# Tuples, Anonymous Types, and Concrete Types

By Bill Wagner

January 2010

The release of .NET Framework 4.0 adds Tuples to the base class library. One challenge for C# developers is that Tuples serve similar purposes to Anonymous Types. It can be a challenge to make the best choice between those two solutions. In this article, I'll look at some of the differences, and explain when I choose which solution. I'll also discuss some of the situations that indicate you should avoid both Tuples and Anonymous types in favor of types you declare.

Tuples can save you a great deal of repetitive tasks. They can also lead to completely unreadable code in a language like C#. Tuples are data values that contain N items. The .NET framework supports tuples containing 1 to 8 members. To use tuples with more than 8 members, you use another tuple for the member of an octtuple.

You are probably familiar with anonymous types. Let's start with a brief refresher on their capabilities. You can create an anonymous type anywhere in your code by instantiating an object (I mean 'object' in the general sense, not 'System.Object') and assigning a value each of its properties:

```
var point = new { X = 50, Y = 100 };
```

point is an object of an anonymous type. You can declare anonymous types with any number of properties, and those properties do not need to be the same type:

```
var person = new {

    FirstName = "Bill",

    LastName = "Wagner",

    DateOfBirh = default(DateTime?)

};
```

The compiler generates an internal sealed class that models the anonymous type. The anonymous type is immutable; all the properties are read only. That class contains overrides of Equals() and GetHashCode() that implement value semantics. In addition, the compiler generates an override of ToString() that displays the value of each of the public properties.

You don't write any of the code for anonymous types. You can't define any new behavior for them. The type names for anonymous types are not valid C# symbols (obviously, the names are valid CLR symbols), so you can't even define extension methods on anonymous types to create additional behavior. For the same reason, you cannot declare methods that take anonymous types as parameters, nor can you return them from methods. (Tomas Petricek has an interesting workaround for returning anonymous types from methods here. http://tomasp.net/blog/cannot-return-anonymous-type-from-method.aspx, which relies on some knowledge of the compiler internals, and is not guaranteed to work in future versions of the language).

So if anonymous types are so limited, why use them at all? In my opinion, there are two related reasons. First, the compiler writes code faster than I do. The compiler creates a page full of code for each new anonymous type. That's code I don't have to write, test, and debug myself. It's a great time saver. Second, anonymous types are great for local values that support algorithms, but aren't part of the overall object model for a system. Because the types are anonymous, they don't clutter the picture of the system as a whole. You don't browse the code for an anonymous type. Anonymous types do not show up in Class View. Anonymous types don't require external documentation. They just quickly provide a small bit of functionality.

Tuples solve a related problem. Tuples are immutable. They are syntactic types: tuples have no behavior described for a specific tuple. There is one advantage tuples have over anonymous types: Tuples are not anonymous types, so they can be used as return values or method parameters. But that advantage comes at a price. Tuples do not have names that carry any meaning to developers. The individual properties on a Tuple are named "Item1", "Item2" and so on.

Like anonymous types, tuples override Equals() and GetHashCode() to impose value semantics. However, those value semantics are strictly syntactic. Tuples carry no knowledge of what they represent, so two different tuples are 'equal' if they contain the same values, even if the meaning is very different. A Tuple that stores a point would is the same code as a tuple that stores the result of rolling two dice, or even the percentage of males and females in a population. All of those could be stored in a Tuple<int, int>, even though they have very different meanings. All Tuples also override ToString() to display the ToString() output of each of the items that make up the Tuple:

This tuple:

```
var origin = Tuple.Create(0, 0);
```

Would display as

```
(0,0)
```

There is no point in displaying the names of members, those names have no semantic meaning.

