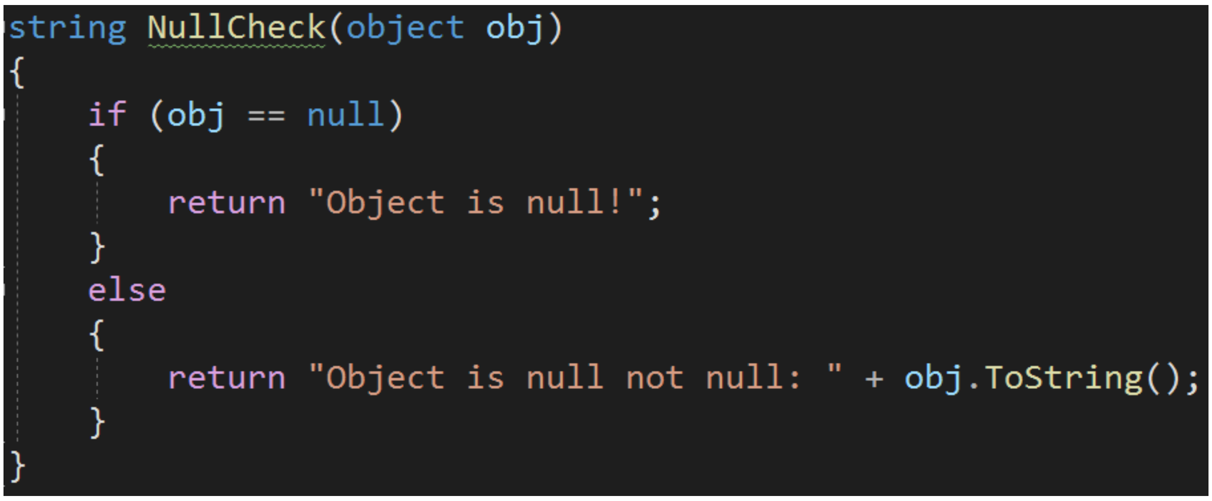
19. What is pattern matching?

Brief summary: Pattern matching is a technique where you test an expression to determine if it has certain characteristics.

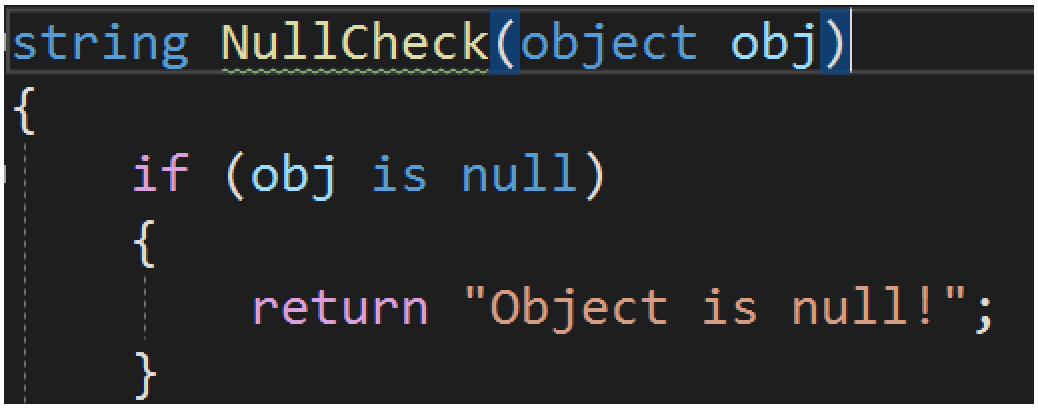
Pattern matching is a technique where you test an expression to determine if it has certain characteristics.

