**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, 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 like 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 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 if the number or types of their parameters differ. They 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 an 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 an 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 an 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.)

Commit: Staging area -> Local.

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 the same namespace (helps 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 metadata of what you clicked.

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**

**GIT**

Origin/master is the remote master branch of what is on GitHub. Your local repo can commit but pushing is what adds your commits to master and updates what your repo considers the master.

Branch is a reference to a commit, which includes the history of its commits, and they can only move forwards.

You can create another branch and merge it with master (git merge *name:* merges named branch with current repo).

Pushing a local branch which has not been merged with master will cause an error, you need to use a different version of push. This will create another remote branch on github, origin/*name,* which now needs a tracker on the local repo.

Pull (fetch and merge – updates remote tracking branches and merges the remote tracking branches into associated current local branches)

2nd person to want to make changes to master must resolve conflicts using merge. New master is child of the master updated from the 1st person and your branch.

HEAD is what commit you are looking at.

git checkout *name*: Switches to named branch. -b creates a new branch from the current commit.

**Collections**

Like arrays, array lists are not really used. Replaced by Generic's List<T>.

Set has Math def of set -> Not ordered, no repeats. Designed to check for membership (ContainsKey or ContainsValue). Fast to Add or Remove.

Dictionary Mapping: Key-value pairs. Both can be any type.

Stack: LIFO. Queue: FIFO. List:

Classes are reference types. Value types like int are not. Structs are value types!

Value types only give a copy of their value(s), so modifying does not change the original.

Reference types give a reference to an object, so modifying them does affect the original.

Value types are deleted from memory once they are no longer in scope.

Reference types need garbage collection since it needs to wait until all variables that reference it pass out of scope.

Managed code has periodic garbage collection to delete unreachable objs.

Unmanaged code needs to manually delete.

Programmer time more valuable than computer time, so use managed code.

Normally == compares value types by value and reference types by reference.

The exception is strings: overloaded to prevent awkward string.Equals()

Should compare object values using .Equals()

Value types are derived from object, so can implicit upcast to an object variable, called boxing.

New object treated as reference type. (Changing original doesn't change the new!)

Can unbox as well by downcasting back to value.

**Testing**

Arrange, Act, Assert

A test method returns void and should be named what it tests.

Should make initial tests that fail and fix them by writing the correct code.

Attribute xunit test with Fact.

**Thursday and Friday 2/21 + 2/22**

Facts don't allow any parameters.

Theories accept sets of parameters, to run the test against all of them.

Flaw in trying to override with "new". you can still run the parent class's implementation using upcasting.

Params type[] name: lets you pass any number of parameters.

Documentation should contain tags such as:

<summary>…</summary>, which gives a brief description of the class/method.

<param name=”*name*”>…</param>, which describes the named parameter.

<exception cref="*name*”>…</exception>, which describes the condition of the exception.

<returns>…</returns>, which describes what the returned value should be.

<remarks>…</remarks>, which gives any further remarks not in summary?

**Constructors** are not inherited, but can manually inherit them by defining child constructor with base:public childClassname() : base() {}, which is done by default already, or

public childClassname(type value) : base(value) {} Thus it uses the parent’s constructor.

C# prefers thin constructors, where setting ??

properties or validate method ??

**.NET** platform – Many languages. Limited interoperability.

C# most popular. Python, Java, TypeScript/JS, F#, R, VB.NET.

Common Language Infrastructure (CLI, shares acronym with Command Line Interface).

Virtual Execution System (VES) is equivalent to JVM.

.NET code (.cs file) compiled by C# compiler into assembly/Common Intermediate Language (CIL) in a .dll or .exe (bytecode), creating cross-platform instruction set for VES. (CIL is barely below other high-level langs.) Has a Common Type System (CTS) for converting basic types, and a Base Class Library (BCL) for sharing often used classes. Exception-handling, types, managed environ (garbage collection). Happens in build.

JIT (Just In Time) Compiler converts that into native instructions for specific CPU right when it is needed. (In .net, this is no longer considered assembly.) Happens in run.

.NET implementations:

.NET Framework is a Windows only implementation. 4.7 most recent.

