# 

[C# 2](#_Toc32332102)

[.Net Core 18](#_Toc32332103)

[Reactive Extensions 19](#_Toc32332104)

[WPF 21](#_Toc32332105)

[Angular Features 41](#_Toc32332106)

[Node 41](#_Toc32332107)

[Presentation Patterns 41](#_Toc32332108)

[Sorting Algorithms 44](#_Toc32332109)

[O notation 45](#_Toc32332110)

[Data Structures 46](#_Toc32332111)

[BitWise Operations 52](#_Toc32332112)

# 

# C#

Code is converted into an assembly which contains CIL and metadata which describes the types in the assembly and a manifest(version, culture and refs) which describes the assembly.

**CIL** is platform agnostic. Its **JIT’d** at runtime(by **CLR**) on a by method process and cached. **Ngen** preJIts code – quicker startup for high graphics apps

Metadata is used by WCF, late binding, serialization, reflection

**CLR**. Locate, load, and manage .NET objects on your behalf. It allows your app to run as long as your code conforms to **CLI** (Common Language Infrastructure)

It provides specific services such as JIT compilation, memory management, app hosting, coordinating threads, Type safety, GC and exception handling, OS independent , CPU compatability.

You interact with the FCL (Framework Class Lib) mscorlib.dll being the main 1.

So the CLR provides the execution environment for your application, and for when you think of the FCL, the Framework Class Library, know that it provides functionality for your application.

**CTS** documents how types must be defined in order to be hosted by the CLR. 5 types – Class, structure, Interface, enum, delegate

**CLS** Rules describing language subset that is understood by all .net languages. [CLSCompliant]

**Object** Base element for refs and structs

**GetHashCode**(): Used for hashtables. Should be unique and never change. Multiplying by a prime (because its unique) results in fewer collisions.

**MemberwiseClone**: Returns a shallow copy

**String** - Immutable. Sequence of chars. Thread safe. S**tring.Intern** you store strings in a hashtable to stop duplicates. You compare strings to see if their equal or to sort them.

Its a ref type as it doesn't have a fixed size unlike value types. (string can be any size. Int can only be up to  2,147,483,647)

**Intern strings**: basically storing strings within a hashtable so that every string created is unique, otherwise they use whats already created.

**String Equality**

**Invariant** Comparison uses alphabet AaBbC… so boy is before Cow. **Ordinal** uses ascii codes so Cow is before boy. **==** comparison always uses **Ordinal** as its faster. String.**Equals()** allows you to pass in a **StringComparison** enum [CurrentCulture, Ordinal, Invariant, (*all with ignore options)*]. **Equal** defaults to **Ordinal**

**Ordering:** Use **CompareTo** which is case sensitive and implements **IComparable** (->int CompareTo (object other);)

**StringBuilder**: stores string as array of chars. **Append, Replace, Insert, Remove, Index, Capacity, Length.** Initial capacity is 16 chars

**Char.** Individual String. 2 bytes corresponds to Unicode. So A = 41 in Unicode and 0010 1001 in Binary. But its 0000 0000 0010 1001 as a Char is 2 bytes. Its 2 bytes in order to be able to display every unicode character. To deal with larger representations you need to look at **surrogate pairs** which contain more digits to display the info along with linkage info for the pair.

Has methods to check what char is such as **IsDigit, IsWhiteSpace, IsLetter**..

When moving chars around we want to encode to remove all the zero’s

Byte[] bytes = **System.Text.EncodingUTF8.GetBytes**(“Hi there”) = 48-69-20-74-68-65-72-65-2E (UTF8 codes for ‘hi there’)

UTF-8. Stores in 8 bits. Uses between 1 and 4 bytes. Needs multiple bytes if its a large character. Used for the Ascii chars .UTF-16 is 2 bytes and uses between 2 and 4 bytes This is what windows uses. UTF-32 is 4 bytes

**Big-endian**and **little-endian**are terms that describe the order in which a sequence of bytes are stored in computer memory.

**Equality Comparison.**

**Value Equality:** Value Type true if its values are equal.

**Referential Equality:** Ref types true if ref pointers equal (point to same object) . **Unless** the ref type overrides the virtual Equals method.

static **ReferenceEquals()** Are they the same type.

**== and =!**. Defined at compile time. Fast but have issues. Object o=5 would use Referential equality. (ref types check if same ref ptr, val types same value)

**Equals(object o)** Compiled at runtime. Object o=5 would use Value equality. **Causes boxing on Value Type**. (ref types check if same value)

static **Equals(obj1, obj2)** ref types check if same value and is null safe.

**IEquatable<T>** Has b**ool Equals(T o)** method. No boxing on value types.

**Order Comparison**

**IComparer** Used by a type(usually a collection) to compare 2 objects and returns int(-1,0 or 1). Method is int **Compare(obj, obj).** Also has Generic impl.

**IComparable**: Used by an object (usually a poco) to compare it to another object. Returns 1 if >, 0 if =, -1 if <. Return value is how it differs from equal. You should leverage a fields I**Comparable** that’s contained within your class. Method is **CompareTo(obj)**. Also has generic impl.

**Var** = implicitly declared variable – Can’t be passed around. Can only be local. Can’t initialize to null.

**Passing data in parameters**

**Optional** parms need to have a default value. Must be known at compile time not runtime i.e. can’t use datetime.now(). Value is baked into calling method at compile time. Must be placed at end (unless there’s a **params** as well). If Optional parms are added at a later time any different assemblies that call the method needs to be recompiled due to the baking above will fail(it wont have the new parm baked in).

**Params** keyword is placed before a collection param and means you can pass a list of records as opposed to 1 record. Must be at end.

**Named arguments:** call Foo (x:1, y:2); Can be in any order. Must be defined after non named params in signature.

**ByVal:** Create a copy and pass that. With ref type makes copy of the ref pointer so changing the object in a method will change it outside. (However this does not work for string class, because its immutable)

Setting a ref type copy to null just deletes that ref pointer and has no effect on the original.

With ValueType its only a copy so not changed outside method.

**ByRef:** Copies the actual object. Changing either type in method will change outside (we are passing in the actual objects). setting a ref to null sets the original to null as well.

C# methods pass **ByVal** by default. Use **ref** to pass **ByRef**

**Out:** Allows you to return multiple values.Out var is passed by ref. Must be initialized before method and assigned value before method ends

**Param passing best practice** is be as generic when passing in parms and be specific when returning values.

IE pass in a Stream(so you can pass in any type of stream but pass out a FileStream so you can pass out a stream! Co/Contravariance ‘forces’ this in Generics

**Stack and Heap :** Both blocks of memory

**Stack** stores local vars and params. It grows/shrinks as methods are entered/exited. Has max size of 1MB

**Heap** stores reference objects. Needs to be cleared by garbage collection.

Value types and reference type pointers live wherever their created, on the stack if their method vars or on the heap if their ref type fields. Reference types are always stored on heap.

**Ref Types** Stores a reference to an object (such as class, interface, delegate, string). Copying it just copies the reference. Reference is just a memory address. Its made up of a reference pointer and an object

**Value Types:** Contains the actual data. Copying it results in a new value type

**Value Types** – It overwrites the virtual methods of Object to use value based semantics. Sealed. Custom constructors must have args

**Diff between ValueTypes and Reference Types**

Copying value type results in a new value type. With Ref type its just a copy of the reference

ValueType is sealed and doesn’t have finalizers

ValueType has no overheads **Ref type has type info and a locking table**

ValueType can’t be null.

ValueType can’t have a custom empty ctr as this is created by the compiler to initialize all its fields.

ValueType is stored as itself in memory. Ref type is stored as the object and a pointer to that object.

Value Type should be immutable as if not any change will be reflected in any copies of that value type

Reference Type can be null because f1 = null means the reference type pointer is not pointing to an object. Value types are stored as themselves.

**Nullable types** Allow value types to be null. ie int? or bool? Good when dealing with db records.

**Null Coalescing Operator** ?? assigns a value if the var is null ie int myData = dr.GetIntFromDB() ?? 100; Same as ISNULL(value, newValue) in SQL

**OOP**:Modular (Easier to maintain), Reusable(lower cost to Develop, faster to develop (higher quality as now more time to better code!), Extendable.

**Cons** Steep learning curve, more code to write(slower code, larger programs), not suitable for all programs

**Inheritance**: Is a relationship. **Composition**: has a relationship

**Polymorphism**. The ability of objects of one type to have one and the same interface, but different implementation of this interface.

Where subclass implement their own implementation of a base class method. Create using virtual.

**Abstraction:**  Exposes essential functionality while hiding others

**Encapsulation:** Hides data

**Virtual methods** decrease performance as the CLR must look through the inheritance chain to see which class to use.

**protected internal** either or

**protected** : Can be used by a class and any derived classes

**internal** : Any class within the assembly

**Sealed**: Can’t inherit from

**Interface Explicit Implementation**. Allows you to resolve namespace collisions. You need to cast object to interface, which hides the method from metadata!!

**Methods**:

Ctrs never inherited

**Automatic prop** public string PetName { get; set; } (In C# 6.0 you can just have Auto property Get and also Auto property initializers)

**as** returns null when casting, **is** returns bool if can’t be cast. Only **as** casts. Casting is evaluated at runtime. Used in **Explicit**

Casting. From baseClass to subClass is **implicit** (subclass contains all of baseClass). SubClass to baseClass is **Explicit** (ie you have to cast)

**Static** - a static constructor is a special constructor that is an ideal place to initialize the values of static data when not known at compile time.

Static constructor will **always get called once per appdomain** when an object is created. Must be parameter-less

Static class must be sealed.

Static method should not change state of containing type.

Static class can’t implement interfaces.

Static operates on the class itself not an instance of

Static methods perform better as the CLR doesn't have to pass around ‘this’

Very difficult to test

Can’t create extension methods off static type’s methods.

**Generics** – Type safety and increase performance. Stops boxing. Cleaner code.

Can be used for interfaces and methods i.e. static void Swap<T>(ref T a, ref T b)

**Default**  reverts value back to initial value ie 0 or null

Constrain by using : where T : {class, struct, interface, new(), baseclass, T}

TypeInference:Means you don’t have to specify <T> as your parms will

**Covariance** Where a subclass can be used instead of a base class. Only deals with ref types. Can be used by generics by using Out keyword. Out defines return values from method. pubic interface IPop<**out** T>. Use a more derived type than originally specified.

**Contravariance**: When base class used instead of sub. Can be used in Generics by using **In** keyword public interface IPush<**in** T>. In defines all parameters. Use a less derived type than originally specified.

**Collections**

**Arrays** Set of data items of the same type and of a known size (object[] arrObj = new object[3] { 3, "DE", DateTime.Now};)

Multi Dim array - int[,] myMatrix = new int[6, 6]; Jagged Array - int[][] myMatrix = new int[6][];

int[][]myMatrix = new int[][]

{

new int[] {0,1,2},

new int[] {3,4,5},

new int[] {6,7,8,9}

};

MultiD array is a single block of mem. Jagged arrays are arrays of single arrays and are more performant as CLR is optimized for single arrays. This is due to bounds checking. Jagged arrays have a Length property that you can access directly. Multi arrays use **GetLength** method where u have to pass in the dimension (jagged arrays only have 1 dimension, 0).

Array methods: Array.Clear() sets the passed arrays entries to empty values

CopyTo() Copys Source array to Dest

Array.Reverse() . Reverses the passed array

Array.Sort() - Sorts the passed array

Array.CreateInstance()

Array.Copy(arr1, arr2, startIndex) copy arr1 to arr2

Resize(array, length) - Creates new array and copies elements from old into it.

In .NET, a collection is any type that supports the **IEnumerable** or IEnumerable<T> interface.

**IEnumerable** returns **IEnumerator** which has 3 methods (**current, movenext, reset**). Its split so the collection can be enumerated by multiple callers at same time.

**Enumerable:** Provide set of static methods to deal with an **IEnumerable** object. Linq operators were added to this class as extension methods

**ICollection<T>** has functionality for C**ount, Add, Remove, Contains, Clear**

**IList<T>** has functionality for **Indexes**

**IDictionary** extends ICollection and has **Keys, Values, ContainsKey(key), TryGetValue(key)** and **Index[key]** properties

**Yield**. Used instead of enumerator. Returns a new value and moves next

**List**: Inserting items in middle is slow, Searching is slow unless binarysearch is used

**LinkedList**: Chain of nodes where each nodes references node before and after.

Elements can be inserted efficiently. Slow to search and to find where to insert

**Queue**: FIFO, Use Enqueue and Dequeue. Peek to see item at head.

**Stack**: LIFO, Use Push and Pop. Peek to see top item

**BitArray**: Dynamically sized collection of bool values. Contains 1 bit instead of 1 byte

**HashSet<T>**: fast lookup. Won’t add duplicate values. Can’t access by index. Implements **ICollection.** Thy have a bunch of Set operations (ie UnionWith, IntersectWith, ExceptWith…). Can also be seen as a Dictionary without the values.

**HashSet Add** and **Remove** methods return Boolean that will be false if they cant complete (ie Cant add if it already exists).

**SortedSet<T>**: Same as HashSet except is sorted

**Dictionary<T,T>** Can be sorted or unsorted**.**  Can’t insert duplicates. **Hashtable** is not typesafe. Its key is hashed and then added to a specific bucket. This bucket is then got when you access by Key. The goal of the hashing function is not to have unique hash keys (and therefore all entries in unique buckets) but to evenly spread the entries in each bucket.

You can add by index (d[3]+=1) if key exists. Check by using **ContainsKey.** Otherwise use **d.Add(key,val)**

You can access (and add) value from index [] but will fail if value doesn’t exist. Use **TryGetValue** instead.

**Sorted**

**SortedDictionary** : No index retrieve, fastish insertion and retrieval

**SortedList** : index retrieve, slow insertion, fast retrieval

**Unsorted:** index retrieve, fast insertion and retrieval

**Dictionary, Hashtable**

**Other Dictionarys**

**OrderedDictionary**(this is like a queue). Its a hashtable with arraylist func to access by index

**ListDictionary** This is a linked list (very fast retrieval for <10 items). Its ordered not sorted

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Structure** | **Retrieval** | **Modification** | **Insertion at End** | **Insert at index** | **Issues/Notes** |
| **Array** | Fast | Fast | Fast | fast | Lightweight, Fixed Size. Perf bad if modifying beyond fixed size |
| **List** - internally its a large Array | Fast | Fast | Fast | Not fast for large sizes as other entries have to be moved up | Not fast to remove/Add in middle of List. Might reach limit and  then performance is worse. |
| **Collections** - implemented as a List | Fastish | Fastish | Fastish |  | Contains virtual methods to override unlike List |
| **LinkedList** - each element knows neighbour | Not Fast | Very Fast | Very Fast | Very Fast | Performance worsens if many insertions/deletions (ie gaps in list).  Fast to insert but not fast when it needs to find out where to  insert |
| **Stack** |  |  |  |  | Performance like List. Only use for LIFO need |
| **Queue** |  |  |  |  | Only use for FIFO |
| **Dictionary** - KVP | Very Fast |  | fa | NA | TypeSafe |
| **HashTable** | Very Fast |  |  | NA | Non TypeSafe |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | add to end | remove  from end | insert at middle | remove  from middle | Random Access | In-order  Access | Search for  specific element | **Notes** |
| Array | O(n) | O(n) | O(n) | O(n) | O(1) | O(1) | O(n) | Most efficient use of memory; use in cases where data size is fixed. |
| List<T> | best case O(1); worst case O(n) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) | Implementation is optimized  for speed. In many cases, List  will be the best choice. |
| Collection<T> | best case O(1); worst case O(n) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) | List is a better choice, unless  publicly exposed as API. |
| LinkedList<T> | O(1) | O(1) | O(1) | O(1) | O(n) | O(1) | O(n) | Many operations are fast,  but watch out for cache  coherency. |
| Stack<T> | best case O(1); worst case O(n) | O(1) | N/A | N/A | N/A | N/A | N/A | Shouldn't be selected for  performance reasons, but  algorithmic ones. |
| Queue<T> | best case O(1); worst case O(n) | O(1) | N/A | N/A | N/A | N/A | N/A | Shouldn't be selected for  performance reasons, but  algorithmic ones. |
| Dictionary<K,T> | best case O(1); worst case O(n) | O(1) | best case O(1); worst case O(n) | O(1) | O(1)\* | O(1)\* | O(1) | Although in-order access time  is constant time, it is usually slower than other structures due to the over-head of looking up the key. |

**Iterator : A collection generator..** amethod that creates an IEnumerable(). Contains a yield which returns item to caller and maintains state.

static **IEnumerable<int>** Fibs (int fibCount) -> has to be iEnumerable or IEnumerator

{

for (int i = 0, prevFib = 1, < fibCount; i++)

{

**yield** return prevFib;

If (prevFib == 13)

**yield break;** -> only way to break early

}

}

**Iterators** can only be used with **Try finally** ONLY as it needs to be disposed.

Good for massive amounts of data or infinite data and other forms of non linear data.

**NOTE** Stops you from stepping into the iterator code when debugging

**Enums:** Use **Enum.TryParse** to convert enum to proper data type as **Enum.IsDefined** uses reflection. Then you can use switch statement if converted enum is out of range.

Int myArrivalStatus=2;

ArrivalStatus status2;

if (Enum.TryParse<ArrivalStatus>(myArrivalStatus.ToString(), out status2))

**Enum.Getname(enumtype, value)** Gets name when you pass in the enum value

**Serialization:** Converting object(s) into different format (byte, xml, json). Deserialization is opposite. Used to store, transmit across app boundry and clone.

