Week 1 Curriculum

* Anatomy of Code (Language, Compiler, Runtime, Platform)
* Environment Setup (IDE(Integrated Development Environment), Code Editor, Version Control, Chat
* Basic Topics (Core C#, Program Structure, Testing, Logging)
  + Logging -
* .NET Building Blocks (Framework, Standard, Core, Project, Solution, Assembly, Library, Application)
  + .NET Standard – interface that can be used in both Framework and Core CLR environments WHY? USEFUL?
  + .NET Core –
  + Project -
  + Solution – a container u**s**ed to organize one or more related code projects(class library project and corresponding test project)
  + Assembly – a collection of types and resources that are built to work together and form a logical unit of functionality(.exe or .dll files)
  + Library Application – defines types and methods that are called by an application
* Common Language Runtime (BCL, CIL, CLI, CLR, CTS, JIT, VES)
  + Base Class Library - .NET Framework provides a set of Base Class Libraries that provide functions and features that can be used with any programming language that implements .NET
  + Common Intermediate Language - .dll libraries that CLR compiles to machine code
  + Common Language Infrastructure – specification you need to follow to write your own .NET infrastructure
  + Common Language Runtime – virtual machine component of .NET Framework that manages execution(JIT compilation of .dll into machine code
  + Common Type System – standard that specifies how definitions and specific values of types are represented in computer memory to allow interoperable language sharing
  + Just In Time Compiler – Compiles .dll into machine code to be run by cpu. Provides multithreading, cross-platform, exceptions, garbage/memory management, inheritance relationships between classes and is interoperable between source language
  + Virtual Execution System – runtime system of CLI that provides environment for executin managed code to provide support required to execute the CIL instruction set (built-in data types, exception handling) CLR is .NET Framework’s implementation of the VES
* Runtime Environment (Garbage Collection, Managed, Unmanaged)
  + Garbage collection – identify and discard those objects that aren’t needed to save resources
  + Managed Code – code that is written to get services of managed runtime environment execution like CLR(multi-threading, x-platform, exceptions, garbage collection/memory management, inheritance, interoperability)
  + Unmanaged Code – Code developed outside .NET Framework(apps that don’t run under control of the CLR)
* Datatypes (Reference, Value)
  + Reference – doesn’t store its value directly, stores address where value is being stored(contains a pointer to another memory location)
    - String, arrays, class, delegates
  + Value – holds a data value within its own memory space(directly contain their values) when passing value type across methods, copies are made
    - Bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct
* Access Modifiers (Internal, Private, Protected, Public)
  + Internal – access limited to the assembly in which type/type member declared
  + Private – within class or struct
  + Protected – within class and derived class instances
  + Public – is public to all
* Extended Modifiers (Abstract, Const, New, Override, Partial, Readonly, Sealed, Static, Virtual)
  + Partial – why? Reason is if there is some kind of code generation, you don’t want your code to be deleted multiple times
  + Abstract – describe common behavior to be inherited by multiple subclasses without implementing the entire class(pure virtual method to be overriden)
  + Const – to declare constant fields and constant local not to be changed
  + New – create objects and invoke a constructor
  + Override – to override a virtual method
  + Readonly – declare a member variable a constant but allows value to be calculated at runtime(different from const that must have value set at compile time)
  + Sealed – restricts inheritance
  + Static – variables shared at class level, accessed with name of the class
  + Virtual – allows to be overriden in derived class
* Class (Constructor, Field, Method, Property, Reference Type)
* Struct (Constructor, Field, Method, Property, Value Type)
* Interface (Method, Property)
* Enum (Collection)
* Semantic Code (DRY, Comments-Inline, Comments-XML, KISS, Separation Of Concerns)
* Object Oriented Programming (Abstraction, Encapsulation, Inheritance, Polymorphism)
* Variance (As, Boxing, Casting, Is, Out, Ref, TypeOf)
  + Enables implicit conversion for methods to have more derived return type
  + Ienumerable<T>
  + Casting – among numeric types, conversions that could lose any data must use casting operator()
  + Boxing – Converting value to object(CLR wraps value in object type and stores it in heap) Implicit
  + Unboxing extracts value from object type. Explicit
  + Why? Supports unified view of the type system where any type can be treated as object
  + Is
  + Out – used for passing the arguments to methods as a reference type, used when a method returns multiple values
  + Ref – used to pass reference types by reference. Allows the called method to replace the object which the reference parameter refers to
  + TypeOf – operator keyword used to get a type at the compile time
* Collections (Array, List, Data Structures, Dictionary, Generics)
  + Generic constraints – allows for compile-time checks for type-safety which is more efficient that runtime ArrayList errors
  + Dictionary – a generic collection usually used to store key/value pairs
  + Data Structures -
  + List -
  + Array -
* Serialization (File I/O, Regular Expressions, Serializer-JSON, Serializer-Text, Serializer-XML)
  + The process of converting an object into a stream of bytes to store the object or transmit it to memory, database or a file. Main purpose to save the state of an object in order to be able to recreate it when needed.
