(T t);
```

```
public static IEnumerable<T> Filter<T>(
    IEnumerable<T> source, Pred<T> criteria) {
    foreach (T item in source){
        if (criteria(item))
            yield return item;
    }
}
```

```
IEnumerable<int> evens = Filter(numbers,
    delegate(int i) {
        return i % 2 == 0;
    }
);
```

# C# 3.0

2008

- Automatic properties
- Object initializers
- Collection initializers
- Static type inference

```
class Foo {  
    public string Bar { get; set; }  
}
```

```
var f = new Foo { Bar = "the_value" };  
  
var manyFoos = new List<Foo> {  
    new Foo { Bar = "first" },  
    new Foo { Bar = "second" },  
};
```

# Extension methods

## Definición

```
public static class IntsExtensions {  
    public static int SumAll(  
        this IEnumerable<int> source) {  
        var res = 0;  
        foreach (var item in source) {  
            res += item;  
        }  
        return res;  
    }  
}
```

## Uso

```
var n = new[] { 1, 2, 3, 4, 5 };  
var sum = n.SumAll();  
// sum == 15
```

# Delegates

Se incluye la definición de varios delegates:

- **Predicate**<T>       ≡       **bool** m(T a)
- **Action**<T><sup>1</sup>       ≡       **void** m(T a)
- ...
- **Action**<T1,T2,T3,T4>   ≡   **void** m(T1 a, T2 b, T3 c, T4 d)
- **Func**<TResult>       ≡       TResult m()
- **Func**<T,TResult>       ≡       TResult m(T a)
- ...
- **Func**<T1,T2,T3,T4,TRes> ≡ TRes m(T1 a, T2 b, T3 c, T4 d)

---

<sup>1</sup>Ya estaba en 2.0

Se agrega sintaxis tipo  $\lambda$  (lambda) para delegates:

### C# 2.0

```
int [] numbers = new int [] { 1, 2, 3, 4, 5 };  
IEnumerable<int> evens = Filter(numbers,  
    delegate(int i) {  
        return i % 2 == 0;  
    }  
);
```

### C# 3.0

```
var numbers = new int [] { 1, 2, 3, 4, 5 };  
var evens = Filter(numbers, i => i % 2 == 0);
```

### C# 3.0 usando los extension methods

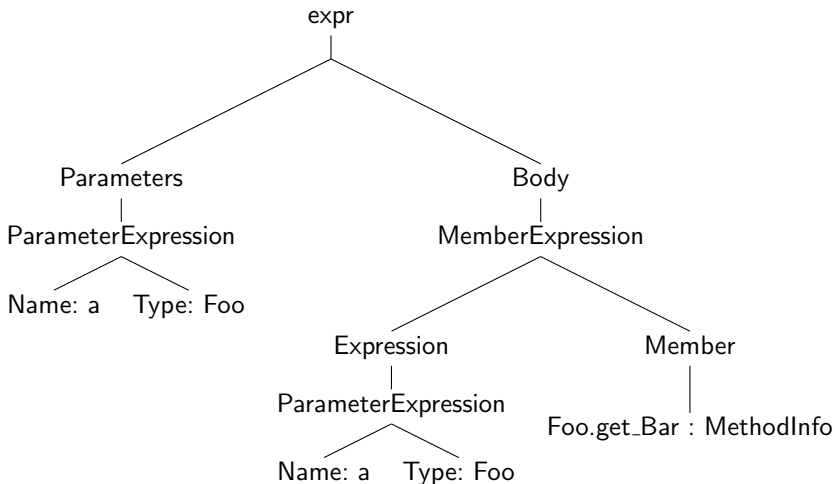
```
var numbers = new int [] { 1, 2, 3, 4, 5 };  
var evens = numbers.Where(i => i % 2 == 0);
```

Las expresiones lambda pueden generar ASTs (Abstract syntax trees) en lugar de delegates. Éstos ASTs pueden manipularse para lograr cosas muy interesantes además de ejecutarlas propiamente dicho.

- Linq-to-SQL, Linq-to-...
- GuardQ (en NetFX)
- Moq

