8 things about Records in C# you probably didn't know

Records are the new data type introduced in 2021 with C# 9 and .NET Core 5.

public record Person(string Name, int Id);

Records are the third way of defining data types in C#; the other two are class and struct.

Since they're a quite new idea in .NET, we should spend some time experimenting with it and trying to understand its possibilities and functionalities.

In this article, we will see 8 properties of Records that you should know before using it, to get the best out of this new data type.

## 1- Records are immutable

By default, Records are immutable. This means that, once you've created one instance, you cannot modify any of its fields:

var me = new Person("Davide", 1);

me.Name = "AnotherMe"; // won't compile!

This operation is not legit.

Even the compiler complains:

Init-only property or indexer 'Person.Name' can only be assigned in an object initializer, or on 'this' or 'base' in an instance constructor or an 'init' accessor.

## 2- Records implement equality

The other main property of Records is that they implement equality out-of-the-box.

[Test]

public void EquivalentInstances\_AreEqual()

{

var me = new Person("Davide", 1);

var anotherMe = new Person("Davide", 1);

Assert.That(anotherMe, Is.EqualTo(me));

Assert.That(me, Is.Not.SameAs(anotherMe));

}

As you can see, I've created two instances of Person with the same fields. They are considered equal, but they are not the same instance.

## 3- Records can be cloned or updated using 'with'

Ok, so if we need to update the field of a Record, what can we do?

We can use the with keyword:

[Test]

public void WithProperty\_CreatesNewInstance()

{

var me = new Person("Davide", 1);

var anotherMe = me with { Id = 2 };

Assert.That(anotherMe, Is.Not.EqualTo(me));

Assert.That(me, Is.Not.SameAs(anotherMe));

}

Take a look at me with { Id = 2 }: that operation creates a clone of me and updates the Id field.

Of course, you can use with to create a new instance identical to the original one.

[Test]

public void With\_CreatesNewInstance()

{

var me = new Person("Davide", 1);

var anotherMe = me with { };

Assert.That(anotherMe, Is.EqualTo(me));

Assert.That(me, Is.Not.SameAs(anotherMe));

}

## 4- Records can be structs and classes

Basically, Records act as Classes.

public record Person(string Name, int Id);

Sometimes that's not what you want. Since C# 10 you can declare Records as Structs:

public record struct Point(int X, int Y);

Clearly, everything we've seen before is still valid.

[Test]

public void EquivalentStructsInstances\_AreEqual()

{

var a = new Point(2, 1);

var b = new Point(2, 1);

Assert.That(b, Is.EqualTo(a));

//Assert.That(a, Is.Not.SameAs(b));// does not compile!

}

Well, almost everything: you cannot use Is.SameAs() because, since structs are value types, two values will always be distinct values. You'll get notified about it by the compiler, with an error that says:

The SameAs constraint always fails on value types as the actual and the expected value cannot be the same reference

## 5- Records are actually not immutable

We've seen that you cannot update existing Records. Well, that's not totally correct.

That assertion is true in the case of "simple" Records like Person:

public record Person(string Name, int Id);

But things change when we use another way of defining Records:

public record Pair

{

public Pair(string Key, string Value)

{

this.Key = Key;

this.Value = Value;

}

public string Key { get; set; }

public string Value { get; set; }

}

We can explicitly declare the properties of the Record to make it look more like plain classes.

Using this approach, we still can use the auto-equality functionality of Records

[Test]

public void ComplexRecordsAreEquatable()

{

var a = new Pair("Capital", "Roma");

var b = new Pair("Capital", "Roma");

Assert.That(b, Is.EqualTo(a));

}

But we can update a single field without creating a brand new instance:

[Test]

public void ComplexRecordsAreNotImmutable()

{

var b = new Pair("Capital", "Roma");

b.Value = "Torino";

Assert.That(b.Value, Is.EqualTo("Torino"));

}

Also, **only simple types are immutable**, even with the basic Record definition.

The ComplexPair type is a Record that accepts in the definition a list of strings.

public record ComplexPair(string Key, string Value, List<string> Metadata);

That list of strings is not immutable: you can add and remove items as you wish:

[Test]

public void ComplexRecordsAreNotImmutable2()

{

var b = new ComplexPair("Capital", "Roma", new List<string> { "City" });

b.Metadata.Add("Another Value");

Assert.That(b.Metadata.Count, Is.EqualTo(2));

}

In the example below, you can see that I added a new item to the Metadata list without creating a new object.

## 6- Records can have subtypes

A neat feature is that we can create a hierarchy of Records in a very simple manner.

Do you remember the Person definition?

public record Person(string Name, int Id);

Well, you can define a subtype just as you would do with plain classes:

public record Employee(string Name, int Id, string Role) : Person(Name, Id);

Of course, all the rules of Boxing and Unboxing are still valid.

[Test]

public void Records\_CanHaveSubtypes()

{

Person meEmp = new Employee("Davide", 1, "Chief");

Assert.That(meEmp, Is.AssignableTo<Employee>());

Assert.That(meEmp, Is.AssignableTo<Person>());

}

## 7- Records can be abstract

...and yes, we can have Abstract Records!

public abstract record Box(int Volume, string Material);

This means that we cannot instantiate new Records whose type is marked ad Abstract.

var box = new Box(2, "Glass"); // cannot create it, it's abstract

On the contrary, we need to create concrete types to instantiate new objects:

public record PlasticBox(int Volume) : Box(Volume, "Plastic");

Again, all the rules we already know are still valid.

[Test]

public void Records\_CanBeAbstract()

{

var plasticBox = new PlasticBox(2);

Assert.That(plasticBox, Is.AssignableTo<Box>());

Assert.That(plasticBox, Is.AssignableTo<PlasticBox>());

}

## 8- Record can be sealed

Finally, Records can be marked as Sealed.

public sealed record Point3D(int X, int Y, int Z);

Marking a Record as Sealed means that we cannot declare subtypes.

public record ColoredPoint3D(int X, int Y, int Z, string RgbColor) : Point3D(X