Types of serializers, **Binary, XML** and **DataContract,** or **ISerializable** to role your own

In all cases you add attributes to what you want and don’t want serialized. And then implement methods that fire before and after Ser and DeS. These methods need to have the [**OnDeserializing**] and [**OnDeserialized**] … attributes

**Binary:** Add [**Serializable**] att to class. [**NonSerializable**] to fields not to properties..

**XMLSerializer [XMLignore]. [XMLinclude]** to add subtypes

**DataContract** : Add **[DataContract][DataMember]** attributes and [**KnownType**] for subclassing.

**Formatters:** how data should be outputted BinaryFormatter, SoapFormatter, XMLDataFormatter

**ProtoBuf Serialization Add [ProtoContract]** and **[ProtoMember(1)]** attributes. Numbers are used as the object names aren't encoded.

**Exceptions**

When throwing the exception up the stack. If we replaced **throw** with **throw ex**, the StackTrace property of the newly propagated exception would no longer reflect the original error.

Rethrowing a less specific exception is something you might do when crossing a trust boundary, so as not to leak technical information to potential hackers. When rethrowing a different exception, you can set the InnerException property with the original exception to aid debugging.

Catching unhandled exceptions

**AppDomain.CurrentDomain.UnhandledException** will catch all unhandled but has no way of preventing app from shutting down

**Application.DispatcherUnhandledException** (on WPF) only catches the main UI thread.

If your thread is a worker thread you need to catch it yourself

**Delegates –** an object that contains a list of references to methods which it then invokes. Like C++ pointers except its type safe

It contains address of method, parms and return type. public delegate string MyDelegate(bool a, bool b, bool c)

For each delegate the CLR creates a class with a ctr that takes the type, method and 3 invoke methods (regular and 2 async)

**Multicast** delegate. When adding or removing from, a new delegate is returned with the updated invocationlist and the previous multicast delegate is discarded! They only return the last value from a non-void method and crash out if an exception is thrown in List

**MulticastDelegate.GetInvocationList** allows you to have control over the delegates with regard to exceptions and return values and using **BeginInvoke**

Delegates can result in a lot of extra code. Instead .Net has built in delegates

**Action**<> and **Func**<> Simple delegates. Both can have <16 parms. Only Func can return a result

Predicate<> Simple Func<>. Takes a single input and returns a bool

Events are basically private delegates. public event CarEngineHandler<CarEventArgs >Exploded; Event args should inherit from EventArgs

Event should return a void and have 2 arguments, object (this will be the broadcaster) and a subclass of EventArgs.

Events should be fired within a protected method

**Anonymous methods**. Inline the method You can use the local variables of the surrounding method.

c1.AboutToBlow += delegate

{

Console.WriteLine("Eek! Going too fast!");

};

**Lambda expressions** like anon methods except less code to write

SetMathHandler((msg, result) =>

{Console.WriteLine("Message: {0}, Result: {1}", msg, result);});

Closure**:**  A function within another scope that has access to the outer scopes variables(which is called a **captured variable**). Useful when you seem to be passing around state

**Indexers**

public Person this[int index]

{

get { return (Person)arPeople[index]; }

set { arPeople.Insert(index, value); }

}

78564

**Increment/Decrement operators** Change a value by 1

int x = 0, y = 0;

**Post Increment** Console.WriteLine (x++); // Outputs 0; x is now 1

**Pre Increment** Console.WriteLine (++y); // Outputs 1; y is now 1

**Operator Overloading [**unary – work in itself, binary -2 vars. Ternary - value = (i >) 0 ? 2 : 3]

public static Point operator + (Point p1, int change)

{

return new Point(p1.X + change, p1.Y + change);

}

**Extension method:** Add a method to an existing type without modifying the type (so you guarantee you don’t incur regression bugs)

public static int ReverseDigits(**this int** i) and you call by… int number = 34; number.ReverseDigits()

Created inside a static class only.

Extension methods operate only on instances, not static classes.

The CLR will first look to see if there’s a ReverseDigits method in the int class. If not it looks at all static classes for the RD method adds thats il code here.

**Anonymous types:** Create a class on the fly. Should only be used within the method they’re declared. To pass them out from method use dynamic

var car = new { Make = make, Color = color, Speed = currSp };

**LINQ** Can query any **sequence** that implements **IEnumerable**() (arrays, xml, datasets, objects). Its strongly typed.

The methods you use to query are called **query operators** (ie First<>, Select, Where, OfType()

var results = from g in array where g.Contains(" ") orderby g select g;

foreach (object gg in results)

…

Query is not run until you run over the result (i.e. the foreach statement): lazy evaluation, unless its a single value (like **First()**) returned or is converted (like **ToList()**)

You can immediately execute like this: int[] results = (from g in array where g.Contains(" ") orderby g select g).ToArray<int>();

In Linq the compiler figures out the best way to create the results

Two ways to write the query.

**Query Syntax:** looks like SQL.

**Fluent Syntax:** Chains the methods together. customers.Where (n => n.Name.Contains ("a")).OrderBy (n => n.Name.Length)

Two types of query architecture. **Local and remote (Interpreted)**

**Local** - deal with query operators in the Enumerable class

**Interpreted** - Operate over sequences that implement **IQueryable. LINQ to SQL** is interpreted

Types of Linq methods

**Filtering:** Where, Take, TakeWhile

**Traversal**: Skip, SkipWhile,

**Projecting:** Select - Projects to new form. One Select increments a counter by 1(eg Used for line numbers).

**SelectMany** - allows us to convert each item in a sequence into another sequence and then stitches altogether into 1 sequence

**Joining**: Join, Zip, GroupJoin

**Grouping:** GroupBy

**Ordering:** OrderBy, ThenBy, Reverse.

**Aggregation:** Sum, Average, Count, Min, Max, Aggregate( when nothing else works!. Very powerful)

Aggregate(seed, obj1, obj2): goes through each item in sequence so you can compare each value to each other

Good for adding weird values (like times), finding the max in a sequence..

**Set:**

**Concat**(Merges 2 sequences),

**Union**(Like Concat except no duplicates),

**Intersect** (whats in both sequences),

**Except**(Whats in A but not in B),

**Distinct**(whats unique in a sequence)

**Conversion - In :** OfType, Cast

**OfType(X)**- Filters **in** only the elements of Type X. Ignores all other types. Enables non generic colls to use linq methods.

**Cast(X)** will try and cast all elements and throw InvalidCastException

**Conversion - Out:** ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable

**Single value returned:** First, Last, Single, SingleOrDefault…

**Quantifier (returns bool)**: Any, All, Contains

Fluent Code. Used to write less code. Usually have a ‘factory’ class that sets all its child vars up. Then every method that you want to add to the fluency is an extension returning this factoryclass. (Think IEnumerable in linq). Can be tricky to design. When using, separate onto separate lines to debug easily. Can be hard to know where an error arose.

**Object Lifetime** Controlled by the CLR garbage collector

Object gets destroyed when its unreachable by any part of your code

Managed heap contains ptr to where next object will go.

When you call ‘obj x=new obj’ the CLR 1. Calculates the amount of memory required for that object

2. Checks to see if there’s enough space on the heap for the object

3. If yes then obj is put on the heap. If no then gc occurs, heap frees up and then obj is added

4. Ptr is then moved to beyond that obj

Garbage collection process

1. CLR creates object graph for each object on the root and also assigns them a generation(0 to start off)
2. Root objects are 1) vars in current method (or any method in its callstack), 2) static vars, 3) vars with finalizers,
3. It walks the graph noting all object root objects touch.
4. Any objects not touched are then deemed to be dead and unreachable.
5. Any unreachable gen 0 object is then removed
6. All other objects are shifted to start of heap
7. All the other gen 0 objects have their generation incremented
8. If memory is still needed the CLR will check gen 1 objects and remove all unreachable 1s
9. If memory is still needed the CLR will check gen 2 objs…

This means newer objects (such as local vars) get removed first whilst long running objects become gen 2 and rarely get touched.

System.GC is a static type that works with the GC(**GC.Collect(?)** forces a gc. Make sure you always call **WaitForPendingFinalizers**() after it). **SuppressFinalize(), GetTotalMemory()**

Only create a finalizer (~obj()) when you’re dealing with unmanaged resources (ie using PInvoke)

**Dispose vs Finalize**. You call **Dispose** manually, the CLR calls **Finalize().** Finalize cleans up when GC’d. Use Finalize() when your not sure if the objects you’re using are still being used by others. i.e. ManualResetEvent.

The LOH heap stores objects >85K. This can cause issues in <64K machines as the space doesn’t get compacted after GC. Different sized large objects can result in memory overflow.

When method is created CLR bookmarks top of stack. And adds variables created in method onto stack. When method ends the CLR reverts back to the bookmark and clears the stack.

When reference is created

Form f = new form// Allocates Form to heap and adds f to stack with pointer of Form

Memory:

The operating system provides memory via VirtualAlloc() only. That works fine, but the granularity is not good: it can only provide 64kB at a time. That's why Microsoft has implemented different heap managers, e.g. the C++ heap manager or the .NET heap manager. Those get memory from the OS in 64kB blocks and provide it to a C++ or .NET program in smaller chunks.

Memory in OS:

There are three areas of memory in an OS

1. Register: Sit on the CPU and are very fast. The OS moves data in and out of these from other memory constantly. Limited size
2. RAM: This is where the physical memory sits.
3. Virtual memory: This memory is on the hard drive. Memory in RAM that’s not being used is copied over to a page file on the HD

**PInvoke** : Calls unmanaged DLLs across process boundries.

Declare the unmanaged function using **DLLImport**. And also describe how to marshall parms and return values

**Class Librarys**

**Assemblies**: {Promote code reuse, Versionable, Self describing} They contain {metadata(describes types), CIL, manifest(describes assembly), Windows file header, optional embed resources

Assemblies are logical constructs, Modules are physical constructs

**Probing** Loading an assembly. **Implicit:** loading from metadata. **Explicit** : Load using Assembly.Load(). You can add folders to probe by adding their paths to the app.config file

Shared assemblies are in the GAC. To be shared they have a strong name which consists of : Friendly name

Version number

Public key

Digital signature(private key and hash of contents)

Optional culture

**Reflection –** obj.GetType() or typeof(object).

**Type** has some members **GetProperties(flag), GetFields(flag), GetMethods**, **GetMembers(bindingFlag)** to get all members as **MemberInfo**[].

**Classes: MemberInfo (**abstract**)** its subclasses include **FieldInfo, PropertyInfo, MethodBase, EventBase**

Use Assembly.Load() to load assembly dynamically

**Late Binding**: Use **Activator.CreateInstance(**Type) it returns an object as the manifest wasn’t known during compile. Need to use reflection to use obj

Custom **Attributes** should be sealed for security reasons. They can be restricted to (class, struct, field…

**Dynamic k**eyword allows other runtimes to talk to the CLR. Used in LINQ. Better than tuples. Its not typesafe.

CLR typechecking is turned off when **dynamic** is encountered

**Expando Objects:**  Create objects on the fly. Their stored in a data structure lie a Dictionary <String, object>

**User mode vs Kernel mode**

Whenever code executes, the thread that’s executing has 1 mode associated with it, User or Kernel.

**User mode**: Less powerful mode Has no access to system resources nor hardware. It protects the application from crashing the system

**Kernel mode**: Privileged mode used by kernel and device drivers. Can access any resource on system. If an exception happens in kernel mode we get the blue screen.

User mode enters Kernel mode by the thread switching to Kernal mode, do the operation and then revert back to User mode. Actually when a user creates a thread its always a user thread. A corresponding kernel mode theread is created by the OS which is then used if the user needs to access the kernel!

**Processes, AppDomains, Contexts** Process contains >0 Appdomains which contain >0 object contexts

Process is a management object. It manages and provides all resources for threads to execute code

It owns

1. **A virtual address space** (2gb on 32bit, 128TB on 64bit) where memory is allocated
2. The actual program referring to an image file on disk that contains the code to be executed.
3. **1 or more threads that execute the code**
4. Working set (physical memory used by process)
5. **Handle table to kernel objects** that hold all kernel objects opened by that process .Handles are unique to each process so a bad address will only take down 1 process.
6. Access token. A security context used for security checks when accessing resources (used when opening a file to see if you have permissions to open that file).
7. Priority class which is the default priority used for each thread.

**Process** is a set of resources (libs and primary thread) and the necessary memory allocations.

Process object’s common members – Start(), Kill(),GetProcessById, ID

Every process has a default AppDomain.

**AppDomain:** There are multiple in a process. These are at least 3, **System, Shared** and **Default** app domain created for each process.

**System** **app domain**. Creates the shared and default App domains. It loads mscorlib.dll into Shared app domain. It also precreates 3 exceptions, **OutOfMemory, StackOverflow** and **FatalExecution** exceptions. This is so they can be thrown if they occur, as you couldn’t create a new **OutOfMemory** exception if tyou are out of memory.

**Shared** app domain; Contains mscorlib.dll, basic System types (string, enum, array). Usually contains no user code.

**Default** app domain: Contains all user code.

Benefits of this additional subdivision.

* AppDs are less exp in cpu and mem and can load/unload quicker than a process
* AppD can crash and still other AppD and the process are OK due to the isolation
* AppD abstracts away from how an OS represents an exe, so its OS neutral

Not easy to share data between AppDomains. WCF does

AppDomain object common members Create(), Load(), Unload(), CurrentDomain

Every AppDomain has a default context(Context 0)

Contexts are a way the CLR is able to ensure objects that have special runtime reqs are handled in an appropriate and consistent manner

Normal objs are called **Context-agile objs.**  An object that requires thread Safety(Synchronization) will get its own context. The obj must derive from **System.ContextBoundObject** class.

**Dynamic Assembly** – created on the fly using **System.Reflection.Emit**

***Asynchronous Programming***

***Delegates***

***Threading***

***…***

***Example and show IAsyncResults***

Threading

Threads are entities scheduled by the kernel to execute code

Thread contains information that needs to be saved when it is preempted (when its stopped and a new thread is ran)

1. State of CPU registers that’s saved in threads stack
2. Current access mode (User or Kernel mode)
3. **Stack for the thread that hold variables and function calls**. Theres actually 2 stacks, one in User space and one In Kernal space.
4. **TLS** which is a private storage area used for data that you don’t want shared. Its rarely used. **Thread.GetData/SetData** methods for it.
5. **Optional message queue if thread calls win32 api**.
6. Priority used in thread scheduling
7. Threads state (running[compute bound], ready, waiting[i/o bound]).

Threads share the same execution environment.

Winforms and WPF apps are event based system where events get fired by message pump. When the ui thread is running some heavy operation the messages don’t get read so the UI becomes unresponsive.

Methods

**Start**() - starts thread

**Join**() - block calling thread execution (apart from message pumping) and waits till thread has finished

**Sleep(time)** - block thread and hands control back to main thread

**Yield()** - same as Sleep but only hands control back to thread if on same processor

**Abort()**

**Interrupt()**: Interrupts a thread that is in the WaitsleepJoin thread state.

**ThreadStart** is the method that will be executed. **ParameterizedThreadStart** is a method with a single object parm that will need to be cast.

These have been replaced by Anonymous methods.

Passing Data to thread. 1. Use closures with Lambda. 2. ParamaterizedThreadStart uses 1 object. 3 All app domain static vars are available to the thread.

A thread will either be waiting or working. **IO-bound** or **Compute-bound.** If you spin the thread in IO bound i.e.

( while (DateTime.Now < nextStartTime)

Thread.Sleep (100);)

Then its actually working. Best to use some blocking functionality like signaling. Sometimes its actually OK to block if only for a very short time.

**Latency: When work take a while to be done -** I/O bound in the synchronization reads

**Blocking:** A blocked thread immediately yields its processor time slice, and from then on consumes no processor time until its blocking condition is satisfied.

Use **Try/catch** within a thread method. Useless around the thread.Start() and an exception that occurs within the thread will not be caught.

**Background** thread will die once all foreground threads die. Need to set thread to background ie thread.IsBackground=true. You should wait for them to die by joining making sure you pass a timeout. Finally blocks will not be called when background thread dies.

**Marshalling** (Moving) Data onto UI threads is done by creating a delegate (which contains data) and passing that to a **Dispatcher** object(in WPF) which then gives it to its UI Thread using **Dispatcher.Invoke**. WIth WInForms you call Form.Invoke.

Textbox.Dispatcher.Invoke(()=> textbox.text = backgroundThreadResult);

**SynchronizationContext:** is a representation of the current environment that our code is running in. We can make a copy of it and use it if on a different thread. It has two main properties **Send** which is the same as **Invoke** (and is synchronous) and **Post** which is **BeginInvoke** and asynchronous**.**

partial class MyWindow : Window{

**SynchronizationContext \_uiSyncContext;**

public MyWindow(){

\_uiSyncContext = SynchronizationContext.Current; //capture the context of the UI thread

new Thread(Work).Start();

}

void Work(){

UpdateMessage("The answer");

}

void UpdateMessage(string message){

// Marshal the delegate to the UI thread: Note we are in the background thread.

\_uiSyncContext.Post(\_ => txtMessage.Text = message);

}

}

**Synchronization** – coordinating concurrent actions for a particular outcome. 3 types

1. **Exclusive locking** – Only 1 thread can access method at a time. Use **Monitor.Enter/Exit**

The locking object should be readonly, private and solely used for locking. Making it private enforces encapsulation. Only 1 thread can lock on that object at any 1 time

Use bool lockTaken = false;

try

{

Monitor.Enter (\_locker, ref lockTaken);

// Do your stuff...

}

finally { if (lockTaken) Monitor.Exit (\_locker); } **//locktaken is only false if exception was thrown**

Lock(…) is syntactic shortcut for Monitor.Enter/Exit with a try finally block.

**Mutex** locks across processes, because you can name it. It also allows you to pass in a timeout that breaks the lock if locked.

var mutex = new Mutex (false, "oreilly.com OneAtATimeDemo")); mutex.WaitOne(TimeSpan.FromSeconds(3))

1. **Non-exclusive locking** - allows a certain # of threads to access a method. Let’s you limit concurrency.

