**Revature Notes**

**Monday 2/18**

**Week Layout**

1 – C#, NET, OOP, testing

2 – SQL (SQL Server), Entity Framework (ORM)

3 – ASP.NET MVC, HTML, CSS [Project 0 due mid-week]

4 – DevOps, Continuous Integration and Deployment, Azure, Docker [Project 1]

5 – Service Oriented Architecture, REST, JS

6 – TypeScript, Angular (JS)

7…9 – Projects 2 and 3, Panels

10 – ??? Specialized Framework

Monday – Assessments (Quiz, Timed Coding, Mock Interviews, QC Audit)

Tuesday…Thursday – Training

Friday – Training and Project Work Time

12:30 to 1:30 is Lunch.

Sometimes Nick may leave, which is also Project Work time.

SDK: Software Development Kit, contains compiler and basic libraries (and more). Code that allows writing code.

**Commands**

cd *path:* change directory (current location).

mkdir *name*: make directory (folder) with given name.

ls: list files in current folder.

rm: Remove deletes files. -r option recursively deletes inside the folder.

.: Current directory

..: Parent directory

git clone *url*: make local copy of git repository from given url

git status: see status of the git directory you’re in

git pull: update git directory from repo.

git add *file*: to include file in what will be committed (stage?)

git reset HEAD *file:* to unstage file from what will be committed.

dotnet –version: Check version of .net.

dotnet build: Compiles code

dotnet run: runs compiled code.

dotnet new console ?

dotnet sln add *file:* Add file to a dotnet solution.

code *path:* Opens path in VScode.

tab autofills if single option (or as much shared across multiple options with same beginning). Double tab will print out all the remaining possible options.

**VSCode – C#**

alt-shift-F is auto-Format code

//compile-time type inference for variables with var, but must be initialized!

//Copies type of right hand side.

//Can use var when type is obvious from context or when it is obnoxiously long??

var somedata = "String type";

//var otherdata; //Left Statement wouldn't work

**Structure**

solution *solName*.sln //set of related projects

project/assembly *projName*.csproj //Assembly are .exe for apps or dll =dynamic linked library for libraries (What .net created automatically with dotnet new)

file (*filename*.cs)

namespace(*solName*.*name*) -> using *name*  //default System

Class (*classname*) //want classname to match filename

methods

properties

fields

Logically organized into namespaces, physically organized into projects?

**C# Tour**

Assemblies contain executable code in the form of Intermediate Language (IL) instructions, and symbolic information in the form of metadata. Before it is executed, the IL code in an assembly is automatically converted to processor-specific code by the Just-In-Time (JIT) compiler of .NET Common Language Runtime.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for #include directives and header files in C#

**Types**

char type represents a UTF-16 code unit.

Nullable value types do not have to be declared before they can be used. For each non-nullable value type T there is a corresponding nullable T?, which can hold an additional value of null.

Single- and multi-dimensional arrays of any type. For example, int[,] is a two-dimensional array of ints, and int[][] is a single-dimensional array of 1d arrays of ints. (Do not have to be declared.)

Class types support single inheritance and polymorphism. (extend and specialize base classes)

Struct types do not support user-specified inheritance; all struct types implicitly inherit from type object. Structs are value types (stores the data of the obj, not a reference to a dynamically allocated object) and thus(?) do not typically require heap allocation. Structs are particularly useful for small data structures that have value semantics, which can make a large difference in the number of memory allocations an app performs. Exs: key-value pairs, coordinates.

An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces. A class or struct that implements an interface must provide implementations of the interface’s function members.

A delegate type represents references to methods with a parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They are also similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

A delegate that references an instance method also references a particular object, and when the instance method is invoked through the delegate, that object becomes this in the invocation. Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. An interesting and useful property of a delegate is it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

The class, struct, interface and delegate types all support generics, whereby they can be parameterized with other types.

An enum type is a type with **named constants**. Every enum has an underlying type, which must be **one of the eight integral** types. The set of values of an enum is the same as its underlying.

Values of reference types are treated as objects simply by viewing the values as type object. Values of value types are treated as objects by performing boxing and unboxing operations: object o = i; // Boxing

int j = (int) o; // Unboxing

**Operators**

The assignment operators and the conditional operator (?:) are right-associative, meaning that operations are performed from right to left. For example, x = y = z is evaluated as x = (y = z).