(Mono ported .net framework to Mac and Linux. Exists, but kind of obsolete.)

.NET Core is a truly cross-platform implementation. 2.2 most recent.

Common Language Runtime (CLR) is the VES in both Framework and Core.

.NET Standard is the 100% common subset (intersection) between Framework and Core.

Not an actual implementation. Lots of libraries. Some are still platform-specific but being added to core.

**OOP**

Abstraction: “Separation between needed functionality and implementation details.”

Knowing what something does without needing to know how it does it. e.g. properties, methods

Encapsulation: “Restricting access to information within objs? Packaging related things together and treating them as one unit.” e.g. access modifiers? Objects containing fields and their methods

Polymorphism: “Many implementations behind a common contract?” Many names. e.g. Method overriding (class redefines parent method) and overloading (same name but dif params)

Inheritance: “Ability for a class to take and extend behavior from another class.” Base, parent fields and methods are present in child, which can add more or specialize them. Is-a relationship.

(Composition: Has-a relationship. e.g. Class has a member of another class, which already defined methods and properties for certain fields. )

**SOLID** principles-

**S**ingle-responsibility principle**:** Class/method should do just one thing; If it does more, divide into smaller pieces.

**O**pen-closed principle: Entities should be open for extension, closed for modification. Should generally add, but not change (affects backwards compatibility and maintenance). e.g. Virtual keyword in parent needed to override.

**L**iskov Substitution principle: Objects should be replaceable by instances of their subtype without affecting correctness of code. e.g. child implicit upcast to parent should run.

**I**nterface Segregations principle:Heavy interface should be split up into multiple light interfaces. e.g. only want some methods, don’t couple with unwanted functionality. Splitting doesn’t really add any cost; Modularity adds benefits.

**D**ependency Inversion Principle**:** Instead of high-lvl code depending on low, or vice versa, both should depend on interfaces. e.g. easier to test and extend using interfaces. (unit testing with dummies, vs integration testing?) [Abs classes can’t create new objs, which would create more dependencies, so better than normal classes, but still prefer interfaces.]

**Serializing**

Serialize, meaning to put in a row, will collect data from across memory into another (text or binary) format. Ideally reversible, thereby allowing transmitting across network.

Doesn't transmit type info or methods. Know other side needs class definition to recreate.

**Asynchronous methods** need async modifier and return a **Task<thing>,** or Task for void, and should have async in name (for documentation purposes).

**.Result** makes you wait for code to finish which is bad, cause contrary to asynch purpose, but we have to because it's Main If Task is returning void, use .Wait() instead, also makes synchronous.

FileMode.Create (over)writes to file with given name. FileMode.Open to read file.

FileStream obj might be null, close with ?.Dispose in finally (stream uses IDisposable).

XmlSerializer needs a stream and does not know about generics, so give typeof obj being serialized: var *name* = new XmlSerializer(typeof(T)); serializer.Serialize(*fileStream*, *obj*);

return (<T>) serializer.Deserialize(*fileStream*); //Deserializer needs to be type casted!

**“using statement”** is different from “using directive” (at top of file). using statement automatically disposes created resources when exiting its block, so we don't need try finally dispose: // should still be try-catching throughout this method

using (var memoryStream = new MemoryStream()) {

using(var fileStream = new FileStream(filename, FileMode.Open)) {

await fileStream.CopyToAsync(memoryStream); //XmlSerializer.DeserializeAsync doesn't exist, so we are using fileStream.CopyToAsync(memoryStream)

//await allows other code to execute while we are running the statement

}

memoryStream.Position = 0; //reset “cursor” to start in order to read contents later

return (T)serializer.Deserialize(memoryStream);

}

**JSON**.NET (aka NewtonSoft JSON) is third-party. Json doesn't care about some xmlAttributes, but does care about XmlIgnore?

File.ReadAllTextAsync(“*fileLocation*”); //Asynchronously read data from file

JsonConvert.DeserializeObject<T>(*readData*); //deserializes json formatted data

JsonConvert.SerializeObject(*dataToWrite*); //serialize data to json format

File.WriteAllTextAsync(“*fileLocation*”, *serializedData*); //Asynch write data to file

Normally, backslash is escape character in strings. To treat backslash literally, use **@ string.**

A **"lambda"** is like a method that's anonymous and can be treated like an ordinary value/object.

