C#

* Visual Studio Code is recommended
* Auto memory management?
* Strongly typed
* Garbage collector frees things that no longer have references: set things you don’t plan to use again to null
* Params keyword is for variable amounts of params

.net Core

* dotnet new (in folder makes it a dotnet/C# project folder)
* dotnet run
* Will expect main in Program.cs?

Basics

* Class names are capitalised
* Child : Base {this is how to extend classes}
* Child can overwrite Base methods.
* If the base method overwritten is not virtual it will complain unless you do new SameMethod() etc
* Overload is when you have the same function twice but different input
* Constructors have the same name as the class and happen on instantiation
* Parent constructor happens when Child is created
* Access: detailed on bottom of example Program.cs file

Next Section

* Object is an instance of a class: MyClass classEx = new MyClass();
* Methods are functions inside classes. Used by MyClass.myMethod();
* Static methods/variables are only made once each for all objects so most(all?) methods should be static
* Abstract classes have method declarations but not implementation. They can only be parent classes
* Import static allows the use of static members without class reference

Getters n Setters

* We use getVar() and setVar(var) inside classes to keep the private vars from being messed with as much
* We can put {get; set;} after a variable declaration to shorthand create them.

Loops

* If else; switch case
* for(int c; c < 10; c++) ; foreach(var element in array); do/while
* continue/break

Interfaces

* Contains abstract methods only (fixed in v8?)
* Interfaces always start with I
* public class MyClass : IMyclass {}
* Abstract methods are like reminders of “required functions” that may differ in implementation
* You can make something expect an interface instead of something specific if you want flexibility

Strings and arrays

* Pretty much the same as Java?
* var objs = new List<ObjName>();
* List properties: Capacity; Count; Item[index]
* List methods: Add(Obj); AddRange(IEnumberable<Obj>); (many various others)
* var myDict = newDictionary<string, Obj>(); allows key/value pairing like php
* IEnumerable<int> etc needs System.Collections.Generic
* yield return(); can give back multiple things in stuff like an IEnumerable<int> etc
* Enumerable has some methods for querying itself
* LINQ queries are for IEnumerable<T> type stuff
* var {answerSet} = {part} in {set} where [logic checks] select {part};
* LINQ queries, requires using System.Linq;
* LINQ queries only happen when the results are called for but that could be multiple times
* Saving the result with .ToList(); causes a call but we then use the saved version instead of new calls

Delegates/Lambdas

* Delegates more efficient than lambdas
* Delegate is a declared thing like an abstract but only the input/output matter

You then set things with the same input/output to them for ease of use

* We can use delegates to return our choice of method as the delegate without bothering to know what it is etc.
* Lambdas: (params) => (do stuff, this auto returns its answer if relevant)

ref and out

* Both for changing the variables outside like C pointers
* Both require method and input to specify ref/out
* out differs in that it's not needed to be set before but must be set during

Events

* Typically for user actions
* Events also help with decoupling. If something in a method changes, it and every dependent needs recompiling/redeploying.
* By having events we can just add new methods that subscribe
* Enable classes/objects to notify others on events
* Events can have multiple subscribers and each subscriber can subscribe to multiple events
* In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class
* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-subscribe-to-and-unsubscribe-from-events>
* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-raise-base-class-events-in-derived-classes>
* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-implement-interface-events>

Event Handlers

* A method inside a class that you want to use with events
* Delegates will decide the structure subscribers will use: void name(object source, EventArgs args)ventArf
* We use the default EventHandler/EventHandler<EventArgs> as our delegate for convinience

Attributes

* Put in [ ] above classes and stuff. Adds metadata like annotations
* [System.Obsolete("warning message")] //can add ,true to change warning to error
* [Conditional("DEBUG")] //only happens if the chosen preprocessing symbol is defined
* [Conditional("A"), Conditional("B")] //do if A || B
* To have “do if A && B” you nest a conditional inside another
* Conditional is only valid on void methods (and some classes?)
* Custom Attributes: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/creating-custom-attributes>
* [Serializable] allows conversion of the object into files?

Reflection

* You can use typeof() to get the implicit type of a var
* GetCustomAttributes method

Errors/Exceptions

* You should use custom exceptions always
* Try catch finally keywords
* Throw exceptions travel up until they find a try/catch
* Debug assist > Custom debug exceptions

Data management

* .net framework Data Providers
* ADO.NET Datasets
* Can be queried with Linq
* Serialization is a related thing
* XML serialization serializes only the public fields and property values of an object into an XML stream, no type information.
* DataContractJsonSerializer class is for JSON to C# and C# to JSON.
* We use [DataContract] and [DataMember] attributes for this
* Json.net is the better external one?

