**Session 3: Making our own Types - Methods, Events, Delegates**



We learned in our last session about how to make classes with properties, constructors, and fields. We're going to continue adding features to our classes so that we can act on our classes and receive notifications about our classes.



**Methods**



Previously in Session 2, we wrote constructors to allow you to control the creation of classes. These are \_methods\_ but this time we will expand on what we've previously learned

[Methods are defined in the official documentation](https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/methods?WT.mc_id=visualstudio-twitch-jefritz) as a code block containing a series of statements that the program can execute. We're going to expand on that definition to also state that a method can (but is not required) return values to its caller. We typically think of methods as an action that our class *performs* or an *acting on* the class.



A method has a **signature** that defines how you can interact with it. The signature is followed immediately by curly braces **{ }** that wrap the code to be executed in the method and takes the following format:



*[Access Modifier]\** **Return Type** **Name**(*[Parameters]*)



By default a method with no access modifiers is considered private



Parameters are comma-delimited and defined similar to our variables with their type and name of the parameter:



int myFirstParameter



Let's take a look at some simple methods in a Student class:

class Student {



public void Enroll() {



// do enrollment steps



}



internal void LeaveClassEarly(DateTime departureTime) {



// leave class code



}



public decimal CalculateGradeForClass(string className) {



// grade calculation code



// This method returns a value of type decimal



return 0.95m;

}



public class Grade {



public string NameOfClass { get; set; }



}



}



var s = new Student();

display(s.CalculateGradeForClass("Quantum Physics"));



var g = new Student.Grade();



display(g);



0.95



| **NameOfClass** |
| --- |
| <null> |

Method access modifiers can be public, protected, private, internal, protected internal. They can also be marked static to indicate that the method runs for all instances of the class and cannot access individual fields and properties of the class. We'll see more about static and other modifiers like override, virtual and abstract in session 4.



**Overloading Methods**



Just like we can provide multiple constructors, as shown in Session 2, we can also provide multiple methods with the same name and different parameters and return types. This is called **Method Overloading** and can be very valuable to provide alternate interactions with your class that have the same intention.



Let's revisit the CalculateGradeForClass method in our last example. Typically, we would *assume* that we are calculating the grade for a student for the current semester or year. What if we wanted to go back and recalculate for a different time? A different method with a similar signature, but additionally accepting a year parameter could be added to provide additional context.



class Student {



public void Enroll() {



// do enrollment steps

}



// This signature assumes calculating the grade for a class THIS YEAR



public decimal CalculateGradeForClass(string className) {



// Let's delegate the calculation to the OTHER implementation of CalculateGradeForClass and pass this year



return CalculateGradeForClass(className, DateTime.Now.Year);



}



public decimal CalculateGradeForClass(int yearEnrolled) {



return 0.8m; //CalculateGradeForClass(className, DateTime.Now.Year);



}



// public decimal CalculateGradeForClass(int classScheduleBlock) {



// return 0.8m; //CalculateGradeForClass(className, DateTime.Now.Year);



// }



public decimal CalculateGradeForClass(string className, int yearEnrolled) {



// grade calculation code



return 0.80m;



}



public class Grade {



public string NameOfClass { get; set; }



}



}



var s = new Student();

display("Grade for this year: " + s.CalculateGradeForClass("Quantum Physics"));



display("Grade for 2019: " + s.CalculateGradeForClass("Quantum Physics", 2019));



Grade for this year: 0.80



Grade for 2019: 0.80



**The Magic params Parameter and Optional Parameters**



There is a special 'catch-all' params keyword you can use in your methods that will catch any arguments that are passed in to the method. Let's update our student with the ability to calculate grade point average:



class Student {



public decimal CalculateGradePointAverage(



int yearOfStudy,



params string[] classes) {



foreach(var c in classes) {



display("Calculating for class " + c);



}



HelloWorld();



return 0.9m;



}



/// <summary>



/// A method to say Hello World!



/// </summary>



public void HelloWorld() {



display("Hello world!");



}



}



var s = new Student();



display(s.CalculateGradePointAverage(2020, "Algebra", "History", "Computer Science", "Art", "English", "German", "Gym", "Study Hall"));



Calculating for class Algebra



Calculating for class History

Calculating for class Computer Science



Calculating for class Art



Calculating for class English

Calculating for class German



Calculating for class Gym



Calculating for class Study Hall



Hello world!

