C# Boot Camp

# History

* Anders Hejlsberg is known as the creator and is still the lead architect of C# today
* Version 1.0 was released in 2002
* As of February 2020, C# 8.0 is the latest released
* C# is largely based on C++ and Java and was designed to be a modern, object-oriented, type-safe, performant language
* Original name: COOL – C-like Object Oriented Language

## Version History

<https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history>

# Types

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/>

C# is a *strongly-typed* language. Each variable and constant has a type, as does each expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value.  Typing rules are enforced at compile-time and most typing errors are discovered during compilation rather than at run-time.

Every type ultimately derives from System.Object (Unified Type System).

## Value Types

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types>

<https://docs.microsoft.com/en-us/archive/blogs/ericlippert/the-truth-about-value-types>

* Derive from System.ValueType
* Structs and enums
* Memory allocated inline in declaring context (can be on the stack).

### Structs

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/struct>

* No inheritance
* Interface implementation is allowed.
* May be declared **readonly** (C# 7.2+)
  + Compiler will ensure all public members and “this” are **readonly**
  + Implications: <https://blogs.msdn.microsoft.com/seteplia/2018/03/07/the-in-modifier-and-the-readonly-structs-in-c/>

### Enums

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/enum>

* Set of named integer constants
* Inherit from **System.Enum**
* No inheritance or interface implementation allowed
* By default, the integral type used is int, but another integral type may be specified if desired
* Enums may be used as bit flags by decorating the enum with the **System.Flags** attribute
  + Named combinations are acceptable, so you must make your discreet flags powers of 2 manually

### Boxing and Unboxing

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing>

Boxing and unboxing refer to the action of converting a value type to a reference type and vice versa.

* Boxing: value type -> reference type
  + Implicit - assign a value type to a reference variable (object or implemented interface)
  + Value wrapped in System.Object and stored on the managed heap
* Unboxing
  + Explicit (using a cast)
  + Reference type on heap is checked to ensure it is a boxed value of the given type then the value is copied into the value type variable.
* Both are computationally expensive (boxing more so) and should be avoided if possible (especially in bulk)
* Value types are not boxed/unboxed when passed by reference using the ref keyword.

## Reference Types

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types>

* Classes, delegates, arrays, interfaces
* Null until explicitly assigned to
* Memory allocated on the heap
* Variable stores only a reference to the object’s location

### Classes

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes>

* Code template of an object that defines members (fields, properties, methods, delegates, events, etc.)
* Instantiated using **new** keyword
* Fully support inheritance
  + Single direct inheritance
* May implement multiple interfaces

### Abstract Classes

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract>

* Only a base class – may not be instantiated
* May or may not contain implementation
* Mark members abstract (no implementation) that required derived types to implement
* Inheritors override abstract members using **override** keyword

### Interfaces

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface>

* define the properties, methods, events, and indexers implementing classes must implement
* Implementation not supplied (ish – C# 8.0 allows default interface implementations so implementing classes wont’ break if additions are made to an interface)
* Convention I<InterfaceName>

#### Explicit Implementation

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation>

* Interface members may be explicitly implemented for that interface by prefixing the member with the interface name: <InterfaceName>.<MemberName>
* Explicitly implemented members are only available through the interface (casting to the interface type, etc.).
* Useful in certain scenarios:
  + A class implements multiple interfaces with the same member name that have different functionality
  + A class implements multiple interfaces that have conflicting member names (ex: property and method with same name)
  + A class implements an interface that has members that don’t pertain to it and don’t make sense to expose directly (ex: System.Array : ICollection<T>.Add())

## Delegates

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/delegate>

* Reference type that references a method (static or instance as well as anonymous)
* Definition looks like a method prototype with delegate keyword – defines signature of compatible methods (parameters and return type must be the same).
* Set method with assignment operator, constructor parameter, or += operator
* Remove method with -= operator

### Multicast Delegates

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/how-to-combine-delegates-multicast-delegates>

* Delegates of the same type can be combined into multicast delegates using the + operator
* Multicast delegates call all methods in their invocation list in order and return the returned value of the last invoked method
* All delegates have multicast capabilities

### Actions

[https://docs.microsoft.com/en-us/dotnet/api/system.action-1](https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=netcore-2.2)

* The introduction of generics in C# has largely removed the need to declare custom delegate types thanks to actions and functions
* C# declares generic actions with up to 16 type parameters : Action<T1, T2, …, T16>
* Derives from Delegate
* Return type is void

### Functions

[https://docs.microsoft.com/en-us/dotnet/api/system.func-2](https://docs.microsoft.com/en-us/dotnet/api/system.func-2?view=netcore-2.2)

* Same as actions but in addition to 16 (up to) parameter types, functions also specify the return type
* Return type specified last: Func<T1, T2, …, T16, TReturn>

## Events

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/index>

* Based on EventHandler delegate and EventArgs base class
  + Delegate signature typically includes the sender object and event arguments
  + Event arguments may contain any data that may be useful to event subscribers
* Use **event** keyword and handler delegate to define event
* Publisher raises event
* One or more subscribers process the event
* Subscription only
  + You can’t null them out (unsubscribe other listeners), etc. (can’t sabotage other subscribers).

## Built-in Types

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/built-in-types-table>

C# has a number of keywords reserved as aliases for predefined types. All of the types (except string and object) are value types. Many of these types have built in methods for helpful functionality like parsing as well as constant values such as maximum, minimum, and other useful values.