**Semaphore**. Use **Wait()** and **Release().** Any thread can release a semaphore unlike lock where only the thread that created it can. Therefore a semaphore of 1 capacity is not the same as a lock

**ReaderWriterLockSlim :** Allows you to have multiple concurrent reads and single writes on a collection. You wrap the following methods around that your doing

public void EnterReadLock();

// Read from a list

public void ExitReadLock();

public void EnterWriteLock();

// write to a list

public void ExitWriteLock();

1. **Signalling** Allows a thread to block until it receives a signal from another thread

**ManualResetEvent** or **AutoResetEvent** You wait by **WaitOne**() and signal by **Set**(). **Reset()** closes the event

Diff between Manual and Auto is Manual allows all threads thru on Set, Auto only allows 1

**CountDownResetEvent** blocks until it’s been set a certain number of times.

**Monitor.Wait/Pulse** Most powerful type of sync. But difficult.

They need to be called inside a lock. When Wait is called it hands back the lock until it’s pulsed where it retakes the lock

public void Main() {

new Thread(Work).Start();

Thread.Sleep(100);

\_go = true;

lock (\_locker)

Monitor.Pulse(\_locker); <- just notifies the Wait where the main work is done

}

void Work() {

lock (\_locker)

while (!\_go)

Monitor.Wait(\_locker); <- we wait here without the lock. When pulsed we reacquire lock,see go is true and break out of while loop and lock.

Console.WriteLine("Woken!!!");

}

**An atomic operation** means it’s isolated from multiple threads. Solely reading or writing is atomic as only 1 thread can to it at any one time. Actually this is only true if it’s a 64bit system. Large datatypes (long, decimal) take more than 32 bits to store…

When a variable is incremented it’s not an atomic operation. 3 things happen. Therefore you need synchronization.

1. reads the value in the memory location;
2. adds one to the value;
3. writes the new value back into the memory location.

Use **Interlocked** when incrementing/decrementing

**Synchronization** is needed to ensure **atomicity. Synchronization** also creates a **memory barrier** around the code so it won’t be moved by the CPU

However atomicity within a lock is violated if an exception is thrown. In following case the bank would lose money. Solution is to have rollback func within a Try/Finally

void Transfer(decimal amount) {

Lock (locker) {

Savingsbalance+=amount;

checkingBalance-=amount ;

}

}

**Deadlock**. When 2 threads each wait for a resource held by the other. Difficult to debug. Good OO can cause it as that code is ‘chained’ which give more chance of locking occurring somewhere

new Thread(() => {

lock (\_lock1) {

Thread.Sleep(1000);

lock (\_lock2) {}

}

}).Start();

lock (\_lock2) {

Thread.Sleep(1000);

lock (\_lock1) {}

}

Overcome Deadlock:

1. **Heirarchial Lock acquisition**

Whenever more than one lock is acquired we grab those locks in a particular set order that will never change, ie an Id of the objects your using within the lock.This is because if we have a threadA and threadB then we want to make sure that they always get the same locks. This way the CLR will note that the lock objects are being used multiple times by the same locks (by looking at the objects lock table). So in above example threadA will always use lock1 whilst threadB hits lock.

Use a method that uses out params to returns locks.

2. **Mutex’s**

You can wait to get a lock with a timespan that leaves if timespan finishes. Create an array of mutex’s that contain the 2 locks. Then use WaitHandle,WaitOne(locks, timespan) with a finally block. If theres no deadlock we enter the finally block where we release each mutex lock.

Tasks, Immutable types help overcome deadlock.

Be wary of locking around classes that call out to methods.

**Thread safety ways around locking. S**haring data is the major issue with Threading (that and too many threads with memory and context switching)

1. Lock around entire object. Can be done if objects methods execute fast
2. Use a SynchronisationContext
3. Keep sharing of data to a minimum

**Collections and Threading**

Locking on collections. Need to lock when

1. Adding/removing/updating record
2. Traversing collection. Better here to lock when creating a copy and then traverse the copy without a lock.

You can use the **ReaderWriterBlockSlim** which allows concurrent reads and single writes.

Or the Concurrent collections.

**ConcurrentQueue**<T>**, ConcurrentStack**<T>**, ConcurrentDictionary**<T>**, ConcurrentBag**<T>**.** These have the following methods **TryAdd, TryTake** which test first if the operation can be performed and then return.

They perform worse than regular collections but better than locks around regular collections.

**BlockingCollection<T>** blocks on the **TryTake** method if nothing exists!

**Non blocking synchronisation :** Faster than blocking ..

**MemoryBarriers :** Prevents CLR from moving code. Also called fence

**Volatile:** a field that doesn’t allow the CPU from caching it. It auto generates a fence around it.

**Interlocked:** Allows you to perform non atomic ops on fields such as **add, subtract read, Increment, decrement**

**ThreadPool** When you create a thread, processing time is used up creating the thread. Use TP instead as it recycles threads. Its threads are background threads. You can change the priority of TPs threads.

How to use ThreadPool:

1. ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
2. Use Delegate.BeginInvoke()
3. Task.Run()

Threadpools helps ensure that not too many threads are created (CPU core limitation) which will result in expensive context switching for time slices

Each process has its own thread pool.

**Race condition** is a bug that occurs when the outcome of a program depends on which of two or more threads reaches a particular block of code first

**High-performance, low-latency tips**

Creating new objects is not so much a problem in C# as it is in C++(as you have to manually deal with deallocation) when you don't need to GC as all that happens is the garbage pointer gets moved up by the size of what you’re allocating.

Destroying objects can be intensive if you have a finalizer.

Need to have proper algorithms, parallel processing, synchronization/asynchronization, caching and lazy creation, lazy evaluation (LINQ).

Use the right collections.

Profile the code. Find problems and fix one at a time, generating benchmark timings at each stage.

### Tasks

A unit of work (or ongoing computation) that if is a thread is in the threadpool.

Task t = Task.Factory.StartNew(()=>…)/ Task.Run(()=>…)

**Factory.StartNew** allows you to pass in **TaskCreationOptions** and a **TaskScheduler**

**TaskCreationOptions: LongRunning**->dedicated thread not off the threadpool

**AttachToParent**-> Parent task waits for its child task (created within a parent task) to complete before it completes

**PreferFairness** -> Tasks should run in order they started

**None**

**TaskScheduler**: **Default:** from threadpool

**Current**: The current thread?

**FromCurrentSynchronisationContext**: runs from UI thread.

Task can also return values. To do so it must be flagged with <T>.. i.e. Task<int> t (this inherits Task). And use **task.Result** property.

Methods:

**Wait(), WaitAny(task[]), WaitAll(task[])**: waits (blocks) for tasks to finish. Don’t use this.

Use **WhenAll, WhenAny** as these don’t block and give you a task which you can **ContinueWith**

***ConfigureAwait*():** If true then attempts to marshal back to original context. Set to false unless it’s the last task. Setting to true on a UI app puts it onto the UI thread.

**Delay(time)**: Task will complete after an async delay. Returns a task

**RunSynchronously()**

**ContinueWith** : Starts a new task when one is finished. ..**StartNew**(()=> {console.write(“hi”)}).**ContinueWith**(()=>{…}) Returns a task.

ContinueWith is better than Wait as the Task Scheduler will most likely use the same thread for the 2 tasks.

Task.Factory.**ContinueWhenAll**(Task[]) gives you back all tasks so you can inspect each task individually to see if they **IsCanceled, IsFaulted,** or **IsCompleted.** It runs when all its tasks completes.

**ContinueWhenAny(Task[])**: Completes when 1 task completes

**TaskContinuationOptions NotOnRanToCompletion**

**NotOnFaulted**

**NotOnCanceled**

**LongRunning**

**AttachedToParent**

**None**

You can also continue by using ***Task.GetAwaiter()*** and implementing its **OnCompleted** method.

var awaiter = task1.GetAwaiter();

awaiter.OnCompleted(()=> Console.WriteLine(awaiter.GetResult());

GetAwaiter is actually what the async/await uses.

Awaiters return the specific exception (not AggregateException) so you don’t have to flatten..

When the **OnCompleted** is finished and a **SynchronisationContext** exists it will be used to move back onto the UI thread. This is why **async/await** updates the UI without having to use Invoke. To not use the SyncContext you need to use **ConfigureAwait(false)**

Properties

**IsCanceled, IsFaulted, IsCompleted**

**Result** (only for Task<t>). Blocks if not called within a **ContinueWith**()

CancellationToken. Pass into method to Cancel.

**TaskCompletionTasks:** Facade around existing operation. Set **tcs.Result** to the methods result and return **tcs.Task**

**Exceptions**

**Aggregate exceptions**: Returns the 2 stack traces, returns multiple exceptions from multiple tasks

**Unobserved expections** : Use **TaskScheduler.UnobservedTaskException** event to raise them

Exceptions get thrown when accessing **Task.Result** and when accessing **Wait methods** Also there is **Task.Exception** property

**Async / Await**

Method runs synchronously till the await keyword. Then clr checks if awaitable(whats called on line after await) is completed. If so it returns. Else it leaves the method. It returns when the awaitable is completed and the switches back onto the Context (Either the UI or threadpool context).

To not switch back to the context you type await AMethodAsync()**.ConfigureAwait(false)**

Dont use Result or Wait in Async.

Always return a Task from Async so it can be validated. (Task returns expection and IsFaulted..)

The awaitable has to be a Task

Exceptions occurring in an async void method cant be caught. You need to have a try catch within an async void method(event handler methods).

**Timers : Two types thread based and UI based.**

1. **Thread timers [These can be accessed by multiple threads and are more precise]**
   1. **Basic thread timer. You pass in details in Ctr.** System.Threading.Timer tmr = new Timer (Tick, "tick...", 5000, 1000); Has a **Change** method which changes the start and interval times.
   2. **Regular timer**. Inherits basic time**r and has Elapsed event,Enabled interval, start, stop**.. methods
2. **UI timers**
   1. **WPF timer:** Has the same method as the above regular but its events get put on the UI Message loop so not as accurate.
   2. **WinForms timer** : same as a)

**Async Coding**

**Old versions**

1. **APM [Async prog model] - good for dealing with events as long as you called the End.. method**

private void LookupHostName() {

object unrelatedObject = "hello";

Dns.BeginGetHostAddresses("oreilly.com", OnHostNameResolved, unrelatedObject);

}

private void OnHostNameResolved(IAsyncResult ar) {

object unrelatedObject = ar.AsyncState;

IPAddress[] addresses = Dns.EndGetHostAddresses(ar);

}

1. **EPM [Event based prog model] - good for calling back on whatever thread ie UI thread**

private void DumpWebPage(Uri uri) {

WebClient webClient = new WebClient();

webClient.DownloadStringCompleted += OnDownloadStringCompleted;

webClient.DownloadStringAsync(uri);

}

private void OnDownloadStringCompleted(object sender, DownloadStringCompletedEventArgs eventArgs) {

m\_TextBlock.Text = eventArgs.Result;

}

**StateBased Machine**

public async Task<int> AlexsMethod()

{

int foo = 3;

await Task.Delay(500);

return foo;

}

The stub method generated by the compiler looks like this:

public Task<int> AlexsMethod()

{

<AlexsMethod>d\_\_0 stateMachine = new <AlexsMethod>d\_\_0();

stateMachine.<>4\_\_this = this;

stateMachine.<>t\_\_builder = AsyncTaskMethodBuilder<int>.Create();

stateMachine.<>1\_\_state = -1;

stateMachine.<>t\_\_builder.Start<<AlexsMethod>d\_\_0>(ref stateMachine);

return stateMachine.<>t\_\_builder.Task;

}

state machine struct generated

<AlexsMethod>d\_\_0

{

public int <>1\_\_state; -> stores the await #

public int <foo>5\_\_1; -> stores foo

public AlexsClass <>4\_\_this; -> class that owns method (if not static)

public AsyncTaskMethodBuilder<int> <>t\_\_builder; -> Create a puppet task to return <int> because return type is Task int

private object <>t\_\_stack; used if await is within a complex statement.

private TaskAwaiter <>u\_\_$awaiter2; storage that helps await to sign up for notification when Task completes

}

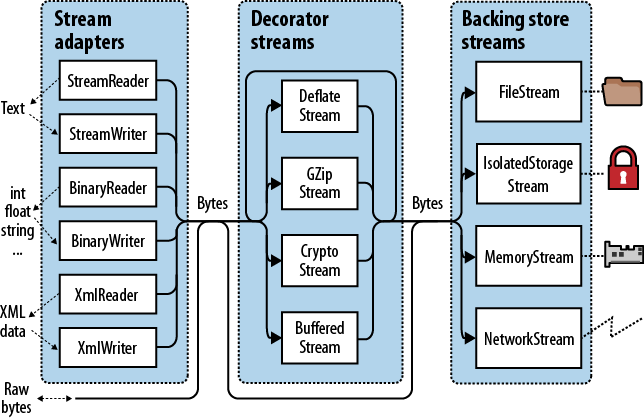
statemachine also has a MoveNext method which contains the original AlexsMethod code. Its called at start and when we return from await

## Streams

**Backing Store streams** provides the raw data (in bytes)

**Decorators** provide binary transforms

**Adapters** offer methods to deal with strings/XML



Abstract **Stream** is the base class used by all. Has functionality to **Read, Write, Seek.**

**Read** receives a block of data from the stream into an array, returning # bytes read and moving stream position up by # bytes. Usually written within a loop as when #bytes read = 0 then we’re finished reading.

**Write** sends data to to stream moving stream position up by length of write.

**Seek** is setting **Position** property!

Wrapping stream within Using block auto calls **Flush()** and **Close()** and kills object!

**FileStream**. Different ways to open file (Create, Append, Open, Truncate)

MemoryStream: Can convert to byteArray

using (FileStream fs = File.Create ("test.txt"))

using (TextWriter writer = new StreamWriter (fs))

{

writer.WriteLine ("Line1");

writer.WriteLine ("Line2");

}

using (FileStream fs = File.OpenRead ("test.txt"))

using (TextReader reader = new StreamReader (fs))

{

Console.WriteLine (reader.ReadLine()); // Line1

Console.WriteLine (reader.ReadLine()); // Line2

}

**AppDomain.CurrentDomain.BaseDirectory** returns the app base directory,which in normal cases is the folder containing the program’s executable.

**CLR via C#**

**Starting an executable.**

1. Windows examines this EXE file’s header to determine whether the application requires a 32 or 64-bit address space.

2. Windows loads the x86, x64, or ARM version of MSCorEE.dll into the process’s address space.

3. Then, the process’s primary thread calls a method defined inside MSCorEE.dll. This method initializes the CLR, loads the EXE assembly, and then calls its entry point method (Main).

4. At this point, the managed application is up and running.

**Running an executable**

Just before Main is called, CLR looks at all members within Main and allocates an internal data structure for each type and their methods.

When a types method is then called for the 1st time the IL is JITed and the native code is saved in dynamically allocated block of memory.

Then the address in memory of the code is added to the data structure above.

When method is called again it will look at data structure, see an address and jump directly to the memory where the native code sits

# .Net Core

Dependency Injection

AddSingleton: Same object is used throughout lifetime of api requests.

AddScoped: Same object is used within a request, but different for each request.

AddTransient: Different object used always

**ASP.NET**

The host is responsible for app startup and lifetime management.

Web apps call **CreateDefaultBuilder** when building a host.

# Reactive Extensions

A push model : eg the Producer/Consumer pattern

**IObservable :** IDisposable Subscribe(Iobserver)

**IObserver**<T>: OnNext, OnError, OnCompleted

**Scheduler:** Gets set when creating observable (**ObserveOn**) and subscribing (**SubscribeOn**)

**Default** gets fro the ThreadPool.

**Current** uses current thread.

**Immediate** runs straight away on current thread

**Creating Observables**

1. Manually by implementing **IObservable**. Lengthy and allows u to call **OnNext** after **OnError** or **OnComplete** or error called which is bad.
2. Override **ObservableBase**. It auto detaches after **OnError** or **OnComplete** called or error thrown.
3. **Observable.Create(observer)**. You pass in the code of the subscribe method.
4. **Observable.Defer(Func<IObservable<TResult>>).** Defers creation of observable until its subscribed to.
5. **Observable**.**FromEventPattern**<RoutedEventHandler, RoutedEventArgs>(h=>btn.Click+=h, h=>btn.Click-=h).
6. **Observable**.**FromEvent**<SomeHandler,T>(h=>someMethod+=h, h=> someMethod -=h). For non event pattern ie (sender, args) events.
7. **Ienumerable ToObservable()**. IEnumerable has ToObservable extMethod
8. **Observable.Generate(initial state, condition, iterate, result)** Create besed on an iterative process.
9. **Observable.Range(start, count).**  A simpler version of Generate.
10. **Some primitives**
    1. **Observable.Return().** Emits one notification and Completes immediately
    2. **Observable.Empty().**Emits nothing and Completes immediately
    3. **Observable.Never().**Emits nothing and never completes
    4. **Observable.Throw()**. Throws an exception

**Creating Operators**

1. Using **Observable.Subscribe**(this IObservable, OnNext, OnError, OnCompleted). Always implement the OnError.
2. Use **CancelToken** with **Observable.Subscribe**(this IObservable, …,cts.Token). this returns a void. You can kill it **by cts.Cancel**().
3. **Observer.Create<T>(** OnNext, OnError, OnCompleted). Returns an IObserver.