The easiest way to understand pattern matching is with an example. Let's say I want to run some code if some value is null, and other if it isn't:



```
string NullCheck(object obj)
{
    if (obj == null)
    {
        return "Object is null!";
    }
    else
    {
        return "Object is null not null: " + obj.ToString();
    }
}
```

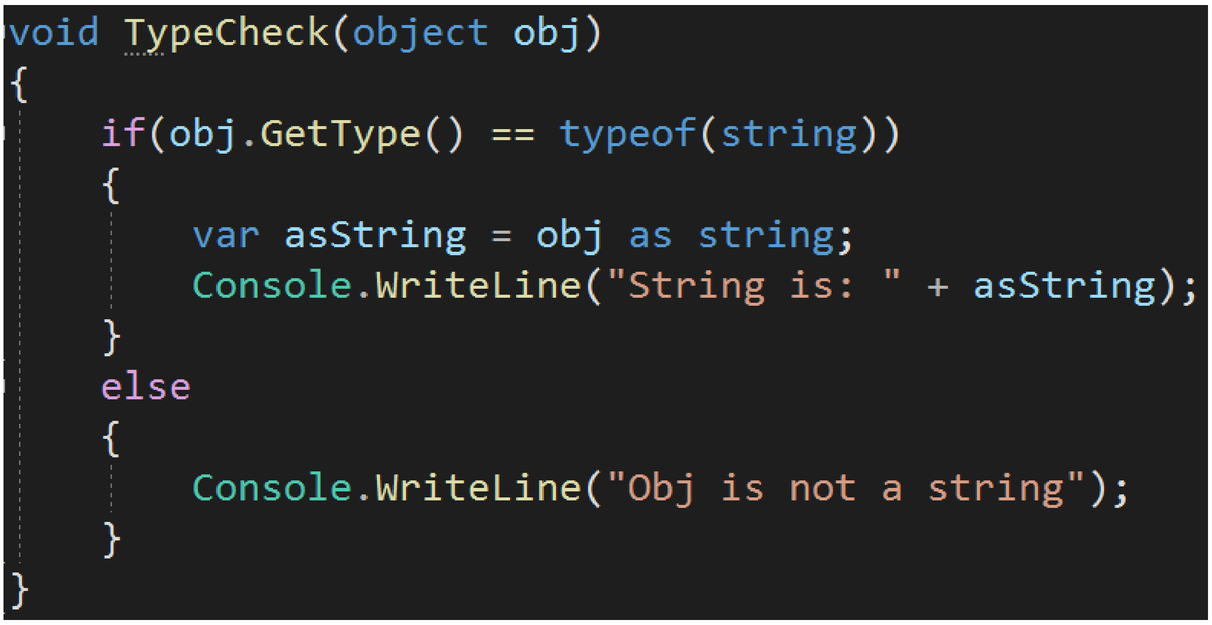
This code is pretty straightforward. There is only one problem - if I wouldn't know exactly what type the obj variable is, it might turn out that it has the == operator overloaded and that it actually does something else than simply checking if the value is null. To avoid this problem we can use **null check** pattern matching.



```
string NullCheck(object obj)
{
    if (obj is null)
    {
        return "Object is null!";
    }
```

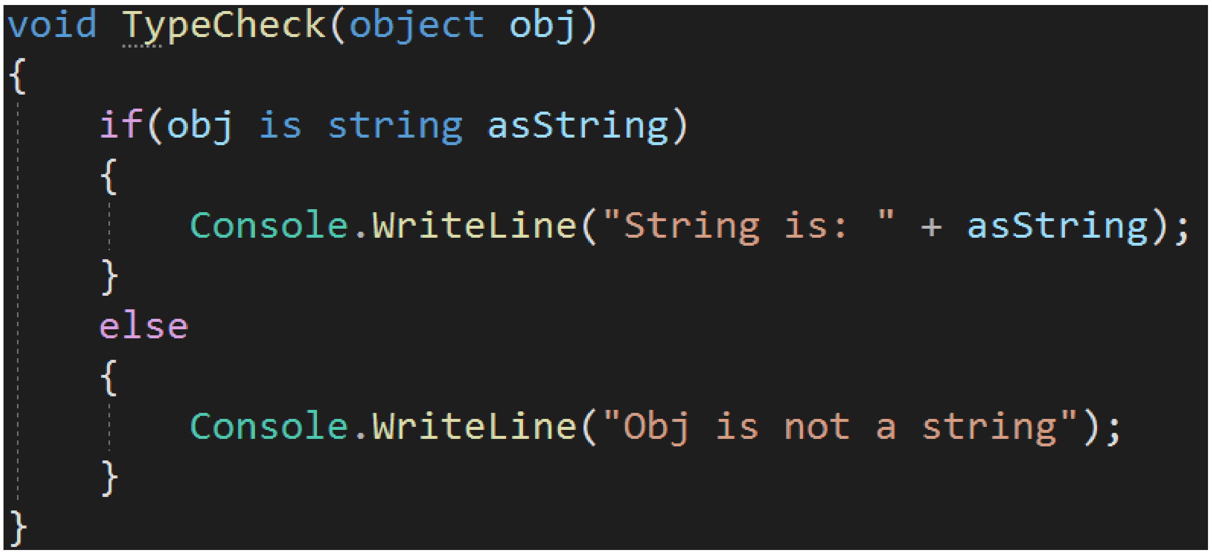
All right. So far pattern matching seems very simple. It allowed us to check if an object **is null**. But it can give us many, many more abilities. Let's walk through some examples.