**LINQ**

Average<TSource>(this IEnumerable<TSource> source, Func<TSource, type> selector);

//Calculate the average of an enumerable obj, based on results from a passed selector function, which takes an input of the Collection’s type and returns type (int, double, etc.) value.

Similar for Sum. Ex: list.Sum(x => x.Count(c => c == 'a') ); //For each string in list, for each character in string, if char is a, add 1 to count for that string, add all returned counts.

Can use Func variable, or method, that matches the typing.

In Visual Studio, can right click and go to properties to get a GUI for modifying the .csproj. (Could also modify directly, but don’t know how to do so properly.)

This enables compiling unsafe code, such as pointers (literal memory location, less abstract than ref variables; shouldn’t be used). In main, place into unsafe block.

We need unsafe code when we are using unmanaged code, like Windows API, that require pointers. Otherwise, avoid it.

**IS null** is a duplicate comparator for == null.

Global assembly cache (C:\Windows\assembly) stores dependencies using strong naming.

Added package manager NuGet for specific projects. dotnet restore fetches dependencies, since integrated with NuGet. (If NuGet is on path, can call nuget restore.)

**NLog** package dependency per project, NLog.config just downloads a .config you can copy over.

<!--

Write events to a file with the date in the filename.

<target xsi:type="File" name="f" fileName="${basedir}/logs/${shortdate}.log"

layout="${longdate} ${uppercase:${level}} ${message}" /> -->

<!--

Write events to the console.

<target xsi:type="Console" name="c" layout="${longdate} ${uppercase:${level}}

-->

Make sure to add target name to logger: <logger … writeTo=”f, c” />

Add “using NLog;” directive to use logger.

Create logger obj: ILogger logger = LogManager.GetCurrentClassLogger();

Multiple event levels, Debug, Info, Warn, Error, and Fatal, which is the worst. Logger logs those of minlevel=”*level*” and above.

Log with: logger.Debug(string); logger.Error(exception); etc.

**Regex** needs using System.Text.RegularExpressions

Regex uses different escape characters, use @strings.

regex syntax - "character classes":

\d for all digits,

\w for all "word characters" (letters, numbers, and underscore)

\s for all whitespace, most of these have an "opposite" version with uppercase

\S for all non-whitespace

[abcd] means, one character, EITHER a, b, c, or d.

a\* means 0 to many a chars. a+ means 1 to many a chars

() are for surrounding groups of characters that you want to extract later.

Group 0 will be whole match and remaining groups are in order of parenthesis.

**Separation of Concerns**:Minimize tight coupling, use loose coupling (easy to switch to another implementation).

**Comment your code**: Use XML comments for the public API (not needed for private), so someone else can auto-generate pages to understand your code.

**DRY**: Don’t Repeat Yourself. Repeated code should be in a separate class/method.

Keep It Simple

Yield return for things with IEnumerable, to return things one at a time. (holds location in method for next call)

**Project 0 initial requirements:** (Come up with idea)

Customers

Stores/locations //location field in store?

Order/order history //OrderHist collection of orders?

Statistics //methods to run

Inventories (per store. when ordered, decrement)

Something you can buy that’s not 1:1, e.g. pizza is dough and cheese.

Buying in multiplies.

Some constraints, e.g. no more than 10 pizza per order, that user can then correct.

Leverage Menu-based console interface? Simple console: 1+enter = make purchase, 2+enter=review orders, etc. DO LAST!

Make sure to keep console and library parts separate. (Library will be used in next proj.)

Have Testing.

Use Serialization to save and load to/from file(s). //Will be replaced by sql.

**More**

3 projects – UI/ConsoleApp, Library, and Tests.

Suggest repositories.

By Wednesday have library + tests, and then console app.

**Tuesday 2/26**

Private Class: Normally don’t exist, but can nest a private class inside another class.

Pluralsight is helpful study website. Microsoft Dev Essentials gives free one month.