* bool (System.Boolean)
  + stores a single boolean value (**true** or **false**)
* byte (System.Byte)
  + 8-bit unsigned integer
* sbyte (System.SByte)
  + 8-bit signed integer
* char (System.Char)
  + UTF-16 character
* decimal (System.Decimal)
  + decimal (high precision) floating point number. Useful when it’s important to avoid rounding errors.
* double (System.Double)
  + Double-precision 64-bit floating point number
* float (System.Single)
  + Single precision 32-bit floating point number
* int (System.Int32)
  + 32-bit signed integer
* uint (System.UInt32)
  + 32-bit unsigned integer
* long (System.Int64)
  + 64-bit signed integer
* ulong (System.Uint64)
  + 64-bit unsigned integer
* short (System.Int16)
  + 16-bit signed integer
* ushort (System.UInt16)
  + 16-bit unsigned integer
* object (System.Object)
  + base class of everything!
* string (System.String)
  + Text sequence of UTF-16 characters
  + Implements **IEnumerable<Char>**
  + Immutable – use **System.Text.StringBuilder** if multiple modifications required
  + May be **interned**, see <https://docs.microsoft.com/en-us/dotnet/api/system.string.isinterned?view=netframework-4.7.2#System_String_IsInterned_System_String_>

## Null

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/null>

* Keyword literal representing a null reference (a reference that does not refer to any object)
* Default value for all reference types
* Many helpful and concise ways to check for null (see **Operators**)

## Operators

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/>

* **= operator** (assignment operator) – assigns value of right-hand operand to left-hand operand
* **== operator** (equality operator)
  + For built in value types, returns true if values are equal
    - Except in the case of floating-point numbers when either value is **NaN** (not a number). In this case the equality operator always returns false.
  + Two operands of the same enum type are equal if the values of their underlying integral types are equal
  + By default, not defined for user-defined structs
    - Can define by overloading == operator
  + By default, tests for reference equality on Object and derived reference types
  + String – compares value equality
* **typeof** – returns the Type object representing the specified type
* **default** – returns the default value of the provided type
* **(T)x** – type cast
* **is** – type compatibility. Returns true if left operand can be cast to type specified in right operand.
  + Can be used with pattern matching (C# 7.0+)
    - **Type pattern** – tests whether an expression can be cast to a specified type and performs the cast if possible
    - **Constant pattern** – tests whether an expression evaluates to a specified constant value
    - **var pattern** – a match that always succeeds and binds the value of an expression to a new local variable
* **as** – type conversion. Returns the left operand cast to the type specified by the right operand if the cast is possible and null if it is not possible.
* **checked** – explicitly enables overflow checking for integral-type arithmetic operations and conversions
  + default behavior is **unchecked**
* **? operator** (conditional operator) – t ? x:y – if t is true, return x, otherwise, return y
* **?? operator** (null-coalescing operator) – x ?? y – returns x if it is non-null and y otherwise
* **?./?[] operator** (null-conditional operator) – for member access or indexers. Returns null if the operand to the left of the ? is null; otherwise, accesses the member or indexer.
  + Thread-safe way to invoke delegates (compiler generates code to use a temporary variable to capture target before verifying null/calling) without adding a lot of distracting code to your code base.
* **Many others** – binary, unary, comparison, logical, indexing, shift, and assignment operators, some of which may be overloaded

### Operator Overloading

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/overloadable-operators>

* User-defined types may overload some operators by defining static member functions using the **operator** keyword.
* Overloading arithmetic operators need not take the same “other” operand. Example: you can add a TimeSpan to a DateTime and get a DateTime back that has the TimeSpan’s duration added to it.

## Equality

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/equality-comparisons>

When comparing two values for equality in C#, there are two types of equality that may be evaluated: **reference equality** and **value equality**.

### Reference Equality (Identity)

* Two object references refer to the same underlying object in memory.
* Testing for reference equality is provided by the **Object.ReferenceEquals(Object, Object)** method
  + Should not be used for value types - always returns false
  + Care should be taken when used with interned/non-interned strings
* Reference equality implies **value equality**

### Value Equality (Equivalence)

* Two objects contain the same value or values (for complex types)
* Should generally be checked using the **Equals** method
  + System.Object (and therefore, all custom classes) check for reference equality in the Equals method by default. The Equals method is virtual and may be overridden.
* May be checked using the == operator for value types and certain immutable reference types (ex: string)

#### Defining Value Equality for Custom Types

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/how-to-define-value-equality-for-a-type>

<https://docs.microsoft.com/en-us/previous-versions/ms173147(v=vs.90)>

* Custom type implementors are free to decide whether to implement custom value equality logic for their type.
* Default implementations
  + **Value Types**
    - System.ValueType overrides Object.Equals(Object) using **reflection** to examine all fields and properties to determine equality. This is slow and should be overridden in derived types.
    - The == and != operators are not defined by default on structs
  + **Reference Types**
    - The Equals method on System.Object tests for reference equality by default
    - The operator == tests for reference equality in reference types by default.
      * Overriding this behavior to test for value equality instead can be useful, but is not recommended for non-immutable types.
* Five guarantees of equivalence should be met in any custom implementation:
  1. **Reflexive property** – x.Equals(x) returns true
  2. **Symmetric property** – x.Equals(y) returns the same value as y.Equals(x)
  3. **Transitive property** – if (x.Equals(y) && y.Equals(z)) returns true, then x.Equals(z) returns true
  4. Successive invocations of x.Equals(y) return the same value as long as the x and y are not modified
  5. x.Equals(null) returns false, but null.Equals(null) throws an exception (only violation of symmectric property)
* Steps to implement equality
  1. Override virtual **Object.Equals(Object)**. In most cases, just call Equals(T) see #2
  2. Implement **IEquatable<T>** and provide its **Equals(T)** method.
     + Should not throw exceptions
     + Typically, should compare member data for value equality
     + When defining Equals on derived classes, it is often best to just check new fields and call the base class’s Equals method to cover the base fields.
  3. Optional but recommended: overload **== and != operators**
  4. Override **Object.GetHashCode** so that equivalent objects produce the same hash code
  5. Optional: implement **IComparable<T>** and overload **>= and <= operators**
* Any new overriding implementation of the Equals method should not throw exceptions and it is recommended that classes that override Equals(Object) should also override **GetHashCode**, and Equals(Type)
  + New **HashCode.Combine** method can be helpful for simple types:
  + <https://docs.microsoft.com/en-us/dotnet/api/system.hashcode.combine?view=netcore-2.2>

# Access Modifiers

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/access-modifiers>

* Access modifiers are used to specify the declared accessibility of types and members
* The following keywords may be used in different contexts and combinations to produce the desired accessibility:
  + **public** – no restrictions
  + **protected** – containing class/derived classes
  + **internal** – containing assembly (project)
  + **private** – containing class

## Accessibility Levels

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels>

* The below accessibility levels may be used:
  + **public** – not restricted
  + **protected** – limited to containing class or types derived from the containing class
  + **internal** – limited to current assembly
  + **protected internal** – limited to current assembly or types derived from the containing class
  + **private** – limited containing class
  + **private protected** – limited to the containing class or types derived from the containing class within the current assembly (C# 7.2+)
* When inheriting from another class, the inherited class may only have the same or more restrictive level of access.
* Overriding members must have consistent accessibility with their overridden counterparts

# Readonly

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly>

* The **readonly** keyword may be used in three contexts:
  + In a field declaration, readonly indicates assignment may only occur in the declaration or in a constructor.
  + In a readonly **struct** definition to indicate the struct is immutable (C# 7.2+)
  + In a ref readonly method return to indicate that method returns a reference and writes are not allowed to that reference.

