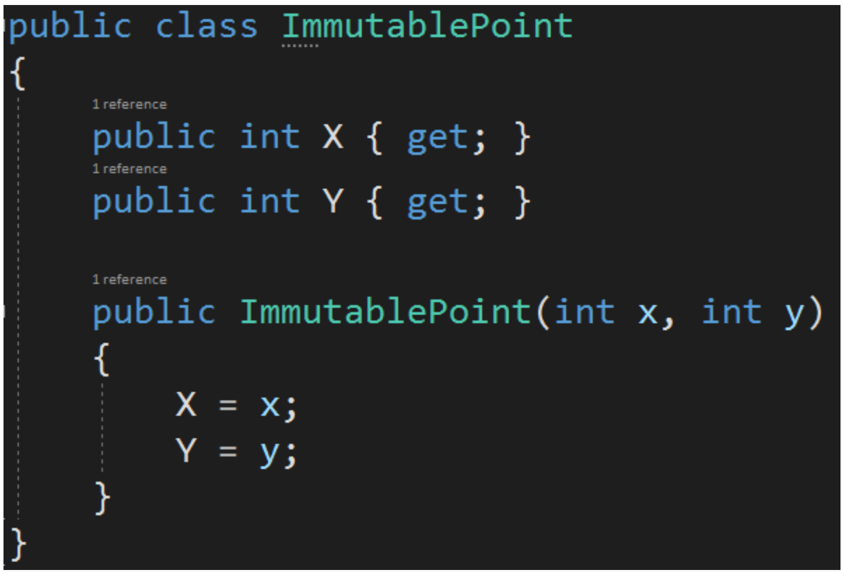
30. What are immutable types and what's their purpose?

Brief summary: Immutability of a type means that once an object of this type is created none of its fields of properties can be updated. Using immutable types over mutable ones gives a lot of benefits, like making the code simpler to understand, maintain and test, as well as making it thread-safe.

Immutability of a type means that once an object of this type is created none of its fields of properties can be updated.

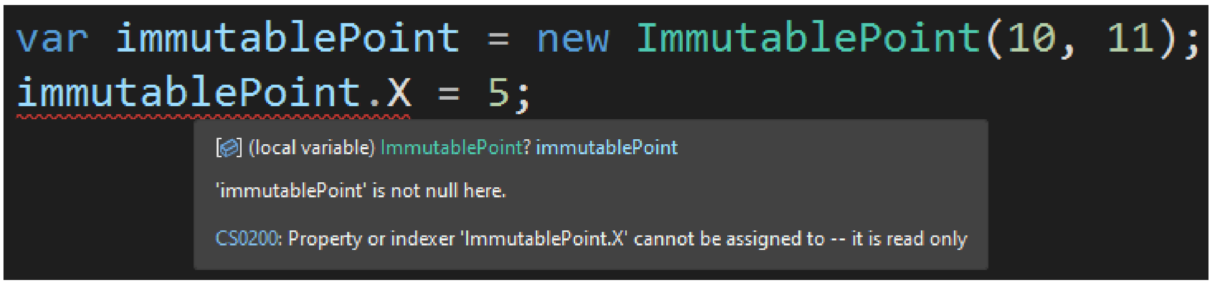
Let's see a simple immutable type:



```
public class ImmutablePoint
{
    1 reference
    public int X { get; }
    1 reference
    public int Y { get; }

    1 reference
    public ImmutablePoint(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

The objects of this class are immutable because the X and Y properties do not have setters. Once we create an object using the constructor (the only place where we can assign values to X and Y) it will not be possible to modify it:



```
var immutablePoint = new ImmutablePoint(10, 11);
immutablePoint.X = 5;
```

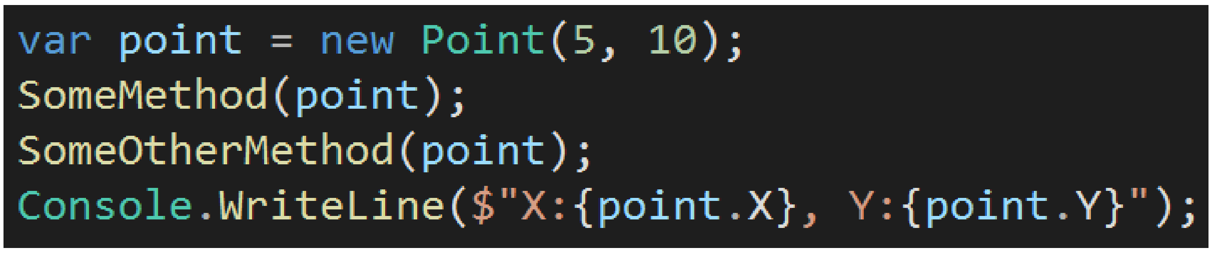
```
(local variable) ImmutablePoint? immutablePoint
'immutablePoint' is not null here.
CS0200: Property or indexer 'ImmutablePoint.X' cannot be assigned to -- it is read only
```

As you can see the concept of immutability is very simple. It can extend to more complex types, for example, collections - once we create an immutable collection, it can't be changed, so no element can be added, removed, or altered.

The question is - why should we bother in creating immutable types?

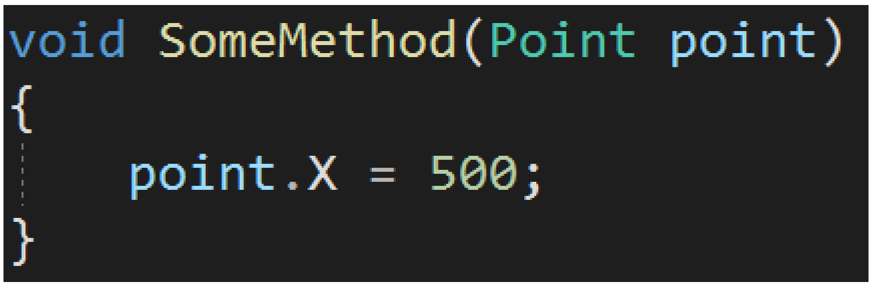
Let's see a couple of the most important benefits of having immutable types:

1) **Clarity and simplicity of the code**
   First, let's see a piece of code that seems simple:



```
var point = new Point(5, 10);
SomeMethod(point);
SomeOtherMethod(point);
Console.WriteLine($"X:{point.X}, Y:{point.Y}");
```

   What do you think will be printed from the fourth line? Well, it's impossible to say, because we don't know what happens in SomeMethod and SomeOtherMethod. Maybe they simply read the values of the point, but maybe they alter it?



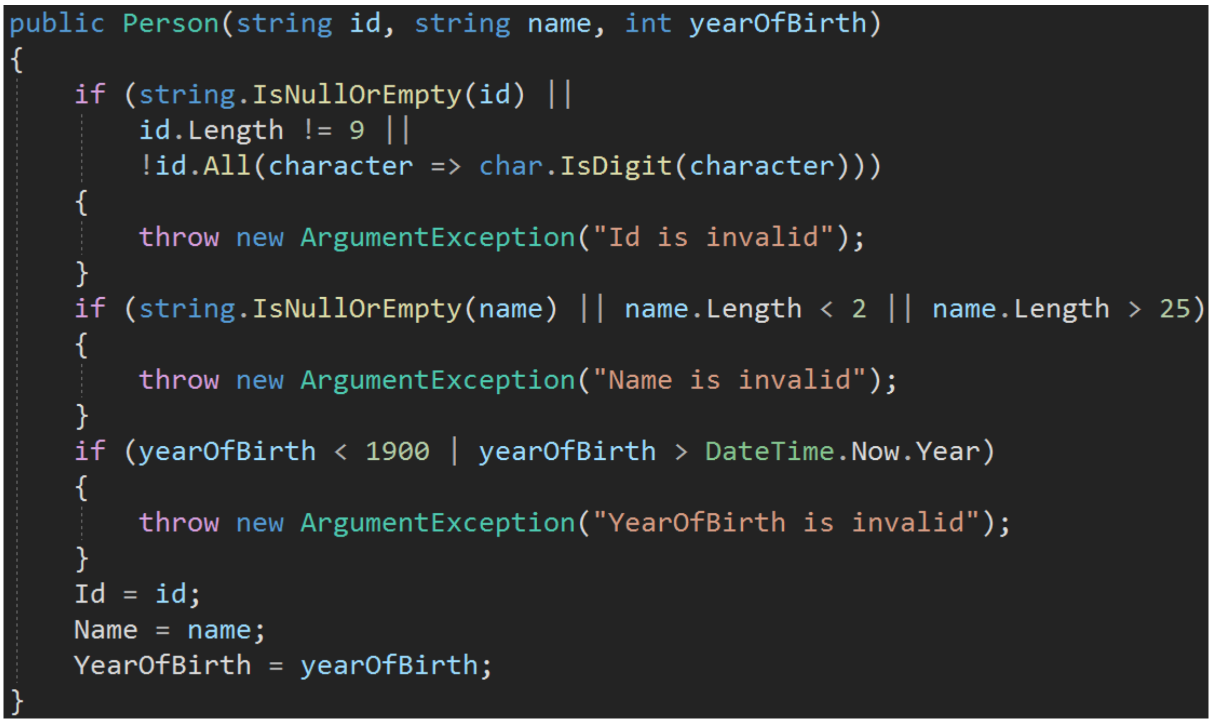
```
void SomeMethod(Point point)
{
    point.X = 500;
}
```

   We won't be sure what the code does and how it behaves until we follow the flow of the code very carefully, checking what exactly every method does with the Point object.
   If the Point was immutable, we wouldn't need to worry - we would be sure that once created, its value remains the same.

2) **Pure functions**
   Pure functions are functions whose results only depend on the input parameters, and they do not have any side effects - they don't alter any state of the application, they don't modify the input parameters. We can call a pure function any time we want with the same set of parameters, in any order, and it will always yield the same result. Because of that, we can **cache** the result, making the parameters the key of the cache. Pure functions are **simple to understand**, as we don't need to be aware of the context in which they are called. **Testing** them is extremely simple, as we only check if their result is as expected. For testing purposes, we don't need to set up any

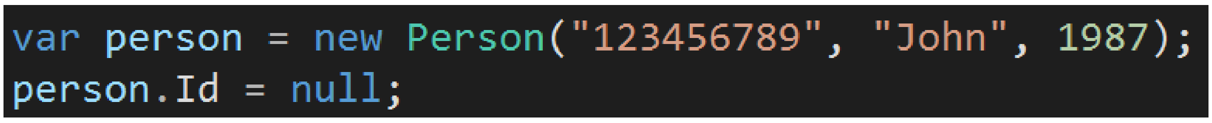
context in which those functions work, as they only depend on the input parameters, and not, for example, the state of the class they live in. Using immutable types and creating pure functions work very well together, and actually, they are two tenets of **functional programming** - a coding paradigm that grows more and more popular for its clarity as well as working great in multithreaded applications. This leads us to the next point:

3) **Thread safety**
   When working with multithreaded applications we must always be very cautious when it comes to making any assumptions about the state of an object - because it can always be the case that another thread altered this state without our knowledge. Using immutable objects wipes this problem out. If an object can't be altered, there is no need to worry that some other thread altered it, right? This makes the creation of multithreaded applications much simpler and less error-prone, and you must know that finding bugs in multithreaded applications can be extremely hard, as they often happen in a non-deterministic manner that can be extremely hard to reproduce.
4) **No invalid objects**
   Let's consider the Person class and its constructor:



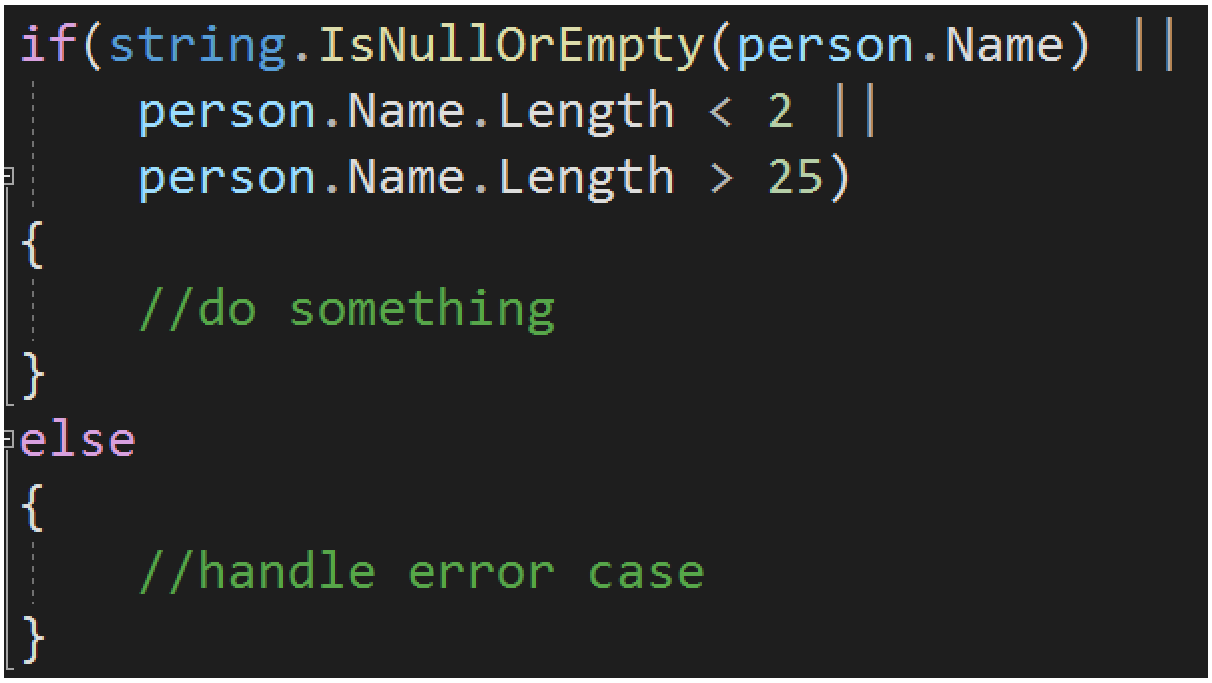
```
public Person(string id, string name, int yearOfBirth)
{
    if (string.IsNullOrEmpty(id) ||
        id.Length != 9 ||
        !id.All(character => char.IsDigit(character)))
    {
        throw new ArgumentException("Id is invalid");
    }
    if (string.IsNullOrEmpty(name) || name.Length < 2 || name.Length > 25)
    {
        throw new ArgumentException("Name is invalid");
    }
    if (yearOfBirth < 1900 | yearOfBirth > DateTime.Now.Year)
    {
        throw new ArgumentException("YearOfBirth is invalid");
    }
    Id = id;
    Name = name;
    YearOfBirth = yearOfBirth;
}
```

   Someone clearly put up a lot of work to make sure that when an object of the Person class is created, it is valid - its Id is correct, the name is not empty, the year of birth is reasonable. If objects of the Person class could be mutated, it would mean that at any time of the application execution they can be rendered invalid:



```
var person = new Person("123456789", "John", 1987);
person.Id = null;
```

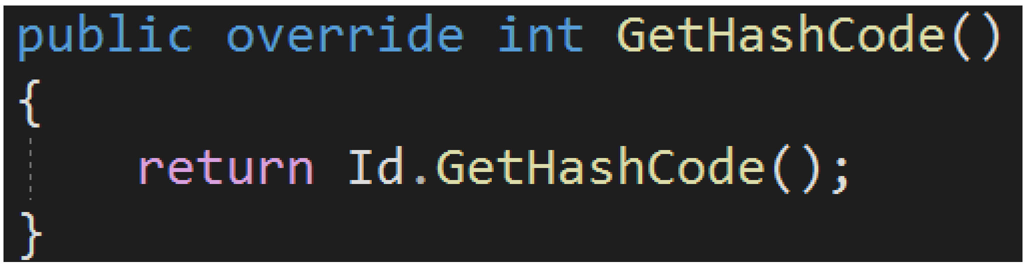
If Person's class objects can become invalid at any moment, that would mean that our code would quickly fill up with lines like that:



```
if(string.IsNullOrEmpty(person.Name) ||
    person.Name.Length < 2 ||
    person.Name.Length > 25)
{
    //do something
}
else
{
    //handle error case
}
```

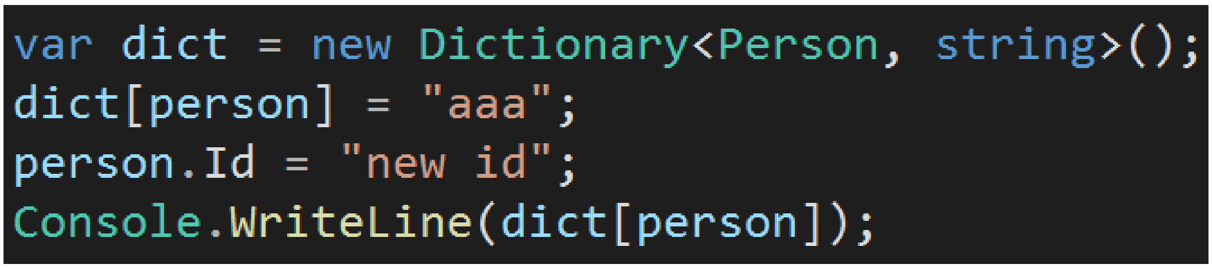
This not only is a code duplication (because we already defined this checks in the Person's class constructor) but it also creates noise in the code, making it harder to understand, maintain and test - because in each of those places we must create tests that will handle both valid and invalid person objects. The easier testing is another benefit of immutable objects, but before we move on to this point, let's consider another aspect of making objects invalid:

5) **Prevention of identity mutation.**
Imagine we want to use the Person class object as a key in the dictionary. We want to use the Person's **Id** as the hash code that the dictionary will use.