Tuples, like anonymous types, should be used when you want immutable data containers that don't define behavior. In addition, only use Tuples for those types that don't contain important types for your business model. Tuples, because of the naming conventions and the limitations on behavior don't carry much semantic information, and can't convey knowledge about your code to other developers. Whenever possible, I prefer anonymous types to Tuples because the property names carry more information. That limits my use of tuples to when I need to pass the type as a method parameter, or as a method return. I also avoid many of the higher-order Tuples. The more members, the more likely to introduce confusion in the code that works with the Tuple. What does "Item7" mean anyway? Furthermore, once you pass 8 members, you must chain Tuple object to create more members:

```
var squares = Tuple.Create(1, 4, 9, 16, 25, 36, 49,



   Tuple.Create(64, 81, 100, 121, 144));
```

That creates an order 8 tuple whose last member is an order 5 tuple containing the overflow. It works, and Tuples do chain this way correctly. I view it as something of a code smell, and you probably should create a real type if you have something like this in your program. I don't have a similar restriction for Anonymous Types, because the property names make them easier to understand.

Even though my own preference is to minimize my use of Tuples, the Tuple types do have capabilities that future C# (or other language) compilers can use to define much more behavior than the Tuple classes have implemented now. Tuples are not sealed. You can derive new types from a Tuple and add other behavior. An ORM might use this capability to build record types on top of Tuples, using the tuple to drive the interaction with the database. The derived class could contain semantically more interesting names for developers. It could also include any other behavior that was associated with the record.

Unless you are writing a code generator, you should avoid deriving from the Tuple classes. You burn the only base class you have, and you would create a less readable public interface. Remember that even though you can add meaningful property names, you can't remove the Item1, Item2, etc. from the public interface. All the users of your class would still see that extra information.

Of course, there is a bit of a middle ground where you can sprinkle in some behavior on a specific Tuple. You can define extension methods on closed generic types. As I showed earlier, a Tuple<int,int> can model a point. You can define behavior using extension methods:

```
public static class PointMethods



{



   public static Tuple<int, int> Scale(this Tuple<int, int> point, int multiplier)
```

```
{

    return Tuple.Create(point.Item1 * multiplier, point.Item2 * multiplier);

}



public static Tuple<int, int> Translate(this Tuple<int, int> point,

    int xMove, int yMove)

{

    return Tuple.Create(point.Item1 + xMove, point.Item2 + yMove);

}



public static int GetDistance(this Tuple<int, int> point)
```

```
  {

    return (int)(Math.Sqrt(point.Item1 * point.Item1 +

      point.Item2 * point.Item2));

  }

}
```

Tuples also support two new interfaces added in .NET 4.0: IStructuralComparable and IStructuralEquatable. IStructuralEquatable defines an Equal() method that uses an IEqualityComparer to determine if each item in the tuple is equal. Both of these interfaces are implemented explicitly for Tuples. StructuralComparable and StructuralEquatable implement ordering and equality tests by comparing each member of the Tuple one after the other. The ordering would be the same as if you ordered a sequence using a LINQ query “orderby Item1, Item2, etc”. Tuples also implement IComparable, and the IComparable interface implementation uses the IStructuralComparer implementation.

IStructuralComparer and IStructuralEquatable may seem to have a great deal of overlap with the other equality and comparison APIs. These do provide some extra capabilities in terms of composing types and preserving structural equality. Above I showed you how you can create tuples with more than 8 members by using another Tuple<> as the last member of the tuple structure. This can go on to whatever depth you’d like (although I don’t recommend it). Each of those Tuple<> members enforces structural equality. Therefore, the containing object can use IStructuralEquality and IStructuralComparable to enforce structural equality regardless of how many members contain other members, as long as those members support structural equality and structural comparison.

It may seem like I spent a great deal of time discussing the shortcomings of Tuple classes in C#. Tuples are useful, but less useful than other idioms in C#. Anonymous Types carry more semantic information than Tuples do. Their property names match those you used to create them. They don’t have more complicated usage once the number of members rises above 8. The only downside is that they can’t be used as return values or parameters. Tuples can. In both cases, these lightweight models are not meant to substitute for a hand coded type. In my own work, I’ve rarely used Tuples except for when I want to return multiple values from a method. For example, I might write a Find() method on a list that returns the sought item, and its index in the list. Of course, Tuples are part of the BCL, and you should expect to find them being used occasionally. You’ll often find them used in mixed language programming. They are especially useful in F#, where the compiler uses Tuples for many of the inferred types it creates.