**Defining an Anonymous Type**

When you define an anonymous type, you do so by making use of the var keyword ter 3) in conjunction with object initialization syntax. We must use the var keyword because the compiler will automatically generate a new class definition at compile time (and we never see the name of this class in our C# code)

The initialization syntax is used to tell the compiler to create private backing

fields and (read-only) properties for the newly created type.

To illustrate, create a new Console Application named AnonymousTypes. Now, add the following

method to your Program class, which composes a new type, on the fly, using the incoming parameter

data:

static void BuildAnonType( string make, string color, int currSp )

{

// Build anon type using incoming args.

var car = new { Make = make, Color = color, Speed = currSp };

// Note you can now use this type to get the property data!

Console.WriteLine("You have a {0} {1} going {2} MPH",

car.Color, car.Make, car.Speed);

// Anon types have custom implementations of each virtual

// method of System.Object. For example:

Console.WriteLine("ToString() == {0}", car.ToString());

}

You can call this method from Main(), as expected. However, do note that an anonymous type can also be created using hard-coded values, as seen here:

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\*\* Fun with Anonymous Types \*\*\*\*\*\n");

// Make an anonymous type representing a car.

var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

// Now show the color and make.

Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

// Now call our helper method to build anonymous type via args.

BuildAnonType("BMW", "Black", 90);

Console.ReadLine();

}

So, at this point, simply understand that anonymous types allow you to quickly model the “shape” of data with very little overhead. This technique is little more than a way to whip up a new data type **on the fly**, which supports barebones encapsulation via properties and acts according to value-based semantics.

To understand that last point, let’s see how the C# compiler builds out anonymous types at compile time, and specifically, how it overrides the members of System.Object.

two anonymous types will yield the same hash value if (and only if)

they have the same set of properties that have been assigned the same values