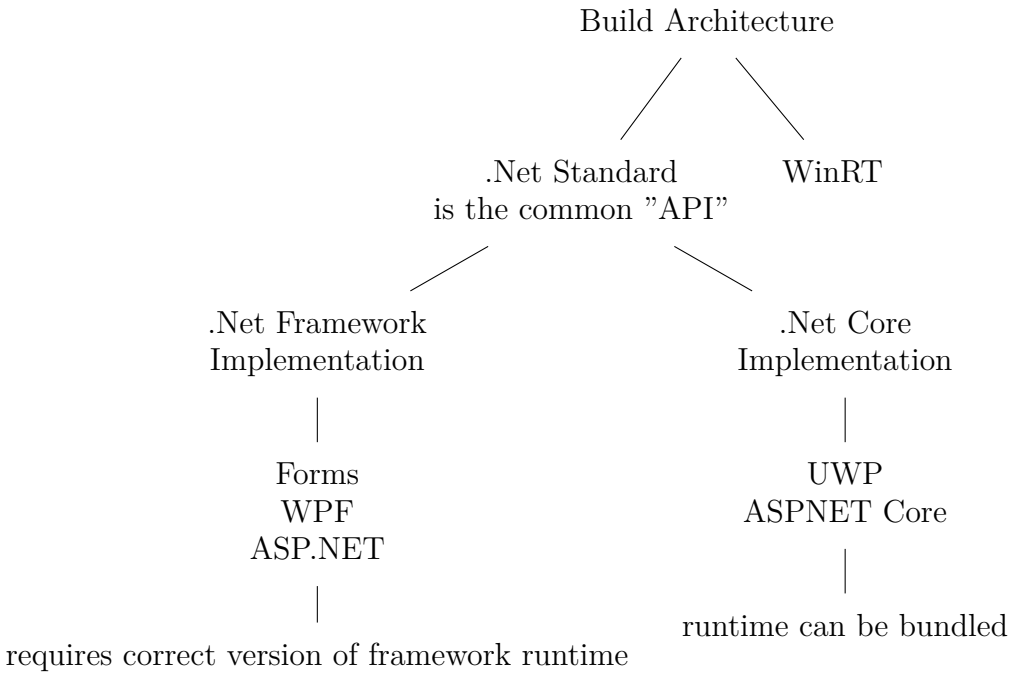
C# Notes

W T

19 May 2021
modified: February 13, 2025

Contents

1 Binary Structure



2 Classes and Structs

Classes are **reference types**, memory is allocated on the **heap**.
Structs are **value types**, memory is allocated on the **stack**.

2.1 Properties

auto-generated with get; set;
use `=>` for expression bodied getter or setter

2.2 Methods

use `=>` for expression bodied method
`fn(argname: value...` named argument function call
`void fn(argname = <default>...` optional argument with default
`void fn(..., params type[] argsname)` variable parameters, .Length property gives length

2.3 Constructors

Default ctor is available if none is declared
`Ctor():this(a,b,c)` to call another ctor (executed before the body)
static ctors are available

Classes can be static, but structs can't.
A struct in a class is allocated inline with the class
When declaring a struct, calling `new Struct()` is optional. However, the object must be fully initialised before it is first used.
Structs are passed by value by default, but the argument modifier `ref` can override this behaviour to pass by reference.
Structs can be declared readonly, which means they must be initialised at construction.
Structs don't support inheritance, but they can implement interfaces, and can contain other structs

`base` can be used in a class to refer to the base object. It can't be used in a struct.

`ref struct` is forced to live on the stack, so can't be part of a class, and can't be captured by lambdas, local fn, or async functions

2.4 Member Modifiers

Visibility	Same Assembly	Other Assembly
`public`	everywhere	everywhere
`protected`	derived only	derived only
`private`	same class only	no access
`internal`	everywhere	no access
`protected internal`	everywhere	derived only
`private protected`	derived only	no access

modifier	target
`new`	methods and optionally fields
`static`	methods, classes, fields
`virtual`	methods
`abstract`	methods, classes
`override`	methods
`sealed`	methods, classes
`extern`	methods (with the [DllImport]attribute)

2.5 Base classes

`Object` Common base class with ToString, GetHashCode, Equals, Finalize (typically implement the `~`dtor), GetType and memberwise Clone methods
`ValueType` Common parent of all value types. It's parent is `Object`.