Threading

* Faster by multitasking
* Desktop example: primary Thread for UI; worker threads for hidden side stuff; communication possibly in its own thread
* You can set a task which will run in the background from Task Parallel library
* async tasks can await things inside themselves. Configureawait(false) to leave the waiting for later
* You would start a task that takes time and then do independent stuff before awaiting the result.
* Httpclient.getStringAsync(url) is a default async type thing

Assemblies

* Basically libraries, come as .exe or .dll
* For libraries that use .net framework you can share them between applications in the global assembly cache
* Assemblies are only loaded if required -> efficient
* Use reflection to get info programatically
* You can load an assembly for inspection using Assembly.ReflectionOnlyLoadFrom
* Different versions are considered different assemblies
* Strong naming is adding a key to prevent naming conflicts
* Strong naming makes Windows require the exact version specified. Configure binding redirects to fix
* Strong naming lets it be referenced by other Strongly named assemblies
* Strong naming lets it go in the global assembly cache
* Strong naming lets you use multiple versions at once (common with plug-in architectures)
* <https://docs.microsoft.com/en-us/dotnet/framework/tools/gacutil-exe-gac-tool>
* ^ For installing/uninstalling/exploring assemblies in the global assembly cache

File System

* System.IO contains:
* FileInfo; DirectoryInfo (instance methods)
* Path; Directory; File (static methods)
* Directory.CreateDirectory(DirPath); File.Create(filePath); File.Exists(filePath);
* filePath = System.IO.Path.Combine(directoryPath, fileName);
* File.Copy(sourceFile, destinationFile, true); //true means overwrite existing
* System.IO.File.Delete(filePath); This will throw an IOException if the file is opened by another process.

File Writing

* Various streams to read/write to various places
* FileStream; IsolatedStorageFileStream; MemoryStream; BufferedStream; NetworkStream
* System.IO.File.WriteAllLines(filePath, stringArray); //WriteAllText for single strings
* StreamWriter is a wrapper for streams that is write only and does so in plaintext

Disposal

* Disposal is for unmanaged resources
* Such as file and pipe handles, registry handles, wait handles, or pointers to blocks of unmanaged memory
* If and object implements IDisposal then they will have a Dispose(); method

Testing

* dotnet add package xUnit
* 3 easy options: xUnit, nUnit, MSTest
* dotnet new sln -o TestFolderName
* dotnet sln add ./Proj4Testing/Proj4Testing.csproj
* dotnet new xunit -o Proj4Testing.Tests
* <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test>

.Net

* dotnet new mvc //auto setup mvc style
* //dotnet new webapi (for use with React/Angular or some other external front end)
* dotnet run
* Default is localhost:5000
* MVC format for links is /[Controller]/[ActionName]/[Parameters]
* @ in cshtml is Razor syntax. You can put C# stuff in there
* Partial views are multi use page parts that get added to the other pages.
* Partial views get a copy of Parents ViewData so changes don’t persist
* @Html.Partial("\_PartialName", MyModel) //MyModel is optional

<https://www.codeproject.com/Articles/476967/What-is-ViewData-ViewBag-and-TempData-MVC-Option-2>

**Group discussion**

Dotnet stuff handles a lot so you don’t often change their main in Program.cs

Dependancy injector gives the service to any function that calls it so you don’t have to always make a new service in every function

string is shorthand for String like int is short for Int32 and long is short for Int64

Attributes can provide metadata or just be pre-function functionality like the below [Authorize]

[Authorize(Roles = "Administrator")] etc attribute which checks if the requester has permissions or something

If (a.HasValue)

b = a.Value;

else

b = -1;

Is the same as ( b = a ?? -1 )

var should be avoided but can be nice shorthand. It gets its type from assignment

Keep it for when what it will become is obvious

int.TryParse(a, out b); returns true/false and puts a into b if possible

Only static things can be used in static methods

MVC Model also contains the relevant utility methods for each Model

Microsoft docs search to get the most up to date stuff

Generics IExample<T> (The T can be anything or multiple, be descriptive) for when you want the same thing for different types so you can give any class to IExample and get the IExample functionality

Don’t use partial classes if possible

**Group presentation**

# Microsoft.AspNet.Identity

For Usermanager and other stuff like that (is this [Authorize]?)

Controller has minimal code. Changes etc go into other methods

For the dbcontext you use .Include(c => c.Catagory) to only load that specific category for efficiency

Using statement only keeps the connection open inside the { brackets } so you don’t have to manually close after

@addTagHelper (investigate this)

Only fetch filedata etc once inside HomeController and then all other Methods can get it from there

Dotnet auto assigns [Key] to Id or ({ModelName}Id)

Keep your choice of Id vs {ModelName}Id consistent

LINQ take and skip command

CORS prevents calling outside sites without permission [authenticateantiforgerytoken]

Everyone should work on everything and learn about every part