One of the most commonly used patterns is the **type test**. I want to run some code if a variable is of some type. Moreover, if it is, I want to cast it to this type. Without pattern matching, I would need to write something like this:



```
void TypeCheck(object obj)
{
    if(obj.GetType() == typeof(string))
    {
        var asString = obj as string;
        Console.WriteLine("String is: " + asString);
    }
    else
    {
        Console.WriteLine("Obj is not a string");
    }
}
```

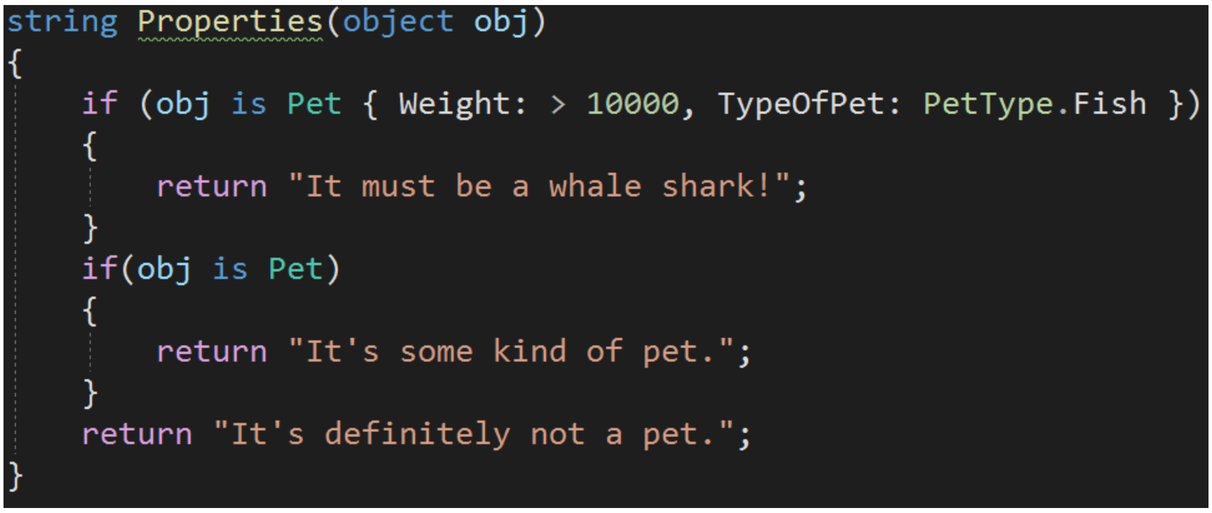
With pattern matching, I can check if an object is a string and cast it in one line:



```
void TypeCheck(object obj)
{
    if(obj is string asString)
    {
        Console.WriteLine("String is: " + asString);
    }
    else
    {
        Console.WriteLine("Obj is not a string");
    }
}
```

This is quite convenient. Please note that the **asString** variable will be available only if the **obj** is a string, so I would not be able to use it anywhere else than inside the **if** statement.