Primary

new T(...){...}: Object creation with initializer

new {...}: Anonymous object initializer

delegate {...}: Anonymous function (anonymous method)

checked(x): Evaluate expression in checked context

unchecked(x): Evaluate expression in unchecked context

typeof(T): Obtain Type object for T

default(T): Obtain default value of type T

Unary

+x: Identity

-x: Negation

~x: Bitwise negation

(T)x: Explicitly convert x to type T

await x: Asynchronously wait for x to complete

Relational and type testing

x is T: Return true if x is a T, false otherwise

x as T: Return x typed as T, or null if x is not a T

Logical XOR

x ^ y: Integer bitwise XOR, boolean logical XOR

Null coalescing

x ?? y: Evaluates to y if x is null, to x otherwise

Conditional

x ? y : z: Evaluates y if x is true, z if x is false

anonymous function

(T x) => y: Anonymous function (lambda expression)

**Statements**

The checked and unchecked statements are used to control the overflow-checking context for integral-type arithmetic operations and conversions. (Uncheck has overflow, checked exception.)

The lock statement is used to obtain the mutual-exclusion lock for a given object, execute statements in a block, and then release the lock.

The using statement is used to obtain a resource, execute a statement, and then dispose of that resource. Ex: using (TextWriter w = File.CreateText("test.txt") ) { w.WriteLine("Line one"); }

Yield:

**Accessibility**

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are six possible forms of accessibility.

public Access not limited

protected Access limited to this class or classes derived from this class

internal Access limited to the current assembly (.exe, .dll, etc.)

protected internal Access limited to derived classes OR classes within the same assembly

private Access limited to this class

private protected Access limited to self or derived classes also within the same assembly

**Generics**

A generic class type’s definition specifies type parameters after the class name with angle brackets enclosing a list of names. Ex: public class Name<TFirst>{ public TFirst First; }

When the generic class is used, type arguments must be provided. Name<string> variable = new Name<string> { First = “one”;} string s = variable.First;

A generic type with type arguments provided is called a constructed type.

Static field has one storage location shared by all instances; non-statics each have their own.

**CLASSES AND OBJECTS**

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Hidden default is object. An implicit conversion exists from a class type to any of its base class types (upcasting). Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class.

Read-only fields are declared with a readonly modifier. Assignment to a readonly field can only occur as part of the field’s declaration or in a constructor in the same class.

**-Parameters** Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The signature of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters and the number, modifiers, and types of its parameters. The signature of a method does not include the return type.

A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed. Can be optional, by specifying a default value when the argument is omitted.

A reference parameter is declared with the ref modifier. The argument passed for a reference parameter must be a variable with a definite value, and during execution of the method, the reference parameter represents the same storage location as the argument variable.

An output parameter is declared with the out modifier. It is used for passing arguments by reference. It's similar to a reference parameter, except that it doesn't require that you explicitly assign a value to the caller-provided argument. (In the method, can assign values to args, so that the caller can use their own variables without needing a new object to hold all return values.)

A parameter array permits a variable number of arguments to be passed to a method. A parameter array is declared with the params modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type.

In an invocation of a method with a parameter array, can pass any number of args of the type, and an array instance is automatically created and initialized with the given args. If zero passed, creates an empty array. (Within the method, the parameter array is exactly like a regular array.) If exactly one passed, acts as a value parameter!

C# requires a local variable to be definitely assigned before its value can be obtained.

**Function Members** -Methods A static method (declared with a static modifier) does not operate on a specific instance and can only directly access static members.

A method declared without a static modifier is an instance method. An instance method operates on a specific invoked instance (explicitly accessed as this) and can access both static and instance members. (It is an error to refer to this in a static method.)

When a *virtual* method (declared with virtual modifier) is invoked, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a *nonvirtual* method invocation, the *compile-time type* of the instance is the determining factor.

When an instance method declaration includes an override modifier, the method overrides an inherited virtual method with the same signature; it specializes an existing inherited virtual method by providing a new implementation of that method.

An abstract method is a virtual method with no implementation. An abstract method is declared with the abstract modifier and is permitted only in a class that is also declared abstract. An abstract method must be overridden in every non-abstract derived class.

-Constructors An instance constructor is a member to initialize an instance of a class. A static constructor is a member to initialize a class itself when it is first loaded, declared with static.

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

-Properties Instead of using getters and setters, C# has properties.

private string \_name;

public string Name {

get{ return \_name; }

set { // inside "set"

// we have implicit argument "value"

// could do null/empty-checks, etc.

\_name = value;}

}

access type *name* { get; set; } = default\_value; Read only properties do not have a set.

Static properties are declared with the static modifier, and instance properties are declared without it.

