**Session 2: Making our own Types with Classes, and Scope**

C# is an object-oriented language in which everything is an object, with the type object as the most basic types that all other types inherit from. In this session, we will learn to create our own types and how they can relate to other types using Interfaces and various scopes.



Refer back to Session 1 for more details about built-in and primitive Types

**First, a word on memory utilization and garbage collection**

Memory with C# and .NET is dynamically allocated and reclaimed by the .NET runtime as you no longer reference variables. The .NET **Garbage Collector** runs occasionally, freeing memory that was previously used by variables. There are ways to tune the garbage collector and in-depth studies of how it manages memory that are much more advanced than this part of the series. If you'd like to read more, check the [official .NET garbage collector documentation](https://docs.microsoft.com/dotnet/standard/garbage-collection/fundamentals?WT.mc_id=visualstudio-twitch-jefritz)



**The Class keyword and defining reference types**

The class keyword is used to create the most common type, a reference type, that you can create and interact with using C#. As a reference type, every instance of your classes will be stored on the memory heap.



Classes are declared with a name, a block of content, and with one or more access modifiers. That syntax is followed by curly-braces { } that enclose the contents of the class definition. Let's declare our first class for use in this notebook:



class Student



{

}



This class, named Student has no modifiers and no content. By default, classes are internal in scope - meaning that they can only be interacted with by other classes in the same library or program. You can explicitly use the internal keyword before the class keyword to ensure that your class declaration is internal in scope.



Classes are typically written into their own files, one class per file, with the filename matching the name of the class inside. In this case, we would create Student.cs



Classes can also be made public, allowing access from outside the library or program that they are compiled with. Inside the class, you can declare and add features like:



* **Constructors:** A method that is used to create an instance of the class



* **Finalizers:** A method that is called when the class is cleaned up by the garbage collector



* **Fields:** A variable owned by the class and typically used in private or protected scope, private by default when no access modeifier is used



* **Constants:** Values defined at compile time that never change for the life of the program



* **Properties:** A flexible mechanism that allows interaction with values stored in a class



* **Methods:** An action-taking code block that contains multiple statements, can accept parameters, and can return values



* **Events:** An interaction point that allows for notification of other classes when something of interest occurs



* **Delegates:** A type that can reference a method based on matching parameters and return types



* **Operators:** Defines how the class interacts with the various built-in .NET/C# operators



Let's add some properties and a constructor to our Student class:



internal class Student



{



/\* A publicly accessible constructor that doesn't do anything.



This public constructor is created by default by the compiler, and does not need to



explicitly be added if no action methods take place during construction



\*/



public Student()



{



}



// Simple publicly accessible properties that define the FirstName, LastName, and Age



// of our Student



public string FirstName { get; set; }

public string LastName { get; set; }



public byte Age { get; set; }



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student();

s.FirstName = "Mary";



s.LastName = "Contrary";



s.Age = 25;

s.IsEnrolled = true;



display(s); // This is NOT a C# or .NET function, but rather a Jupyter notebook feature to display

//the content inside parenthesis ( )

var fritz = new Student() {



FirstName = "Jeff",



LastName = "Fritz",



Age = 20,



IsEnrolled = false



};

display(fritz);



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| Mary | Contrary | 25 | True |
| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| Jeff | Fritz | 20 | False |

There's more information in the [class keyword reference in the official documentation](https://docs.microsoft.com/dotnet/csharp/language-reference/keywords/class?WT.mc_id=visualstudio-twitch-jefritz)

**Constructors**



Constructors are an active block of code that is defined with an access modifier and optionally some parameters to help with the initialization of the object. In the above class, our constructor does not take any arguments and it publicly accessible - any code can use that constructor to create an instance of this class.



The var s = new Student(); statement creates a new Student object and calls the constructor method. We can have our Student class do some initialization there if it makes sense in configuring the object.



public class Student {



public Student()



{

// A student is enrolled by default



IsEnrolled = true;



}



public string FirstName { get; set; }



public string LastName { get; set; }



public byte Age { get; set; }



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student();



s.FirstName = "Joe";



s.LastName = "Bag O'Donuts";



s.Age = 30;



display(s)



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| Joe | Bag O'Donuts | 30 | True |

Try adding other initialization values in the constructor method and tinkering with the output.

Constructors can also be **overloaded**, that is, you can create multiple constructors that accept various input parameters to allow your class to be created different ways. Let's revisit our student and add a constructor with input variables for age, since every student has an age.



public class Student {



public Student(string firstName, string lastName, byte age)



{



this.FirstName = firstName;



this.LastName = lastName;



this.Age = age;



// A student is enrolled by default



IsEnrolled = true;



}



public string FirstName { get; set; }



public string LastName { get; set; }



public byte Age { get; set; }



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student("Jeff", "Fritz", 30);



display(s)



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| Jeff | Fritz | 30 | True |

Improve the above code to ensure that EVERY student created has a name and an age.

