26. What is the purpose of the GetHashCode method?

Brief summary: The GetHashCode method generates an integer for an object, based on this object's fields and properties. This integer, called hash, is most often used in hashed collections like HashSet or Dictionary.

We are not yet quite done with collections. In the next lecture, we will discuss Dictionaries. But to understand Dictionaries we must first understand the GetHashCode method, so let's do it in this lecture.

GetHashCode is one of the few methods that belong to the System.Object type. In other words, we can call it on any object in C#. Before we understand what it does, let's see it in action:

```
var anyObject1 = "123";
Console.WriteLine(
    $"'{anyObject1}' hashcode is {anyObject1.GetHashCode()}");

var anyObject2 = 123;
Console.WriteLine(
    $"{anyObject2} hashcode is {anyObject2.GetHashCode()}");
```

```
'123' hashcode is -2093679465
123 hashcode is 123
```

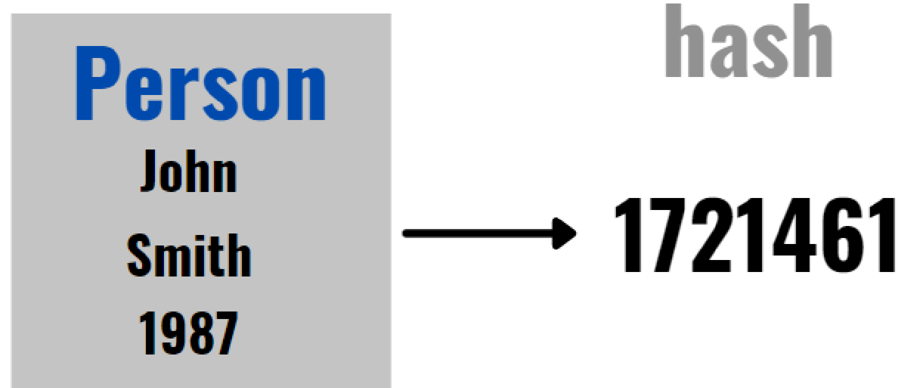
For now, those values look enigmatic, but hopefully, we will understand them better later in the lecture.

The GetHashCode method is a **hash function** implementation for an object. Let's see the definition of a hash function:

*"A **hash function** is a one-way cryptographic algorithm that maps an input of any size to a unique output of a fixed length of bits. "*

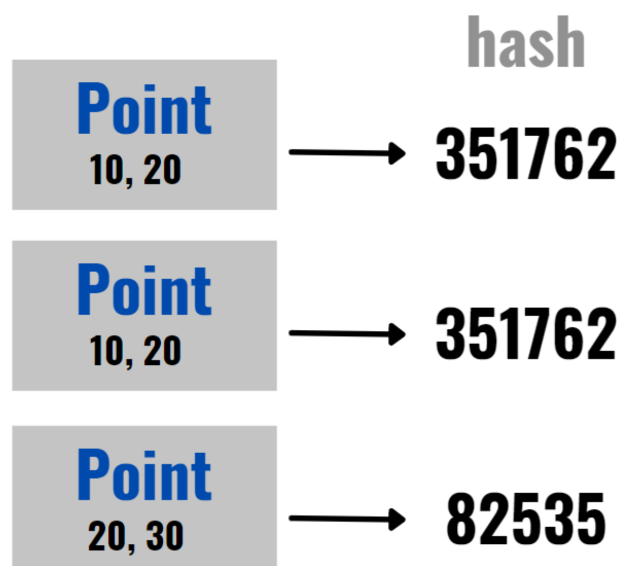
At least for me this sounds completely vague. First, let's understand what the result of this hash function is. In C# is an **integer**. In simple terms, hash is a number

calculated for some object from its components. Here is an object of Person class and some hash calculated for it.



We will take a closer look at how hash is actually calculated a bit later, but for now, let's just say that it's a function of the values of fields and properties belonging to the object. In this case, we could for example associate each letter with some number, which would allow us to translate words "John" and "Smith" to integers. Then, we would somehow combine those integers with the integer representing the year of birth, and as a result, we would have the hash code of the person. According to that, if we created another object of the Person class with name John, LastName Smith, and YearOfBirth 1987, the hash should be the same. On the other hand, if this other object had a different year of birth, its hash would be different.