# Virtual

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual>

* **virtual** keyword Indicates that a method, property, event, or indexer may be overridden in a derived class
* Can only be used on instance members (not static)
* Use **override** keyword to override a virtual member
* Inherited virtual properties may be overridden in further derived classes (used **sealed** keyword to stop virtual chain)
* When a virtual method is invoked, the overriding method of the most derived type in the ancestry of the run-time type of the object is used (including the type of the run-time object itself).

# Sealed

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed>

* **sealed** keyword be applied to classes to prevent other classes from inheriting from them
* May be applied to overriding methods or properties in a derived class, forbidding further derived classes from overriding specific virtual methods or properties

# This

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/this>

* **this** keyword refers to the current instance of the class
* Must be used to indicate a class member when a local variable uses the same name
  + ex: this.name = name;
* May be used to modify the first parameter of a static method to declare an **extension method**
* May be used with square brackets to declare an **indexer**

# Base

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/base>

* **base** keyword may be used to access members of the base class from within a derived class
* permitted within constructors, instance methods or instance accessors
* Cannot be used within a static method

# Var

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/var>

* **var** keyword may be used to declare method-scoped variables
* The compiler will implicitly type the variable on your behalf

# Static

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>

* Modifier indicating the member belongs to the type itself instead of an instance of the type.
* Can be used with classes, fields, methods, properties, operators, events, and constructors, but it cannot be used with indexers, finalizers, or types other than classes
* Static classes/members cannot be derived from/overridden

# Methods

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods>

* A method **signature** along with a code block containing a series of statements to be executed
* May be declared in classes or structs
* The modifiers, return type, name, and parameters define a method’s **signature**
  + If a method does not return anything must use the **void** keyword in place of a return type
* Expression body methods may be used as shorthand to return a single item or expression from a method
* By default, all parameters are passed to methods by value
  + For value types, this means the value is copied
    - The method will not affect the source value
  + For reference types, this means the reference is copied
    - The method will not affect the value of the reference itself (ex: setting the reference variable to null and persisting this change in the calling context)
    - The method may modify the data belonging to the referenced object

## Parameter and Return Modifiers

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier>

* Modifiers may be applied to method parameters and returns to pass them by reference:
  + The **out** keyword specifies that the method *will* set the value of the argument
  + The **ref** keyword specifies that the method *may* set the value of the argument
  + The **in** keyword specifies that the method does not modify the value of the argument (C# 7.2+)
    - Implications: <https://blogs.msdn.microsoft.com/seteplia/2018/03/07/the-in-modifier-and-the-readonly-structs-in-c/>
* **Value types** may also be returned by reference by using the **ref** keyword, which can be more performant with large structs would require copying each value when returning.

## Constructors

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors>

* Special methods used to create an instance (usually) of a class or struct
* Name is the same as that of its type
* No return type (instance constructors implicitly return an instance of their type)
* Can be used to initialize variables, inject dependencies, limit instantiation, etc.

### Instance

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors>

* Instance constructors are called using the **new** keyword when instantiating structs and non-static classes
* Can be used to initialize non-static members
* Multiple constructors may be defined with different parameters
* Instance constructors may call other instance constructors in a nested way by using the **this** keyword for constructors in the same class or the **base** keyword for constructors of the base class (nested constructor called first) before the constructor body.
* Struct constructors must initialize all members

#### Default

* If no instance constructor is specified for a class, C# creates a default constructor that has no parameters and sets member variables to their default values.
* A default, parameterless constructor is generated for all structs; it may not be explicitly defined.

### Static

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors>

* Static constructors may be used to initialize static members of a type or perform any actions that only need to be performed once
* The C# compiler will initialize members to default values if a static constructor does not initialize them
* Always parameterless
* No access modifiers
* Called only once (automatically – never directly) for any type before the first instance is created or any static members are referenced

#### Private

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/private-constructors>

* Private constructors can be useful in certain scenarios:
  + With other accessible constructors to create a “base” constructor that implements functionality that all (or many) other constructors call (shared code)
  + With no other constructors to force the use of a static method to get an instance of a type (as in the factory pattern)

## Finalizers

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/destructors>

* Or **destructors**, are methods called to perform final cleanup when a class (not struct) instance is being collected by the garbage collector.
* Invoked automatically – cannot be called
* Cannot be inherited or overloaded and does not have modifiers or parameters
* Finalizer of a derived class implicitly calls finalizer of base class after it runs
* Cannot control when it is run
* IDisposable + using bock is typically preferred, but if the type doesn’t lend itself to this usage patter, finalizers are a good way to ensure unmanaged resources, etc. are freed after the instance goes out of scope.

## Extension Methods

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>

* “Extend” a class by supplying static methods that may be used as though they were member methods
* First method parameter: this T
* Extension methods cannot be used to replace or hide existing methods on the type they are extending. For example, it would not make sense to implement a parameterless ToString extension method since every object already has a ToString method inherited from object.