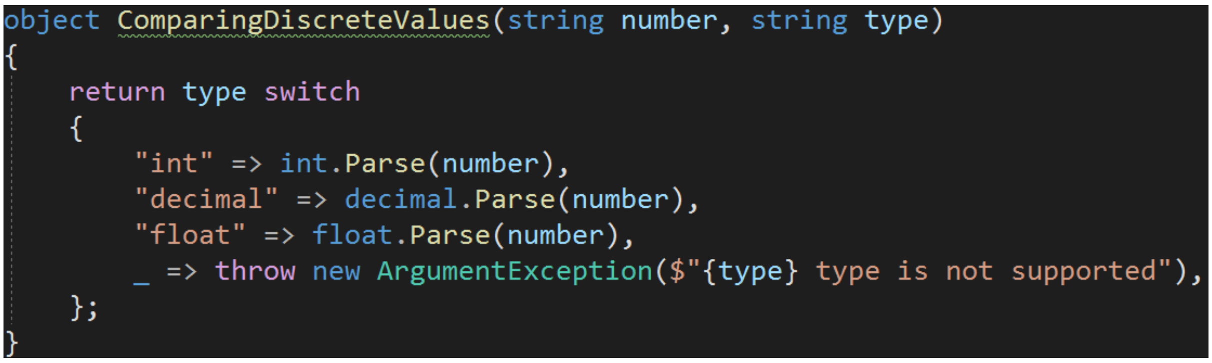
We can also check some **particular properties** of the checked object:



```
string Properties(object obj)
{
    if (obj is Pet { Weight: > 10000, TypeOfPet: PetType.Fish })
    {
        return "It must be a whale shark!";
    }
    if(obj is Pet)
    {
        return "It's some kind of pet.";
    }
    return "It's definitely not a pet.";
}
```

Here we checked if an object is a Pet with Weight larger than 10000 and PetType equal to Fish.

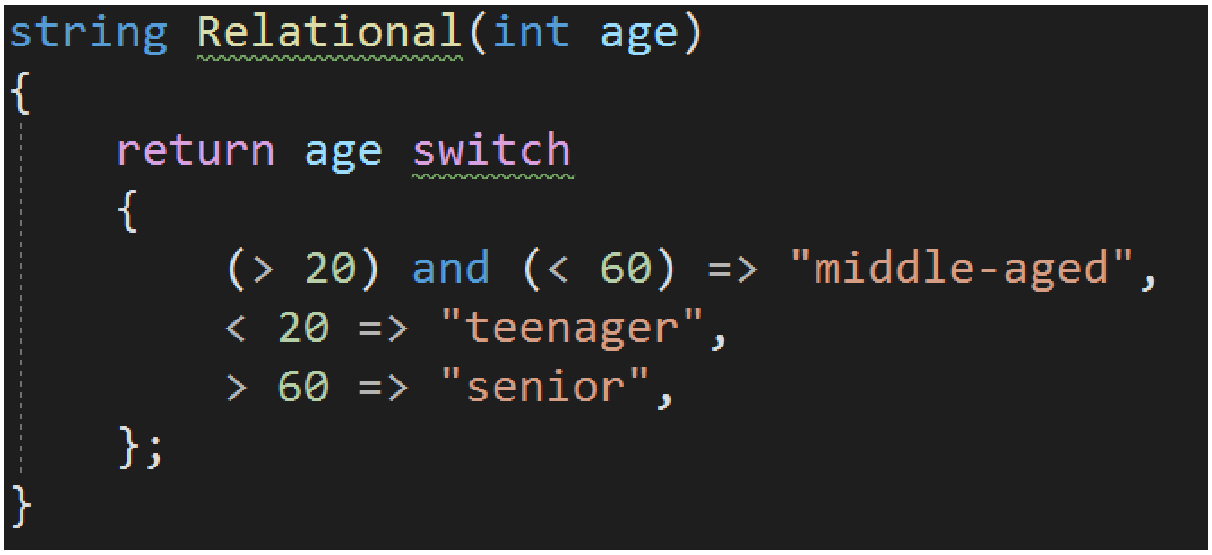
All right. The next type of pattern matching is **comparing discrete values**. This is very similar to using a plain old switch statement. Let's say I have a method taking a string that should represent a number, and another string saying what type of number it is (int, decimal, or float). Depending on the second parameter I want to convert the first parameter to the given type:



```
object ComparingDiscreteValues(string number, string type)
{
    return type switch
    {
        "int" => int.Parse(number),
        "decimal" => decimal.Parse(number),
        "float" => float.Parse(number),
        _ => throw new ArgumentException($"{type} type is not supported"),
    };
}
```