Microsoft Virtual Academy as well. Programming in C# jump start recommended for basics.

**Azure** – Create a resource: “Create new” button, name resource group.

Needs a server: “Create new” button, globally unique name. Give admin a name (not admin) and password. Pick server location. (Don’t have to redo all this but need to know how.)

Name database.

How to change firewall to allow new IP to access it: Go to server. Click firewall settings. Add name, current address (or range) and SAVE.

**In MSSMS**, connect to *servername*.database.windows.net. Make sure correct database is selected in drop down list on toolbar! Often resets.

ctrl+n is new query, click button in toolbar.

Strings are in single quotes ‘’, not doubles “” ?

**Relational Databases**

Divided into Tables, which are made up of Rows and Columns.

Database Management Systems (DBMS)

Relational Database Management Systems (RDBMS)

Structured Query Language (SQL)

SQL Server (AKA T-SQL or Transact SQL). Note: a SQL server is both software language and database, both are named SQL Server in this case. T-SQL is just a name for the language.

Other versions like MySQL, Oracle SQL, PostgreSQL, SQLite.

Functional Dependency: For two sets of columns, X and Y, X🡪Y if for each possible set of X values, there is exactly one possible set of Y values. Given same X, you know same Y. (Y is a fact about X.) Often sets are one column, but not always!

Candidate Key (of a table): Minimal set of columns in a table (as in cannot remove one column and still have the following be true) such that every other column depends on the set.

Can have multiples in a table.

The values of any candidate key can uniquely identify a row.

Composite key: A key of more than one column.

Primary Key (of a table), PK: One of a table’s candidate keys that we choose to actually identify the row. Remember can be multiple columns. (Generally, choose simplest to compare and begin name with \_ . )

Foreign Key: A column that references a primary key of a table to establish a relationship between the rows (can be two separate tables or same table).

Non-atomic value: Column with more than one value (a list). Hard to read values, you have to search through it!

Update anomaly: Problem arising from redundant data, you might not update all copies. (Similar to copy-paste problem; use methods)

Deletion anomaly: If we delete some data (row), we sometimes lose other data.

(Similar to tight-coupling problem.)

Insertion anomaly: Impossible to insert some data without associated data. Related to above.

Normalization:

3rd is considered fully normalized. For simple databases, higher forms can often satisfy the lower forms. Generally, not enough advantage to enforce lower forms (BCNF, 4-6, and lower).

1st: No duplicate rows, enforced with primary key. (Theorists assume this from definition of Relation DB.) Needs atomic values. No “repeating groups of columns” (shouldn’t have color1, color2, color3).

Could copy rows but with different color values, where color column is added to primary key (forcing composite key). Or could add rows (new pk value, individual color values).

Best is to make another table, in 1:N relationship, for the N side of the relationship and use a foreign key in the 1 side to N’s table, or, for M:N relationship, make a new table for both the M and N, with a composite primary key of both, which are FKs back to appropriate tables.

2nd: No non-candidate key column may depend on part of any composite candidate key; No partial dependency, depend on whole key. If no composite keys, then 1st NF implies 2nd NF.

3rd: No non-candidate key column should depend on any other non-candidate key column; No transitive dependencies (everything already dependent on candidate keys).

Summary: (1st) Depend on the keys, (2nd) the whole keys, and (3rd) nothing but the keys.

Normalization – Pros: Less redundancy means less errors. Easier to evolve data model. Easier to read and write consistently.

-Cons: Makes some queries slower (Faster to read denormalized). [Many get faster.]

**SQL**

--Comment with dash dash

Many databases per server.

Azure SQL DB doesn’t support USE *dbname;* Must use dropdown.

Each database has multiple schemas, which namespace/scope for database objects?

Whitespace doesn’t matter. Semicolon isn’t needed. SQL syntax is case insensitive.

Select (highlight) statement and press F5 to run.

SQL’s default string comparison is case insensitive. It depends on “collation”, which is configurable.

Strings are in single quotes! ‘ ’

SQL doesn’t need to track conceptual candidate keys and dependencies.

SELECT statement: Returns a value. (Others just do something, like void return.)

