# Top-Level Statements and Records Demo

## Start

1. Have VSIXs installed
2. Open ComingSoon.sln
3. Open Program.cs

using System.Console;

class Program

{

static void Main()

{

var person = new Person

{

FirstName = "Scott",

LastName = "Hunter"

};

DisplayPerson(person);

static void DisplayPerson(Person person)

{

WriteLine($"{person.FirstName} {person.LastName}");

}

}

}

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

## Top-Level Statements

This is a pretty simple C# application, but there’s so much ceremony! Why do we also have a static Main() method inside of a class? Couldn’t we just remove all that?

Remove the Program and Main() declarations, leaving the code inside Main().

using System.Console;

var person = new Person

{

FirstName = "Scott",

LastName = "Hunter"

};

DisplayPerson(person);

static void DisplayPerson(Person person)

{

WriteLine($"{person.FirstName} {person.LastName}");

}

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

Now, we can continue writing code at the top-level as if I were in my Main method. Note that the DisplayPerson() method still works because it is a *local function* in my Main method. However, I can’t write any top-level code below where the Person class is defined. That’s where “the Main method stops”.

## Init-only Properties

Our Person class is somewhat representative of a general trend in C# that can often be a bit frustrating: C# is really fantastic when you’re working with mutating data, but seems to penalize you when you need to work with immutable data. In this case, we have public, mutable auto-properties so that we can construct Person with an object initializer. That’s a really convenient and popular style! However, that means that someone could come along later and mutate my object:

var person = new Person

{

FirstName = "Scott",

LastName = "Hunter"

};

person.LastName = "Hanselman";

But what if I want those properties to be immutable? In that case, I would need to add a constructor because read-only auto-properties can only be set in a constructor, and then I wouldn’t be able to use an object initializer. If only there were a way to define the auto-properties so that they can be assigned during *initialization* and not just *construction*.

That’s a new feature we call init-only properties. (Change the ‘set’ keywords to ‘init’.)

class Person

{

public string FirstName { get; init; }

public string LastName { get; init; }

}

Note that the object initializer used to construct Person still works as expected, but the line that sets “person.LastName” after initialization now has a compiler error. (Delete the line.)

## Records

Add the “data” modifier to Person to make it a record.

data class Person

{

public string FirstName { get; init; }

public string LastName { get; init; }

}

Records are classes (or structs) that are meant to represent *values*, not objects. You can write the same things in a record that you can in any other class, but you get some extra things that are targeted at working with them as immutable data.

## With Expressions

You work with immutable data differently than mutable data. Whenever you want to represent a change, you create a new, modified version of the data representing its value after the change. This is often called “non-destructive mutation”.

Immutable libraries such as our own Roslyn syntax tree API for C# often laboriously declare a multitude of “With” methods, such as “WithFirstName()” and “WithLastName()”, which return a copy of the target object with that particular property changed.

With records in C# 9.0 we wanted to remove the need for these manual With methods, and build non-destructive mutation into the language. We can now just to “mutate” our immutable class using a with-expression:

var person = new Person

{

FirstName = "Scott",

LastName = "Hunter"

};

var otherPerson = person with

{

LastName = "Hanselman"

};

DisplayPerson(person);

DisplayPerson(otherPerson);

The with-expression uses object-initializer syntax to specify the changes, and you can of course specify more than one at a time.

## Value Equality

Records automatically provide value equality. Add the following lines after DisplayPerson(otherGuy) to demonstrate that we can create an object that has value equality with the original object, but does not have reference equality.

var originalPerson = otherPerson with

{

LastName = "Hunter"

};

DisplayPerson(originalPerson);

WriteLine($"Equals: {Equals(person, originalPerson)}");

WriteLine($"ReferenceEquals: {ReferenceEquals(person, originalPerson)}");

## Shorthand property syntax

We think that for records the vast majority of data members are going to be public init-only auto-properties. For that reason we want to make it so that the default when you write just a type and a name should be public init-only auto-property, not private field:

data class Person

{

string FirstName;

string LastName;

}

This lends itself to beautiful one-liner record declarations:

data class Person { string FirstName; string LastName; }

That’s a no-nonsense type declaration right there! However, this doesn’t work in the prototype yet, which is why you see errors in the consuming code above.

## Positional Records

For small records you are likely to want a constructor and a deconstructor. You can do that automatically by adding a parameter list onto the record name itself:

data class Person(string FirstName, string LastName);

However, this requires us to change our construction from an object initializer to a constructor call (unless we want to add a public parameterless constructor).

var person = new Person("Scott", "Hunter");

## Deconstruction

Positional records automatically generate a deconstruct method for us. So, we can easily pull the values out a record – just like we can with tuples:

static void DisplayPerson(Person person)

{

var (f, l) = person;

WriteLine($"{f} {l}");

}

## Inheritance

(Can’t show yet; back to slides)