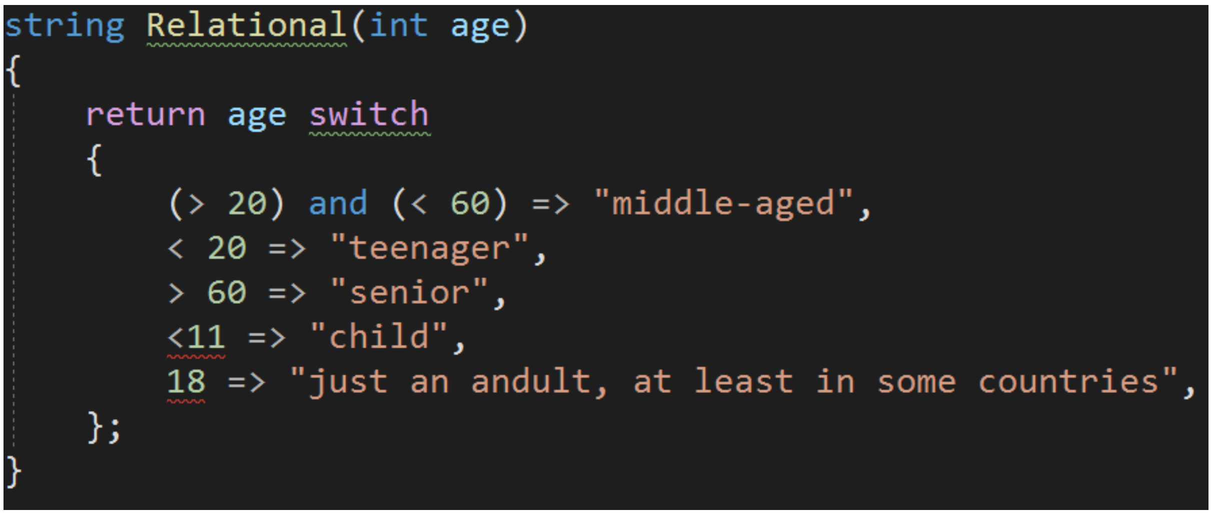
Please notice the special "_" case. This is a **discard pattern** and it works similarly as **default** in the switch statement. It will be executed if the type parameter is not equal to any of the specified values.

Let's get to more complex types of pattern matching. The next one is a **relational pattern**. It allows us to check how a given value compares to constants:



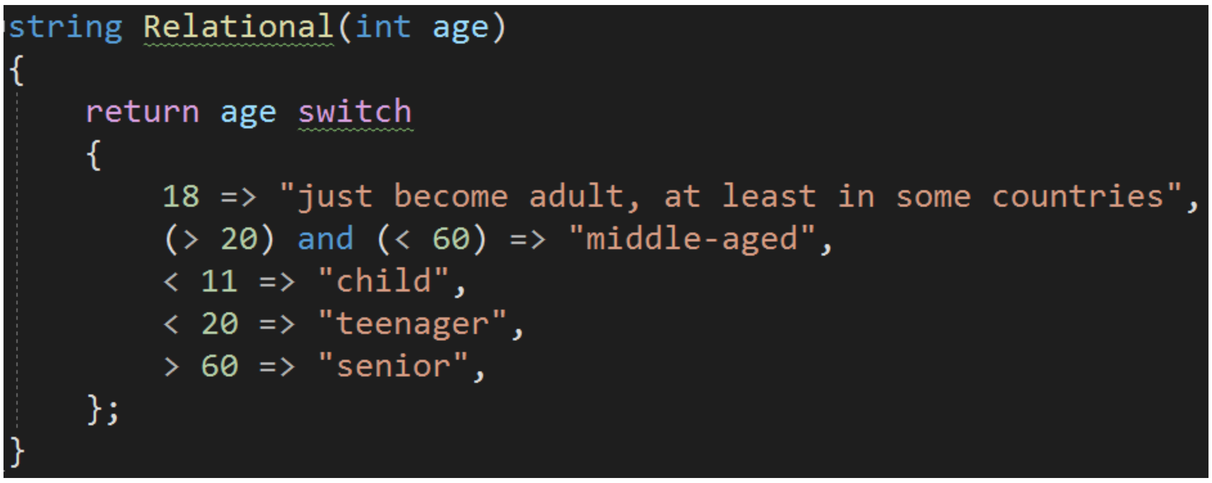
```
string Relational(int age)
{
    return age switch
    {
        (> 20) and (< 60) => "middle-aged",
        < 20 => "teenager",
        > 60 => "senior",
    };
}
```

