The Constant Evolution of C#



Mike Woodring

Programmer | Learner | Teacher

@mcwoodring linkedin.com/in/woodring



"The only constant is change."

Heraclitus (of Ephesus), common translation



"Life is flux."

Heraclitus (of Ephesus), literal translation



(Observed)
Principles of
C# Evolution

Incremental, not abrupt

Opt-in, not mandatory

Nonbreaking, not disruptive

Codification of common patterns & best practices that emerge in the community

Simpler and/or faster and/or safer code



The Constant Conventional Evolution of C#



Mike Woodring

Programmer | Learner | Teacher

@mcwoodring linkedin.com/in/woodring



C# Evolution – a Sampling

Main - Before

```
c:\> program.exe
                                                                      Hello, world!
class Program
    static void Main()
       System.Console.WriteLine("Hello, world!");
```

Main - After

c:\> program.exe Hello, world! System.Console.WriteLine("Hello, world!");

Main – with Command Line Args

```
using System;
Console.WriteLine("Hello, world!");
for (var n = 0; n < args.Length; n++)
{
    Console.WriteLine($"args[{n}] = {args[n]}");
}
return 0;</pre>
```

```
c:\> program.exe C# rocks
Hello, world!
args[0] = C#
args[1] = rocks
```

Canonical Properties - Before

```
public class Point
                                        Point.cs
   private int _x;
   private int _y;
   public int X
       get { return _x; }
       set { _x = value; }
   public int Y
       get { return _y; }
       set { _y = value; }
   public override string ToString() {
       return $"({X}, {Y})";
```

```
Program.cs
var pt = new Point { X = 30, Y = 12 };
System.Console.WriteLine(pt.ToString());
```

Auto Properties - After

```
public class Point
{
   public int X { get; set; }
   public int Y { get; set; }

   public override string ToString() {
      return $"({X}, {Y})";
   }
}
```

```
Program.cs
var pt = new Point { X = 30, Y = 12 };
System.Console.WriteLine(pt.ToString());
```

Init Properties

```
public class Point
{
   public int X { get; init; }
   public int Y { get; init; }

   public override string ToString() {
      return $"({X}, {Y})";
   }
}
```

```
Program.cs
var pt = new Point { X = 30, Y = 12 };
System.Console.WriteLine(pt.ToString());
```

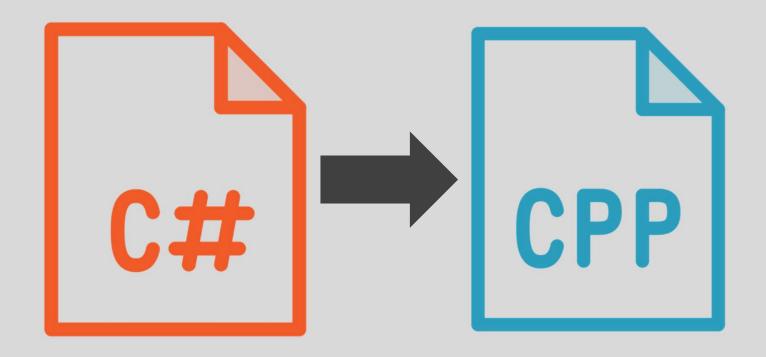
Expression-bodied Members

Point.cs

```
public class Point
{
    public int X { get; init; }
    public int Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

Exception-safe Usage of Disposable Resources





Exception-safe Usage of Disposable Resources

```
class Widget
    public void DoSomething()
        // useful code not shown
    public void Cleanup()
        // useful code not shown
                           Widget.cs
```

```
var w = new Widget();
try
    w.DoSomething();
finally
    w.Cleanup();
                           Program.cs
```

Exception-safe Usage of Disposable Resources

```
class Widget : IDisposable
    public void DoSomething()
        // useful code not shown
    public void Dispose()
        // useful code not shown
                           Widget.cs
```

```
public interface IDisposable
{
    void Dispose();
}
.NET BCL
```

```
using (var w = new Widget())
{
    w.DoSomething();
}
    Program.cs
```

Relational Patterns - Before

```
enum Generation { BabyBoomer, GenX, Millenial, GenZ, GenA }
class Person
   public int BirthYear { get; set; }
   public Generation Generation
           if ((BirthYear >= 1946) && (BirthYear <= 1964))
               return Generation.BabyBoomer;
           else if ((BirthYear >= 1965) && (BirthYear <= 1980))
               return Generation.GenX;
           else if ((BirthYear >= 1981) && (BirthYear <= 1996))
               return Generation.Millenial;
           else if ((BirthYear >= 1997) && (BirthYear <= 2012))
               return Generation.GenZ;
           else
               return Generation.GenA;
                                                             Person.cs
```

```
var p = new Person { BirthYear = 1950 };
System.Console.WriteLine(p.Generation);
```

Relational Patterns - After

```
enum Generation { BabyBoomer, GenX, Millenial, GenZ, GenA }
class Person
    public int BirthYear { get; set; }
    public Generation Generation =>
        BirthYear switch
            (>= 1946) and (<= 1964) => Generation.BabyBoomer,
            (>= 1965) and (<= 1980) => Generation.GenX,
            (>= 1981) and (<= 1996) => Generation.Millenial,
            (>= 1997) and (<= 2012) => Generation.GenZ,
            => Generation.GenA
        };
```

Person.cs

Summary



Principles

- Incremental
- Non-breaking
- Opt-in
- Simpler and/or safer and/or faster

Courses Referenced



Jason Roberts, Exception Handling in C#



Elton Stoneman, <u>IDisposable Best Practices for C# Developers</u>



Deborah Kurata, Object-Oriented Programming Fundamentals in C#



C# Development Fundamentals (Skill Path)



C#: The Big Picture



Approachable (to C++ & Java devs)

Managed (with help from the CLR)

Resilient & safe, with native performance

Acquired BCL skills are transferable

Conventional evolution is to be expected