When a property declaration includes a virtual, abstract, or override modifier, it applies to the *accessor(s)* of the property.

-Indexer An indexer is a member that enables objects to be indexed in the same way as an array. (Think List obj accessed thru []). An indexer is declared like a property except that the name of the member is this followed by a parameter list written between the delimiters. The parameters are available in the accessor(s) of the indexer. Indexers can be overloaded, meaning that a class can declare multiple indexers as long as the number or types of their parameters differ. Indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

-Event An event is a member that enables a class or object to provide notifications. It is declared like a field except it includes an event keyword and the type must be a delegate type.

The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handlers are present, the field is null.

Clients react to events through event handlers. Event handlers are attached using the += operator and removed using the -= operator. Example:

static void ListChanged(object sender, EventArgs e) { changeCount++; }

public static void Usage() {

List<string> names = new List<string>();

names.Changed += new EventHandler(ListChanged);

names.Add("Liz");

} //Add triggers Changed which runs ListChanged, incrementing changeCount.

-Operators All operators must be declared as public and static. An operator is a member that defines the meaning of applying a particular expression operator to instances of the class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators.

-Finalizers The finalizer for an instance is invoked automatically during garbage collection. A finalizer is a member that implements the actions required to finalize an instance of a class. Finalizers cannot have parameters, they cannot have accessibility modifiers, and they cannot be invoked explicitly. Classes should generally not implement finalizers, for many reasons.

**Arrays** Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at runtime using new, which specifies the fixed length of the array instance, and automatically initializes elements of an array to their default, for example, zero for numeric types and null for reference types. Multi-dimensional arrays of size n specified with n-1 commas in the [] delimiters. An array with elements of an array type is also called a jagged array because the lengths of the element arrays do not have to be the same: int[][] a = new int[2][]; a[0] = new int[10]; a[1] = new int[5]; Arrays can be initialized in {}, thereby skipping need to give size, or repeat new and array type.

**Interfaces** An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines. Interfaces may employ multiple inheritance, and classes and structs can implement multiple interfaces. When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type.

C# also supports explicit interface member implementations, enabling the class or struct to avoid making the members public, but these can only be accessed when of the interface type.

**Enums** Define enums when you need to define a type that can have a set of discrete values, each a named constant. They use one of the integral value types as their underlying storage. When an enum member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the enum type) or the value of the textually preceding enum member plus one. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type and is a distinct valid value of that enum type. Enum values can be converted to integral values and back using type casts.

**Attributes** C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this additional declarative information by defining and using attributes. All attribute classes derive from the Attribute base class provided by the standard library. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. The metadata defined by attributes can be read and manipulated at runtime using reflection. When a particular attribute is requested using this technique, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

**Tuesday 2/19**

**Git** Software for versioning control.

**GitHub** Hosts repositories. Adds some access controls.

Central Version Control System (VCS): One central version (everyone has to merge with?)

Distributed - : Everyone has a copy. Can agree on one (GitHub).

Working directory/tree: Has files that everyone (non-git aware programs) can see.

Index/staging area: Temporary place to prepare for commit.

Local Repo: Permanent record of history in the commits.

Remote Repo: Usually GitHub. Default for clone is origin. Someone else’s record of history.

Add: Working tree -> staging area.

Reset: Staging -> tree. (Takes out. The rest all copy over.)

Push: Local -> Remote.

Pull: Remote -> Local and leftwards.

Stash: Temp stack changes? Move stuff out of the way without committing.

Diff: Tree 🡨🡪Staging

Diff -- : Staging 🡨🡪 Local

**Modifiers**

Static classes cannot be instantiated or derived from, they’re just containers for static members.

readonly members are like const but can be set in a constructor.

abstract members must be in an abstract class (other methods can be non-abstract).

sealed classes cannot be derived from (mainly to prevent overriding).

partial classes are spread across multiple files in same namespace (allow computer generated code).

|  |  |  |
| --- | --- | --- |
|  | Class | Member |
| static | Y | Y |
| virtual | N | Y |
| override | N | Y |
| new | N | Y |
| const | N | Y |
| readonly | N | Y |
| abstract | Y | Y |
| sealed | Y | N |
| partial | Y | N |

**Errors**

In VSC, ctrl+click to see where you defined, or a simple view from a library of, what you clicked on.

Can catch all exceptions to log them, but make sure to throw again if you don’t handle it. Can wrap exception in another exception if want both stack traces.

Finally block happens regardless of if an exception happened or not, usually to close resources. Code after finally block won’t happen if exception wasn’t handled.

**Wednesday 2/20**