3 Functions and Methods

`ref` pass by reference
`out` output parameter, introduced the argument name at this point
`in` input parameter, must be initialised before call.
Extension methods are declared as static methods within a static class and the first parameter is `this ClassName name`

4 Interfaces

Properties are supported (get; set;). Fields are not.
Default implementations can be provided since C#8.
Static members are supported since C#8 and require an implementation in the interface. But since C#10 they can be declared `static abstract` which is then implemented on the class.
Covariant (more derived) return types can be returned by the implementating class since C#9.

5 Types and Enums

`Nullable<T>` adds the HasValue and Value properties.
Boxing. Value types (primitives and structs are boxed when assigned to an object or suitable interface type. This creates a copy of the value
`enum MyEnum: <type>` if `<type>` is missing, then defaults to `int`
`[Flags] enum MyEnum` uses a bit field, and members can be defined as unions of other members using the `|` and & operator
`Enum` Helper class with TryParse, GetNames, GetValues, IsDefined, GetUnderlying(type) methods

6 Pattern Matching

`if (x is int d {...}` match type and assign to d
`x = y switch {...}` switch expression

`x = obj switch { {prop:  == ...} => ...` switch expression with obj.prop
`x = obj switch { var (a, b,...)  when a==b => ...` deconstructing switch
Since C#9, supports recursive properties.

7 Tuples and Records

...todo...

8 Operators

- `is` type check, returns bool
- `as` type conversion, returns an object or null
- `sizeof` unsafe operator, for ValueTypes only
- `typeof` returns a `Type` from an expression
- `?.` null or member access
- `??` null coalesce, returns the first non-null value
- `?[]` null or indexed access

8.1 Equality

If you need to provide special functionality to support equality, you should overload `operator==`, `operator!=` and override `Equals()`.

- `static bool operator==(MyClass l, MyClass r){...} and !=` should be overloaded to provide special implementation
- `override bool Equals(object r) {...}` should perform runtime type check and then transfer to operator==

By default, **equality for class is by reference**, and **for structs it is by value**.

There is a non overrideable method called `ReferenceEqual` which will always test object references for equality

8.2 Overloadable Operators

- Unary

  `+`
  `-`
  `!`
  `~`
  `++`
  `--`
  `true`
  `false`

- Binary

  `+`
  `-`
  `*`
  `/`
  `%`
  `&`
  `|`
  `^`
  `<<`
  `>>`

- Comparison

  `==`
  `!=`
  `<`
  `>`
  `<=`
  `>=`

8.3 Index operator

`public ElemType this[IndexType]` {`get; set;`}

8.4 Conversion operators

`public static implicit|explicit operator TargetClass(SourceClass)` this is static, so it can be implemented on a 3rd class, but the suitable host might depend on accessibility or logical considerations.
Conversion can only be performed between unrelated classes not in the same inheritance path.

9 Arrays

Initialisation can be done with or without the `new MyArray[int]()` constructor and with or without the initialiser list {}. If the initialiser list is not provided then default values are assigned (typically 0 for numerics and null for references).

`[,,..]` multidimensional
`[][]...` jagged

`Clone()` and `Array.Copy()` do shallow copies of the elements (i.e. the references not the objects)

Covariance is supported in arrays, i.e. you can store more derived objects in the array than it is declared as.

`Array.Sort()` defaults to Quicksort, but overloads are provided for primitive numeric types.

10 Generics

10.1 Classes and Structs

`public class MyGeneric<T,U> where T:...constraint...` where the constraint can be one of:

- `class`
- `struct` which would include primitive types like int
- class or interface name
- U secondary type parameter in the declaration
- new() the instantiating class must have a parameterless ctor

Generic structs are supported

10.2 Generic Interfaces

`public interface MyInterface<in A, out B>` The implementing class can implement using base class of A, and subclass of B:

- covariance B can be returned as a more derived class
- contravariance A can be accepted as a less derived class

10.3 Static members

There is one instantiation of static members for each instantion of the generic.
`public class Partial<T>:Base<int, T>` partial instantiation, not all of the types are specified.