0.9



Other arguments can be made optional by adding a default value to them:



class Student {



public decimal CalculateGradePointAverage(



short enrollmentYear = 2020,



params string[] classes) {



display("Calculating GPA for year " + enrollmentYear);



foreach(var c in classes) {



display("Calculating for class " + c);



}



return 0.9m;



}



public decimal CalculateGradePointAverage(



short enrollmentYear = 0) {



if (enrollmentYear == 0) enrollmentYear = (short)DateTime.Now.Year;



display("Calculating GPA for all classes in the year " + enrollmentYear);



return CalculateGradePointAverage(enrollmentYear, "Art", "History", "Physics");



}



}



var s = new Student();



display(s.CalculateGradePointAverage(2019, "Algebra", "History", "Computer Science"));



display(s.CalculateGradePointAverage(classes: new [] {"Algebra", "History", "Computer Science"}));



display(s.CalculateGradePointAverage());



Calculating GPA for year 2019



Calculating for class Algebra



Calculating for class History



Calculating for class Computer Science



0.9



Calculating GPA for year 2020



Calculating for class Algebra



Calculating for class History



Calculating for class Computer Science



0.9



Calculating GPA for all classes in the year 2021



Calculating GPA for year 2021



Calculating for class Art



Calculating for class History



Calculating for class Physics



0.9



**Out and Ref**



Parameters passed into the method can **ALSO** take modifiers to set their interactions with the method. The parameters can be passed both in and **OUT** as well as by value and by reference into the method. This may seem a litle strange, so let's take a look at some samples.



We'll look at the out and ref keywords in this demo. out specifies that a parameter is set as an output and ref indicates that a parameter is passed by reference. If these keywords are used in a method's parameter signature, they must also be used when executing the method.



class Student {



// Enroll and return a success indicator as well as the new StudentId



public bool Enroll(



short year,



out string studentId) {



// Enrollment code



studentId = "1234567";



return true;



}



public void DoSomething(ref int myValue) {



myValue++;



display("Inside of DoSomething myValue is: " + myValue);



}



public void Hello(ref string myName) {



myName = myName + " Jones";



display("Hello: " + myName);



}



public void DeleteItemsFromShoppingCart(ref int numberOfItems) {



numberOfItems = 0;



}



}

var s= new Student();



// string id = "";



short year = 2020;



// You can declare the output variable inline



// by declaring its type like this:



// display(s.Enroll(year, out string id));



display(s.Enroll(year, out id));



display("student id: " + id);



display("Year: " + year);



int myVar = 2020;



s.DoSomething(ref myVar);



display("myVar is: " + myVar);



var myName = "Jeff";



s.Hello(ref myName);



display(myName);



True



student id: 1234567



Year: 2020



Inside of DoSomething myValue is: 2021

myVar is: 2021



Hello: Jeff Jones



Jeff Jones



The id is set in our method and MUST be set before the method ends to be returned properly. Let's look at the DoSomething method that receives a value **ByReference** as indicated by the ref keyword. Check out the behavior of myValue and tinker with adding and removing the ref keyword to see its behavior.



int myValue = 10;



s.DoSomething(ref myValue);

display(myValue);



Inside of DoSomething myValue is: 11



11



The myValue integer is a **value type** and is passed as a reference into the DoSomething method where it is modified. The reference to the myValue variable is updated inside the method, and its value reflects that update.



Reference types, those types we create with the class keyword like Student are **ALWAYS** passed by reference into methods.



class Student {



// Grade as a reference type is always passed by reference



public void CalculateGrade(Grade g) {



g.FinalScore = 0.9m;



/\*



g = new Grade {



NameOfClass = "Calculus",



FinalScore = 0.75m



}; \*/



}



public class Grade {



public string NameOfClass {get; set;}



public decimal FinalScore { get; set; }

}



}

var s= new Student();



var g= new Student.Grade() { NameOfClass="Physics", FinalScore=0.7m};



display(g);



s.CalculateGrade(g);



display(g);



| **NameOfClass** | **FinalScore** |
| --- | --- |
| Physics | 0.7 |
| **NameOfClass** | **FinalScore** |
| Physics | 0.9 |

**Delegates**



Now that we know what a method is and how to interact with them, sometimes we want to pass a pointer to that method around our program. This pointer to the method is called a **Delegate** and allows us to call the method from another location. Delegates are defined with the method signature that they need to match in order to reference that method. The [official documentation on delegates](https://docs.microsoft.com/dotnet/csharp/programming-guide/delegates?WT.mc_id=visualstudio-twitch-jefritz) has more on defining a delegate.