# Properties

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties>

* Member that provides access to some private backing field (implicit or explicit)
  + Simple properties that require no internal logic may be auto-implemented by just defining the get/set accessor signatures. These are called **auto-implemented properties**.
* Can be as simple as a wrapper around a private field or contain validation and computation code
* Have special methods called **accessors** (get, set)
* Value passed to set accessor using implicit **value** parameter
* Access modifiers or omitting accessors can be used to make properties read-write, read-only, write-only, or inaccessible
* Can be static

# Indexers

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/index>

* Like properties, but cannot be static or auto-implemented
* Allow an instance of a class or struct to be indexed with square bracket syntax just like arrays
* Index value is passed to body of get/set accessor

# Attributes

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/>

* Mechanism for applying metadata to assemblies, types, methods, properties, parameters, etc.
* Attributes may be interrogated using Reflection
* One or more attributes may be applied to a given element declaration
* Convention: attribute names end with “Attribute” suffix, but the suffix is not necessary when using attributes
* Attributes can accept parameters: positional – must be specified in order and before other parameters; unnamed, and named – optional
* Attributes may specify the target element type to which they apply (assembly, field, event, class, method, etc.)

# Exceptions

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions/index>

* Derive from System.Exception
* May be thrown due to errors or “exceptional” conditions that warrant intervention
  + The method cannot complete for some reason (can’t access resource, invalid parameter, etc.)
  + Inappropriate call to object is made due to its state (writing to a read-only file, etc.)
* Throwing an exception immediately ends execution of the code block and bubbles up the stack until a suitable try-catch block is found
* Contain call-stack information (StackTrace) as well as other useful information
* May be caught an analyzed for logging errors, cleaning up resources, notifying user, etc.

## Throwing Exceptions

* Exceptions should only be thrown due to error conditions, not to control program flow
* The most specific exception for a given error should be thrown
* Public and protected methods should throw exceptions whenever they cannot complete their intended functions
* Possible exceptions should be documented

# Try-Catch

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch>

* try block followed by one or more catch clauses
* When an exception is thrown within a try block, catch clauses are examined in the order they are written for a match
  + This means catch clauses for more specific exceptions should come before those of more generic exceptions
  + Exception arguments (exception type) as well as exception filters (“when”) may be used to specify matching exceptions
  + Exception filters are preferred to catching and rethrowing exceptions because they leave the stack unharmed
* throw statement may be used in a catch block to rethrow the caught exception

# Try-Finally

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-finally>

* The finally block typically executes when the try block is exited (either normally by completing, returning, breaking, etc. or an exception is thrown)
* Finally block **always** executes, **except** in certain cases when a uncaught exception causes the application to crash
* Typically used to cleanup resources allocated in a try block

# Try-Catch-Finally

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally>

* Typical pattern is to acquire and use resources in try block, catch exceptions in catch block, and cleanup resources in a finally block

# Generics

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/index>

* Generic interfaces, classes, methods, delegates, and events accept type parameter(s) to be specified by the code instantiating/calling it
* Makes code more “generic” while preserving benefits of strong typing
* Constraints
  + class/struct, interfaces, base classes, new()
* Covariance (interfaces and delegates only) use “out” generic modifier on your type parameter, ex. Buffer<out T> - only for returns (out), not method parameters
  + Allows methods to return a type that is more derived than the generic type parameter
* Contravariance (interfaces and delegates only) use “in” generic modifier on your type parameter, ex. Buffer<in T> - only for method parameters (in), not returns
  + Allows methods to take a type that is more derived that the generic type parameter

## Constructed Types and Unbound Generics

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types#constructed-types>

* A generic type definition is said to be *unbound* and functions as a template or blueprint for building types that have the type parameters specified (constructed types)
* Constructed types may be used just about anywhere a non-generic type may be used
* Unbound generic types may only be referenced within the **typeof operator** by leaving type parameters blank
  + Ex: var unboundType = typeof(MyGenericClass<,,>);

## System.Collection.Generics

* List<T>  - Indexed like an array, but automatically resizes itself as it grows
* Queue<T> - FIFO
* Stack<T> - LIFO
* HashSet<T> - collection of unique items, unordered.  Allows for set operations (intersection, union, etc.)
* LinkedList<T> - doubly linked-list.  Fast insertion, removal.
* Dictionary<TKey, TValue> - collection of key/value pairs. Keys are unique.
* SortedDictionary<TKey, TValue>, SortedList<TKey, TValue>, SortedSet<T> - maintain sort based on key/or type (SortedSet)

# Nullables

<https://docs.microsoft.com/en-us/dotnet/api/system.nullable-1?redirectedfrom=MSDN&view=netcore-2.2>

* Nullable<T> is a value type (struct) that represents a value type that can be assigned null
* Null can be checked by comparing to null or checking HasValue property

# Collections

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections>

* A collection is a type responsible for managing and providing access to a group of related elements in memory
* Collections can be broadly categorized into lists, dictionaries, and sets
  + **Lists** – provides access to elements by position in order (index)
    - Tend to use memory efficiently and provide efficient element access
    - Arrays, List<T>, Collection<T>, ReadOnlyCollection<T>, ObservableCollection<T>, IList<T>
  + **Dictionaries –** provide element access by a key
    - Provide very fast key-based lookup (but this is usually not as fast as looking up by index)
    - Dictionary<TKey, TValue>, IDictionary<TKey, TValue>
    - Often based on a hash table
  + **Sets –** treat collections as a group and allow operations to be performed on the collection as a whole (ex: union/intersection both join two entire collections to create a new collection).
    - HashSet<T>, ISet<T>
    - Often based on a hash table
    - Typically no element lookup – just enumerate through the set
* Operations
  + Reading
    - Enumerate the elements – request all items to be returned (one after another)
      * Lists usually return elements in indexed order, but dictionary and set types typically shouldn’t be depended on to return elements in any particular order (unless they are of an ordered type of course)
      * All collections support enumeration
    - Look up element (by index or key)
      * + Not all collections support lookup, even some list types that return items in order (Ex: sets, linked lists, stacks, queues)
  + Writing
    - Some collection types support adding and removing elements
    - List collections also support inserting elements into a particular position

## Collection Interfaces

### IEnumerable<T>

[https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=netstandard-2.0)

* Indicates the collection of items can be enumerated (or iterated through)
* **LINQ** extensions methods are written against this interface
* Defines one method: GetEnumerator() which returns an **IEnumerator<T>**
* Read-only – the collection can only be enumerated, not modified
* Enumerated **lazily** – items are not enumerated until they are accessed with foreach, etc.
  + Allows enumeration of things beyond typical in-memory collections -> can stream data over a network for example
  + Beware of pitfalls! Think of enumerable as a sequence of calls that returns values one at a time rather than a collection.