10.4 Generic Methods

Similar use of `where` to specify constraints.
Type inference is supported so explicit declaration of type parameters isn't usually needed.
Generic methods can be overloaded by providing an implementation for a full or partial instantiation

11 Collections

11.1 IEnumerator and IEnumerable

A collection implements `IEnumerable` and `IEnumerable<out T>` which has a method `IEnumerator(<T>) GetEnumerator()`. The IEnumerator returned has methods `MoveNext()` and `Reset()` and the `Current` property. Idiomatic use of the LINQ functions and the `foreach` loop means that these methods are not often invoked directly.

11.2 Iterator Blocks

`IEnumerable`s can created from factory methods that are implemented using `yield` statements to return within a loop.

11.3 .NET Collections

11.3.1 `System.Collections` - Non-generic Collections

- `ArrayList`
- `Hashtable`
- `Queue`
- `Stack`
- `SortedList`

11.3.2 `System.Collections.Generic` - Generic Collections

- `List<T>`
- `Dictionary<Key,Value>`
- `Queue<T>`
- `Stack<T>`
- `SortedList<Key,Value>`
- `HashSet<T>`
- `LinkedList<T>`
- `SortedDictionary<Key,Value>`

`SortedList` is array based, while `SortedDictionary` is tree based.

11.3.3 `System.Collections.Concurrent` - Thread-safe Collections

- `ConcurrentDictionary<Key, Value>`
- `ConcurrentQueue<T>`
- `ConcurrentStack<T>`
- `ConcurrentBag<T>`
- `BlockingCollection<T>`

N.B the `BlockingCollection<T>` is designed to control the flow of the data in terms of size of the collection rather than coordinating access to it. The other concurrent collections will use thread level locking and lock free mechnisms.

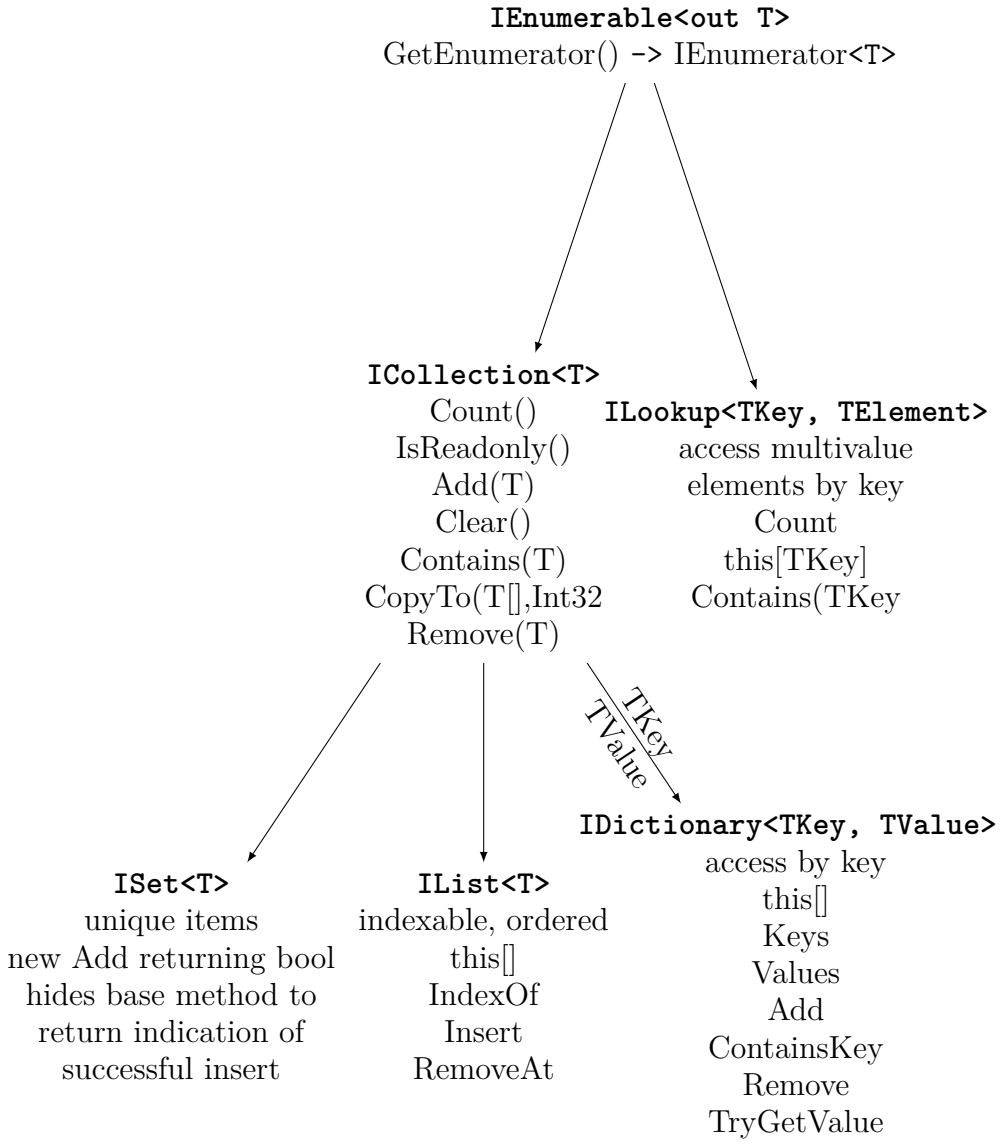
11.3.4 `System.Collections.Immutable` - Immutable Collections