Also, if we calculate the hash for the second time, the result should be the same as it was for the first time, assuming the object was not modified. Also, if we have two objects that are different instances, but we consider them equal (for example, two instances of the Point class, both having X=10 and Y=20) the hash code for both of them should be the same.



At this point you probably wonder "okay, but what is the use for hash codes?". Well, their main use is that they work as keys in **hashed collections**. This may sound cryptic by now but don't worry - we will soon learn about one of the most useful C#'s hashed collections - the Dictionary. If you used Dictionaries before you know that each value is stored under a key. The key can be any object, even a complex one, but the Dicionary needs to be able to translate it to an integer, and that's exactly where the GetHashCode method comes in handy. We will learn more about it in the lecture about the Dictionaries.

Back to the hash functions, that "map" complex objects into integers. The very important trait of the hash function is that it should **uniformly distribute its values**. That means, if I call GetHashCode methods for 100000 **different** objects of the Point type, I should get very little or no duplicated hashcodes.

But **it is possible to have duplicated hash codes.** This situation is called "**hash code conflic**t" and it's perfectly normal. Many people consider hash codes the be the "identifiers" of objects and think that two different objects of the same type can't have the same hashcodes. But this is not true, and it cannot be. Let me prove it to you.

Consider a Point type. It contains two fields: X and Y. Both X and Y are ints, so each of them can have a value between int.MinValue to int.MaxValue, so in other words - the range of the integer. For simplicity, let's say that the minimal value of the integer is -2 000 000 000 and maximal is 2 000 000 000. This means, we can have 4 billion different X coordinates and 4 billion Y coordinates, which in total gives **4 billion*4 billion** different Points, which is 16 quintillions! On the other hand, the **hash itself is an integer**, so we can only have **4 billion** different values, so much, much less than different Points. So when creating different Points, we will sooner or later simply run out of different hashes. It's sometimes referred to as the "balls into bins problem". If we have more balls than bins, and each ball is stored in some bin, it must mean that in some bins there is more than one ball.

Let's summarize the hash function. If I have two different objects of some type, ideally their hash codes should be different. If I have plenty of different objects of the same type, there should be as few duplicated hash codes as possible. Finally, if I have two objects I consider equal, their hash code should be the same.

Let's see some implementations of the GetHashCode method for some types. Here is the implementation for the int type:

```
// The absolute value of the int contained.
public override int GetHashCode() {
    return m_value;
}
```

For integers, the implementation is as simple as it can ever be. The integer value itself is a perfect hashcode. It will be the same for two equal integers, and it will be different for two different integers. There will be no duplicates at all because for each integer possible the hash code will be different.

Now, let's consider the Point type:

```
class Point
{
    public int X { get; }
    public int Y { get; }
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

Before we think of our own implementation of the GetHashCode method, let's see what is the default. As we said, the GetHashCode method is defined in the System.Object class, so I can call it on any object even if I did not override it. Let's see some Points:

```
var point1 = new Point(10, 20);
var point2 = new Point(10, 20);
var point3 = new Point(20, 30);
```

As you can see **point1** and **point2** are the same, so I would like them to have the same hash code. **point3** is different, so it should have a different hash code. Let's see the result:

```
X=10, Y=20 hash code is 43942917
X=10, Y=20 hash code is 59941933
X=20, Y=30 hash code is 2606490
```

Well, that's not what we wanted. The two first hash codes should be the same. To understand why is that so, we must understand what's the default implementation of the GetHashCode method:

- for reference types, it bases on the reference itself, so the "address" of the object in memory
- for value types it is calculated based on the values stored in the object

That explains why two Points, even with the same X and Y, have different hash codes. We declared the Point as a class, so a reference type. point1 and point2 are two different objects with two different references. The hash code is built based on the reference, so it's different for both of them.

So if we want to have the same hashcodes for the Points with the same X and Y, we can simply change the Point class to a struct:

```
struct Point
```

And now, we can re-run the application:

```
X=10, Y=20 hash code is 1644919074
X=10, Y=20 hash code is 1644919074
X=20, Y=30 hash code is 1644919094
```

Now we have what we wanted - the first two Points have the same hash codes.

But let's change it back to a class, and let's try to implement the GetHashCode method ourselves:

```
class Point
{
    2 references
    public int X { get; }
    2 references
    public int Y { get; }
    3 references
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    - references
    public override int GetHashCode()
    {
        //?
    }
```

Most of the base types in C# already provide a good implementation of the GetHashCode method. Those methods are usually strongly based on pure math and also pretty low-level, and because of that, I don't want to get into details on how they work. Just so you have an idea, here is a **fragment** of the GetHashCode implementation for string:

```
// 32 bit machines.
int* pint = (int *)src;
int len = this.Length;
while (len > 2)
{
    hash1 = ((hash1 << 5) + hash1 + (hash1 >> 27)) ^ pint[0];
    hash2 = ((hash2 << 5) + hash2 + (hash2 >> 27)) ^ pint[1];
    pint += 2;
    len  -= 4;
}

if (len > 0)
{
    hash1 = ((hash1 << 5) + hash1 + (hash1 >> 27)) ^ pint[0];
}

int     c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
```

As you can see this is pretty low-level stuff. Luckily for us, the hard work has already been done by others. When defining custom types, we can simply combine the hashcodes of the values stored in the object into a single hashcode. For the Point class, it would look like this:

```
public override int GetHashCode()
{
    return HashCode.Combine(X, Y);
}
```

HashCode.Combine takes any objects as parameters, so for example for a person class we could easily use it like this:

```
class Person
{
    public string FirstName { get; }
    public string LastName { get; }
    public int YearOfBirth { get; }

    public override int GetHashCode()
    {
        return HashCode.Combine(FirstName, LastName, YearOfBirth);
    }
```

Also remember, that we do not always need to combine all properties and fields of a type to get a valid hash code. For example, if we had SocialSecurityNumber in the Person class, which by definition identifies a person, it would be perfectly fine to use it as the only component of the hash code. We always consider two Person objects equal if they have the same social security number, and we can ignore other fields (if they were different, it would most likely mean there is some error in data itself, as two different people should never have the same social security number).

We know how to implement the GetHashCode method now, but the question that we need to answer is this: when should do it?

The answer is simple - if the type is going to be used as a key of any hashed collection, like a Dictionary or the Hashtable, and the default implementation is not working for us.

For reference types, we usually don't want the default GetHashCode, as it compares objects by reference. As with the Point class - we had two Point objects with the same X and Y, yet their hash codes were different, so when used as keys in the Dictionary, they would be considered two different keys. In this case, we usually want to override the GetHashCode method and HashCode.Combine can be a great help (of course, in some situations hashes based on the reference itself are fine. It all depends on the context).

Later in the course, we will learn about records. **Records** are reference types that provide their own, value-based GetHashCode method.

For value types, it is a bit tricky. There is a default implementation that works fine and uses the values stored in the fields or properties of the type to calculate the hash code. The problem is that this default implementation uses **reflection**, and as we learned in the "What is reflection?" lecture, it's painfully slow. Because of that, it's a good idea to provide a custom implementation of the GetHashCode method

in value types we create, especially if they are going to be used as hashed collection keys a lot.

When overriding the GetHashCode method it is important to also override the Equals method. We will explain the reason for that in the lecture about the Dictionaries.

Let’s summarize. The GetHashCode method generates an integer for an object, based on this object’s fields and properties. This integer, called hash, is most often used in hashed collections like HashSet or Dictionary.

Bonus questions:

- "**Can two objects of the same type, different by value, have the same hash codes?**"
  Yes. Hash code duplications (or “hash code conflicts”) can happen, simply because the count of distinct hash codes is equal to the range of the integer, and there are many types that can have much more distinct objects than this count.

- "**Why it may be a good idea to provide a custom implementation of the GetHashCode method for structs?**"
  Because the default implementation uses reflection, and because of that is slow. A custom implementation may be significantly faster, and if we use this struct as a key in hashed collections extensively, it may improve the performance very much.