SELECT \* FROM *tablename* is all columns and all rows in given table.

Replace \* with column names to return those specific named columns.

Can rename using AS [*name*]. SQL Server uses [], standard is “”. Can use without either, if name has no spaces in it.

Can compute new values from column values. Ex: Select FirstName + ‘ ’ + LastName AS [Full Name] From SalesLT.Customer; -- Can’t copy this over because ‘’ are wrong characters!

Use WHERE *condition(s)* clauseto return specific rows matching that condition. Allows AND, OR, NOT and not equal (!= or <>).

Use ORDER BY *column(s)* clause to sort results. Orders by first column, and on ties it orders by second. Each individual column comparison is ascending by default, but can descending with DESC. (can be different for separate columns). Ex: Select \* From SalesLT.Customer Order By FirstName Desc, LastName;

We have ordered comparison of numbers, dates, times and strings. Strings are in dictionary (lexicographic) order, but this is affected by collation. Comparators are <, <=, >, >=.

Ex: Select \* From SalesLT.Customer Where FirstName >= 'c' AND FirstName < 'd';

-- Can’t use > b because would include names that start with b. Everything with c and more letters is sorted after c, so use >=c.

LIKE comparator: In SQL Server, [abc] is a or b or c. % is any number of any character (ultimate wildcard). \_ for one of any character.

In SQL Server, comment and uncomment shortcuts are the same (ctrl+k,c and ctrl-k,u).

Data types –

Numeric:

TINYINT (1byte, equivalent to C# byte),

SMALLINT (2bytes, equivalent to C# short),

INT 4 bytes, like C#, this is what we generally use (when there are no special requirements)

BIGINT (8bytes, equivalent to C# long)

Floating Point:

FLOAT, REAL, and DECIMAL/NUMERIC (highest precision and custom precision).

Decimal(4,3) means 4 digits with 3 after the decimal point.

Currency: MONEY

String: Use 1byte-per-char encoding specified in collation.

CHAR/CHARACTER(n), fixed-length character array (string) of size n, not really used.

VARCHAR/CHARACTER VARYING(n), variable length char array (string) up to size n.

NCHAR(n), Unicode char.

NVARCHAR(n), Unicode varchar. This is what we use all the time.

Can use (max) to set to maximum size for a varchar.

‘abc’ is a varchar, implicitly converted to nvarchar when comparing.

N’abc’ is a nvarchar (or nchar?).

Date:

DATE for dates.

TIME for times.

DATETIME for time and day, but low-precision and limited range (default c# datetime value is out of range of this) so use ->

DATETIME2(n) for high-precision, wide-range timestamps. N is for the precision.

DATETIMEOFFSET for intervals of time.

We can EXTRACT to get parts of these, like YEAR from a DATETIME2.

Implicit conversions from strings, useful for comparisons.

Advanced SELECT -

GROUP BY clause:

Useful with aggregate functions, which take in many values and return one. Built-in are COUNT, SUM, AVG, MAX, and MIN.

Group By accepts a list of columns, and then all rows which share the same value of all those columns are combined into one row in the result set. Columns which are not part of group by cannot be used in select because it can be different in the rows that were originally combined), so either you make it part of the group by (make it a basis for combining the rows), or you run some aggregate function which says how to turn those many values into one value.

Can’t use an aggregate in the Where clause!

First rows from the tables are filtered with Where, then we run any aggregations with Group By. If we want to run with conditions on the aggregate rows, then we need the HAVING clause.

Ex: Select FirstName From SalesLT.Customer Where LastName <= ‘n’ Group By FirstName Having Count(Firstname) =1;

Note we can use columns in condition that are not in are Select because…

Runs in order we write them except Select clause is after Having and Before Order By:

5, Select

1. From

2. Where

3. Group By

4. Having

6. Order By

SOLID Principles -

**S**ingle Responsibility: If multiple responsibilities, should be able to break it down into smaller components.

**O**pen/Closed: Children classes should generally add functionality, but not change the original.

**L**iskov Substitution: Child class should work in any case as its base class. (Implicit upcasting)

**I**nterface Segregation: Similar to single responsibility, but for interfaces. Child class shouldn’t have to implement methods it won’t need. Want modularity.