**Control Observable/Observer relationship**

1. **Observable.DelaySubscription**(this source, timespan). Delays subcription emitting by a timespan.
2. **O.TakeUntil**(this source, timespan). Stops emitting after a certain length of time.
3. **O.TakeUntil**(this source, **Observable**). Stops emitting when another observable emits
4. **O.SkipUntil**(this source, timespan). Subscribed but only receive data after a certain length of time.
5. **O. SkipUntil** (this source, **Observable**). Receive data when another observable emits
6. **O.Skip(int).** Skips X number of emits
7. **TakeWhile(func).**  Takes a predicate.
8. **SkipWhile(func).**  Takes a predicate.
9. **Repeat(int): Resubscribes** a number of times (or infinitely).
10. **Do(Action):**  Does something!

**Observable**

* **Finally:** Runs when sequence has ended. Good for cleanup.
* **Interval**: Returns incrementing ints at a set interval
* **Timer()**: Either emits once or a continuos emit based on time)
* **Defer:** Only sends messages from when u subscribe.
* **Merge:** Merges observables into 1 observable
* **Range**
* **Throttle(time, scheduler):** ignores results within the timespan
* **DistinctUntilChanged**(): ignores results unless different
* **Switch**(): transforms an observable of observable to observable of the latest results.i.e. switches to the latest observable result
* **Buffer(#itemsinbatch, #items to skip ):** gets as parameters the number of items you want in a batch and the number of items to skip before opening a new batch. So if its (2,1) you skip have the 2nd item in batch in next batch. Also has a **Scheduler**
* **ObserveOnDispatcher**() : used by WPF to go back onto UI context
* **SubscribeOn(Scheduler):** Sets the scheduler
* **Subscribe(IObserver** or its Next,error , complete methods): passes in iObserver and returns a disposable

It's all about **IObservable<T>** which is basically a sequence which can be finite or continuous. A **producer** emits items onto the **sequence** and **consumers** passively receive these items and act on them. **It is a push model** where producers emit data onto the sequence and consumers who have registered interest in the sequence via the Subscribe method receive that data. In between the producer and the consumer we can create multiple stages where we apply operations such as transformations, filters, projections and the like.

Each operation is applied on an IObservable<T> and returns an IObservable<T>.

**Cold Observable**: The Observable starts emitting when a consumer subscribes. Each subscriber gets a unique and complete sequence.

**Hot Observable** : The observable is emitting whether it has any consumers. Each consumer gets the same seqeuence.

# WPF

Its UI presentation software for Windows based applications.

Its architecture is 3 layers; Managed(PresentationCore and PresentationFramework) which connect with an UnmanagedLayer (MILCore) of which Visual is the bridge. The UnmanagedLayer connects to the System layer to access DirectX, Kernel, User32 (manages memory).

Benefits. It allows UI code to be separate from business logic. Uses DirectX as opposed to GDI and is faster and much richer. Databinding. Reusability with templates

Scales up or down depending on system DPI – therefore resolution independent.

Also vector based rendering (which means it doesn’t lose resolution when scaled)

**Assemblies :**

**Milcore.dll** converts object into DirectX objects. Unmanaged code. This is known as the **MIL** (**Media integration layer** and it maintains the **visual tree**)

**PresentationCore.dll** holds base types such as UIElement, visual. Its a wrapper around MilCore

**PresentationFramework.dll** holds top level types such as FrameworkElements controls, windows and styles

**WindowsBase.dll** contains DependencyObject and DispatcherObject

**Architecture**

**Managed** : **PresentationCore**, **PresentationFramework**

**Unmanaged** : **Milcore**

**OS Layer** : Kernel, DirectX, GDI...

Each app has 2 threads, 1 to manage the UI and 1 to render the controls

**Base Classes:**

**DispatcherObject -** Verifies if code is running on the correct thread and access the dispatcher to marshal code to the UI thread

The Dispatcher is owned by the Application and has queues of messages (based on priority). DispatcherObject has a reference to the dispatcher and sends messages to it. (MyTextBlock.Dispatcher.Invoke(()=> MyTextBlock.Text = “Hello from former diff thread!”;)

**DependencyObject** –has functionality to Get, Clear, read, coerce Dependency properties..

**Visual** – class that encapsulates drawing instructions. **Provides the link between WPF and milcore**. Where ‘controls’ coordinates are set up. Instructions are cached

**UIElement** –**LIFE** adds support for layout(**Measure** and **Arrange**), **input**, focus, **events**. IE mouse and key actions to the ‘control’

**FrameworkElement** – implements some of UIElement members. Also support for **databinding**, animation and **styles.** Has **Parent, DataContext** property.

**Shape** – Rectangle, Line, Ellipse…

**Control** – Most of the controls.. sets font, foreground and background colors. **Template support**

**ContentControl** – base class for controls that contain some content.

**ItemsControl** – show collection of items – Listview TreeView

**Panel** – Base class for all layout containers – elements that contain at least 1 child, Has **children, datacontext** property.

**Logical Tree** : displays the controls as rendered on screen. WPF uses this tree a lot when doing resource lookup, routed events and property inheritance. Its a simplification of the Visual Tree. Not really a tree as it contains logical islands due to templates. (this is where **TemplateBinding** comes in). It can contain **CLR** objects (ie a student class for an **listboxitem**) as well as DP’s

**Visual Tree** : Displays the controls broken down into their core visual components ie **Button** is a **button chrome, content presenter** and **textblock** at its core. Can **only** be used by Dependency properties.

Use **LogicalTreeHelper** methods [**GetChildren**](https://msdn.microsoft.com/en-us/library/ms598654(v=vs.110).aspx)**(do),**[**GetParent**](https://msdn.microsoft.com/en-us/library/system.windows.logicaltreehelper.getparent(v=vs.110).aspx)**(do), and**[**FindLogicalNode**](https://msdn.microsoft.com/en-us/library/system.windows.logicaltreehelper.findlogicalnode(v=vs.110).aspx)**(do, string)** for tree traversal. **VisualTreeHelper** methodsare **GetChildCount(), GetChild(do, int)**

**Dependency Properties**

Doesn’t store value like regular prop. Instead it uses DP framework to determine value when needed. It stores the value in an internal storage system and uses the DP to get/set the value.

allows change notification, inherited, default values and economical property storage

DPs can **only be added to Dependency Objects** –

You register DPs by passing in certain values

The property name, Data type, type that owns the DP, optional metadata, optional validation callback

Some metadata options are **DefaultValue, CoerceValueCallback, ValidateValueCallback (**returns bool. If false then exception thrown)**, PropertyChangedCallback.** Use **FrameworkPropertyMetadata** to get more options when creating a DP(such as inheritance).

Some **FrameworkPropertyMetadata** options: Default, inherited, Affects Measure, affects arrange, IsSealed, all callbacks

Property wrappers are added to wrap it in a traditional property. These use **GetValue** and **SetValue** methods defined in the base. Their not necessary.

Their only used by the programmer within code. Xaml calls **DependencyObject.Get/SetValue** directly..

**GetValue** gets the latest value from **SetValue** or Default value if SetValue has never been called. The DP is static which means the DOs DP is the same property for each DO, which results in a lot of saved memory and better performance.

public class Button : ButtonBase

{

// The dependency property

public **static readonly DependencyProperty** IsDefaultProperty;

//Register property

Button.IsDefaultProperty = **DependencyProperty.Register**(“IsDefault”,t**ypeof(bool), typeof(Button),**

new FrameworkPropertyMetadata(false, new **PropertyChangedCallback**(OnIsDefaultChanged)));

// A .NET property wrapper (optional)

public bool IsDefault {

get { return (bool)**GetValue**(Button.IsDefaultProperty); }

set { **SetValue**(Button.IsDefaultProperty, value); }

}

Default base value for property are set by following factors

1. The default value

2. The inherited value (if the FrameworkPropertyMetadata.Inherits flag is set )

3. The value from a styles /templates setters and triggers Theme Style-> ControlStyle-> Parent template

5. The local value

When getting final value you go through the following if they pertain

1. Get default value from above

6. Evaluate expression if any

7. Run animation if any

8. Run CoerceValueCallback to correct value

9. Run ValidateValueCallback to correct value

DP **ChangeNotification**  *how property triggers and binding work*

When a DP changes the dp triggers a protected method named **OnPropertyChangedCallback**(). This method passes the information along to two WPF services (**data binding** and **triggers**). It also calls the **PropertyChangedCallback**, if one is defined.

**DP Change Notification**

1. **CoerceValueCallback** is called first passing in value. This returns a new value. **Only changes value in control Not in the DP**
2. **ValidateValueCallback** is then called passing in value. – this is used to catch errors. Returns bool. If false then exception raised
3. Finally if both previous succeed then **PropertyChangedCallback** is called

DP property value inheritance flows down the logical tree not down a classes inheritance.

**Dependency Object/Property under the hood.**

**DependencyProperty** is a static class. It contains a static hashtable of all unique DPs used in the app(1 per DP). These entries contain the DP, its default data and metadata such as callbacks. The key for each hashtable entry is the DP and DO names as 1 string ie TextBoxText

Each DependencyObject instance in the App has an array called EffectiveValues. If any of its DPs have their value set by either local, animation or binding then this EffectiveValues will get the DP’s value. Otherwise they will just get the value for Default (from the DP itself), style, template or trigger. So only DPs that have the DO.SetValue() called will store data(in the array).

**Attached Property** – a **static** property that may apply to many controls but are contained within another class. Used a lot in layouts Eg. Grid.Row=”1”

Attached properties are so as they allow extensibility to the framework. Otherwise they would have to be defined in framework element(grid.row, Dockpanel.Dock...)

The Ap stores the value like a DP but the value is used by another control..

~~Attached properties are examined "upwards" explicitly, in the code of specific objects. In the case of Grid, upon determining where to place its items, it checks for the value of Grid.Row and Grid.Column attached properties on each contained item.~~

~~Attached properties can only be used by child classes of the parent class (the parent contains the AP) IE if a button is within a grid, it can use the grid.row ap. If its not within the grid it can’t use its APs…~~

Attached Prop and DP differ in the fact that AP is used by different class to where its defined. Aps use the GetValue and SetValue methods directly instead of the property wrapper because the AP can be set on any property. And they pass in a Framework element that uses the value.

**APS can be created by any class, DPS only within a DO**

The Element that has the AP will then call SetValue/GetValue passing itself in usually with a relevant value.

Ie if a TextBlock is using Grid.Row it passes itself into the SetRow method which is in the Grid class. The grid class then knows that the TextBlock needs to be with 1 of its rows. The grid gets the row number from the TextBlock and adds the TextBlock to that specific row!

public static void SetRow(UIElement element, int value)         {  
             element.SetValue(Grid.RowProperty, value); {

Where element is TextBlock.

Then the parent control will get all its children (usually in MeasureOverride or ArrangeOverride) and work on them when they have the AP defined…

[https://blogs.msdn.microsoft.com/helloworld/2010/06/25/understanding-wpf-dependency-property-and-attached-property/](https://blogs.msdn.microsoft.com/helloworld/2010/06/25/understanding-wpf-dependency-property-and-attached-property/%20)

**Behaviours** are a declarative way of adding functionality to elements that also fits in with the MVVM principle. Its the XAML equivalent of C# extension methods.

**Attached Behaviour :** Is a static class that contains >0 attached properties which use a callback to get at the calling Element (Dep Object). This then allows you to write code to access the element usually by way of events.

The issue with Attached Behaviours is having to use a public Static class thus no encapsulation. Also memory leaks

<ListBox..

WPF:ListBoxBehaviors.DoubleClickCommand ="{Binding ShowDetailsCommand}"></ListBox>

**Blend Behaviour:** These are from Interactivity library. They allow you to inherit a Generic Behaviour<T> class which gives you access to the element <T> when you override an OnAttached method. No need for public static classes!

**CommandBehaviorBase<Control>**: Used by Prism for behaviours. Takes a command and parm and fires ExecuteCommand

**Namespaces**

At least one namespace is needed which will define the object your describing in XAML. This is usually xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" which maps to System.Windows.Controls. Notice it doesnt have a shortcut. This means you can write <Button as opposed to <XX:Button.

Also xmlns:=x"http://schemas.microsoft.com/winfx/2006/xaml" which maps to System.Windows.Markup. Note the prefix x. This is a shortcut(1 namespace doesn't need a prefix per xaml file) which needs to be used x: anytime you want to access any objects from its namespace.

The clr-namespace directive enables you to place a .NET namespace directly inside XAML.

The namespace address is defined within the AssemblyInfo class of assemblies under a **XmlnsDefinitionAttribute**. Multiple assemblies can have the same address and so can be loaded under the same shortcut. Just have to make sure that all the assemblies that share a namespace address do not introduce 2 classes with the same name.

Use **XmlnsPrefixAttribute** to set the prefix(ie like the x above).

**XAML**

**Its a declarative way of constructing objects and setting properties**

Allows the UI to be separated from the code.

Every element in a xaml doc maps to a .Net class. Elements contain attributes which map to the classes properties.

Classes used in XAML need a no arguments constructor. And public properties.

Its compiled into BAML which is then embedded into the assembly. Also a .g.cs file is created for your window class. (the window class is defined at the top of the xaml file ie <Window x:Class="WindowsApplication1.Window1"

The .g.cs is added to the assembly and the baml is embedded into the assembly

InitializeComponent() method in class extracts the baml and build the UI

Types of XAML

WPF Xaml –

Silverlight Xaml – subset of WPF Xaml

WF Xaml – for Windows Workflow Foundation

XPS Xaml – used for electronic docs

**TypeConverters** are used to convert data types to and from other types which XAML can understand. Ie String to Color. The type converters are attributes that sit either on property or class level.

To create your own Inherit from **TypeConverter** and override **ConvertTo** and **ConvertFrom.**

**Property-Element syntax** – Used for complex properties. Allows an object which has many props to be used for a property

<Grid Name="grid1">

<Grid.Background>

<LinearGradientBrush>

<LinearGradientBrush.GradientStops>

<GradientStop Offset="0.00" Color="Red" />

**MarkUp Extensions** – Used when getting value of existing object or get dynamic value..

<Button ... Foreground="**{x:Static SystemColors.ActiveCaptionBrush}**" >

**Extensions are just classes with default constructors** which inherits from **MarkUpExtension** and overrides **ProvideValue** ()

Returning this from ProvideValue causes the extension to be evaluated again.

ProvideValue contains a IServiceProvider which allows you to get the XAML Target by calling serviceProvider.GetService(typeof(IProvideValueTarget)) along with the target property.

**Markup Examples**

**Null**

**Static -** Allows you to use static classes and enums

**StaticResource** – a static property

**DynamicResource** – resources that’s reapplied if changed

**Binding** – a binding

**x:Null** = Returns null

**x:Static** returns a static object from wpf framework

**x:Reference** You pass a name of type and it returns the live object in question Source={x:Reference rgvCircles}

**x:Type** returns the type in question

**x:Array** Creates an array

**TemplateBinding** – a value from the control on which you’re applying a template

**Children of Object Elements**

There can be three types

1. **Content** : <Button> OK </Button>
2. **Collection Items** : <ListBox>

<ListBoxItem Content=”Item 1”/>

<ListBoxItem Content=”Item 2”/>

</ListBox>

1. **Type Conversions** : <SolidColorBrush>White</SolidColorBrush>

The **Xaml** parser will firstly check if object inherits from **IList**, then **IDictionary**, then checks to see if it has a **ContentProperty**. Otherwise checks if object has **type converter** and child is plain text. Otherwise it will raise an error.

**Nested Elements** – elements can deal with nested elements in 1 of 3 ways

If the parent implements IList the parser calls IList.Add and passes in the child

If the parent implements IDictionary the parser calls IDictionary.Add and passes in the child

If the parent is decorated with the ContentProperty attribute, the parser uses the child to set the property . Grid uses this

If the object has a type converter it will run this.

Else raise an error.

Primitive types can be created by supplying the string representation of the data eg <sys:DateTime>10/30/2013 4:30 PM</sys:DateTime>

Xaml can be created in

Code only

Code and compiled xaml – This is the most common approach.

Code and uncompiled xaml. XamlReader parses the xaml on the fly

**Layout**

All inherit from **Panel** whose **Children** property takes a collection of UIElements

**Custom Layout Panel:** Override **Panel** class and **FE.MeasureOverride** and **FE.ArrangeOverride** methods using **Panel.Children** property within each method.

A Panel does 3 things.

1. Store list of children;
2. Sizes these children
3. Positions these children

**StackPanel** – places elements in horizontal or vertical stack. Good for small sections of a large complex window. Set **Orientation** to **Horizontal** if you want left to right. Only as big as biggest element.

**StackPanel** measures its children to infinity.

**WrapPanel** – places elements in horizontal or vertical wrapped lines . Good for toolbars. Set **FlowDirection** to **RightToLeft** to flip.

Has **ItemWidth/ItemHeight** properties to define its childrens size

**DockPanel** – aligns elements against an entire edge. Good for the overall ‘main window’. Uses attached properties that its children use. **LastChildFill** will fill the remaining space with the last child! The order in which controls are added to the DockPanel control is crucial in determining

the layout. The element without docking must come last!!

**Grid** – Uses rows and columns. Most commonly used. Grid.**ColumnSpan,** Grid**RowSpan**.

**UniformGrid**. Like grid but forces all cells to have same size. Rarely used. **Columns, Rows** properties

**Canvas** – elements can be placed anywhere by fixed coords. Its a resource lightweight panel. You define location based on the Canvas attached properties. Ie <Canvas>

<Button C**anvas.Top="20" Canvas.Left="30"**>Button</Button>

</Canvas>

dadYou can only define 1 horizontal and 1 vertical AP for each control within the canvas.

**VirtualizingStackPanel** used by databound lists for performance. You need to turn it on (**IsVirtualizing**=true). Also **VirtualizationMode** reuses the container.

Used by **ListBox, ListView** and **DataGrid.** Defined as their **ItemsPanelTemplate**. Exists but is turned off for **TreeView**.