The cool thing about pattern matching is that it has very good IDE and compiler support, and we get an error when we do something silly. For example, let me add some more cases here:



```
string Relational(int age)
{
    return age switch
    {
        (> 20) and (< 60) => "middle-aged",
        < 20 => "teenager",
        > 60 => "senior",
        <11 => "child",
        18 => "just an andult, at least in some countries",
    };
}
```

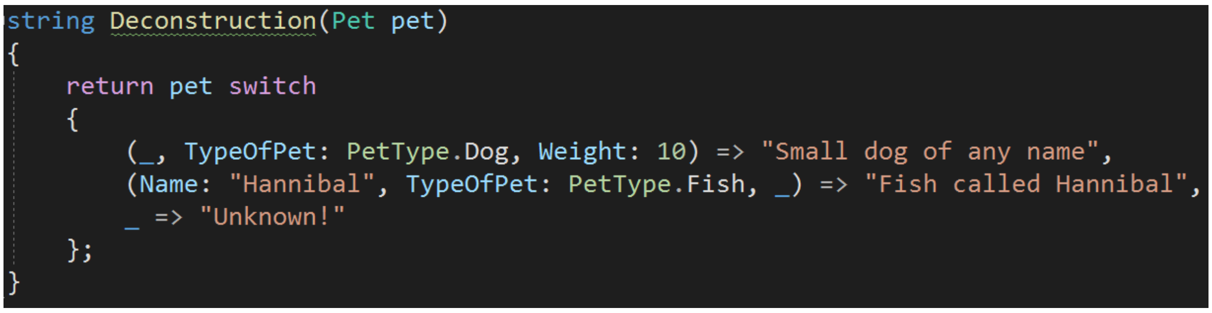
This code doesn’t compile, because the last two cases are unreachable. The cases are executed from top to bottom, so when the age parameter is 10, we will hit the “less than 20” case. We will never reach the “less than 11” case. Let’s fix the order of the cases:



```
string Relational(int age)
{
    return age switch
    {
        18 => "just become adult, at least in some countries",
        (> 20) and (< 60) => "middle-aged",
        < 11 => "child",
        < 20 => "teenager",
        > 60 => "senior",
    };
}
```

