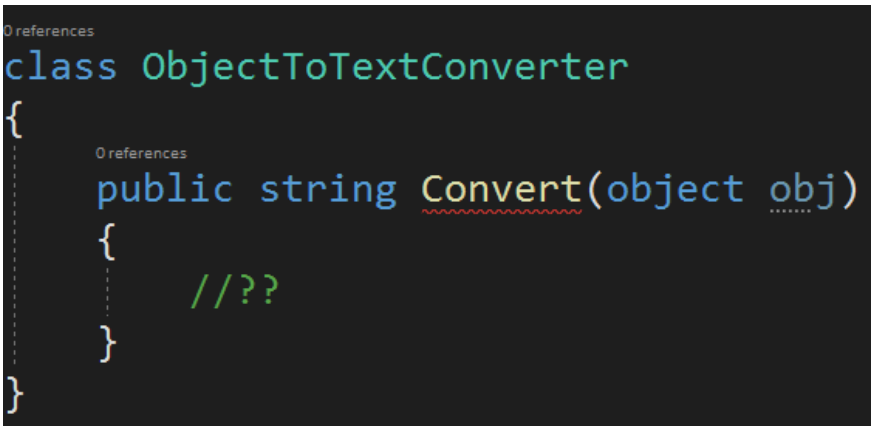
16. What is reflection?

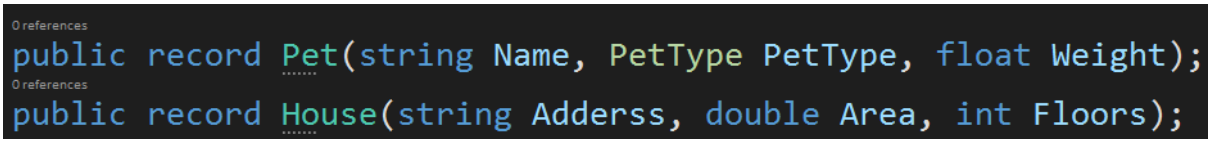
Brief summary: Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example, using reflection, we can list all fields and their values belonging to a given object, even if at compile time we don't know what type it is exactly.

Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example, using reflection, we can list all fields and their values belonging to a given object, even if at compile time we don't know what type it is exactly.

This all probably sounds a bit mysterious to you, so let's consider the following use case: we want to write a class that can take various objects and save them to a text file. There are already mechanisms that do it, and store objects are JSON or XMLs, but let's say we want some custom format so we must implement it ourselves. We want this class to be completely generic, so it can take any type of object.



Let's consider two sample types this class could convert to text. For brevity, defined them as records, which we will learn about later in the course.



If a Pet object is being converted, I would like the result to be for example:
"Name is Taiga, PetType is Dog, Weight is 30.0"
Similarly, for a House I would like to have:

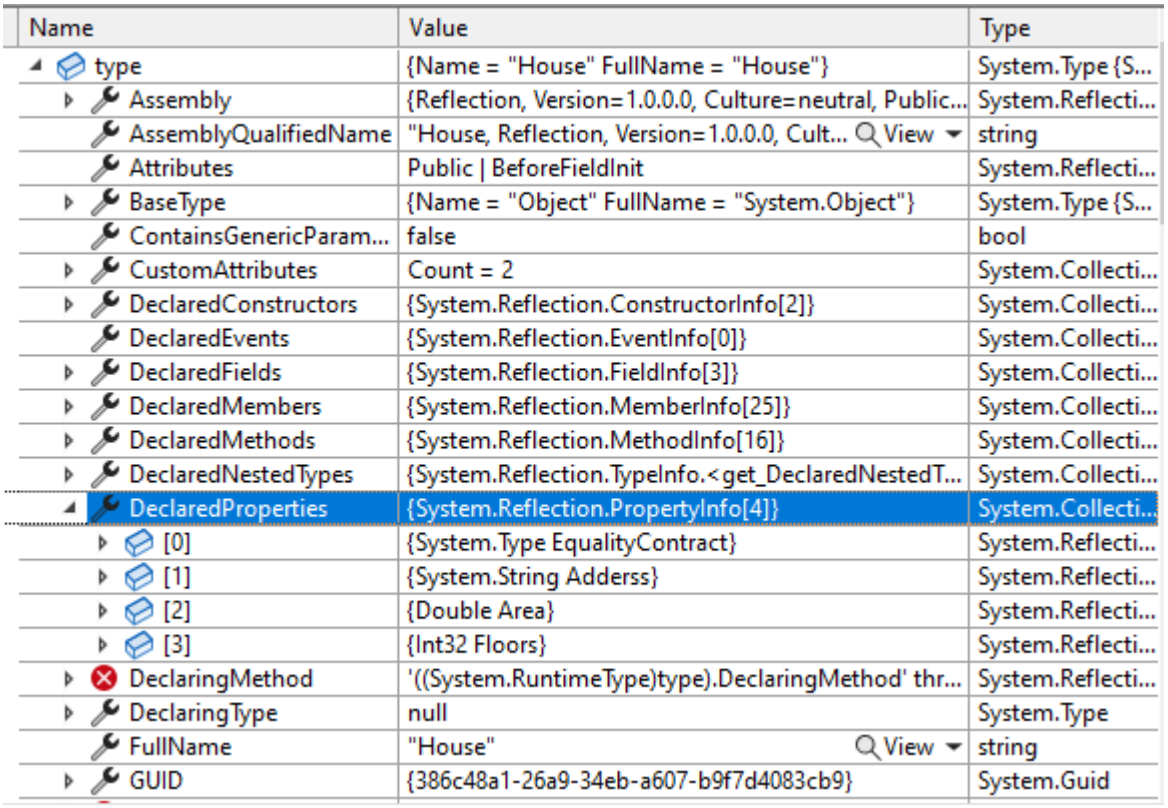
“Address is 123 Maple Road, Berrytown, Area is 170.6, Floors is 2”.

The problem is that in the Convert method, we have no idea what type we deal with. We can’t cast “obj” to anything concrete, as the types may vary. Also, to implement what we want we will not only need the values of the properties (which we could access if we only had more concrete type than System.Object) but we will also need the need their names, which is not available at runtime. In other words, when calling house.Floors we can get the number 2, but we can’t get the “Floors” string.

Well, actually, we can, but only if we use **reflection**. Reflection allows us to access information about some type at runtime. We can not only access the values of some fields but also their names. Moreover, we could access information about methods, constructors, access modifiers, and so on. Let’s see this in practice. First of all, we will use the GetType method on the obj object. It will return a Type object, which provides all information about a type:

```
0 references
public string Convert(object obj)
{
    Type type = obj.GetType();
}
```

Let’s see the **type** object in the debugger:



Name	Value	Type
type	{Name = "House" FullName = "House"}	System.Type {S...
Assembly	{Reflection, Version=1.0.0.0, Culture=neutral, Public...	System.Reflecti...
AssemblyQualifiedName	"House, Reflection, Version=1.0.0.0, Cult... View	string
Attributes	Public | BeforeFieldInit	System.Reflecti...
BaseType	{Name = "Object" FullName = "System.Object"}	System.Type {S...
ContainsGenericParam...	false	bool
CustomAttributes	Count = 2	System.Collecti...
DeclaredConstructors	{System.Reflection.ConstructorInfo[2]}	System.Collecti...
DeclaredEvents	{System.Reflection.EventInfo[0]}	System.Collecti...
DeclaredFields	{System.Reflection.FieldInfo[3]}	System.Collecti...
DeclaredMembers	{System.Reflection.MemberInfo[25]}	System.Collecti...
DeclaredMethods	{System.Reflection.MethodInfo[16]}	System.Collecti...
DeclaredNestedTypes	{System.Reflection.TypeInfo.<get_DeclaredNestedT...	System.Collecti...
DeclaredProperties	{System.Reflection.PropertyInfo[4]}	System.Collecti...
[0]	{System.Type EqualityContract}	System.Reflecti...
[1]	{System.String Adderss}	System.Reflecti...
[2]	{Double Area}	System.Reflecti...
[3]	{Int32 Floors}	System.Reflecti...
DeclaringMethod	'((System.RuntimeType)type).DeclaringMethod' thr...	System.Reflecti...
DeclaringType	null	System.Type
FullName	"House" View	string
GUID	{386c48a1-26a9-34eb-a607-b9f7d4083cb9}	System.Guid