#### IEnumerator<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerator-1>

* State machine that does the enumerating for an enumerable
  + Separated from the IEnumerable<T> collection itself so that multiple clients could potentially enumerate the same collection independently, without interfering with each other
  + This also means IEnumerable should return a new enumerator for each call to GetEnumerator
* Most of the built-in collection enumerators check if the collection has changed each time MoveNext is called. If a change has occurred, an exception is thrown.
* Exposes **Current** property and **MoveNext** and **Reset** methods for enumerating/restarting
* This simple interaction model is what allows for lazy-enumeration
* Implements IDisposable, but IEnumerable (non-generic) does not

#### Enumerable Covariance

* Non-array read-write collections do not allow covariance because of the inherent danger of allowing the adding of an element that is of a completely different type
* Covariance is safe for IEnumerable because it is read-only and is therefore allowed

#### Yield

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield>

* Used within methods that return IEnumerable or Enumerator with return or break without an explicit IEnumerable/IEnumerator instance needed (the compiler creates them on your behalf)
* Returns items one by one
* Compiler generates the enumerator state machine on your behalf
* Cannot be used inside of try/catch block (try/finally, using OK)

#### Iterators

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators>

* Any method or get accessor that utilizes yield return, is known as an **iterator**
* Cannot have ref or out parameters
* Don’t support Reset (just get a new instance of enumerator)

#### For Each Loops

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in>

* **foreach** loop executes a block of code for each element in an instance that implements IEnumerable or
  + has a public parameterless GetEnumerator method (that returns a class, struct, or interface) and
  + the return type of GetEnumerator has a public Current property and a public parameterless MoveNext method that returns a boolean
* The foreach iteration variable is read-only (this helps avoid a lot of confusion)
* The collection itself cannot be modified in a foreach loop (adding, replacing, removing, inserting, etc.) – an exception will be thrown, but modifying the values of reference types is allowed
* Compiler translates array foreach loops into for loops for efficiency’s sake

### ICollection<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icollection-1>

* Implements **IEnumerable<T>**
* Represents a collection of items as well as modification functionality (IEnumerable represents something a little more abstract – a collection of items that may be stored elsewhere, say in a database, which could be fetched over and over – no count and no modification)
* Base interface for classes in System.Collections.Generic namespace
* Adds Count property as well as methods for modifying the collection (Add, Clear, Remove, etc.)
* The three categories of collections have implementing interfaces: IList<T>, IDictionary<TKey, TValue>, ISet<T>
* Some implementing classes will implement interface members explicitly

### IList<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ilist-1>

* ICollection<T> with indexer property and IndexOf(), Insert(), and RemoveAt() methods

### IDictionary<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.idictionary-2>

* ICollection<KeyValuePair<TKey,TValue>> with indexer property (by key) and Add() and Remove() methods for collection modification
* Can return a collection of keys, values
* TryGetValue(), ContainsKey() methods

### ISet<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.iset-1>

* Exposes Add() method
* Set operations: ExceptWith(), IntersectWith(), SymmetricExceptWith(), UnionWith()
* Set comparison: IsProperSubsetOf(), IsProperSupersetOf(), IsSubsetOf(), IsSupersetOf(), Overlaps(), SetEquals()

### Read-only Interfaces

ICollection<T> and implementing interfaces each (except ISet<T>) have a corresponding read-only interface.

#### IReadOnlyCollection<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlycollection-1>

* Essentially IEnumerable<T> with a Count property

##### ReadOnlyCollection<T>

* A wrapper around an IList<T> that implements IList<T> (modification-members explicitly so) but doesn’t allow modification (throws exception if modification attempted)
* Doesn’t prevent a consumer from using reflection to get at original list if they are so motivated

#### IReadOnlyList<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlylist-1>

* IReadOnlyCollection<T> plus a read-only indexer property

#### IReadOnlyDictionary<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ireadonlydictionary-2>

* Same as IDictionary<TKey,TValue> but without Add() and Remove() methods, and read-only indexer property

### Obsolete Interfaces

* The non-generic counterparts, IEnumerable, ICollection, IList, and IDictionary are largely superseded by their generic counterparts.
* **IEnumerable<T>** does implement its non-generic predecessor to ensure that older code could still enumerate generic collections since the generic interface can be cast to the non-generic.

## Arrays

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/>

* Reference type wrapper around contiguous memory block of elements (values or references themselves)
* When the array initialized but individual elements are not explicitly initialized, each element will be implicitly initialized to the default value for the type (ex: 0 for int, null for reference types)
* **Pros:** lightweight, index-based list of ordered elements that is supported by a rich API set and special C# syntax
* **Cons:** fixed size
* Implements ICollection and IList which includes nonsensical functionality like Add (throws exception)
  + ICollection.Add implemented **explicitly**.
  + Should check ICollection.IsReadOnly property first on ICollection (returns false for arrays)
  + Interestingly, arrays are not read-only, but in the ICollection sense, they are (no add, remove, etc.)
* .NET bounds-checks array indices (in an optimized way – compiler can remove checks if not needed)
* Read-only Length and LongLength properties
* Accepts values of more derived types (class array) or implementors (interface array)
* Each distinct declared array of type X in your code causes the creation of a type in the type system that derives directly from System.Array (your code cannot directly derive from this special class)
* Instance functionality: CopyTo, Clone, GetValue, SetValue, etc.
* Static functionality: AsReadOnly, Copy, ConstrainedCopy, Clear, BinarySearch, FindIndex, Sort, Reverse, etc.
* Arrays use zero-based indexing by default, but it is possible to create arrays that use other bounds
  + Must create using Array class’ static method CreateInstance
  + Very rare and not recommended (implemented to support interoperability with other languages)
  + To support this at runtime, use GetUpperBounds and GetLowerBounds instance methods