**Chaining Constructors**



Constructors can call other constructors. That sounds strange, but makes sense if you have constructor methods that each provide a part of the definition of the class. You can call the other constructors by use the : this() method suffix. Let's improve our Student class from above to call the parameterless constructor from the constructor that takes an age parameter.



public class Student {



public Student()



{



// A student is enrolled by default



display("Running the parameterless constructor");



IsEnrolled = true;



}



public Student(byte age) : this()



{



this.Age = age;



display("Running the Age constructor");



}



public string FirstName { get; set; }



public string LastName { get; set; }



public byte Age { get; set; }



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student(30);

display(s)

Running the parameterless constructor



Running the Age constructor



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| <null> | <null> | 30 | True |

Now we can see that our properties are being initialized properly and the second constructor is called even though we are only triggered the constructor with the age.



Try setting the parameterless constructor to call the age-parameter constructor. What happens?

**Properties**



We've already seen some properties, [*auto-implemented* properties](https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties?WT.mc_id=visualstudio-twitch-jefritz) in our Student class. These declarations of a type with an access modifier and { get; set; } means that these are properties that can be read and written to in our class. Besides public, properties can be declared with private, internal, protected, and protected internal scope. We already know what private and internal are, but protected and protected internal grant access to the property to classes that **inherit** from this class. We'll learn more about inheritance in a future lesson.



The Properties are able to be assigned to *INSIDE* our class by using this.PROPERTYNAME = value notation. We're able to read and write the property value with similar notation as noted in the previous samples.



What if we wanted to do **MORE** with our properties when we set and retrieve their values? Let's take a look at making the Age property only capable of being set internally to our Student class. We can do this by adding an access modifier to the set statement.



public class Student {



private Student() { IsEnrolled = true; }



public Student(int age) : this()



{



this.Age = age;



}



public string FirstName { get; set; }



public string LastName { get; set; }



internal int Age { get; private set; }



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student(30);



display(s);

// s.Age = 25;

// display(s);



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| <null> | <null> | 30 | True |

Perhaps we know the birthdate of all students is June 1, 2000. We could update our Age property to remove the SET operation and make it only a GET with a calculation against the known birthdate.



public class Student {



public Student() { IsEnrolled = true; }



public string FirstName { get; set; }



public string LastName { get; set; }



public byte Age {



get



{



// Total days / 365 to convert to years... not 100% accurate, but close enough for a demo



return (byte)Math.Floor(DateTime.Now.Subtract(new DateTime(2000, 6, 1)).TotalDays / 365d);



}



}



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student();



// s.Age = 30;

display(s)



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| <null> | <null> | 21 | True |

**Default Values or Initializing Auto-Implemented Properties**



There are a few techniques that you can use to set default or initial values for properties when a class is created. This initial or default value can be overwritten with a different value once the class has been created.



We've seen an approach previously in the samples here where the IsEnrolled property is set to true in the constructor. Additionally, you can add the = equals operator with an initial value after a property definition to set the initial or default value.



public class Student {



public string FirstName { get; set; }



public string LastName { get; set; }



public byte Age { get; set; }



// Define the IsEnrolled property with a default value of True



public bool IsEnrolled { get; set; } = true;



}



// Create a new Student object and inspect the value of the Student to see if IsEnrolled is set to true



var s = new Student();



display(s)



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| <null> | <null> | 0 | True |

**Expression Body Definitions for Properties**



You will see folks refer to the => operator as the **fat-arrow** operator and C# folks will read this operator as "such that" with an expression on the right side with some code to be executed when interacting with the property. This [expression body definition](https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/properties?WT.mc_id=visualstudio-twitch-jefritz#expression-body-definitions) allows for a very terse expression of what to do when interacting with the property.



We can use these expression body definitions for get, set and the property itself. Take a look at the following sample code:



public class Student {



public string FirstName { get; set; }



public string LastName { get; set; }



// Let's create a simple GET that puts together the full-name of our student



// public string Name => string.Concat(FirstName, " ", LastName);



// public string Name => string.Format("{0} {1}", FirstName, LastName);



public string Name => $"{FirstName} {LastName}";



// Total days / 365 to convert to years... not 100% accurate, but close enough for a demo



// Read-only and property that is calculated each time it is accessed



public byte Age => (byte)Math.Floor(DateTime.Now.Subtract(new DateTime(2000, 6, 1)).TotalDays / 365d);



// Default value set to True



public bool IsEnrolled { get; set; } = true;



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student();



s.FirstName = "Sally";



s.LastName = "Smith";



display(s)

| **FirstName** | **LastName** | **Name** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- | --- |
| Sally | Smith | Sally Smith | 21 | True |

**Fields**



Working with the fixed birthdate in the previous example is a bit clumsy. We would REALLY like to have a BirthDate value stored with our class and the Age calculated off of that. **Fields** are variables that belong to the class and can be accessed depending on their access modifier. Some of the supported access modifiers include