`ImmutableArray<T>`
`ImmutableList<T>`
Can be instantated explicityly, or built using `ToImmutable...` extension method or the `Immutable...<T>.Builder` class.

11.3.5 Wrapper Classes

- `ReadOnlyCollection<T>`
  - Provides a read-only view of a list, ensuring that the underlying collection cannot be modified through the wrapper.
- `ObservableCollection<T>`
  - A dynamic data collection that provides notifications when items are added, removed, or changed. It is often used in data binding scenarios, particularly in WPF (Windows Presentation Foundation).
- `KeyedCollection<TKey, TItem>`
  - A collection that contains elements with keys, allowing you to look up elements by their key. It combines features of both lists and dictionaries.
- `SynchronizedCollection<T>`
  - A thread-safe collection that provides synchronized access to the underlying collection, ensuring that it can be safely accessed by multiple threads.
- `Collection<T>`
  - A base class for creating custom collections. It provides a wrapper around a ‘List¡T¿‘ and allows you to customize the behavior of the collection.
- `ReadOnlyDictionary<TKey, TValue>`
  - Provides a read-only wrapper around a dictionary, ensuring that the underlying dictionary cannot be modified through the wrapper.
- `NameValueCollection`
  - A specialized collection of associated String keys and String values that can be accessed with either the key or the index.

11.4 Base Interfaces



11.4.1 IEnumerable

`IEnumerable` simply promises that it is possible to traverse a sequence of elements and is the basis of `foreach` loops. Its only method is `GetEnumerator`. LINQ methods are built on top of the IEnumerable interface using the Enumerable extension class.

11.4.2 ICollection implementations

An `ICollection` extends the `IEnumerable` by adding the ability to manipulate the contents and do basic element insertion and removal. No ability to retrieve the actual element is supported, only adding one and removing one.
Behaviours:

- `Add` fails if the underlying collection is readonly, otherwise doesn't check for duplicates
- `Remove` returns bools to indicate if anything was removed
- `Count`

- `Clear`
- `CopyTo` copy into a contiguous array

Implementations:

- `LinkedList` doesn't implement the IList interface, because indexed operations can't be done efficiently with the linked list structure
- `Stack` implemented as an dynamic array
- `Queue` implemented as a **circular** dynamic array.

11.4.3 IList implementations

`IList` extends on the `ICollection` by adding the notion of a stable order, so when items are added they keep their position in the collection unless it is changed by the program. This means `IList` supports indexed operations, and typically this is efficiently done used array based contiguous memory. Also inserts can now be done at arbitrary locations. It won't sort the elements automatically, so this means it needs to be done explicitly. Once it is sorted, `BinarySearch` can be used to achieve $O(\log N)$ searches, via the same comparer. Find is also supported through a custom predicate but generally it will employ a linear search.
Behaviours:

- `Item[int]` retrieve element based on its position within the order
- `IndexOf` return the position of an item
- `Insert` insert an item at a specified position
- `RemoveAt` remove an item at a specified position

Implementations:

- `List` is based on a dynamic array, so indexed operations are very efficient. The non-generic equivalent is `ArrayList`.

11.4.4 IDictionary Implementations

IDictionary implementations use collections of `KeyValuePair<TKey,TValue>` elements, and hence inherit the `ICollection and IEnumerable interfaces via the KVP class`.