### Array Covariance

<https://blogs.msdn.microsoft.com/ericlippert/2007/10/17/covariance-and-contravariance-in-c-part-two-array-covariance/>

* You can always implicitly cast an array of a derived type to an array of a base type
* This is generally not a good idea and can lead to runtime problems (compiler cannot catch them)

### Multidimensional Arrays

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/multidimensional-arrays>

* Arrays can have more than one dimension. The number of indices is called the dimension or rank.
* Separate dimensions/lengths/indices with commas when declaring/instantiating/indexing
  + int[ , , ] a3 = new int[4,4,3];
  + a3[2,2,2] = 5;
* Length property returns the entire length of the array (all dimensions multiplied)
* Use GetLength instance method to get the length of a single dimension (zero-based index)
* Rank property returns the number of dimensions
* Contiguous memory (like single-dimensional arrays)

### Jagged Arrays

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/jagged-arrays>

* Can be created using an array of arrays
* C# syntax supports jagged arrays by multiplying square bracket sets
  + float[][] jaggedArray = new float[3][];
  + NOT var jaggedArray = new float[float[]];
  + Note: jagged array dimensions may only be declared one at a time
* Unless the need is for a truly jagged (inner arrays are of different lengths within the same dimension), multidimensional arrays should be preferred.
* Conceptually, can be implemented with other collection types
  + Ex: List<List<int>>

## Lists

### List<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1>

* Acts much like an array, but supports adding and removing elements (dynamically sized)
* Actually implemented as a wrapper around an array (Capacity property tells the current underlying array size)
* Adds rich functionality beyond IList – AddRange, RemoveRange as well as anything an array can do (binary search, etc.), except multi-dimensional capabilities
* Exposes an AsReadOnly method that returns the original list wrapped in a **ReadOnlyCollection<T>** to prevent modification

### Collection<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1>

* Implements IList<T> but, unlike List<T>, exposes virtual and protected methods so that deriving classes may be customized. This is accomplished like **ReadOnlyCollection<T>**, by encapsulating a List<T>
* Not as efficient as List<T>
* Ex: Add and Insert methods are not virtual, but they do call the virtual protected InsertItem method.

#### ObservableCollection<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.observablecollection-1>

* List that provides change notifications via INotifyCollectionChanged
* Derived from Collection<T>

### Specialized Lists

These are lists but without indexed-based lookup.

#### LinkedList<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1>

* List with fast adding and removing of elements
* Purpose: handle frequent list adds/moves/removals in a performant way
* Implemented as a doubly-linked list

##### LinkedListNode<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlistnode-1>

* Represents a doubly-linked list node that contains a T value and references to the next and previous nodes

#### Stack<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1>

* Last-in-first-out (LIFO) collection
* Can only push (add item to top) and pop (take item from top) (No insertion into middle)
* Think “the plate dispenser at the salad bar”
* Implements IEnumerable<T> in a “peek” (non-destructive) way
* Implemented with an array under the covers

#### Queue<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1>

* First-in-first-out (FIFO) collection
* Can only add to bottom (enqueue) and remove from top (dequeue)
* Implements IEnumerable<T> in a “peek” (non-destructive) way
* Implemented with an array under the covers

## Dictionaries

### Dictionary<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2>

* Basic dictionary
* Fast lookup, add, and remove
* Basically implemented as a hash table of linked list buckets
  + Can override **IEqualityComparer<T>** to customize how equality is determined (must implement GetHashCode method)
* No guaranteed order of enumeration
* May be wrapped in a **ReadOnlyDictionary<TKey,TValue>** to prevent modification
  + No AsReadOnly method as in List – must call constructor directly
  + Implements IDictionary<TKey,TValue> but throws exceptions when modification attempted

#### KeyValuePair<Tkey,Tvalue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.keyvaluepair-2>

* Represents a dictionary entry
* When enumerating a dictionary, key-value pairs are returned

### SortedList<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2>

* Dictionary that keeps items sorted by key
* Implemented as a pair of lists (one of keys and one of values) which are sorted by the key
* Lookup by key is relatively fast (binary search for key)
* Modifications are slow (as with List<T>)
* Can override **IComparer<T>** to customize how sort order is determined
* May be wrapped in a **ReadOnlyDictionary<TKey,TValue>** to prevent modification
  + No AsReadOnly method as in List – must call constructor directly
  + Implements IDictionary<TKey,TValue> but throws exceptions when modification attempted

### SortedDictionary<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2>

* Functionally the same as **SortedList<TKey,TValue>**, but uses a red-black tree (self-balancing binary search tree) internally
* Optimized for fast element lookup/modification (modifications are faster than SortedList)
* May be wrapped in a **ReadOnlyDictionary<TKey,TValue>** to prevent modification
  + No AsReadOnly method as in List – must call constructor directly
  + Implements IDictionary<TKey,TValue> but throws exceptions when modification attempted

### KeyedCollection<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.keyedcollection-2>

* Abstract base class for a collection whose keys are embedded in the values
  + Must override GetKeyForItem method
* A list and a read-only dictionary of items (implemented as one of each) that are kept in sync
  + Be careful using integers as keys. In this case, default indexer indexes by key, not index
* May be wrapped in a **ReadOnlyCollection<TValue>** to prevent modification

## Sets

### HashSet<T >

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1>

* Elements are unique and have no order
* Implemented as a hash table internally
  + No way to lookup elements by key or index (can’t reverse hash function) -> must enumerate
  + Fast insertion/check for element existence
  + Can override **IEqualityComparer<T>** to customize how equality is determined (must implement GetHashCode method)
* Implements ISet<T>, so it implements a rich collection of set operations
  + These operations modify the set inline (good for performance, but original set is lost)
  + LINQ versions of these operations returns a new collection

### SortedSet<T >

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1>

* Same functionality as HashSet<T> but uses a balanced tree internally to keep elements sorted
* Sorting only affects enumeration
* Can override **IComparer<T>** to customize how sort order is determined

## Concurrent Collections

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent>

Key Concepts:

* In environments in which multiple threads may execute concurrently on the same data set of data, there are many pitfalls that can lead to data corruption
* Atomic – from Greek *atomos*, meaning “indivisible”. Atomic operations are guaranteed to isolated from concurrent processes (they are indivisible). There are different ways using hardware and software to enable atomicity, but in this context, it suffices to say that atomic operations are necessary to create collections that are thread-safe.