```
public class Foo { public string Bar { get; set; } }
```

```
Expression<Func<Foo, string>> expr = a => a.Bar;
```



## Demo: GuardQ, Linq, Moq

# C# 4.0

2010

# Optional & named parameters

## Definición

```
public void M(int x, int y = 5, int z = 7) {  
    // ...  
}
```

## Uso

```
M(1, 2); // equiv M(1, 2, 7);  
M(x: 3, z: 8, y: 2); // equiv M(3, 2, 8);  
M(x: 3, z: 8); // equiv M(3, 5, 8);
```

```
public void BuyMoreStuff( Item [] cart ,  
    ref Decimal totalCost , Item i )  
{  
    CodeContract . Requires( totalCost >= 0 );  
    CodeContract . Requires( cart != null );  
    CodeContract . Requires(  
        CodeContract . ForAll( cart , s => s != i ));  
  
    CodeContract . Ensures(  
        CodeContract . Exists( cart , s => s == i );  
    CodeContract . Ensures( totalCost >=  
        CodeContract . OldValue( totalCost ));  
    CodeContract . EnsuresOnThrow<IOException>(  
        totalCost == CodeContract . OldValue( totalCost ));  
  
    // Do some stuff  
    // ...  
}
```

# Con C# 3.0

```
public class Animal { }
```

```
public class Dog : Animal { }
```

```
public interface IProducer<T> {  
    T Produce();  
}
```

```
public class DogProducer : IProducer<Dog> {  
    Dog Produce() { ... }  
}
```

La siguiente línea **NO** compila:

```
IProducer<Animal> ap = new DogProducer();
```

# Con C# 4.0 y covarianza

```
public class Animal { }
```

```
public class Dog : Animal { }
```

```
public interface IProducer<out T> {  
    T Produce();  
}
```

```
public class DogProducer : IProducer<Dog> {  
    Dog Produce() { ... }  
}
```

La siguiente línea **SI** compila:

```
IProducer<Animal> ap = new DogProducer();
```

# Con C# 3.0

```
public class Animal { }
```

```
public class Dog : Animal { }
```

```
public interface IConsumer<T> {  
    void Consume(T a);  
}
```

```
public class AnimalConsumer : IConsumer<Animal> {  
    void Consume(Animal a) { ... }  
}
```

La siguiente línea **NO** compila:

```
IConsumer<Dog> dc = new AnimalConsumer();
```

# Con C# 4.0 y contravarianza

```
public class Animal { }
```

```
public class Dog : Animal { }
```

```
public interface IConsumer<in T> {  
    void Consume(T a);  
}
```

```
public class AnimalConsumer : IConsumer<Animal> {  
    void Consume(Animal a) { ... }  
}
```

La siguiente línea **SI** compila:

```
IConsumer<Dog> dc = new AnimalConsumer();
```

# dynamic keyword y DLR

- Se trata de mejorar la interoperabilidad con lenguajes dinámicos (Python, Ruby, Javascript, ...).
- C# sigue siendo un lenguaje estático.
- DLR encapsula cómo trabajar con lenguajes dinámicos.
- El keyword `dynamic` sirve para indicar un objeto que va a ser usado con *metho lookup* dinámico. O sea que los errores van a aparecer en *runtime* y no en *compile-time*.

```
dynamic d = GetDynamicObject (...);  
d.M(7);
```

# Recursos

- Visual C# Developer Center  
<http://msdn.microsoft.com/en-us/vcsharp/>
- Moq  
<http://moq.googlecode.com/>
- NetFX  
<http://netfx.googlecode.com/>
- C# Future  
<http://code.msdn.microsoft.com/csharpfuture/>
- The Future of C# 4.0  
<http://tinyurl.com/6rndow>
- C# 4.0 team on a podcast  
<http://tinyurl.com/65o43j>  
<http://tinyurl.com/57b9tx>

# Preguntas

¿?

# Contacto

## **Brian J. Cardiff**

`bcardiff@manas.com.ar`

`http://weblogs.manas.com.ar/bcardiff`