**SelectiveScrollingGrid** used by Grid. It allows you to freeze certain cells.

Note: Virtualizing can take a performance hit if you have a few hundred entries and are scrolling a lot as the items are constantly being created, discarded and recreated.

**Border.CornerRadius** – rounds corners of border

**Grid Sizing**

Absolute size “100”

Automatic Size – Auto – each row/col is given the exact amount of space it needs. Measures its children to infinity!

Proportional size - \* Space is divided between a no of rows/columns. This is the Default.

<RowDefinition Height="\*"></RowDefinition>

<RowDefinition Height="2\*"></RowDefinition> 2nd row is twice the height of the first row

**GridSplitter.**

**If Vertical:Width=3 HorizontalAlignment = bottom VerticalAlignment=stretch**

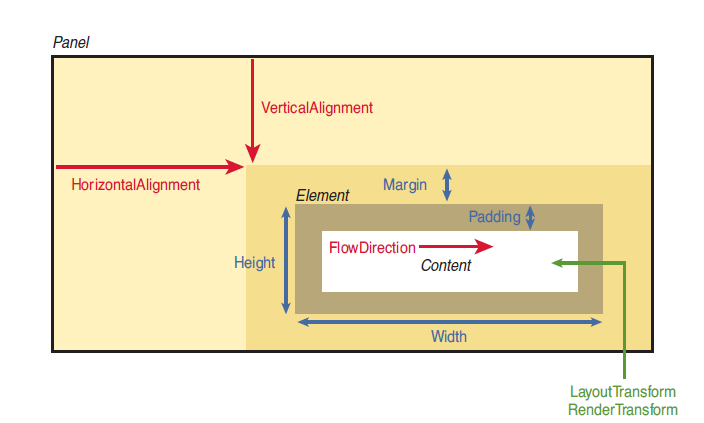
**If Horizontal Height=3, HorizontalAlignment=Stretch VerticalAlignment=Bottom**

<GridSplitter Background="Black" Grid.Row="1" Height="5" ShowsPreview="True"

HorizontalAlignment="Stretch" VerticalAlignment="Top"></GridSplitter>

**When displaying elements 2 things happen**

1. Measure pass : Each container loops thru its controls getting the requested dimensions for each of its children elements. Returns desired size
2. Arrange pass: The system the tries to arrange its children in their appropriate position, cutting them off if they can’t be displayed





**Control Size**

**Height**(defaults to Double.Nan), **MinHeight, MaxHeight.** Try not to use Height

**DesireHeight**(calc. during layout based off above 3 values). **RenderSize** is the actual Size after layout. Same as **ActualSize**. All 3 are readonly

**Margin** – adds space around element. All framework elements have it. Uses **Thickness** structure which accepts 4 doubles

**Padding** – adds space between element and its content. All controls (+ border) have it. (panels, shapes don't have padding)

**Visibility**

**Visible, Collapsed, Hidden. Hidden** still participates in layout unlike **Collapsed.** IE it keeps its **ActualSize**

**Controlling Position** (works with Framework Elements)

**Alignment : *Enables an element to control what to do with any extra space its parent panel gives it* NB** Only useful if it has extra space!

**Horizontal** and **VerticalAlignment.** **Stretch** is default for both. **Left, Right** for **Horizontal** and **Top, Bottom** for **Vertical.** Both have **Center** and **Stretch**

**ContentAlignment** (Works with Controls only)Aligns the content within the control

**Horizontal** and **VerticalContentAlignment.** Left is default for Horizontal, Top for Vertical. **TextAlignment** for **TextBlocks**

**Controls. EACH CONTROL CAN ONLY HAVE 1 PARENT**

1. **Content Controls**

They contain a single item (which obviously be a panel of items! Such as a GroupBox).

They have a **HasContent** boolproperty.

They have a **ContentTemplate** property.

They have a content property of type object. If the control has no visual attributes (ie its not a UIElement) then its ToString() is called.

**Button**., **checkbox** is a **togglebutton**

Has **IsDefault**, **IsCancel** properties

CheckBox has **IsChecked** property

**RadioButton**. To group they just need to be within a panel control. Also setting **GroupName** property

**Label**. Like a **TextBlock** except it can accept key strokes. So **<Label Target=tBox>**\_This</Label> will set focus in the tBox **textbox** when Alt -T is selected.

**Frame**. Can contain any kind of content. Such as HTML

**GroupBox, Expander.** They both have a **Header** property

**Tabbing**: Done by placement in xaml. Can override by **TabIndex**=”1”. Its an AP. K**eyboardNavigation.IsTabStop**="False" wont tab into control.

1. **Item Controls**

**ComboBox, ListBox, ListView, TabControl, DataGrid** are the relevant **Selector** Item controls.

**Items** property contains its items. Will render as expected if **UIElement** or else as a **TextBlock** containing its ToString(). Good for items known at compile time (ie for Tabs)

It is readonly if **Items** is being used., but can contain any arbitrary item. Its not a DP so you can’t bind to it. Ie you must populate it with code. You can only populate it 1 at a time (ie within a foreach loop)

Also called Direct mode and is also set by creating **items** within the **itemscontrol** in xaml.

**ItemsSource** is used to populate from a collection

Both **Items** and **ItemsSource** can’t both be set at the same time. But you can always retrieve from **Items** property!

**DisplayMemberPath** displays a property of an item in the collection

**SelectedValuePath** sets the value of the selected property in the collection i.e. Student.ID (where the ItemSource is Students)

**SelectedItem** is the first item in the selected list.

**SelectedValue**. Its the **SelectedItem** if the **SelectedValuePath** is empty otherwise the property stated in the **SelectedValuePath**

**IsSynchronizedWithCurrentItem** : syncs the [**CurrentItem**](http://msdn.microsoft.com/en-us/library/system.componentmodel.icollectionview.currentitem.aspx)of the default [**CollectionView**](http://msdn.microsoft.com/en-us/library/system.componentmodel.icollectionview.aspx)of the bound collection with the SelectedItem of your control.

**ItemsPanel** used to customize how items are arranged. Ie <stackPanel orientation=”Horizontal”/> displays the items horizontally

**<ListBox.ItemsPanel>**

**<ItemsPanelTemplate>**

**<WrapPanel />**

**</ItemsPanelTemplate>**

**</ListBox.ItemsPanel>**

IsItemsHost is an alternative way to specifiy the panel by defining it in the ItemsControl.Template, not in the ItemsPanelTemplate

<ScrollViewer Padding="{TemplateBinding Padding}">

**<WrapPanel IsItemsHost="True" />**

</ScrollViewer>

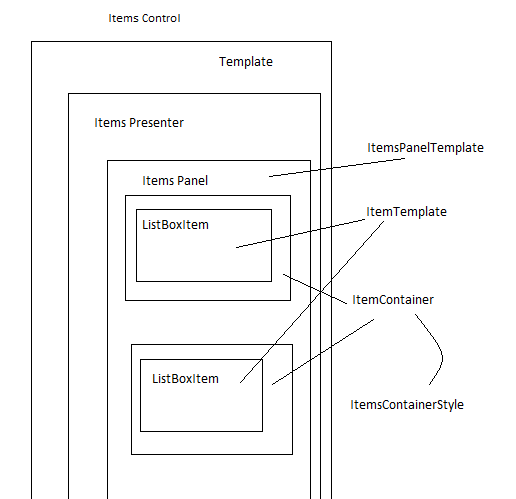
**ItemTemplate**. A **DataTemplate** of how content of data item will be displayed

[ItemTemplateSelector](ms-xhelp:///?method=page&id=p:system.windows.controls.itemscontrol.itemtemplateselector&vendor=Microsoft&topicVersion=110&topicLocale=EN-US)

**ItemContainerStyle** How the container for the ListItem will look

**ComboBoxItem**, **ListBoxItem** and **TabItem** are properties for their respective Items Controls which give you extra properties. **TabItem** is needed as it has a Header property which creates a specific header for each tab(else the header is the same as each tabs content.)

**SelectionMode: Single, Multiple, Extended**(using shift or ctrl key to select multiple)

****

**DataGrid** has functionality for sorting, freezing, reordering...

It has specific columntypes such as **DataGridTextColumn, DataGridHyperlinkColumn, DataGridCheckBoxColumn, DataGridComboBoxColumn** and **DataGridTemplateColumn.**

DataGrid columns are **not** part of the visual or logical tree, so they **don't** inherit the **DataContext.**

You can use **AddOwner** functionality to add DataContext to column.

FrameworkElement.DataContextProperty.AddOwner(typeof(DataGridColumn));

Then **OverrideMetadata** for DataContext on DataGrid

FrameworkElement.DataContextProperty.OverrideMetadata ( typeof(DataGrid),               
new FrameworkPropertyMetadata   
   (null, FrameworkPropertyMetadataOptions.Inherits,   
   new PropertyChangedCallback(OnDataContextChanged)));

Finally:

public static void OnDataContextChanged(DependencyObject d, DependencyPropertyChangedEventArgs e){

DataGrid grid = d as DataGrid;

foreach (DataGridColumn col in grid.Columns)

col.SetValue(FrameworkElement.DataContextProperty, e.NewValue);

}

**RowDetailsTemplate** is a **DataTemplate** for each Grid row

**ComboBox**.

**Text** property,

**IsEditable:** Allows you to type into cbobox as well as select dropdown

**Note** ItemControls should not be displayed within a Panel that has auto height (stackpanel or grid with row height as Auto) as they will display all the items and you cant scroll vertically. Use grid with row height of \* instead.

**MenuBar** stacks its items horizontally. You need to use its **ItemsPanel** to arrange it differently. **MenuItems** are its items property. **IsMainMenu** property gives menu focus when you press Alt

**ContextMenu** can’t be embedded into tree. It needs to be added using a controls AttachedProperty (<ListBox.ContextMenu>..)

As its not part of the visual tree use **PlacementTarget** when binding

<ContextMenu> <MenuItem Header="Proc Details"

Command="{Binding Path= GetProcessDetailsCommand}"

CommandParameter="{Binding PlacementTarget.SelectedItem, RelativeSource={RelativeSource AncestorType={x:Type SelectiveScrollingGrid }}}" />..

**ToolBar. OverflowMode** property**. Always, AsNeeded, Never. Never** will cut off some controls if its not got space

1. **Other Controls**

**Image. Source** property which takes a URI. **Stretch** prop defines how its displayed

**TextBox. Text** property**. TextWrapping** set to Wrap to show all text

**Textblock. Text** property**. Inline** property (same as content) can contain Run, Bold, … Not a control to make it more lightweight.

**ProgressBar: IsIndeterminate** shows actual value or generic progress. **Maximum, Minimum, Value, LargeChange, SmallChange**

**Slider: TickFrequency** Interval between ticks displayed on control. **TickPlacement** shows location on control. **Maximum, Minimum, Value, LargeChange, SmallChange**

**Decorators:** These are controls with a single child element such as **Border** and **AdornerDecorator. Its a lightweight version of ContentControl.**

**Adorner**  an element that sits on top of another element and has a higher Z order. **ViewBox** and **border**

**AdornerDecorator: A**dds support for drawing in adorner layer such as focus cues. This is why its near the tip of the visual tree for each Control.

**AdornedElementPlaceholder** : Notifies where the **adorner** is. Used when creating custom error template

**ViewBox** : Auto scales, stretchs its contents. Good for scaling fixed size controls.

**Run**: Inline content element which contains text

**ScrollBars**.. They don’t appear if controls container is of infinite size such as Stack or grid row when its height is \*. To overcome this wrap the control within a **ScrollViewer** or use a container that constricts its growth ie grid row when height is auto.

<Stack>

<ScrollViewer ScrollViewer.VerticalScrollBarVisibility="Auto">

<DataGrid …/>

</ScrollViewer>

**Scrollviewer** is not recommended around an **ItemsControl** as it will display all of its items. Not good if you’re leveraging virtualization. This also happens if you contain the **ItemsControl** within a **StackPanel**

**ScrollViewer** measures its child to infinity in each scrollable direction

**ScrollView.IsDeferredScrollingEnabled**

**Virtualization**

2 types

**DataVirtualization:** Only returns subset of results;

**UI Virtualization**: Optimization to avoid creating container objects for items that are not visible

1. Set **ItemsControl.ItemsPanel** to **VirtualizationStackPanel**
2. Set the **VirtualizingStackPanel.IsVirtualizing = “True”** within **VirtualizationStackPanel**  element**.** This only creates Item containers for visible items
3. Set **VirtualizingStackPanel.VirtualizationMode = “Recycling”.**  This reuses the container objs

**Note** These are all set up for **ListView** and **ListBox**. Not for **ComboBox** or **TreeView**

**Note**  VIrtualization is stopped if you do the following

Add Item containers manually

Add grouping to control

Set **IsVirtualizing** to false

Weak Event. WeakEventManager<DataGrid, MouseButtonEventArgs>.AddHandler(grid, "MouseLeftButtonDown", grid\_MouseLeftButtonDown);

Allows subscriber to be destroyed .

**Routed Events.** Allows the event to originate in one element but be raised in another. Different from .NET events as otherwise you would have to create the same events for some of the controls relatives

3 types

**Direct Events**: Ordinary net events. They don’t pass to other elements. **MouseMove** is an example as you wouldn’t need to know about a cells parents mouse coordinates.

**Bubbling events** : these travel up the containment hierarchy, from element to parent to parent…

**Tunneling events** : these travel down the hierarchy. They allow you to preview an event before it reaches the appropriate control. They go from the topmost parent on the visual tree and goes down till the element that fired the event.

Source: Where event started

OriginalSource

Handled : When set to true then stops carrying event further. (Unless you subscribe to event using UIElement.AddHandler!!)

public static readonly RoutedEvent ClickEvent;

static ButtonBase()

{

ButtonBase.ClickEvent = E**ventManager.RegisterRoutedEvent**("Click", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));

public event RoutedEventHandler Click

{

add{AddHandler(ButtonBase.ClickEvent, value);}

remove{RemoveHandler(ButtonBase.ClickEvent, value);}

}

protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)

{

RaiseEvent(new RoutedEventArgs(Button.ClickEvent, this));

}

}

**AttachedEvent**: < StackPanel **Button.Click="DoSomething"** Margin="5">

**WPF events** : there are 5 categories

Lifetime events : occur when the element is init, loaded or unloaded

Mouse, keyboard, stylus and multitouch events

**Application Class** It’s a singleton

Call by code : Application app = new Application();

App.Run(new Window());

Call by Xaml : <Application x:Class="TestApplication.App"

StartupUri="Window1.xaml"/>

App Events: Startup, Exit, DIspatcherUnhandledException, Activated, Deactivated

Application.Current accesses the current application

Application resources. Contains the strings and images used by the app

**Commands** <Button Command="ApplicationCommands.New">New</Button>

**ICommand**

void Execute(object parameter);

bool CanExecute(object parameter);

event EventHandler CanExecuteChanged; - > to fire CanExecute requery. Call CommandManager.RequerySuggested

**Routed Command** – only wpf implementation of ICommand. Empty Shell that depends on either System or custom commands. Their Execute methods are in the code behind.

Their Execute/CanExecute have no logic. Instead they raise events that traverse Element tree looking for **CommandBindings** .

The **CommandBindings** have their own event handlers that contain the command logic.

Their good as the RoutedCommand is decoupled from the handler. Also Input Gestures are available

They hook into the the **CommandManager**. As such you have no control over when the CanExecute method is raised again. ***Major reason not to use them.***

**RoutedUICommand**. Same as **RoutedCommand** except it has a Text property which is displayed on window.

**CommandBindings** - XAML where you define the command. Used with RoutedCommands and code behind. Contains properties to define the Command and also the events **Execute**, **CanExecute** (and preview..).

**Difference** between **RoutedCommand** and **DelegateCommand**.

1. **RoutedCommand**  implementation has to be written in the code-behind. With the **DelegateCommand** its put in a different class such as a VM.
2. **DelegateCommand** doesn’t support routing through the element tree. This is fine if the interface is based off 1 VM but if its split up then you need to use **RoutedCommands** to invoke child commands. (However you can also use Prism **CompositeCommands**

**Resources**

Contained in **FrameworkElements** and **FrameworkTemplates**

**Binary Resources.** Can be embedded, or local as in in the file directory. Mainly used for images.

When embedded, set **BuildAction** to **Resource.**

<Image Source="pack://application:,,,/Resources/1.jpg"/> - local image (all you need is Source="Resources/1.jpg")

<Image Source="pack://application:,,,/WpfRemote3;component/Ress/3.jpg"/> - remote image (WpfRemote3 is the assembly name. Component and application are always needed)

When content (ie does not have to recompile when resource changes) set **BuildAction** to **Content.** Good for content that changes frequently.

To use resources not added to project you use **siteOfOrigin**(instead of **application**)

<Image Source="pack://siteOfOrigin:,,,/OfficeFrontDoor.jpg"/> make sure the jpg is in the bin folder

**Logical Resources**

Every **FrameworkElement** has its own resource **dictionary**. However you should only use App resources if the resource is used widely.

**FrameworkElement.FindResource[**“Name”] - throws error if no resource.

**FrameworkElement.TryFindResource[**“Name”] - no error

**FrameworkElement.Resource[**“Name”] - throws error if none.

**Resource Lookup**: This is where we search up the **logical tree** for a resource, from element to parent to ancestors to **Application**. It then goes to Themes and System defined resources. Exception is thrown if nothing is found. The same key can be used in different collections. The 1 nearest the element is used.

**Static resource** – **Gets from resource only once**. Gets loaded at Application load. All static resources share the same resource.. If its modified (in the resource dictionary) the SRs also change. You need to replace the entry in the resource dictionary with a new object to not change the Static resource

To not share a resource use **x:Shared=”False”**

Can be any CLR object

Resolved at compile time

**Dynamic resource** – **Gets from resource dict every time.** reapplied if the resource changes. Gets loaded only when needed. Can **only** be used on DP’s.