As you can see there is quite a lot of data in here. We have some information about constructors, methods, base type, and also properties which I highlighted. We can see all properties we declared in the House type, and also an extra EqualityContract property which is is autogenerated for records. We will ignore it when converting the object to string.

All right. Let's use this data to achieve what we want. First, I want to read all properties from the given object, except the EqualityContract:

```
Type type = obj.GetType();

var properties = type
    .GetProperties()
    .Where(property => property.Name != "EqualityContract");
```

This gives me an IEnumerable<PropertyInfo>. Now I want to build a string for each PropertyInfos, accessing the property name as well as its value, and then join the strings together. I will use LINQ to do it:

```
return string.Join(
    ", ",
    properties
        .Select(property =>
        $"{property.Name} is {property.GetValue(obj)}"));
```

The Select method comes from LINQ, and it simply maps every property to a string.

All right. This is the final method:

```
public string Convert(object obj)
{
    Type type = obj.GetType();

    var properties = type
        .GetProperties()
        .Where(property => property.Name != "EqualityContract");

    return string.Join(
        ", ",
        properties
            .Select(property =>
            $"{property.Name} is {property.GetValue(obj)}"));
}
```

Let's make sure it works:

```
var converter = new ObjectToTextConverter();

Console.WriteLine(converter.Convert(
    new House("123 Maple Road, Berrytown", 170.6d, 2)));

Console.WriteLine(converter.Convert(
    new Pet("Taiga", PetType.Dog, 30)));
```

The result of this code is:

```
Adderss is 123 Maple Road, Berrytown, Area is 170.6, Floors is 2
Name is Taiga, PetType is Dog, Weight is 30
```

Great! Seems everything is working. We used reflection to access the information about some type at runtime and read the values and names of its properties.

Reflection gives us much more abilities. Here are some of them:

- loading dlls at runtime and using them
- instantiating a new instance of some object of a specific type at runtime. For example, we can create an object of a type defined in a dll we loaded reading private fields or properties, executing private methods (don't overuse it!)
- finding all classes derived from a specific base type or implementing a specific interface
- reading the attributes. This is for example what NUnit does when it runs the tests. It finds all methods with the [Test] attribute and executes them. We will learn more about attributes in the next lecture
- running a method by its name, for example, if the user of the application selected if from some dropdown
- debugging. For example, sometimes it is necessary to find out the list of currently loaded assemblies
- creating new types at runtime (System.Reflection.Emit namespace is used for that)
- and many more

As you can see reflection is a powerful tool, but as such should be used with caution. The code that heavily relies on reflection is usually hard to maintain and understand. It's also prone to errors. For example, when you call a method by its name, but someone changes the name without your knowledge, the code will crash the next time it's run because no method with the name exists anymore.

Also, Using reflection has a relatively big impact on **performance**. At one of the projects I worked on I was asked to improve the performance of some process. This application was using reflection a lot, mostly to load some types and attributes from dlls at runtime. It turned out that the results of those loads can be cached, and only this improvement made the process work twice as fast as before. We will learn more about this mechanism in the "What is caching?" lecture.

Use reflection with caution. If there is a convenient way of implementing the same logic without it, go for it. If not, reflection may be a lifesaver but keep an eye on the performance.

Let's summarize. Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example call a method with the name equal to a given string, or list all fields and their values belonging to a given object.

Bonus questions:

- "**What are the downsides of using reflection?**"
  Using reflection has a relatively big impact on performance. Also, it makes the code hard to understand and maintain. It may also tempt some programmers to "hack" some code, for example, to access private fields at runtime, which may lead to unexpected results and hard-to-understand bugs.