

35. What are anonymous types?

Brief summary: Anonymous types are types without names. They provide a convenient way of encapsulating a set of read-only properties into a single object without having to explicitly define a type first.

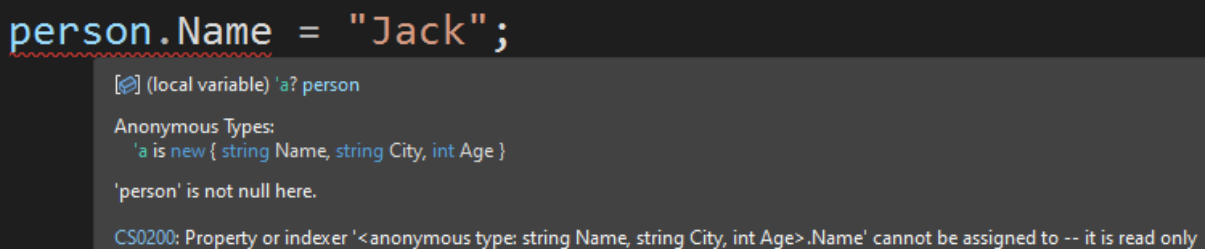
Anonymous types are types without names. Anonymous types provide a convenient way of encapsulating a set of read-only properties into a single object without having to explicitly define a type first.

```
var person = new { Name = "Martin", City = "Savannah", Age = 45 };  
  
Console.WriteLine($"This person's name is {person.Name}," +  
    $"he lives in {person.City} and is {person.Age} years old");
```

As you can see, to create an object of an anonymous type we simply use the “new” keyword and then put any properties we want in the curly braces. Here we created an anonymous type with three properties - Name of type string, City of type string, and Age of type int.

The properties of anonymous types are read-only, so code modifying them will not compile:

```
person.Name = "Jack";
```



(local variable) 'a?' person
Anonymous Types:
'a' is new { string Name, string City, int Age }
'person' is not null here.
CS0200: Property or indexer '<anonymous type: string Name, string City, int Age>.Name' cannot be assigned to -- it is read only

To understand better what may be the use case for anonymous types, let's consider a simple coding challenge. First, let's define a collection of Pets:

```

var pets = new[]
{
    new Pet("Hannibal", PetType.Fish, 1.1f),
    new Pet("Anthony", PetType.Cat, 2f),
    new Pet("Ed", PetType.Cat, 0.7f),
    new Pet("Taiga", PetType.Dog, 35f),
    new Pet("Rex", PetType.Dog, 40f),
    new Pet("Lucky", PetType.Dog, 5f),
    new Pet("Storm", PetType.Cat, 0.9f),
    new Pet("Nyan", PetType.Cat, 2.2f)
};

```

Each Pet has a name, type, and weight. What we want to do is to build a collection of strings that will contain data about each pet type and average weight for pets of this type. The result should be sorted by weight ascending. In other words, it should look like this:

```

Average weight for type Fish is 1.1
Average weight for type Cat is 1.45
Average weight for type Dog is 26.666666

```

We will use LINQ to do it. If you don't know LINQ, check out my other course "LINQ tutorial: Master the Key C# Library". In the last lecture of this course, you can find a discount coupon.

All right. We need to group those pets by type:

```

var averageWeightsData = pets
    .GroupBy(pet => pet.PetType)

```

For each of the groups, I want to calculate the average weight:

```

var averageWeightsData = pets
    .GroupBy(pet => pet.PetType)
    .Select(grouping => grouping.Average(pet => pet.Weight))

```

This is what I want, but there is one problem. I only selected the **average weights** of each group now, but I lost the information about the name of each of those

groups. I must change this code to not select floats (as the average weight is a float) but pairs of PetTypes-floats.

But how should I represent those pairs? I could define a class, struct, or a record for it:

```
record PetTypeAverageWeightPair(  
    PetType PetType, float AverageWeight);
```

...but this seems like a relatively big effort. I created a whole separate type for this very specific piece of data. I will probably never use it in a different context. Not to mention that its name is a bit awkward, but how else should we call it? There is really no good name for this very specific set of data.