* **private** fields can be accessed anywhere *inside* the class



* **public** fields can be accessed by any code that can interact with the class



* **internal** fields can be accessed by any code inside the current library or program



Let's update that last sample with a BirthDate field and a constructor that accepts a birthDate parameter.



public class Student {



// The new BirthDate field is declared with its Type and Access Modifier explicitly



private DateTime \_BirthDate;



public Student(DateTime birthDate)



{



this.\_BirthDate = birthDate;



}



public string FirstName { get; set; }



public string LastName { get; set; }



public DateTime BirthDate => \_BirthDate;



// Total days / 365 to convert to years... not 100% accurate, but close enough for a demo



public int Age => (int)Math.Floor(DateTime.Now.Subtract(BirthDate).TotalDays / 365d);



public bool IsEnrolled { get; set; } = true;



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student(new DateTime(2003, 10, 1));



display(s.Age);



display(s);

17



| **FirstName** | **LastName** | **BirthDate** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- | --- |
| <null> | <null> | 2003-10-01 00:00:00Z | 17 | True |

**Encapsulation**



A technique that you may want to use to interact with your properties and their persisted values with backing fields. This technique is called **encapsulation** and was originally the only way to create properties. Consider the FirstName property in the below sample. The value is stored and fetched from the \_FirstName field.



The LastName property also encapsulates the \_LastName field, but we have added a notification message (this display command is specific to Jupyter notebooks) to report when the LastName property is modified.



public class Student {



// The new BirthDate field is declared with its Type and Access Modifier explicitly



private DateTime \_BirthDate;



private string \_FirstName;



private string \_LastName;



public Student(string firstName, string lastName, DateTime birthDate)



{



this.\_FirstName = firstName;



this.\_LastName = lastName;



this.\_BirthDate = birthDate;



}



// The FirstName property is ENCAPSULATED in the \_FirstName field



public string FirstName {



get { return \_FirstName; }



set { \_FirstName = value; }



}



// Using encapsulation techniques like this allows us to add code to take actions when the property is interacted with



// ... In this case, we're notifying when the LastName is changed.



public string LastName {



get { return \_LastName; }



set {



// Notify the teachers about the name change



display("Name change notification!");



\_LastName = value;



}



}



// Total days / 365 to convert to years... not 100% accurate, but close enough for a demo



public int Age => (int)Math.Floor(DateTime.Now.Subtract(\_BirthDate).TotalDays / 365d);



public bool IsEnrolled { get; set; } = true;



}



// Create a new Student object and assign it to the variable name `s`



var s = new Student("Jeff", "Stevens", new DateTime(2003, 10, 1));



s.LastName = "Fritz";

display(s)

Name change notification!



| **FirstName** | **LastName** | **Age** | **IsEnrolled** |
| --- | --- | --- | --- |
| Jeff | Fritz | 17 | True |

**Constants**



Constants are values that are defined at compile time and do not change for the life of the application. The const keyword declares the constant with access modifiers and variable type. From outside the class, you can then access the constant value using the name of the class and only the constant name from inside the class.



More details about [constants](https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/constants?WT.mc_id=visualstudio-twitch-jefritz) are available in the official documentation.

public class Student {



// The new BirthDate field is declared with its Type and Access Modifier explicitly



private DateTime BirthDate;



private string \_FirstName;



private string \_LastName;



// Here is a publicly accessible constant



public const int MaximumYearsEnrolled = 5;



public Student() { IsEnrolled = true; }



public Student(string firstName, string lastName, DateTime birthDate) : this()



{



this.BirthDate = birthDate;



// We access the constant here by using its name



display("From the constructor, the maximum years enrolled: " + MaximumYearsEnrolled);



}



public string FirstName { get; set; }



public string LastName { get; set; }



public int Age { get; set; }



public bool IsEnrolled { get; set; }



}



// Create a new Student object and assign it to the variable name `s`

var s = new Student("Jeff", "Stevens", new DateTime(2003, 10, 1));



// We can display the constant here



display(Student.MaximumYearsEnrolled);



// This does not work, as the constant is only accessible through the use of the Student typename



//display(s.MaximumYearsEnrolled);



// This also does not work, as constants cannot be modified. Try un-commenting and running.



// Student.MaximumYearsEnrolled = 7;



(37,1): error CS0131: The left-hand side of an assignment must be a variable, property or indexer



**Shortcut with C# 9: Records**



Starting with [C# 9](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/record), we can use the record keyword to define a reference type. You can use the short form of record with a constructor-like signature to create a record with positional syntax for a collection of properties. The C# compiler will automatically create a constructor for you as well as the property definitions and a ToString() method.



public record StudentRecord (string FirstName, string LastName, int Age);



var s = new StudentRecord("Sally", "Smith", 30);



display(s);



display(s.ToString());



(8,1): error CS8852: Init-only property or indexer 'StudentRecord.LastName' can only be assigned in an object initializer, or on 'this' or 'base' in an instance constructor or an 'init' accessor.