The standard .NET collections are not thread-safe. The collections System.Collections.Concurrent namespace includes multiple thread-safe counterparts to the standard collections which may be used in such contexts.

So why not just use thread synchronization mechanisms (locks, etc.) to limit threads in critical sections where the collections are modified?

* The code to do this correctly can be quite complicated to avoid deadlocks in large codebases with multiple critical paths
* Added awareness required to add locking to new code/maintain existing code
* Potentially a large performance degradation
  + Concurrent collections use many techniques or more complex algorithms (which may include, but are not limited to, simple locks) to achieve thread-safety while avoiding performance hits
    - Ex: ConcurrentDictionary uses an optimistic concurrency technique that calculates changes as though no other thread interfered, then atomically checks for interference and applies the changes if none has occurred. If interference has occurred, it starts its calculations over. This scales very well by minimizing contentions with locks, etc.
    - Ex: ConcurrentDicitionary also uses fine-grain locking (only affected portions of the dictionary – e.g., hashtable buckets – are locked during modification) to minimize contention.

The .NET concurrent collections are robust, thread-safe (from internal data corruption), and scalable. The thread-safe operations provided by these collections can NOT protect against race conditions your code may create between calls to them, so it is advisable to try to accomplish entire discrete operations within a single atomic call. For example, if you needed to increment a value in a concurrent dictionary and then return its updated value. You could use the AddOrUpdate method followed by the TryGetValue method but this would introduce a race condition in which another thread could modify the collection between the two method calls. Instead, you should simply return the updated value returned by the AddOrUpdate method. In multithreaded scenarios the developer’s task is often more complicated than just migrating to a concurrent collection; rather it is often about using the tools (API surface) at your disposal to accomplish task in a thread-safe way (usually by implementing each necessary operation in a way that it may complete atomically).

Microsoft’s documentation for each type will have a **Thread Safety** section that identifies which functionality is and is not thread-safe.

### General Purpose Collections

#### ConcurrentDictionary<TKey,TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2>

* Thread-safe collection of key-value pairs
* Implements **IDictionary<TKey,TValue>** and adds new methods that are useful in multithreaded scenarios
  + IDictionary<TKey,TValue>.Add/Remove are implemented explicitly to discourage their use
    - These methods make some assumptions about the state of the dictionary before they are called: namely about the existence of the key being added/removed (exceptions thrown if a duplicate key is added or if a non-existent key is removed). These assumptions may be valid and safe in a single-threaded environment but are not in a multi-threaded environment because the result of any check preceding the operation could be invalidated by another thread modifying the collection just before the operation. The methods below are more appropriate for multithreaded environments and should be preferred.
  + Try-methods – fail gracefully by returning true for success and not throwing exceptions due to unexpected state
    - TryGetValue – tries to get the value with the specified key
    - TryAdd – adds a new key to the dictionary if it does not already exist
    - TryRemove – removes the specified key from the dictionary if it exists
    - TryUpdate – updates the value for an existing key if it already contains a specific value
  + Or-methods – won’t fail (unless a supplied delegate throws an exception, etc.)
    - AddOrUpdate – adds a key value pair to the dictionary or updates the value for the key based on the existing value if it already exists (two delegates may be supplied: one for add and one for update)
    - GetOrAdd – gets the value for a specified key or adds the specified value (or value from the specified delegate) if it doesn’t exist (Add delegate may supplied)
  + These operations are atomic and thread-safe (see documentation for information locking implementation), with the exception of the execution of the delegates they accept as parameters. So for example, the AddOrUpdate method executes the supplied delegates, but outside of any locks to generate the values for consumption by the atomic code that attempts to update the value. If this atomic operation fails due because another thread updated the collection concurrently, the process (including delegate execution) starts again. This cycle repeats until successful. This means the supplied delegates may execute more times than the containing operation.
  + TIP: If a non-returned result of an atomic operation is required, it may be helpful to set a **closure** variable within the supplied delegates that may be consumed outside of the “atomic” operation
    - If using this approach, you should use the method overloads that accept delegates for both cases (get/add or add/update) and explicitly set the closure variable in both delegates. This is because, as stated previously, the setup code (delegates) may execute multiple times, so you cannot assume the captured closure variable will retain its default value just because the final result path retained the default variable value.
    - This can cause your code to be complicated and difficult to debug. Alternatively, you may (not always) break your operation into multiple independent atomic operations to derive the needed information. For example, if you need to know whether the chosen route was add or update and both could potentially give the same return value, you could replace the add value to a known impossible update value, then detect this and call AddOrUpdate again to correct the set value if necessary.

### Producer-Consumer Collections

These collections are not suitable as general-purpose collections because they have no direct element access. They are more suited for scenarios in which one or more tasks is producing data to be produced and one or more threads consuming this data.

#### ConcurrentQueue<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1>

* Thread-safe first-in-first-out (FIFO) collection
* Enqueue method just like **Queue<T>** as enqueuing should never fail due to activity in other threads
* TryDequeue and TryPeek are implemented as try-methods with out-parameters which provide thread-safety

#### ConcurrentStack<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1>

* Thread-safe last-in-first-out (LIFO) collection
* Push method just like **Stack<T>** as pushing should never fail due to activity in other threads
* TryPop and TryPeek are implemented as try-methods with out-parameters which provide thread-safety

#### ConcurrentBag<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1>

* Thread-safe collection of T that allows adding/removing items but the order of removal is not guaranteed
* Add method and TryTake, TryPeek implemented as try-methods with out-parameters for thread safety
* Maintains an independent thread-local collection for each thread that attempts to access it
  + Each thread adds to and removes from its own collection until a TryTake is attempted and the thread local collection is empty, then the item is *stolen*from another thread’s collection (requires thread synchronization)
    - Optimized for scenarios where the same thread is used to produce and consume
  + Within any thread, it typically behaves like a stack
  + Very fast performance when no threads are starved, but *stealing*does require thread synchronization and incurs a performance hit