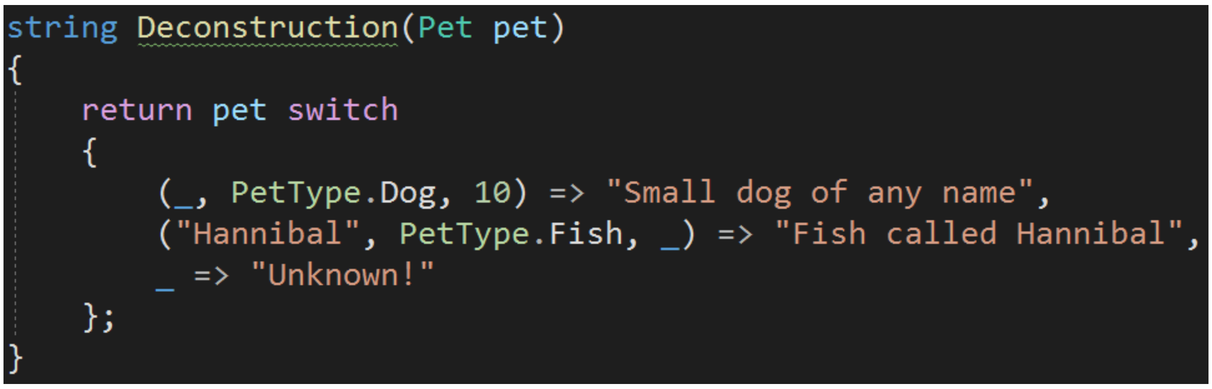
Great. Now, this should work as expected. This code actually demonstrates one more pattern - a **logical pattern**. We used it when we checked if the age is less than 20 **and** more than 60.

We can also use pattern matching with deconstruction. Check out the “What is deconstruction?” lecture to learn more.



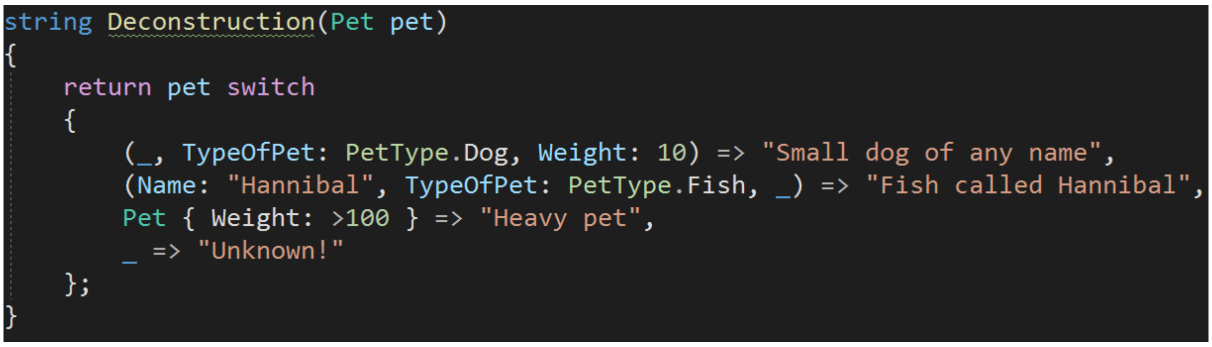
```
string Deconstruction(Pet pet)
{
    return pet switch
    {
        (_, TypeOfPet: PetType.Dog, Weight: 10) => "Small dog of any name",
        (Name: "Hannibal", TypeOfPet: PetType.Fish, _) => "Fish called Hannibal",
        _ => "Unknown!"
    };
}
```

We could omit the parameter names (but personally I would rather leave them for readability).



```
string Deconstruction(Pet pet)
{
    return pet switch
    {
        (_, PetType.Dog, 10) => "Small dog of any name",
        ("Hannibal", PetType.Fish, _) => "Fish called Hannibal",
        _ => "Unknown!"
    };
}
```

We can also mix deconstruction with checking particular properties:



```
string Deconstruction(Pet pet)
{
    return pet switch
    {
        (_, TypeOfPet: PetType.Dog, Weight: 10) => "Small dog of any name",
        (Name: "Hannibal", TypeOfPet: PetType.Fish, _) => "Fish called Hannibal",
        Pet { Weight: >100 } => "Heavy pet",
        _ => "Unknown!"
    };
}
```

All right. We learned some of the most basic usages of pattern matching. There is also a question when to use them, and when to use plain old if and switch statements. In my opinion, you should simply use those that you find more readable. You can mix both to get what's best in any of them.

To summarize: pattern matching is a technique where you test an expression to determine if it has certain characteristics.

Bonus questions:

- **"How can we check if an object is of a given type, and cast to it this type in the same statement?"**
  We can use pattern matching for that. For example, we could write "if obj is string text". This way, we will cast the object to the string variable called text, but only if this object is of type string.