Delegates are sometimes referred to as **Callback Functions**



Let's take a look at passing a delegate for a pointer into another method.



class Student {



public delegate int CalculateHandler(



int myArg1,



int myArg2);



public int Calculate(



int arg1,



int arg2,



CalculateHandler handler) {



var output = handler(arg1, arg2);



return output;



}



public int Add(int arg1, int arg2) {



var output = arg1 + arg2;



display("Added: " + output);



return output;



}



public int Subtract(int arg1, int arg2) {



var output = arg1 - arg2;



display("Subtracted: " + output);



return output;



}



}



var s = new Student();



// instantiating a delegate is used with



// a NEW keyword wrapping the name of



// the method to be assigned



var calcHandler = new Student.CalculateHandler(s.Add);



s.Calculate(10, 5, calcHandler);



Added: 15



We can also assign **anonymous methods** to a delegate to be passed around. An anonymous method is defined with some paraenthesis enclosing the parameter list and an expression body indicated with the fat-arrow => notation.



Consider this code to work with our Student CalculateHandler:



var s = new Student();



var multiply = new Student.CalculateHandler(



(foo, bar) => foo \* bar);



display(s.Calculate(10, 5, multiply));



50



We can also directly cast a delegate when the type is specifed, allowing for implicit conversion to the delegate type. This makes our code a little more terse, but the intent is still clear:



var s = new Student();



s.Calculate(10, 5, s.Add);



Added: 15



**Multicast delegates**



Defined delegate types can be [**multicast**](https://docs.microsoft.com/dotnet/csharp/programming-guide/delegates/how-to-combine-delegates-multicast-delegates?WT.mc_id=visualstudio-twitch-jefritz) allowing them to point to multiple methods to be called. This sounds a LITTLE weird, but it means that we can stack executions and pass that entire stack into another method. Let's take a look at that Student and CalculateHandler from our previous example again.



var s = new Student();



// Define an initial delegate variable



var calculation = new Student.CalculateHandler(s.Subtract);



calculation += new Student.CalculateHandler(s.Add);



calculation += new Student.CalculateHandler((arg1, arg2) => {



var outValue = arg1 \* arg2;



display("Multiplied: " + outValue);



return outValue;



});



calculation -= new Student.CalculateHandler(s.Subtract);



s.Calculate(10, 5, calculation);



Added: 15



Multiplied: 50



Delegates can be combined with + and - operators to stack them and remove them from the stack to be executed. As we'll see with **Events** in the next section, adding and removing references to delegates is very important.



This is different from executing a collection of delegates as seen below.



class Student {



public delegate int CalculateHandler(int myArg1, int myArg2);



public int Calculate(int arg1, int arg2, params CalculateHandler[] handlers) {



foreach (var handler in handlers) {



display(handler(arg1, arg2));



}



return 0;



}



public int Add(int arg1, int arg2) {



var output = arg1 + arg2;



display("Added: " + output);



return output;



}



public int Subtract(int arg1, int arg2) {



var output = arg1 - arg2;



display("Subtracted: " + output);



return output;



}



}



var s = new Student();



// instantiating a delegate is used with a NEW keyword wrapping the name of the method to be assigned



s.Calculate(10, 5, s.Subtract, s.Add);



Subtracted: 5

5



Added: 15



15



**Events**



[Events](https://docs.microsoft.com/dotnet/csharp/programming-guide/events?WT.mc_id=visualstudio-twitch-jefritz) allow us to notify when something has happened inside of our class. Events build on the concept of delegates as they reference another method that should be called when the event is **raised**. We define an event with access modifiers and by .NET standard practice, two arguments:



* the sender



* a class that contains any arguments about the event being raised.



The return type from an event is a delegate of type EventHandler that defines these two arguments.



Let's look at a simple example:



class EnrolledEventArgs : EventArgs {



public short YearEnrolled {get; set;}



}



class Student {



public delegate void EnrolledEventHandler(



object sender,



EnrolledEventArgs args);



public event EnrolledEventHandler Enrolled;



public void Enroll() {



// do some long running process on another service, another machine, somewhere else



Enrolled(this,



new EnrolledEventArgs {



YearEnrolled = 2021



});



}



}



var s = new Student();



s.Enrolled += (sender, args)



=> display("I'm now enrolled for the year " + args.YearEnrolled);



s.Enroll();



I'm now enrolled for the year 2021



**Generic EventHandler definition**



Insted of writing your own delegate for each event, [you can use a generic EventHandler<T>](https://docs.microsoft.com/dotnet/csharp/programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines?WT.mc_id=visualstudio-twitch-jefritz#example) and specify the type of the EventArgs that you would like the Event to return.



class Student {



// A generic event handler that has the same signature and EventArguments of the type inside <>



public event EventHandler<EnrolledEventArgs> Enrolled;



public void Enroll() {



// do some long running process on another service, another machine, somewhere else



Enrolled(this, new EnrolledEventArgs {YearEnrolled = 2021});



}



}



var s = new Student();



s.Enrolled += (sender, args) => display("I'm now enrolled for the year " + args.YearEnrolled);



s.Enroll();



I'm now enrolled for the year 2021



**Partial Keyword**



Classes can be defined across multiple files in separate class code blocks and then stitched together as a single class object. This type of file layout and class design helps facilitate generating a class from metadata and allowing customization of that class in a second or third file. You can use the [partial keyword to define each of the parts of the class](https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods?WT.mc_id=visualstudio-twitch-jefritz).