  + Regular Expressions -
* Exception Handling (Catch, Custom Exceptions, Finally, Throw, Try)
  + Provides a way to transfer control from 1 part of the program to the other
  + Try
  + Cath
  + Finally
  + Throw
  + Custom Exceptions - They can greatly simplify and improve the error handling and thus increase the overall code quality
* Test Driven Development (Code-Coverage, Unit-Testing)
  + What it is - to write and correct the failed tests before writing new code (before development).
  + Why we use it/useful - helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests
  + Technology: xUnit,
  + Unit-Testing - is the process through which units of source code are tested to verify if they work properly
* Application Debugging (Breakpoints, Debugger, Logging)
  + Logging - Helps with debugging, especially when it’s a problem out in production.
    - Technology
      * (Nlog(XML config. and Serilog(code))
    - Syntax??
    - Logging hierarchy
      1. Fatal
      2. Error
      3. Warning
      4. Info
      5. Debug
      6. Trace
  + Breakpoints
  + Debugger
* SOLID (Single-Responsibility, Open-Closed, Liskov-Substitution, Interface-Segregation, Dependency-Inversion)
  + Single- Responsibility principle – while each class above all and also each method, each interface and each assembly etc. should have exactly 1 responsibility? What is a responsibility? Should have 1 reason to change
  + Open-Closed Principle – code should be open to extension, closed to modification(its an attitude to how we evolve our code base) not every version I shouldn’t redesign everything. My code should be able to be extended with new features later
  + Liskov-Substitution principle – anywhere in code it should be possible to replace an object with instances of their subtypes, without affecting the correctness of the behavior(I have a shape parent class: any code that accepts a shape should accept another shape)
  + Interface Segregation principle – prefer many granular interfaces rather than few broad purpose interfaces(if im writing a method of my own and I need to depend on some object that implements data access)
  + Dependency Inversion Principle – Code should depend on abstractions (interfaces) not concrete classes.
* Design Patterns (Creational-Singleton, Creational-Factory)
* Delegates (Action, Event, Function, Lambda, LINQ, Predicate)
  + Type safe pointers. You can pass methods as parameters to a delegate to allow it to point to the method. Used to define call-back methods and implement event handling
  + Action - It can contain minimum 1 and maximum of 16 input parameters and does not contain any output parameter. The Action delegate is generally used for those methods which do not contain any return value, or in other words, Action delegate is used with those methods whose return type is void.
  + Event
  + Function - ??
  + Lambda - Lambdaexpression is a concise way to represent an anonymous method. ... Lambdaexpression uses the type inference feature of C**#** 3.0 which allows the compiler to infer the type of the variable based on the context. It's is very convenient because that saves us a lot of typing
  + LINQ – language integrated query is uniform query syntax in C# and VB.NET to retrieve data from different sources and formats. Eliminating the mismatch between programming languages and databases, as well as providing a single querying interface for different types of data sources.
  + Predicate - A predicate is also a delegate like Func and Action delegates. It represents a method that contains a set of criteria and checks whether the passed parameter meets those criteria or not. A predicate delegate methods must take one input parameter and return a boolean
* Multithreading (Async-Await, Task, Thread)
  + Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval.
* Git (Add, Commit, Log, Pull, Push, Status)