**D**ependency Inversion: Decrease coupling.

VES is Virtual Execution System

vs

CLR is Common Language Runtime

func vs lamda

Can use an alias (AS *name*) before it is defined.

Ex: Select e1.\*, e2.\* From Employee AS e1 CROSS JOIN Employee AS e2;

CROSS JOIN: Same as cartesian product, multiply. Every row combination, with result having the columns from both tables.

INNER JOIN: Most common type of join, takes row pairs which match given condition. Most common type of condition is ON t1.ForeignKey = t2.PrimaryKKey (that FK references).

OUTER JOINS -

FULL JOIN: Keep all rows from both tables, with null where there is no matching row from the other table.

LEFT JOIN: Keep all rows from left, with nulls on right.

RIGHT JOIN: Keep all rows from right, with nulls on left.

JOINS are done in FROM clause.

SUBQUERIES

--every track that has never been purchased

Select \*

From Track

Where TrackId NOT IN (Select TrackId From InvoiceLine);

--Most popular track

Select \*

From Track

Where TrackId = (Select TOP(1) TrackId --TOP(n) takes just first N results from query

From InvoiceLine

Group By TrackId

Order By Count(\*) Desc); --If you use order by in a subquery, need top (or some other things) to work

Subqueries return a table, so can join on them.

For readability, can define subquery above: WITH *name* AS (subquery) query;

EXISTS and NOT EXISTS: If any rows returned, true. If none, false. And the opposite.

Can use comparators (>=, <=, etc.) as well as ANY and ALL for or-ing or and-ing the condition check on all the rows.

Union, Intersect and Except (Set Difference) from Math are used.

The numbers and types of the two queries' result columns must be compatible.

By default, gives distinct rows. Use query *setOperation* ALL query, to allow duplicates.

Union gives row values in either result set (OR), Intersect gives values in both result sets (AND), and except is values in one result set but not the other result set (A-B).

Data Manipulation Language (DML) operates on rows.

5 operations: Select, Insert, Update, Deleted, Truncate.

Insert Into *tablename* (*columnNames*,…) Values (*values*,…), *moreRows*, …;

Can leave off column name section, but this is more readable/less error prone, and allows skipping columns with acceptable default values.

Can be more complex, like inserting the result of a query. Doesn't need "Values".

Update *tablename* Set columnName = newValue, column2 =new2,… Where condition; --leaving out where condition would affect every row in the table.

Substring(*stringName*, *startingPosition*, *length*)

Delete From *tablename* Where condition; --w/o where, would delete every row, one at a time

--Truncate Table *tablename*; --deletes every row, all at once. Faster, but less info in logs.

Deleting a row which has a foreign key pointing at it would violate referential integrity, so DB throws an error. 1 solution is to set those keys to null. Other is to delete all dependent rows.

Data Definition Language (DDL) operates on tables at a time. Can't see individual rows. Works with views, function, procedure, triggers, constraints, etc.

Create, Alter, and Drop, at multiple levels.

Create Schema *name*; --GO is keyword to separate Batches of commands. Some commands need to be in their own batch.

Create Table *schemaName.tablename* (*column1Name c1type, column2Name c2type*,…);

ALTER allows adding or deleting columns (and more).

Alter Table *schemaName.tablename* Add *columnName columnType*;

Drop allows deleting table's rows and its definition. Ex: Drop Table *schemaName.tablename;*

Constraints – Can set constraints after column's type one after another (no commas).

Not Null: Value must be given. (Default allows null.)

Primary Key: Makes PK. Enforces uniqueness, implies Not Null (but being explicit is helpful), sets clustered index.

Unique: Self-explanatory.

Can set constraints after defining all columns, which allows specifying groups of columns. These should be named. Ex: Constraint *constraintname constraintType* (*columns*)

Check(*x*): Arbitrary condition x that must be true for any row.

Default(*x*): If no value given, sets value to this given default, x. (If constraint not given, is null).

