12. What is deconstruction?

Brief summary: Deconstruction is a mechanism that allows breaking a tuple or a positional record into individual variables. It is also possible to define how deconstruction should work for user-defined types by implementing the Deconstruct method.

Deconstruction is a mechanism that allows breaking a tuple or a positional record into individual variables. It is also possible to define how deconstruction should work for user-defined types by implementing the Deconstruct method.

Deconstruction was first introduced with C# 7.

First, let's see some code.

```
(int sum, int count, double average) AnalyzeNumbers(
    IEnumerable<int> numbers)
{
    var sum = 0;
    var count = 0;
    foreach(var number in numbers)
    {
        sum += number;
        count++;
    }
    var average = (double)sum / count;
    return (sum, count, average);
}
```

This method takes a collection of integers and returns the **sum**, **count,** and **average** as a three-element tuple. For simplicity, I skipped handling empty collections. Now let's see how this method could be used:

```
var numbers = new[] { 1, 4, 2, 6, 11, 5, 83, 1, 2 };
var analysisResult = AnalyzeNumbers(numbers);

if (analysisResult.count == 0)
{
    Console.WriteLine("The collection is empty");
}
else
{
    Console.WriteLine($"The collection has {analysisResult.count}" +
      $" elements, with total sum of {analysisResult.sum} " +
      $"and average of {analysisResult.average}");
}
var numbersAverageSize = analysisResult.average > 100 ?
    "large" :
    "small";

Console.WriteLine(
    $"The numbers in the collection are " +
    $"relatively {numbersAverageSize}");
```

Since we use each of the tuple's elements quite often, let's store them in variables:

```
var numbers = new[] { 1, 4, 2, 6, 11, 5, 83, 1, 2 };
var analysisResult = AnalyzeNumbers(numbers);
var count = analysisResult.count;
var sum = analysisResult.sum;
var average = analysisResult.average;

if (count == 0)
{
    Console.WriteLine("The collection is empty");
}
else
{
    Console.WriteLine($"The collection has {count}" +
      $" elements, with total sum of {sum} " +
      $"and average of {average}");
}
var numbersAverageSize = average > 100 ?
    "large" :
    "small";

Console.WriteLine(
    $"The numbers in the collection are " +
    $"relatively {numbersAverageSize}");
```

This works, but it’s a bit cumbersome. It would be better if we could create those three variables in the same line the AnalyzeNumbers method is executed. And that’s exactly what **deconstruction** is for. Let’s see this in code:

```
var numbers = new[] { 1, 4, 2, 6, 11, 5, 83, 1, 2 };
var (count, sum, average) = AnalyzeNumbers(numbers);
```

In the second line, we declared three variables and assigned the first element of the tuple to the first one, the second to the second one, and the third to the third one. The count of variables must be equal to the count of tuple elements. Because of that, the following code will not compile:

```
var (count, sum) = AnalyzeNumbers(numbers);
CS8132: Cannot deconstruct a tuple of '3' elements into '2' variables.
```

But we don’t need to declare every variable if we don’t want to. Let’s say that for some reason I don’t care about the second tuple’s element, which is the sum. I can skip it by using the **discard**:

```
var (count, _, average) = AnalyzeNumbers(numbers);
```

Discard is a special, write-only variable, and we can’t use it after it’s assigned. Its only purpose is to be a placeholder for ignored elements of a tuple:

```
var (count, _, average) = AnalyzeNumbers(numbers);
var sum = _;
CS0103: The name '_' does not exist in the current context
```

It is also possible to deconstruct tuples into variables that we already have. In this case, we just need to skip the “var” keyword:

```
int sum;
double average;
(sum, _ , average) = AnalyzeNumbers(numbers);
```

We can also mix using the existing variables with declaring new ones:

```
int sum;
double average;
(sum, var count, average) = AnalyzeNumbers(numbers);
```

All right. So far we've been deconstructing ValueTuples. We can also deconstruct ordinary tuples…

```
var tuple = new Tuple<string, bool, int>("abc", true, 10);
var (text, boolean, number) = tuple;
```

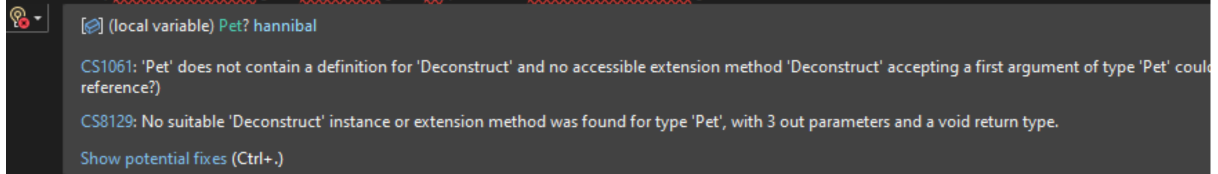
…as well as positional records:

```
var bob = new Person("Bob", 1950, "USA");
var (name, _, country) = bob;

1 reference
record Person(string Name, int YearOfBirth, string Country);
```

Let's define a new class:

```
class Pet
{
    1 reference
    public string Name { get; }
    1 reference
    public PetType PetType { get; }
    1 reference
    public float Weight { get; }

    0 references
    public Pet(string name, PetType petType, float weight)
    {
        Name = name;
        PetType = petType;
        Weight = weight;
    }
}
```

Classes, by default, do not support being deconstructed:



But we can provide our own Deconstruct method to enable it. Such a method must be void, and it must have one **out** parameter for each variable that will be created as the result. Let's add the Deconstruct method to the Pet class:

```
0 references
public void Deconstruct(
    out string name,
    out PetType petType,
    out float weight)
{
    name = Name;
    petType = PetType;
    weight = Weight;
}
```

Now we can deconstruct the Pet object into three variables:

```
var taiga = new Pet("Taiga", PetType.Dog, 30f);
var (petName, petType, weight) = taiga;
```

We can define as many Deconstruct methods in a class as we want. We can also add the Deconstruct method to structs, records, and interfaces.

Even if we did not create some class and we don't have access to its source code, we can still "add" the Deconstruct method to it using **extension methods**. Let's see this in practice. Let's say I wished I could deconstruct a DateTime object:

```
var date = new DateTime(2020, 1, 8);
var (year, month, day) = date;
```

Unfortunately, this doesn't work, because DateTime does not have the Deconstruct method implemented. Let's fix it by defining the Deconstruct extension method:

```
static class DateTimeExtensions
{
    1 reference
    public static void Deconstruct(
        this DateTime date,
        out int year,
        out int month,
        out int day)
    {
        year = date.Year;
        month = date.Month;
        day = date.Day;
    }
}
```

Now the deconstruction works as expected:

```
var date = new DateTime(2020, 1, 8);
var (year, month, day) = date;
```

Bonus questions:

- "**What is the difference between the destructor and the Deconstruct method?**"
  The destructor is a method that's called on an object when this object is being removed from memory by the Garbage Collector. The Deconstruct method allows the object to be deconstructed into single variables. It is by default generated for tuples, ValueTuples, and positional records, but we can also define it in custom types.

- "**How can we define deconstruction for types that we did not create and we don't have access to their source code?**"
  We can define the Deconstruct method as an extension method for this type.