Dynamic resources are good for dealing with System colors/fonts.

Can only be DPs.

Resolved at compile time

Resource xaml needs to be defined above its calling as xaml loads top down. This is not needed for Dynamic resources.

Resource Dictionary – Used when sharing resources among projects. You then merge it into your app resources

Resource Tips. Don’t create if just used once, inline instead. Best to merge a usercontrol resource as opposed to having defined in UC xaml as its cleaner. Dynamic resources are basically like bindings, they don’t use up too much memory.Dont hardcode resources in App.resources. Merge them instead.

**Styles :** Separates property values from UI Elements. Their stored in Resources. So basically every **FrameworkElement** and **FrameworkTemplate**

<Window.Resources>

<Style x:Key="BigFontButtonStyle">

<Setter Property="Control.FontFamily" Value="Times New Roman" />

Styles get applied to an element either by e**xplicit** reference (by setting the [**Style**](https://msdn.microsoft.com/en-us/library/system.windows.frameworkelement.style(v=vs.110).aspx) **property**) or **implicitly** by associating a style with the **CLR type** of the element.

Each element can use 1 style <Button Style={StaticResource BigFontButtonStyle }/>

**BasedOn** is like inheritance for styles and uses base styles key. If that’s implicit then something like BasedOn=”{StaticResource {x:Type Button}”

Styles can also be set by **TargetType** <Style TargetType="Button" x:Key=”test”>

If you leave out the **x:Key** then the style is implicit (actually the key is still there, but hidden and is the name of the TargetType) This is known as **Typed style.** Having the key included is called a **Named Style.**

With a named style, it’s okay for the target element to be a subclass of the TargetType. **But a typed style typically gets applied only to elements whose type matches exactly**. This is done to prevent surprises.

**TargetType** styles can be painful i.e. In a complex window, **Textblock** element is used by many elements.

Due to DP owner issues (some Panels DPs are owned by other control types) its best to be specific when setting **TargetType** (i.e. Don’t set **TargetType=Control)**.

Styles can be overridden by setting properties on the local value of the control or by overriding its style

**Style="{x:Null}"** in an element ignores the style

Controls can have multiple style properties

Styles set in **FrameworkElement**.**Style**. Can also be set in **FrameworkElement.FocusVisualStyle, ItemsControl.ItemContainerStyle** and **Datagrid.CellStyle**

~~DataTemplates and Control Templates can set their own style.~~

**Tip:** If a control doesn’t pick up its default style for some reason the just set the style using the implicit key. Ie

<Rectangle … Style="{StaticResource {x:Type Rectangle}}"/>

**Triggers** Style gets changed automatically by DP change

**Property Trigger** – gets changed by the DP change. Use <**Trigger Property**=IsEnabled **Value**=.../>> Can only be used by DPs. Use **SourceName** to define the element whose property you’re looking at (if its not the current element).

**Data Trigger** – gets changed by data change Use <**DataTrigger Binding**={Binding **Path**=...} **Value**=...>. Theres no **Property** as its defined in the **Path**

**Setter:** Used for both P/D triggers. Contains **TargetName** for element you’re changing, **Property** and **Value.**

**Event Trigger** – Fires animation when event occurs

**Styles and DataTemplate** and **ControlTemplate**  all have triggers collections. **FrameworkElement** has Event triggers collection

**Template Trigger** has a **higher** precedence than a **Style Trigger** so it will overwrite it.However using a style trigger allows you not to have to write your own template trigger but of course this will only work if the property you trigger is not used by the defaults templates trigger.

<Style x:Key="BigFontButtonStyle">

<Style.Triggers>

<Trigger **Property**="Button.IsPressed" Value="True">

<Setter Property="Control.Foreground" Value="Red" />

</Trigger>

<DataTrigger **Binding**="{Binding source={} Path=HasErrors}" Value="True">

<Setter Property="Control.Background" Value="Auburn" />

</DataTrigger>

<EventTrigger RoutedEvent="Mouse.MouseEnter">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Duration="0:0:0.2" To="22"

Storyboard.TargetProperty="FontSize"/>

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger></Style.Triggers>

**Logical OR Triggers** Multiple triggers applied to the same element with the same **Setter** results

**Logical AND Triggers** You need to use MultiTriggers

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="HasItems" Value="false" />

<Condition Property="Width" Value="Auto" />

</MultiTrigger.Conditions>

<Setter Property="MinWidth" Value="120"/>

</MultiTrigger>

**Behavior** : encapsulate a common bit of UI functionality.

<Ellipse Canvas.Left="10" Canvas.Top="70" Fill="Blue" Width="80" Height="60">

<i:Interaction.Behaviors>

<custom:DragInCanvasBehavior></custom:DragInCanvasBehavior>

</i:Interaction.Behaviors>

</Ellipse>

**Templates** Allow you to replace an elements visual tree with anything.

A template is just a tree of visual elements that defines a member of the logical tree. As it builds the logical tree, the framework watches for controls and data items that have corresponding templates. When such an element is encountered, the appropriate template is “inflated” into the actual visuals that represent the logical item and inserted into the visual tree.

Templates inherit from FrameworkTemplate class of which there are 3 types

**DataTemplate**: look for a data object. **DataType** property

**ControlTemplate**: Look for a control. **TargetType** property

**ItemPanelTemplate**: Layout of items in an ItemsControl. Can **only** contain one child which **must** **be a Panel.** The **ItemsPanelTemplate** is inflated within the **ItemsPresenter**, thereby creating a panel for the layout of the ItemsControl’s items.

Difference between Data and Control template is that dataT works on an existing control. ControlT creates a new control

Template is just a factory that knows how to spit out a copy of a UI tree.

The default visuals for controls are templates. They are also the reason there's only a limited # of properties for controls look and feel.

A control can have multiple template properties that can be changed.

Use **TemplateBinding** to bind to the templates parent source. Its lightweight and has limited props. Can only be used inside the Visual Tree. So can’t be used inside a templates triggers. Doesn’t do 2way binding. However use regular binding when binding the template to the VM.

TemplateBinding is the same as {**Binding RelativeSource** = **{RelativeSource TemplatedParent}**,…}

**Control Template** : a control is made up of other UI controls. The controls template is the xaml of these other controls. You can replace the controls template with whatever you want. Eg a round button. Note the controls functionality does not change. This is how all controls are.

**TargetType** **must** be set on a control template

**<ControlTemplate.Triggers>** To setup triggers for template

Use names in template elements so they can be accessed by trigger.

Respect Visual States with the **Visual State Manager (VSM).** States for each framework element are defined as **TemplatePart** attributes at the elements class definition.

You can’t change part of an existing template (EG just change the arrow on an expander) Instead you need to get the entire template, change the relevant bits and use that. To get the template you use string xaml = XamlWriter.Save(someControl.Template);

**ContentPresenter :** Used within an Control template and allows you to use the content of the control in the template

<Viewbox>

<ContentPresenter Margin=”20” Content=”{TemplateBinding Content}”/>

</Viewbox>

**ItemsPresenter :** Used in ItemsControl to specify the place where the **ItemsPanel** is added

**Data Template** Maps some XAML to a type

Setting DataTemplate.DataType allows the data template to be applied automatically to the data type. This is very powerful.

Some template properties

**ContentTemplate** Content controls

**ItemTemplate**  item controls

HeaderTemplate

**SelectedContentTemplate** tab control

**DetailsTemplate** DataGridRow

**RowDetailsTemplate** DataGrid

**RowHeaderTemplate** DataGrid

**ColumnHeaderTemplate** GridView,

**CellTemplate** DataGridTemplateColumn, GridViewColumn

**CellEditingTemplate** DataGridTemplateColumn

**DataTemplateSelector** Choose template to use on the fly. Very powerful. Override **SelectTemplate**(collectionItem, Dependency Object). Can be used on **ItemTemplateSelector**, **ContentTemplateSelector**

**VisualStateManager** : Documents parts and states of a control which can be used when creating a template for that control.

**Custom Controls.**

**User Control:** Used to group different controls together. Doesn't support themes.

User control inherits from **ContentControl**

When to use:

Don’t need theming, templates, styles

Wont be shared across WPF applications

**Custom Control**: Look-less. Enhances a control. Its look is created by a theme(generic.xaml file).

inherits from **Control** and has a static ctr(which overrides the default style). Has no InitializeComponent()..

All WPF common controls(Button, panel…) are custom controls.

When to use:

Extend existing control

When you want to customize how it looks

When its to be shared across applications

With Custom Control You add a generic.xaml resource dictionary to a Themes subfolder in project. This is where the style will be.

Use **TemplateBinding** or **TemplatedParent** binding.

To access controls internally override **OnApplyTemplate** and call **base.GetTemplateChild**(*controlname*)

EventHandlers need to be attached in ctr

**ANIMATIONS**

**Linear Animation.** Automate a property change in a linear way. Their named <TypeName>Animation ie **DoubleAnimation**

**Key frame-based Animation:** Use serveral key frames in order. Progress is usually linear. Named <TypeName>AnimationUsingKeyFrames ie **StringAnimationUsingKeyFrames.**

**Path based Animation:** Movement of visual objects along a complex path.Named <TypeName>AnimationUsingPath ie **DoubleAnimationUsingPath**

**Basic Animation**

DoubleAnimation widthAnimation = new DoubleAnimation();

widthAnimation.From = 160;

widthAnimation.To = this.Width - 30;

widthAnimation.Duration = TimeSpan.FromSeconds(5);

cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);

**Visual**

**Brushes**

**SolidColorBrush:** Creates a solid color. Can be written in HEX #FFAABBCCDD where FF is opacity and the other pairs are RGB

**LinearGradientBrush**: Blends 2 or more colors. Has **StartPoint** and **EndPoint** where both are between (0,0) and (1,1). Contains a collection of **GradientStops. GradientStop**  contains a **Color** and **OffSet** property. LGB has **SpreadMethod** enum(**Pad**, **Reflect**, **Repeat** ) which is used if Offset doesn't reach 1

<LinearGradientBrush StartPoint="0,0" EndPoint="0,.5" SpreadMethod="Reflect">

<GradientStop Color="Gray" Offset="0"/>

<GradientStop Color="Silver" Offset=".2"/>

</LinearGradientBrush>

**RadialGradientBrush:** Radiates out in circles. **GradientOrigin** is the center of the gradient(defaults to “.5,.5”. **RadiusX, RadiusY** are the horizontal/vertical radii of the outermost circle of the gradient. **Center** is the center of the outermost circle!

<RadialGradientBrush x:Key="rgBrush" Center=".3, 0.4">

<GradientStop Color="Red" Offset="0"/>

<GradientStop Color="Yellow" Offset="1"/>

</RadialGradientBrush>

**ImageBrush**. Paints an image using **ImageSource** property. **ViewBox** displays a section of the image. (starting coord, x length, y length). **Viewport** sets the position and dimensions of the base tile. Ie ((0,0).5,.5) will show 4 tiles

**Stretch** enum : **None**: image is displayed in its original size with any part of image outside dimensions not painted

**Fill:** Stretches the image to fit the dimensions

**Uniform** Stretched to fit dimensions but aspect ratio is kept. Any outside dimension not painted.

**UniformFill** Stretched to fit dimensions but aspect ratio is kept. Content can be clipped if diff from aspect ratio.

**TileMode.** Tiles the image. Only used if Stretch is none.

**VisualBrush.** Displays an element using the **Visual** property**.**

**Shapes**

**Fill:** Inside of shape. **Stroke**: edge of shape

**Rectangle and Ellipse:** Use **Height** and **Width**. **RadiusX**/**RadiusY** round corners on rectangle. **Stretch** will work instead of Height And Width.

**Line**: Connects two points. Use **X1 Y1** and **X2 Y2** properties. **Stroke** to choose color.

**Polyline.** Draws a complex line. Use **Points** to define the line. **Stroke** to choose color. <Polyline Stroke="Red" Points="30, 30 40, 40 40, 30"/>

**Polygon.** Similar to Polyline but the first and points are used to close the line. Uses **Fill**

**Path:** Combines **Geometry** objects within **GeometryGroup**. **EllipseGeometry**, **LineGeo**, **PAthGeo**, **RectangleGeo**, **CombinedGeo**(combination of 2 geometry objects), **StreamGeo**(like **PathGeo** but is readonly after created)

<Path Fill="Aqua" Margin="1">

<Path.Data>

<GeometryGroup FillRule="Nonzero">

<EllipseGeometry RadiusX="40" RadiusY="50"/>

<RectangleGeometry Rect="0,0,10,100"/>

</GeometryGroup>

</Path.Data>

</Path>.

**Transformations** Alters the shape of elements by altering the coordinate system used to drawing it.

**LayoutTransformation**: Happens before element is rendered

**RenderTransform**: Happens after the element rendered. Has **RenderTransformOrigin** where you state the start of transform(0,0) is default.

Type of RenderTransforms

**RotateTransform**: Rotates the element. **CenterX, CenterY** are center points. **Angle** is angle to rotate.

**ScaleTransform**: Scales the element. **CenterX, CenterY** are center points. **ScaleX** is horizontal scaling **ScaleY** vertical(defaults are 1). Setting **Scale** props to -1 will flip the element.

**SkewTransform**. Slants the element. **CenterX, CenterY** are center points **AngleX** horiz skew, **AngleY** vertical skew.

**TranslateTransform**. Shifts the element. **X** shifts it horizontally, **Y** vertically. Both default to 0.

**TransformGroup.** Groups multipleTransforms.

<TextBlock.RenderTransform>

<TransformGroup>

<RotateTransform Angle="45"/>

<ScaleTransform ScaleX="-1" ScaleY=".9"/>

</TransformGroup>

</TextBlock.RenderTransform>

**Clipping** Clips part of the element

<Button Height="100" Width="100" Name="Button1">

<Button.Clip>

<!--Since the Button is 100 by 100 pixels, an EllipseGeometry

with a Center of 50,50 and a RadiusX and RadiusY of 50 will clip the button so that it appears circular-- <EllipseGeometry Center="50,50" RadiusX="50" RadiusY="50"/>

</Button.Clip> Button

</Button>

**Multimedia**

**SoundPlayer. Load(), Play(), Stop() SoundLocation.** Only plays uncompressed wav files.

**MediaPlayer,MediaElement.** Both play audio and video. Machine needs to have at least **WinMediaPlayer** 10 **MediaPlayer** has no visuals. Methods are **Load() Play() Pause() Stop(). Position** starts at defined timespan**. MediaEnded, MediaOpened, MediaFailed** are events.

**Binding** Updates a **target**(UI element) from a **source** (Data source)

**The binding engine**. WPF checks if your **DataContext** inherits the **INotifyPropertyChanged** interface, subscribes to the event provided by the interface and listens to any changes. If such an event is raised, the binding engine will just call the **Getter** again.

You can also use BindingExpression to to explicitly update.

**BindingExpression** be = itemNameTextBox.**GetBindingExpression**(TextBox.TextProperty);

be.**UpdateSource**();

**Datacontext** is simply an **inherited attached** property. If you don't set it on a child control, it will take the value the parent has until it reached the root. The only exception to this are **ContentControls**and **ContentPresenter**. They will not inherit the **DataContext** but will change them depending on the content. So these controls always have by default the Content as their DataContext.

Can be set in code or XAML. Code is good when you want to clear the binding.

**Binding** bind = new Binding("FontSize");

bind.**Source** = tbEvent;

**BindingOperations.SetBinding**(txtFilter, TextBox.TextProperty, bind);

Clearing binding. Can clear 1 DP or all DPS on a DO.

**BindingOperation.ClearBinding(**DO,DP)

**BindingOperation.ClearAllBindings**(DO)

Get Binding = Binding b = **BindingOperations.GetBinding**(DO,DP).

BindingOperations.GetBindingExpression(DO,DP) : Allows you to get at source

Add it in XAML . You can also use static **BindingOperations** class (**SetBinding**, **ClearBinding**)

**Modes :** Direction of data flow

**OneWay**(target property is updated when source changes)

**TwoWay**,

**OneTime** (good to reduce overhead when the source won’t change), Binding gets disconnected

**One Way to Source**(reverse of one way. Good way of setting a non DP property)

**Default** (Either TwoWay for user settable props. OneWay for everything else)

**UpdateSourceTrigger:** Deals with updating source from Target ie used in TwoWay

**PropertyChanged** : Source is updated when target prop changes

**LostFocus** : Source is updated when target prop changes AND loses focus

**Explicit :** Need to call **BindingExpression.UpdateSource**

Binding to objects. Property must be public

You bind to either **Source :** Some sort of object

**RelativeSource :** can bind to a relation of the source

Mode: **Self**

**TemplatedParent**: The parent of the template the binding is in. (more props than templatebinding)

**FindAncestor** : Pass in **AncestorType** and **AncestorLevel**

**PreviousData**: the previous data item in the data bound collection

**AncestorType**: type of control

**AncestorLevel**: Level of parent. Used when you want to skip a level (1 is current level)

**ElementName :** When binding to another element. This is used instead of Source

**DataContext:** When multiple controls bind to same source set the datacontext of their parent to the source. (this is implicit binding). You leave the source empty in each child binding.

{**Binding}** means the source is set somewhere up the tree AND your binding to the entire object. (in a vIew it might be the vm)

You would not be setting Path in this case.