Foreign Key: Makes FK. Constraint *constraintName* Foreign Key (*yourColumnNames*) References *TableName* (*otherColumnName*);

Identity: Cannot provide a value. Sets value based off the value in the previously created row. Can set the initial value and the amount it increments with Identity(*x, y*). Default is 1,1 (auto-incrementing). Does not allow repeating IDs of deleted rows.

--Adding column without a default is not allowed. One fix is allowing null, fixing existing rows, and then adding Not Null constraint

[Maybe? Constraint *columnName\_NotNull* Check(c*olumnName* IS Not Null); ]

Alter Column *columnName type constraints(s)*; is to change the definition of a column.

Convert(*type, argument*) is for explicit type conversion.

Can have computed columns, can cache the value on the row or be recomputed whenever you need it. Ex: Alter Table *name* ADD *columnName* As (*computation*);

Create View *schemaName.viewName* AS *selectQuery*;

Views can be selected from, like a table. Ex: Select \* From *schemaName.viewName*;

Can add rows to views, which also adds to underlying table.

Ex: Select Into *schemaName.viewName* (*columnName,…*) Values (*values,…*);

Can run Insert, Update, and Delete on views, but only on columns that map to real columns in the base tables (not computed columns).

Variables

Declare with: Declare @*varName* AS *type*;

Set with: Set @*varName* = *value*;

Table Variable: Declare @*tableVarName* AS Table *col1Name type, col2Name type*,…)

Can insert into @*tableVar*, etc.

Functions

CREATE FUNCTION *schema*.*funcName*(@*param pType*)

Returns *rType*

AS

BEGIN

--example function

Declare @result int;

Select @result = Count(\*)

From Movie.Movie

Where YEAR(ReleaseDate) = @year;

Return @result;

END

--Functions can only read. not write!

--In SQL, string indexing starts at 1!

Ex: Substring(columnName, 1, 1) extracts one char starting at first character.

Multiplicity

0/1 to 1: 1's table has a Unique, Not Null FK to 0/1 side.

1 to N: N's table has a Not Null FK to 1 side.

M to N: New table, with PK consisting of two FKs to the M and N tables.

Called junction table, or join table.

Triggers

Create Trigger *schema.triggerName* On *schema.tableName*

//

AS

BEGIN

--Inside, two special variables, inserted and deleted

UPDATE

Set

//

END

Can put triggers on insert, update or delete.

Can be Before, After, or Instead Of the operation.

Procedures are like functions, but they allow any sql command, they don't have to return anything, and must be called with Execute. Can contain while loops, if-else, and try catch.

Ex: Wasn't finished…

Create Procedure Movie.procedureName (param )

AS

Begin

Begin Try

If(Exists(Select \* From Movie.Movie))

Begin

Set @rowsChanged = (Select Count(\*) From Movie.Movie);

Update Movie.Movie

Set Title = @newName;

End

Else

Begin

Set @rowsChanged = 0;

Raiserror('no movies found!', 16, 1)

End

End Try

Begin Catch

Print Error\_Message();

End Catch

End

-- Raiserror is correct spelling (there is no 2nd e, for some reason).

If Executing, make sure Declare is in same batch.

**ACID** –

**A**tomic transactions: perform all operations successfully, or none of them.

**C**onsistent: operations should not be allowed to violate constraints. (go from one valid state to another)

**I**solation: Even if multiple transactions run concurrently, end result should be as if each operated on their own, sequentially

**D**urable: Not complete until results are written to persistent storage.

**Isolation Levels –**

Read\_uncommitted: Allows reading data that has not been committed, called dirty read.

Read\_committed: Default. Allows other finished transactions to modify a row between one transaction's selects, called nonrepeatable read.

Repeatable\_read: Allows other finished transactions to insert rows that meet conditions one transaction is filtering one, called phantom read.

Serializable: Uses locks. Most isolation, lowest performance.

Begin Transaction

Statement(s)

End Transaction

Using a Rollback Transaction returns state to before beginning the transaction.

ADO.NET

Originally meant active data object, just ADO.

Now it is a namespace/brand name for all .NET data access stuff.

System.Data.SqlClient

Connection strings: List of key value pairs, used for giving server, database, login information (username, password), and other options.