Behaviours:

- `Item[]` If the item doesn't exist, when retrieving it will throw and error, but when setting it will add a new key and set the value.
- `TryGetValue` is used to check whether item exists before getting it
- `Add` Will throw an error if the item doesn't exist
- `Remove` returns a boolean to indicate whether item was successfully found and removed

Implementations:

- `Dictionary` a hash based collection of KVPs
- `SortedDictionary` a tree based collection of KVPs

- **SortedList** The keys and values are stored separatedly in individual arrays sorted by the value of the key, and $O(\log N)$ search is achieved by a binary search. Retrieval of the Keys and Values properties are lightweight.

11.4.5 IList/IDictionary Blend

OrderedDictionary implements both the IList and IDictionary interfaces. It stores the data as a hash stucture of KeyValuePairs, but separately it maintains a list(array) of Keys to track the order of insertion. **NamedValueCollection** and **HydridDictionary** in the Specialized collections are two more examples

11.4.6 ISet Implementations

ISet extends **ICollecion** by adding the behaviour similar to mathematical sets, so the elements are forced to be unique, and set operations are supported.
Implementations:

- **HashSet** based on a hashed stucture
- **SortedSet** based on a red-black tree

Behaviours:

- **Add** hides the base function and enforces uniqueness, returning a bool to indicate success
- various set operations such as **UnionWIth**, **IntersectWith**, **ExceptWith**, **IsProperSubsetOf**

11.5 Helper Interfaces

Item classes should implement `IComparable<in T>` and `IEquatable<T>`, if used in a collection that needs them.
When implementing `IComparable<T>.CompareTo(T)`, the operators `<,>,<=,>=` should be overloaded to use the implemented function.
When implementing `IEquatable<T>.Equals(T)`, Object.Equals() should be overridden and operators `==,!=` should be overloaded to use the implemented function.
Some collection methods allow a custom comparer or custom equality comparer to be passed in. In that case:
IComparer implementation is needed to customise sorting
IEqualityComparer implementation is needed to support keys and hashes

11.5.1 Special Collections

BitArray
BitVector32

12 LINQ and Collections Extension Methods

LINQ keyword	Extension Method	
`from`		introduces a cursor variable and collectio
		name, as in `from x in c`
`where`	`Where`	provides a predicate
`select`	`Select`	projects an expression `select new {...`
`orderby`	`OrderBy, ThenBy`	
`orderby descending`	`OrderByDescending`, `ThenBy...`	
`group by`	`GroupBy`	
`join`	`Join`	as in `from a in x join b`
		`in y on a.p equals b.q`
`into`		introduces continuation variable to hold
`let`		introduces variable derived from the curr
`select many`	`SelectMany`	
`distinct`	`Distinct`	
`take`	`Take`	
`skip`	`Skip`	
`first`	`First`	
`single`	`Single, SingleOrDefault`	throws exception if ¿1

12.1 `Span<T>`

...todo...

13 `async` and `await`

...todo...

14 Events and Delegates

`public delegate RetType MyDelegate(ArgType arg,...)` declares a new delegate type.
`MyDelegate del = new MyDelegate(method);` or just `MyDelegate del = method;` creates an instance.
`MyDelegate del = new MyDelegate(delegate (...) {});` or just `MyDelegate del = delegate (...) {});` creates an empty instance.
`del += method2` adds an instance, `del -= ...` to remove.
`del()` delegates are called like functions. An exception in a delegate will cause all calls to terminate

`delegate (args...) {}` anonymous method
`(args)=>{....}` lambda
`(args)=>expression` lambda (expression valued)

14.1 Events

Events are encapsulated inside a publisher class and can only be raised from within the class. This is the key difference from a delegate.
`public event MyDelegate MyEvent;` declares an event

MyEvent+=... to add a handler
MyEvent?.Invoke(args) to call

14.2 Action<T1,... and Func<T1,...out Tret>

Generic delegates save you from having to declare delegate types explicitly. **Action** returns a void, while **Func** returns a type.

15 Miscellaneous

@"string" literal string, saves escaping characters
$"string{*expression*}" string interpolation
using a = <name> define namespace alias
a::<type name> reference namespace alias
extern alias MyAlias reference external assembly, even alternate versions of assembly
#define/undef
#if/elif/else/endif
#region/endregion
#line
#pragma