Check out the use of the partial keyword to define two partial Teacher classes and even a **partial method** called GetAge that is defined in one part and executed from the other. Try removing the implementation of GetAge from the second class and see how the DisplayAge reflects this.



// This is one part of the class and would be stored in one file



public partial class Teacher {

public string Name { get; set; }



public void DisplayAge() {



GetAge();



}



// Definition of the GetAge method

partial void GetAge();



}



public partial class Teacher {



private DateTime \_BirthDate;



public DateTime BirthDate {



get { return \_BirthDate; }

set { \_BirthDate = value;}



}



// Implementation of the GetAge method



partial void GetAge() {



display("Get Age: " + (int)DateTime.Now.Subtract(\_BirthDate).TotalDays / 365);



}



}



var t = new Teacher { Name="Fritz", BirthDate = new DateTime(2000, 8, 28)};



display(t);



t.DisplayAge();



| **Name** | **BirthDate** |
| --- | --- |
| Fritz | 2000-08-28 00:00:00Z |

Get Age: 21



**Enumerations**



An [enumeration type](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/enum?WT.mc_id=visualstudio-twitch-jefritz) is a value type that can be used to describe a collection of related named constants that actually reference an integral type. We use the enum keyword to define these types with an access modifier and the name of the type. Values are then listed, separated by commas inside the enum block.



By default, underlying values start with the value 0 and increment by 1. Let's take a look at an example that describes the values of a dimmer lightswitch:



public enum LightSwitch {



Off,

FiftyPercent,



Bright,



Nuclear,



On



}

LightSwitch hall = LightSwitch.Bright;



display(hall);



// We can force a value back to an integer with a simple cast:



display((int)LightSwitch.Off)



Bright



0



Enumerations can be defined inside and outside of classes, and be used anywhere another variable type would be referenced. We can assign different values to the constants inside the enum with an = operator and the numeric value to be assigned.



class Student {



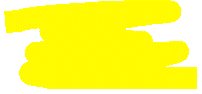
public enum EnrolledState {



NotEnrolled = 0,

Enrolled = 5,

OnMentorship = 10,



Internship = 11,

MilitaryLeave = 20

}



public EnrolledState Enroll() {



// evaluate student's scenario, make a decision



return EnrolledState.MilitaryLeave;



}



}



var s = new Student();



display(s.Enroll());

MilitaryLeave



**Enum Flags**



Enums can also be used with bitwise operations so that you can store and pass along any number of values stored in one variable. Use the [Flags] attribute to instruct the C# compiler to enable this feature and set explicit binary values for the enum constants. You can use both integer values or binary values in your definitions.



Then, you can store multiple values by using the bitwise OR operator | and can interact with those values [using other bitwise operators](https://docs.microsoft.com/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#enumeration-logical-operators?WT.mc_id=visualstudio-twitch-jefritz).



[Flags]



public enum FileAccess {



// We can define values using integers and the powers of 2:



Read = 1,



Write = 2,

Execute = 4



}



var access = FileAccess.Read | FileAccess.Write | FileAccess.Execute;



display(access);



Read, Write, Execute

[Flags]



public enum DaysOfWeek



{

// We can also use binary values directly by using the 0b binary prefix.



// The \_ character is used as a separator and is ignored by the compiler

None = 0b\_0000\_0000, // 0

Monday = 0b\_0000\_0001, // 1



Tuesday = 0b\_0000\_0010, // 2

Wednesday = 0b\_0000\_0100, // 4



Thursday = 0b\_0000\_1000, // 8

Friday = 0b\_0001\_0000, // 16

Saturday = 0b\_0010\_0000, // 32



Sunday = 0b\_0100\_0000, // 64

Weekend = Saturday | Sunday



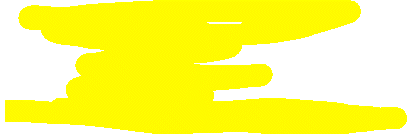
}



var workWeek = DaysOfWeek.Monday |

DaysOfWeek.Tuesday |

DaysOfWeek.Wednesday |



DaysOfWeek.Thursday |

DaysOfWeek.Friday;



display(workWeek);

Monday, Tuesday, Wednesday, Thursday, Friday

