Anonymous types provide a convenient way to encapsulate a set of **read-only properties** into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the new operator together with an object initializer..

The following example shows an anonymous type that is initialized with two properties named Amount and Message.

C#

var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following

// statement to verify that their inferred types are int and string.

Console.WriteLine(v.Amount + v.Message);

Anonymous types typically are used in the [select](https://msdn.microsoft.com/en-us/library/bb384087.aspx) clause of a query expression to return a subset of the properties from each object in the source sequence.

Anonymous types contain one or more **public read-only** properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property **cannot be null**, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named Product. Class Product includes Color and Price properties, together with other properties that you are not interested in. Variable products is a collection of Product objects. The anonymous type declaration starts with the new keyword. The declaration initializes a new type that uses only two properties from Product. This causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You must provide a name for a property that is being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are Color and Price.

C#

var productQuery =

from prod in products

select new { prod.Color, prod.Price };

foreach (var v in productQuery)

{

Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);

}

Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using var. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type. You can create an array of anonymously typed elements by combining an implicitly typed local variable and an implicitly typed array, as shown in the following example.

C#

var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};

Anonymous types are class types that derive directly from object, and that cannot be cast to any type except object. The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

You **cannot** declare a **field, a property, an event, or the return type of a method** as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type object. However, doing this defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Because the Equals and GetHashCode methods on anonymous types are defined in terms of the Equals and GetHashCode methods of the properties, two instances of the same anonymous type are equal only if all their properties are equal.