Difference between **Source** and **DataContext**: all elements children will auto inherit DataContext but not Source

**Path** : The property of the **Source/Element** to use. This is optional if you want to bind to an entire object. (ie binding a coll to a listbox etc)

**TargetNullValue :** A placeholder value thats used if the source value is null..TargetNullValue=Nothing is selected.

**FallbackValue**: Value to use if binding fails

**NotifyOnSourceUpdated: bool** whether to raise event if source updates

**NotifyOnTargetUpdated: bool** whether to raise event if target updates

[**NotifyOnValidationError**](ms-xhelp:///?method=page&id=p:system.windows.data.binding.notifyonvalidationerror&vendor=Microsoft&topicVersion=110&topicLocale=EN-US): Whether to raise the attached event **Error**

**IsAsync.** Whether to bind on a background thread. Should only be used for badly performing code.

**Displaying data in Binding**

**StringFormat** : Used with string objects. <TextBlock Text=”{Binding StringFormat=’Num of items is {0}’}”/>.

If the property is not a string you must use **XXXStringFormat** where XXX is the element type. Ie **ContentStringFormat** as **labels Content** is an object

Other formatters are  **HeaderStringFormat, ItemStringFormat**

**DataTemplate:**

**ValueConverters**  Used for changing data between target and source More powerful then Binding.StringFormat

You implement IValueConverter which contains **Convert** and **ConvertBack** methods

<Textbox Text="{Binding Path=UnitCost, Converter={StaticResource PriceConverter}}"/>

You pass in data and also the targettype(which is the DP) and a parameter and a culture..

Theres a few built in converters such as **BooleanToVisibilityConverter.**

**Types of Binding**

Binding regular binding

**TemplateBinding** binds a template to its parent control.

Only does 1 way binding, no support for ValueConverters, only works on DPs.

When binding to a non dependency object (like a regular c# object) you need to implement **INotifyPropertyChanged** on the object

When binding to a collection of objects you bind to the **ItemsSource**. You can then use **DisplayNamePath** to display a property. You can also use a datatemplate to bind to a custom framework element using either ITemsTemplate or by setting the templates type to the object within the collection

i.e. <DataTemplate DataType="{x:Type local:Product}">

Collection needs to be an **ObservableCollection** to auto update list when the collection is updated

**ObservableCollection<>** This implements **INotifyCollectionChanged** which has a **CollectionChanged** event which will notify the target that was defined in the xaml of the change.

If the elements within the **ObservableCollection** implement **INotifyChanged** then the target will get updated when any of these elements changes.

The target always needs to be a DependencyProperty

Its property on a ViewModel needs to call **INotifyPropertyChanged** if the data is loaded later than at startup (ie if populated with an async call)

**DataValidation**

Need to implement **IDataErrorInfo** (or **INotifyDataErrorInfo** in 4.5) This will add a default red border over control.

You also need to add **ValidatesOnDataErrors=True** or (**ValidatesOnNotifyDataErrors**for **INotifyDataError**)to binding of control.

**INotifyDataErrorInfo**: bool **HasErrors**, Ienumberable **GetErrors**, **ErrorsChanged**

You can write a **Validationrule** which gets checked each time the data is updated.

Then in control you need to set <**Binding.ValidationRules> <local:JpgValidationRule/>**

To customize you can add an **Validation**.**ErrorTemplate attached property** to the control. **Validation.Error ap** is true if control has error.

This contains an AdornedElementPlaceholder which represents the control

DP’s become invalid if **ValidatePropertyCallback** returns false.

Binding Error Properties

**ValidatesOnExceptions**: Raises error if exception in code that runs when property is updated.

**ValidatesOnDataErrors**: If using **IDataErrorInfo** and error is raised on property

**ValidatesOnNotifyDataErrors**: if using **INotifyDataError**and error is raised on property

[**NotifyOnValidationError**](ms-xhelp:///?method=page&id=p:system.windows.data.binding.notifyonvalidationerror&vendor=Microsoft&topicVersion=110&topicLocale=EN-US): Whether to raise the attached event **Error**

**ValidationRules**: Collection of ValidationRules

Adorner properties

**Validation**.**ErrorTemplate:** An AP that you add to control to display error

**Validation**.**HasError:** An AP that’s true if you have errors

**Validation**.**Errors:** An AP that contains the errors. You get it by Validation.Errors[0]

**CollectionView** Created during binding. It sits between your data source and the control (eg between coll and listbox)

It is the **CollectionViewSource**

Its a window onto your data source. Tracks the current item and supports **sorting, filtering, navigation** and **grouping**

Observable collections return ListCollectionView. IEnumberable collections return CollectionView which offers poor performance

Members include [**CurrentItem**](ms-xhelp:///?method=page&id=p:system.windows.data.collectionview.currentitem&vendor=Microsoft&topicVersion=110&topicLocale=EN-US)**, Count, MoveToFirst, Last, Next, Previous** and **MoveToPosition; CurrentChanged** event

You get the dataview by the following code

**ListCollectionView** view =(ListCollectionView)**CollectionViewSource**.**GetDefaultView**(lstProducts.ItemsSource);

**Filtering**

view.Filter = new Predicate<object>(FilterProduct);

public bool FilterProduct(Object item)

{

Product product = (Product)item;

return (product.UnitCost > 100);

}

**Sorting**

view.SortDescriptions.Clear();

view.SortDescriptions.Add(new SortDescription("ModelName", ListSortDirection.Ascending));

**Navigation**

view.MoveCurrentToPrevious.

View.IsCurrentBeforeLast...

# 

## DataProviders

**ObjectDataProvider.** Binds to .net object. Has these added functionalities

Allows you to bind to a method (like GetData())

Can use a parameterized constructor

Can be populated async by a worker thread

Can be pop by an Instance object

<ObjectDataProvider x:Key="objDPStudents" IsAsynchronous="True"

ObjectType="{x:Type localClasses:TestObjectDataProvider}"

MethodName="GetStudents">

<ObjectDataProvider.ConstructorParameters>

<sys:Int32>10</sys:Int32>

</ObjectDataProvider.ConstructorParameters>

</ObjectDataProvider>

<ObjectDataProvider x:Key="SpouseTypes" ObjectType="{x:ObjectDataProvider sys:Enum}" MethodName="GetNames">

<ObjectDataProvider.MethodParameters>

<x:Type Type="{x:Type comm:SpouseType}"/>

**XMLDataProvider**. Binds to XML file using XPath. You can use **HierarchicalDataTemplate.**

<XmlDataProvider x:Key="xmlData" XPath="GameStats" Source="data\gamestats.xml">

<x:XData>

<GameStats xmlns="">

<GameStat Type="Beginner">

<HighScore>1203</HighScore>

</GameStat>

<GameStat Type="Beginner">

<HighScore>1203</HighScore>

</GameStat>

</GameStats>

</x:XData>

</XmlDataProvider>

<ComboBox ItemsSource="{Binding XPath=GameStat/@Type, Source={StaticResource xmlData}}>

Some XPath examples

XPath= **GameStat/@Type :** Gets all type attributes from each GameStat entry

**GameStat[2]/HighScore :** Gets the HighScore leaf from the 2nd Gamestat entry

**. :** Gets the current record

**\*** : Gets all records

**Hierarchical Data Templates**

<HierarchicalDataTemplate DataType="GameStats"

ItemsSource="{Binding XPath=\*}">

<TextBlock Text="All Game stats"/>

</HierarchicalDataTemplate>

<HierarchicalDataTemplate DataType="GameStat"

ItemsSource="{Binding XPath=\*}">

<TextBlock Text="{Binding XPath=@Type}"/>

</HierarchicalDataTemplate>

<DataTemplate DataType="HighScore">

<TextBlock Text="{Binding XPath=.}"/>

</DataTemplate>

<TreeView.ItemTemplate>

<HierarchicalDataTemplate ItemsSource="{Binding Modules}">

<HierarchicalDataTemplate.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal" Margin="2">

<TextBlock Text="{Binding ModuleName}"

FontSize="14" FontWeight="Bold"/>

<TextBlock Margin="5,0,0,0" FontSize="12"

Text="{Binding FileName, StringFormat=(\{0\})}"

VerticalAlignment="Center"/>

</StackPanel>

</DataTemplate>

</HierarchicalDataTemplate.ItemTemplate>

**Binding to ADO.NET objects.**

**Bind to DataSet:** adoGridDataSet.DataContext = aset;

<Grid x:Name="adoGridDataSet" Background="Gray" Margin="6">

<ListBox **ItemsSource="{Binding Path=Customers}"**

**DisplayMemberPath="ContactName"**/>

</Grid>

**Bind to DataTable:** adoGridDataSet.DataContext = aset.Customers;

<Grid x:Name="adoGridDataSet" Background="Gray" Margin="6">

<ListBox **ItemsSource="{Binding}"**

**DisplayMemberPath="ContactName"**/>

</Grid>

**Asynchronous** in Binding

There is a property IsASync but if your property is written right it shouldn’t need it. F you need async have a method called from the property that runs on an async task or background thread.

**Dispatching**..

UI in WPF(as is WinForms) is managed by a single thread. This thread processes windows messages (message pumping). This thread is contained within the Dispatcher object which is held within the Application class which is the root element in the VisualTree. The dispatcher contains a queue of work items for the thread to process and gives them to Thread using DIspatcher.Run().

Each control inherits from DispatcherObject which has a reference to a Dispatcher object.

When the Dispatcher is doing a lot of work the UI appears unresponsive. So we want to run long running processes on a separate thread and then push the result back onto the Dispatchers thread. We do this by using

Control.Dispatcher.BeginInvoke(new Action(() =>...));

This puts the delegate onto the UI thread where its run

PRiority enum can also be passed to Invoke method.

As Dispatcher doesn't have an interface you should create an adapter pattern for Dispatcher using an Interface so you can mock it in Testing

public interface IDispatcherService

{

void Dispatch(Action action);

}

**Increase WPF Performance**

**Use Virtualization panels** (**IsVirtualization**=True and **VirtualizationMode**=Recycled)

1. Limit Dynamic resources and try and add resources to Application or Page level.
2. Actually try not to use app or even page wide resources and keep the resources as close to using elements as needed.
3. Simplify your visual tree
4. Use Freezable Objects (stops having to add/remove from elements Changed event) Egs are Brush, Pen, Geometry.
5. Use Grid only when needed.
6. The more elements you have the more processing is needed in its Measure/Arrange pass throughs. When a new element is added, all the elements are measured/arranged again.
7. Use ObservableCollection as opposed to List when ItemSource binding
8. Try to not call ObservableCollection notifycollection changed after each update. Maybe every X updates
9. When binding to CLR object implement **ICustomTypeDescriptor.GetProperties()** or else WPF has to use reflection.
10. Fix binding errors so its not writing to trace
11. Try not to use relativeSource in ListboxItem templates because templates binding is created before listboxitem is added to tree. Define APs on LBI instead.
12. Hardcode sizes when you can
13. Use GAC( so it doesnt have to check the hash of Strong named assembly), Splash screen, NGen

**Some .NET/WPF Questions**

• WPF-what is WPF and its architecture and benefits

Its UI presentation software for Windows based applications.

Its architecture is 3 layers; Managed(PresentationCore and PresentationFramework) which connect with an UnmanagedLayer (MILCore) of which Visual is the bridge. The UnmanagedLayer connects to the System layer to access DirectX, Kernel, User32 (manages memory).

Benefits. It allows UI code to be separate from business logic.Uses DirectX as opposed to GDI and is faster and much richer. Databinding. Reusability with templates

• WPF vs Winforms

WPF has a higher learning curve and is not as mature. Also RAD apps are quicker in Winforms

• WPF-Why MVVM should be used?

Separates Presentation from Business layer (View from Model)

Easier to Unit test

• WPF-Control template & DataTemplate differences

A Control Template is rendered for its own sake. A Data Template is used to display underlying data. CT uses template binding DT uses standard databinding. DT uses a DataType CT uses a TargetType

• WPF-Event Bubbling and Tunneling

Bubbling goes up the logical tree and Tunneling goes down..

• WPF-Binding types

• WPF-Explain how WPF bindings works (With Interfaces names and its members details)

Data that will be bound needs to implement the **INotifyPropertyChanged (INotifyCollectionChanged** if coll**) OnPropertyChanged** method

• WPF-Explains about WPF commands and interfaces details

Class needs to implement **ICommand** interface **CanExecute**() and **Execute**()

• WPF-When do you use code behind in WPF applications? Any scenarios in your project?

Tie your vm up to the datacontext. Try not to use it but use if you dont need to test and its much easier than using the vm

• WPF-Explain about Visual Tree

Related to Logical tree but shows elements at their lowest level Controls are Textblocks, borders ContentPresenters

• WPF-How do you know a control in parent control in the parent-child relationship. explain with methods and details

* If a control’s content can be of any data type, how do controls know how to render their own content?
* How does WPF take raw data of any type and flow that through a control hierarchy and eventually convert that into something visible, tangible, and responsive to user interactions?
* What is the first class that is loaded by any new WPF Application project?

# Angular Features

**Components** :Encapsulates template, behavior and data of view. There is 1 root component always. Component is a class. It binds to the view.

**Directives:** Works with the DOM by adding behavior

**Routers**: Navigation

**Services**: Encapsulates non view functionality

# Node

Runtime environment for javascript outside a browser

# Presentation Patterns

**Types of state**. UI apps have 3 types of state that must be managed

**View State**: The state of the UI ie controls being enabled, font color…

**Session State**: The state of the data in memory after its taken from the data store

**Persisted State** The state of the data in the data store

**SOLID**

**S** : Single responsibility for each class

**0:** Class should be extensible but closed for modification ie encapsulation

**L** a subclass should be able to replace a base class without any alteration

**I**: Use specific interfaces as opposed to 1 general purpose interface

**D:** Use loose coupling

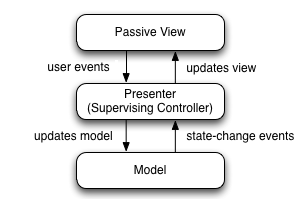
**DRY**  Don’t repeat yourself

**SoC Separation of concerns**. Have different layers in your app that overlap as little as possible ie ui, business , data layer. MVC is good impl of this

**MVP pattern:** What you see on the screen is the View, the data it displays is the model, and the Presenter hooks the two together. The view relies on a Presenter to populate it with model data, react to user input, provide input validation. The view doesn’t observe the model which is the main difference between this and the MVC.

**Benefits**: View State/Logic moved to presenter.

**Issues**: presenter is coupled with view



**Presentation Model (PM) Pattern** . The PM pattern is similar to MVP in that it separates a view from its behavior and state. However an abstraction of a view is created, called the Presentation Model which contains a view state and its behavior. A view, then, becomes merely a rendering of a Presentation Model.

**MVVM (**Identical to the PM pattern)

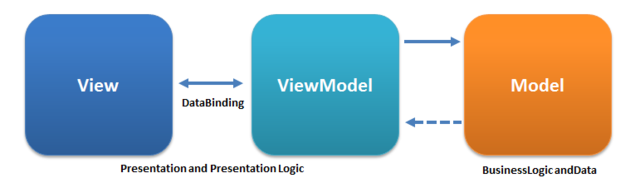
The viewmodel replaces the presenter in MVP and the Data binding replaces the coupling of presenter and view in MVP.

The VM is responsible for view state and logic and communicates with view through binding.

**Benefits:** Separates model from view

View separate from code that runs it

Testability

**Issues:** Overkill for small apps.  


**MVC**

The **View** gets its data from the **Model** and displays it. The **Controller** then takes data from **View** and gives it to **model**. **Model** is where the session state is. The **model** also updates the **view** ( - did this happen in DEN?)

**Benefits**: model is extracted out

**Issues:** View logic and view state are still tightly woven.

View is coupled to the model and controller



1. Explain about the data structures you used till now

2. C# - What are OOPS properties and concepts

**Inheritance** : Inherits base functionality. Promotes code reuse and reduces

Encapsulation : Hides away how data and functionality operates

Polymorphism : Different behaviors of the same signature. Subclasses have all the functionality of base classes

Abstraction : Can’t be created.. Must be inherited

3. C# - How polymorphism works

4. C# - Events, Delegates and Multi-cast Delegates

5. C# - Any design patterns used (Explained Singleton, Facade, Factory and Abstract Factory etc)

**Facade** : Simplify some other interface(s) by only exposing certain operations.

**Strategy**: Abstract changing behaviors into separate classes

**Template**: Abstract parts of an algorithm into separate classes

**Adapter**: Wrap an interface around another that doesn’t work fully on its own..

**Factory**: Make it the center of object creation

**Decorator** : Open to extension. Closed for moderation

**Command:** decouple the requester of an action from the object that performs the action.

**Iterator:** Allows a collection to be accessed sequentially while hiding the underlying functionality

6. C# - Explain about thread deadlocking and Singleton pattern in a thread-safe environment

7. C#-Explain about garbage collection

When object is created CLR calculates its size and see if that amt is avail on the heap. If so it adds the object to the heap and changes the end of heap pointer. Otherwise it calls a Garbage collection. All new objects since last GC are assigned level 0. All others have level incremented. GC then goes thru level 0 objs and removes dead ones. If more space needed it moves on to Level 1 ...

8. C#-How to you free the memory of the objects in class

Its non deterministic. All you can do is make sure there are no references to it.

9. C#-Value types vs Reference types

Struct is value type is made up of primitive types (char, int, bool) its contained in a single space in memory. Reference type is an object which is when created is contained in 2 locations in memory; the object and the pointer to the object. When you copy a value type you create a new value type.

10. C#-Stack vs Heap

Types are Value types, reference types and references. They are split into short and long lived. Any variable that lasts longer than a method is long lived. All long lived types go on the heap, shortlived to the stack. If the compiler is unsure it’ll presume their long lived. Thats why ref types always go on the heap as the compiler is unsure if the ref is dead..