```
public override int GetHashCode()
{
    return Id.GetHashCode();
}
```

If the Id is mutable, we will lose the object in the Dictionary:

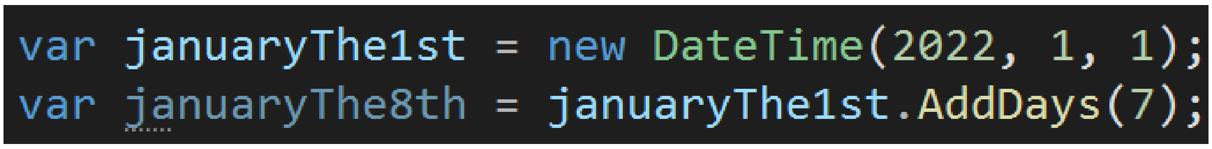


```
var dict = new Dictionary<Person, string>();
dict[person] = "aaa";
person.Id = "new id";
Console.WriteLine(dict[person]);
```

First, we've used the person object as the key in the Dictionary, using its Id's hash code. Then the id changed. Because of that, the fourth line will throw an exception, because there is no key with the hashcode built by the "new id" string in the Dictionary - the only key there is the one built with the old id. As a rule of thumb, if an object is meant to be a key in the dictionary, it should be immutable.

6) **Easier testing**
Immutable objects make code easier to understand, and they also give us a guarantee that once a valid object had been created, it will remain valid forever. This makes testing much simpler because we have fewer paths of code to test, as handling of invalid objects is simply not needed. Also, we don't need to test if a state of an object had been changed, which is sometimes tricky especially if it's the private state that changes. As mentioned before, using immutable objects makes it easier to create pure functions, and they are extremely simple to test.

All right. Seems like immutable objects can be really beneficial. But following this "nothing ever changes" rule can be demanding. After all, we sometimes need to change *something*. Let's consider the DateTime type, which is immutable in C#. It provides a method called AddDays:



```
var januaryThe1st = new DateTime(2022, 1, 1);
var januaryThe8th = januaryThe1st.AddDays(7);
```

Adding 7 days to January the 1st won't make it a different date. It will produce another date. It makes perfect sense - after all, a date never changes, and January the 1st 2022 will always be January the 1st 2022.

Such "apparent modification" of immutable objects is called **a non-destructive mutation**. It is an operation of creating a new object based on another immutable object. The immutable object won't be modified, but the result of "modification" will become a new object. We will learn more about it in the next lecture "**What are records and record structs?**"

All right. We learned what immutable types are and what are the most important benefits of using them. But we must also be aware of the important **disadvantage** they have: with the non-destructive mutation, each update of an object actually creates a new object, allocating new memory. The old object must be cleaned up by the Garbage Collector. It's usually not an issue with small types, but remember, even collections can be immutable. Imagine having a list of million elements, and that adding a new item to it means actually building a whole new collection of size million and one.

It may sound scary, but don't be discouraged to use immutable types. First of all - there are implementations of collections that actually make this quite efficient. Second - not all applications suffer from performance loss when using immutable types, and the benefits are often bigger than the costs. Garbage Collector is a smart tool, and most often you won't even notice the performance impact of introducing immutable types. Nevertheless, there are cases when performance is critical. For example, I wouldn't recommend making every type immutable when developing video games, as it would make the Garbage Collector kick off more often, and remember that when Garbage Collector works, all other threads are frozen until it finishes. In the case of video games, it could lead to a performance decrease that would be noticed by the players, and we definitely don't want that to happen.

Bonus questions:

- "**What are pure functions?**"
  Pure functions are functions whose results only depend on the input parameters, and they do not have any side effects like changing the state of the class they belong to or modifying the objects passed as an input.

- "**What are the benefits of using immutable types?**"
  The code using immutable types is simple to understand. Immutable types make it easy to create pure functions. Using immutable types makes it easier to work with multithreaded applications, as there is no risk that one thread will modify a value that the other thread is using. Immutable objects retain their identity and validity. Mutable objects make testing problematic. Testing code using immutable types is simpler.

- "**What is the non-destructive mutation?**"
  The non-destructive mutation is an operation of creating a new object based on another immutable object. The immutable object won't be modified, but the result of "modification" will become a new object. The real-life analogy could be adding 7 days to a date of January the 1st. It will not change the date of January the 1st, but it will produce a new date of January the 8th.