The solution is to use an anonymous type. An anonymous type is a type defined right where it's needed, without even giving it a name. It's perfect for use cases like ours - where the type is **small and temporary, and we don't intend to use it anywhere else**:

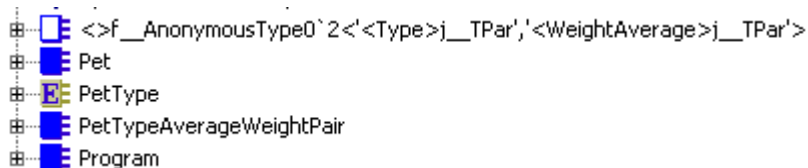
```
var averageWeightsData = pets  
    .GroupBy(pet => pet.PetType)  
    .Select(grouping => new  
    {  
        Type = grouping.Key,  
        WeightAverage = grouping.Average(pet => pet.Weight)  
    })
```

The final code would look like this:

```
var averageWeightsData = pets  
    .GroupBy(pet => pet.PetType)  
    .Select(grouping => new  
    {  
        Type = grouping.Key,  
        WeightAverage = grouping.Average(pet => pet.Weight)  
    })  
    .OrderBy(data => data.WeightAverage)  
    .Select(data => $"Average weight for type " +  
    $"{data.Type} is {data.WeightAverage}");
```

Because the anonymous type we declared doesn't even have a name, we will not be able to use it anywhere else - because how could we refer to it if we don't know its name?

Actually, the compiler gives it a name that can be seen in the Common Intermediate Language, but even if we use the decompiler to find it, it won't be possible to use it. Just to satisfy your curiosity, I checked how the compiler named this particular type:



```
<>f__AnonymousType0`2<'<Type>j__TPar', '<WeightAverage>j__TPar'>
Pet
PetType
PetTypeAverageWeightPair
Program
```

The name of the anonymous type is at the top. As you can see it's not very readable. Please note that from the perspective of Common Language Runtime anonymous types are no different than any other types.

Let's list the **most important information** about anonymous types:

- they contain only read-only properties
- no other kinds of class members, such as methods or events, are valid
- if no names are given to the properties of the anonymous type, the compiler will use the name of the property that was used to set the value of the anonymous type's property. For example, if instead of this:

```
.Select(grouping => new
{
    Type = grouping.Key,
    WeightAverage = grouping.Average(pet => pet.Weight)
})
```

...we would have this:

```
.Select(grouping => new
{
    grouping.Key,
    WeightAverage = grouping.Average(pet => pet.Weight)
})
```

...the name of the first property would be "Key", the same as the name of the property we assigned to it. When the value is not a property or a field, it

must be given a name. So in the case of WeightAverage, whose value is calculated, we must give it a name - otherwise, it will not compile, which we can see here:

```
.Select(grouping => new
{
    grouping.Key,
    grouping.Average(pet => pet.Weight)
})
```

- Anonymous types are class objects, derived directly from System.Object. They can't be cast to any other type.
- They override the Equals and GetHashCode methods to support value-based equality. Two anonymous objects with the same values will have the same hashcodes, and the Equals method will return true for them. Please note that the == operator is not overloaded, so it will return false (because they differ by reference).
- They support non-destructive mutation with the "with" keyword. Remember: non-destructive mutation is not changing the original object, but rather creating a new one with changed values.

```
var someData = new { number = 5, text = "hello!" };
var changedData = someData with { number = 10 };
```

Bonus questions:

- **"Can we modify the value of an anonymous type property?"**
No. All properties of anonymous types are read-only.
- **"When should we, and when should we not use anonymous types?"**
The best use case for anonymous types is when the type we want to use is simple and local to some specific context and it will not be used anywhere else. It's very often used as a temporary object in complex LINQ queries. If the type is complex or we want to reuse it, it should not be anonymous. Also, anonymous types can only provide read-only properties; they can't have methods, fields, events, etc, so if we need any of those features the anonymous types will not work for us.

- **"Are anonymous types value or reference types?"**

They are reference types since they are classes, but they support value-based Equality with the Equals method. In other words, two anonymous objects with the same values of properties will be considered equal by the Equals method even if their references are different.