# Sorting Algorithms

**Bubble Sort**: Swaps adjacent values if in wrong order making repeated pass throughs. - Polynomial

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best** | **Average Case** | **Worst Case** |
| **Time** | *O(n)* | *O(n2)* | *O(n2)* |
| **Space** | *O(1)* | *O(1)* | *O(1)* |

**Selection Sort:** starts at start of list and goes through unsorted part finding smallest element and placing that in‘sorted part’. Repeats (in unsorted part) until and end of list: Polynomial

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best** | **Average Case** | **Worst Case** |
| **Time** | *O(n)* | *O(n2)* | *O(n2)* |
| **Space** | *O(1)* | *O(1)* | *O(1)* |

**Insertion Sort:** Makes 1 pass left to right and sorts items as their encountered moving along the sorted index until done - Polynomial!

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best** | **Average Case** | **Worst Case** |
| **Time** | *O(n)* | *O(n2)* | *O(n2)* |
| **Space** | *O(1)* | *O(1)* | *O(1)* |

**Heap Sort:**  Create a max heap (O(n)). Then delete the values.(O(log n))

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best** | **Average Case** | **Worst Case** |
| **Time** | *O(n log n)* | *O(n log n)* | *O(n log n)* |
| **Space** | *O(1)* | *O(1)* | *O(1)* |

**Merge Sort:** Divide list into half over and over until we get to individual elements and then compare with adjacent element. Then get 2 pairs and sort them individually until only 1 list remains. Not good for almost sorted lists as you see from Best case time

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best** | **Average Case** | **Worst Case** |
| **Time** | *O(n log n)* | *O(n log n))* | *O(n log n)* |
| **Space** | *O(n)* | *O(n)* | *O(n)* |

**Quick Sort**: Create pivot. Cycle thru array and put everything lower than pivot to its left and everything greater to its right. Pivot is now sorted. Then repeat until is…

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best** | **Average Case** | **Worst Case** |
| **Time** | *O(n log n)* | *O(n log n))* | *O(n2)* |
| **Space** | *O(1)* | *O(1)* | *O(1)* |

**Binary Search**: needs to be sorted. Divides list in 2 and discards half where value isn't at. Repeats.: O(Log n)

**Linear search** : goes through each element in list 0(n)

# O notation

Complexity classes

• O(1) denotes constant running time

• O(log n) denotes logarithmic running time (BST Contains method)

• O(n) denotes linear running time (traversing a collection)

• O(n log n) denotes log-linear running time (merge sort)

• O(*n\*\*c*) denotes Quadratic running time. Polynomial (bubble sort) (each item in list of n will have n operations on it)

• O(c\*\*N) denotes Exponential time. (C is a constant raised to a power bases on list size)

def f(x):

for i in range(1000): #1000 - Constant

ans = i

for i in range(x): # 2x - Linear

ans += 1

for i in range(x): # 2x squared - Polynomial

for j in range(x):

ans += 1

#Complexity is 1000 + 2x + 2x2, if each line takes one step

def genExponentialComplexity(L):

res = []

if len(L) == 0:

return [[]]

smaller = genExponentialComplexity(L[:-1]) # go thru all but last element

extra = L[-1:] # get last element

new = []

for small in smaller:

new.append(small + extra)

return smaller + new

2°+ 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^N = (2^N+1) - 1 nodes.

(N-l) + (N-2) + (N - 3) + .. . + 2 + 1. The sum of 1 through N -1 is N(N-1)/2 = O(N^2)

int sum(Node node) {

if (node == nUll) {

return a;

}

return sum(node . left) + node. value + sum(node.right);

}

We said that the runtime of a recursive function(the above func) with multiple branches is typically O(branches^depth).

There are two branches at each call, so we're looking at 0 (2^depth ) .

# Data Structures

**Linked List:** Used everywhere. Other data structures (tree, stack, Queue) use them. Size is ‘infinite’.

**Doubly linked list** has nodes that reference the previous node.

**Circularly linked list** is simple linked list whose tail, the last node, references the head, the first node.

**LinkedListNode** class. Contains a **Value** and a **Next** (of ListNode) which points to the next entry. This is the data structure( you can also have a previous member which makes it a doubly linked list.

class LinkedListNode<T> {

public LinkedListNode(T value) {

this.Value = value;

}

public T Value { get; internal set; }

public LinkedListNode<T> Next { get; internal set; }

}

With the **LinkedList** class you have a **LinkListNode**<T> property for head and tail. Implement **ICollection.** The only tricky is area is the **Remove** method where you have to update the previous.Next to current.Next..

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds node to end of list | O(1) |
| Add **Without tail** | Adds node to end of list after traversing it | O(n) |
| Remove | Traverses list and remove node if found | O(n) |
| Contains | Traverses list and returns true if found | O(n) |

Cycle through a list

public bool Contains(T item) {

LinkedListNode<T> node = head;

while (node!=null) {

if (node.Value.Equals(item))

return true;

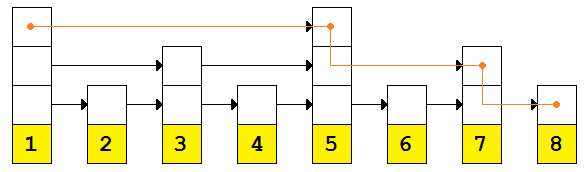
node = node.Next;

}

return false;

}

**SkipList**: Its an ordered list stored in a linked List structure which decreases traversing through list to O(log n) by having different layers in the list to traverse through. Set 100 of nodes to level 1 50% to lvl 2, 25% to lvl 3 etc…



Above we see that to get from 1 to 5 takes 1 comparison.

class SkipListNode<T> {

public SkipListNode(T value, int height) {

this.Value = value;

this.Next = new SkipListNodeT[height]

}

public T Value { get; internal set; }

**public SkipListNode<T>[] Next { get; internal set; } //array of links**

}

public bool Contains(T item) {

SkipListNode<T> cur = \_head;

for (int i = \_levels - 1; i >= 0; i--) {

while (cur.Next[i] != null) {

int cmp = cur.Next[i].Value.CompareTo(item);

if (cmp > 0) // The value is too large, so go down one level

break;

if (cmp == 0) // Found it!

return true;

cur = cur.Next[i];

}

}

return false;

}

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds node to end of list | O(log n) |
| Remove | Traverses list and remove node if found | O(log n) |
| Contains | Traverses list and returns true if found | O(log n) |

**ArrayList**: Again variations of it are used everywhere. Allows indexing in constant time. Uses an array so data is contiguous. However size is finite so need to grow and recreate it if we reach limit.

Linear array - doesn’t grow. Dynamic array - does grow.

class MyArray<T> : IList<T> {

T[] \_items;

public MyArray(int capacity) {

\_items = new T[capacity];

}

void GrowArray() {

int length = \_items.Length == 0 ? 16 : \_items.Length << 1;

T[] newArray = new T[length];

\_items.CopyTo(newArray, 0);

\_items = newArray;

}

public void Insert(int index, T item) {

if (Count == \_items.Length)

GrowArray();

**Array.Copy(\_items, index, \_items, index + 1, Count - index)**; //Shifts items following index slot.

\_items[index] = item;

Count++;

}

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds item to end of array | O(1) unless we need to Grow then O(n) |
| Insert | Adds item to array to at specific index | O(n) |
| Remove | Traverses array and remove item if found | O(n) |
| Contains | Traverses list and returns true if found | O(n) |
| Item | Gets or Sets value at specific index | O(1) |

**Stack** Use a linked list. Have the head as the only place for insertion and deletion.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Push | Adds item to top of stack | O(1) |
| Pop | Removes item from top of stack and returns it | O(1) |
| Peek | Returns item from top of stack without removing it | O(1) |

**Queue** Use a linked list. Have head for removal and tail for insertion

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Enqueue | Adds item to end of queue | O(1) |
| Dequeue | Removes item from start of queue and returns it | O(1) |
| Peek | Returns item from start of queue without removing it | O(1) |

**Set** Using a list will result in O(n), whereas using a tree is O(log n)

public class Set<T> : IEnumerable<T>

where T : IComparable<T>

{ }

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Union | Returns distinct items in 2 sets. [1, 2, 3, 4] union [3, 4, 5, 6] = [1, 2, 3, 4, 5, 6] | O(nm) for list |
| Intersection | Whats in both. [1, 2, 3, 4] intersect [3, 4, 5, 6] = [3, 4] | O(nm) for list |
| Difference | Whats in the first not second. [1, 2, 3, 4] difference[3, 4, 5, 6] = [1,2] | O(nm) for list |
| Symmetric Difference | Only exists in either [1, 2, 3, 4] symmetric difference [3, 4, 5, 6] = [1, 2, 5, 6] | O(nm) for list |

**Hash Table:**  Hash tables are a collection type that store key–value pairs in a manner that provides fast insertion, lookup, and removal operations.

Backed by an array. Note array should be prime number length to deal with collisions

**Hashing function** accept a key and return an output unique only to that specific key.

This is known as hashing, which is the concept that an input and an output have a one-to-one correspondence to map information.

Hash functions return a unique address in memory for that data

Needs to return the same result every time and also to evenly spread the keys throughout the array

If key is integer use key % array size. If key is string then…

**Collisions:** Will happen eventually even with a good hashing function. Some solutions:

Next open slot: find nearest empty slot. Tricky when removing the value.

LinkedListChain : each array slot is actually another data source like a linkedlist

Have a large hash table!

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add |  | O(1) unless we need to Grow then O(n ) |
| Insert |  | O(1) |
| Remove |  | O(1) |
| Item |  | O(1) |

**Binary Search Tree**

Values to the left of root node (and all subsequent parents) are lower, to the right are greater or equal. Each node can have 0,1 or 2 children.

Hardest thing is removing nodes. **Tree is not balanced.**

class BinaryTreeNode<T> : **IComparable<T>**

**where T : IComparable<T>** {

public BinaryTreeNode(T value) {

this.Value = value;

}

**public BinaryTreeNode<T> LeftChild { get; set; }**

**public BinaryTreeNode<T> RightChild { get; set;}**

public T Value { get; private set; }

public int CompareTo(T other) {

Return Value.CompareTo(other);

}

}

class BinaryTree<T> : IEnumerable<T>

w**here T : IComparable<T> {**

**BinaryTreeNode<T> \_head;**

Adding to Binary Tree involves traversing down 1 side of the tree till the location is found which is ideal for recursion.

public void Add(T value) {

BinaryTreeNode<T> newNode = new BinaryTreeNode<T>(value);

if (\_head == null) {

\_head = newNode;

Count++;

return;

}

AddTo(\_head, value);

}

This method traverses the tree..

private void AddTo(BinaryTreeNode<T> node, T value) {

if (node.CompareTo(value) > 0) {//new node is less

if (node.LeftChild == null) {

node.LeftChild = new BinaryTreeNode<T>(value);

Count++;

}

else

AddTo(node.LeftChild, value);

}

else if (node.CompareTo(value) < 0) { //new node is greater

if (node.RightChild == null) {

node.RightChild = new BinaryTreeNode<T>(value);

Count++;

}

else

AddTo(node.RightChild, value);

}

}

Values Removing a node is different depending on a number of things

1. **Node to be removed has no right child.** This is simple as its left child if exists just replaces the node to remove.
2. **Node to be removed has right child which has no left child.** This is same as 1 with its right child replacing node to remove.
3. **Node to be removed has right child which has left chil**d. Here the left most child of the removed nodes right child replaces node to remove.

This is because the left most node will always be less than all to the right of the node to be replaced and also more than those to the left of the node to be replaced.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds to correct location in tree | O(log n) avg O(n) worst case. |
| Remove | Removes first occurence from tree | O(log n) avg O(n) worst case. |
| Contains | Returns true if node exists in tree | O(log n) avg O(n) worst case. |

Traversing the tree.

Here you go through the tree performing an action on each node.

1. **PreOrder**: Starts at root, does all left side then right side
2. **PostOrder**: Does left bottom up, then right bottom up and finally node
3. **InOrder**: Does from lowest to highest value

public void OrderTraversal(Action<T> action) {

Some…OrderTraversal(action, \_**head**);

}

void PreOrderTraversal(Action<T> action, BinaryTreeNode<T> node) {

if (node != null) {

action(node.Value);

PreOrderTraversal(action, node.LeftChild);

PreOrderTraversal(action, node.RightChild);

}

}

void PostOrderTraversal(Action<T> action, BinaryTreeNode<T> node) {

if (node != null) {

PostOrderTraversal(action, node.LeftChild);

PostOrderTraversal(action, node.RightChild);

action(node.Value);

}

}

void InOrderTraversal(Action<T> action, BinaryTreeNode<T> node) {

if (node != null) {

InOrderTraversal(action, node.LeftChild);

action(node.Value);

InOrderTraversal(action, node.RightChild);

}

}

**AVL Tree:**

Like a Binary Search Tree except its balanced. Therefore Left and right nodes can’t differ by more than 1 level.

Rotation is used to balance. **Right, Left, Left-right** and **right-left** are the way to balance depending on which side was heavy and the level differences

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds to correct location in tree. Makes sure to balance. | O(log n) avg . |
| Remove | Removes first occurrence from tree | O(log n). |
| Contains | Returns true if node exists in tree | O(log n) all case. |

**Heap**

Returns the min or max in the collection. The child values are always less than the node. The tree will be complete(apart from the last level). No emphasis on ordering.

**Data is stored in an array.** The root node is always array[0].This provides a formula to get at a nodes relations.

To get at **parent** = (index-1)/2

To get **left child** = (2 \* index) +1; to get **right child** = (2\*index) + 2

**Add**. You add a node to the leaf level. Then you swap it with its parents until its at the correct level based on its value

**Remove**. Usually you remove the root. You replace the node with the last added node. Then you swap to move the new node down to its correct position in the heap. You always choose the largest child when you could choose either.

public void Add(T value) {

if (\_count >= \_items.Length) {

GrowBackingArray();

}

\_items[\_count] = value;

int index = \_count;

while (index > 0 && \_items[index].CompareTo(\_items[Parent(index)]) > 0)

{ //while not the root and item is > than what comparing to.

Swap(index, Parent(index));

index = Parent(index);

}

\_count++;

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds to heap. | O(log n) avg . |
| RemoveMax | Removes root | O(log n). |
| Contains | Returns true if node exists in heap | O(log n) |

}

**B Tree:** Its a sorted, balanced tree structure.

Non root node has at minimum T-1 and maximum 2xT-1 values. T is known as the Degree or Order or Factor.

So for a factor of 50 a child node will have at least 50 nodes. If root has 50 then there can be 2500 ( 50x50) records of which we would only to have to search at the worst through 100 records.

**Balancing Operations** In order to keep tree balanced nodes are moved around when other nodes are added/removed.

**Push Down:** Merges two adjacent children.

 Each of those children have T - 1 values.

 The parent value is added as the median value.

 The resulting node has 2 × T - 1 values.

**Rotate value:** Move value from right node up to root, while moving other root node down to left child node.

This can also happen in opposite direction.

**Split Nodes:** Opposite of push down

Move median value up to root and split node that median value was in.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Performance** |
| Add | Adds to tree. | O(log n) . |
| Remove | Removes from tree | O(log n). |
| Contains | Returns true if node exists in tree | O(log n) |

Searching (node, valuetofind)

**Adding Node:**

**Searching node**

Foreach value in node

If valuetofind = value then return true

If valuetofind < value

Search(<leftchild>, valuetofind)

End

Return search(<lastchild>, valuetofind) // this is to the right of the rightmost root node.

# BitWise Operations

Performed by the processor. Faster than calculating

**Not.** Bits are flipped. Not x = -x -1

NOT 0111 (decimal 7)

= 1000 (decimal 8)

**And** : Values are set to 0 unless their both 1.

0101 (decimal 5)

AND 0011 (decimal 3)

= 0001 (decimal 1)

**Or** : Values are set to 1 unless their both 0.

0010 (decimal 2)

OR 1000 (decimal 8)

= 1010 (decimal 10)

**XOR**: Set to 1 if either are 1 else 0

0101 (decimal 5)

XOR 0011 (decimal 3)

= 0110 (decimal 6)

**Most Significant bit:** digit on left of number

**Least Significant bit:** digit on right of number

**Bit Shifting:** Shift the chars in 1 direction losing the first char but gaining a new 0.

**Left shift <<** adds a zero to the right. Left Shift doubles the number

0110 << 1 = 1100 ie 6 become 12

**Right shift:**There are two types

1. **Logical >>** adds a 0 to the left (most significant bit. Right shift halves the number (with truncation if its odd)

0111 >> 1 = 0011 ie 7 become 3

If negative(Bold 1 signifies signed bit) **1**011 >> 1 = 0010 ie -5 becomes 2.

1. Arithmetic >>> Same as above for positive

For negative keeps the digit in left(msb)

Eg **1**011 >>> 1 = **1**101 ie -5 becomes -3

**A = 0, bit = new value you want to add to A.** [**https://www.youtube.com/watch?v=NLKQEOgBAnw&t=21s**](https://www.youtube.com/watch?v=NLKQEOgBAnw&t=21s)

**Set bit :** A |= 1 << bit

**Clear bit :** A &= ~(1 << bit)

**Test bit :** (A & 1 << bit) != 0

**Arithmetic**

|  |  |  |  |
| --- | --- | --- | --- |
| **Subtracting** | **Adding** | **Multiplying** |  |
| **0-0 = 0**  **1-0 = 1**   * 1. **= 1 \*1 carries**   **1-1 = 0** | **0+0 = 0**  **0+1 = 1**  **1+0 = 1**  **1+1 = 10**  **1+1+1 = 11** | **0 \* 0 = 0**  **1 \* 0 = 0**  **0 \* 1 = 0**  **1\*1 = 1**  **~~1\*1 \* 1 = 11~~** |  |

**Twos Complement**

Invert the bits in the positive representation (010 -> 101) and add 1 (-> 110). Then prepend 1(all negative numbers start with 1) to number (-> 1110)

So 2 [010] becomes -2 [1110]