#### IProducerConsumerCollection<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.iproducerconsumercollection-1>

* Implemented by CuncurrentQueue<T>, ConcurrentStack<T>, and ConcurrentBag<T>
* TryAdd and TryTake methods, and Count property
* Implements IEnumerable<T> and ICollection<T>
* Exists primarily to provide functionality for **BlockingCollection<T>** from the collections that implement it

#### BlockingCollection<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1>

* Wrapper class that encapsulates an **IProducerConsumerCollection<T>** instance(ConcurrentQueue<T> by default)and provides concurrent addition and removal of items from multiple threads using Add/Take methods as well as bounding and blocking
* The maximum element capacity may be specified and Add calls will block until space is available
  + This throttles the producing threads so they cannot get too far ahead of consuming threads
* Take method blocks if empty until items are available and throws an InvalidOperationException if called and the collection is empty and no more items are expected to be added (this is indicated by calling the CompleteAddingMethod)
  + Add/Take are cancellable using a CancellationToken and TryAdd/TryTake accept a timeout parameter
* Static methods TryAddToAny/TryTakeFromAny accept an array of BlockingCollections
  + Example using an array of BlockingCollections in a pipeline: <https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/how-to-use-arrays-of-blockingcollections>
* Implements IDisposable

### Custom Partitioners

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/custom-partitioners-for-plinq-and-tpl>

The Task Parallel Library’s Parallel.Foreach executes a foreach operation in which iterations may run in parallel. Likewise in PLINQ (Parallel LINQ), queries may be executed in parallel. In each of these scenarios the respective library provides default partitioners which partition the data into multiple sections to be accessed concurrently. These methods also provide overloads in which a custom partitioner may be supplied. The System.Collections.Concurrent namespace also includes the partitioner base classes which may be extended to create such custom partitioners.

### Missing Concurrent Version Alternatives

There is not a thread-safe version of every standard .NET collection type, but with a little creativity, you can get similar results. Below is a list of possible thread-safe alternatives to some standard collections.

* List<T>/T[]
  + ConcurrentDictionary<int,T> where the index is the dictionary key
  + T[] with all accessing code wrapped in locks
* HashSet<T>
  + ConcurrentDictionary<T,T> with each key equal to its value, so uniqueness of values is ensured
* SortedList<TKey,TValue>/SortedDictionary<TKey,TValue>
  + ConcurrentDictionary<TKey,TValue> and sort elements before enumerating

### Concurrent Collection Tips and Best Practices

* Try to use a only a single concurrent collection method call for each required operation to avoid race conditions
* There is necessarily overhead in making collections thread-safe and scalable which degrades performance when compared with their single-threaded counterparts, so don’t use concurrent collections in non-concurrent scenarios
* CPU-bound work that needs to interact with the collection only sparingly is the best type of scenario for using multi-threading + concurrent collections. If the work to be done isn’t doesn’t require much computation and will access the collection frequently, the overhead introduced by the concurrent collection along with the thread contention for access to a shared resource will typically outweigh any gains made by multi-threading.
* Avoid code that requires interacting with the aggregate state of concurrent collections
  + Querying the aggregate state of a concurrent collection is often quite expensive. For example, if a ConcurrentDictionary’s IsEmpty property is used, the entire dictionary must be locked so that every bucket may be interrogated. So typically it is best to avoid frequent collection interactions that involve the entire collection: The following collection methods/properties may require extensive locks (slow):
    - **ConcurrentDictionary**: IsEmpty, Count, Clear(), ToArray(), CopyTo(), Values, and Keys.
    - **ConcurrentBag**: IsEmpty, Count, ToArray(), CopyTo(), GetEnumerator()
* Concurrent collections do not strictly block modifications from occurring while they are being enumerated.
  + ConcurrentBag, ConcurrentStack, and ConcurrentQueue each take a snapshot of the collection (slow) when the enumerator is requested for enumeration and any subsequent changes are ignored while enumerating.
  + ConcurrentDictionary allows modifications during enumeration, but the results may be in an indeterminate state.
    - ToArray will create a snapshot which can be enumerated, as do Keys and Values collections
* When to use concurrent collections (if you don’t implement thread synchronization yourself):
  + Concurrent collections are always required if multiple threads may modify the collection concurrently.
  + Concurrent collections are required if a single thread modifies the collection while multiple threads read it.
  + If a collection is only read from multiple threads (while not being modified), the standard collections may be safe to use, but this may be risky if the collection makes any modifications lazily under the hood when read (only verifiable via documentation/source code).
    - An alternative in certain multi-reader scenarios may be **Immutable Collections**

## Immutable Collections

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable>

## Obsolete Collections

* Some collection types were added to .NET before the introduction of generics and are weakly-typed as a result (ex: System.Collections.ArrayList). These should be considered obsolete and their generic equivalent or counterparts should be preferred.

# Lambda Expressions

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions>

* A lambda expression is an unnamed block of code
* (comma-separated parameters) => { code }
  + Parameter types may be inferred (by the compiler if possible) by using only their names or specified explicitly
  + => is referred to as the “lambda operator” or can be read “goes to”
  + Lambda expressions cannot be assigned to implicitly-typed variables (no var)
* Implemented as a custom class and a delegate
  + Compiler generates a class/method behind the scenes that matches the signature of the lambda (parameters and return type) with the same code body as the lambda.
  + Compiler generates a delegate with a class instance and method reference to call the lambda code

## Anonymous Methods

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-methods>

* Anonymous methods have been largely superseded by lambdas as the preferred method of creating inline code
* button1.Click += delegate(System.Object o, System.EventArgs e)

{ System.Windows.Forms.MessageBox.Show("Click!"); };

## Closures

* A closure is code and the supporting data environment
* Necessity of conditions created by lambdas because data external to the lambda could be used inside of the lambda code. The compiler must find a way to pass this information to the code.
* All data is *conceptually* passed by reference: every usage of the closure variables within the code references a common memory location for that data (even value types).
  + This means closure variables are shared variables and should be treated as such regarding race conditions, etc. as lambda code could be executed in a different thread than its creation context if, for example, it is executed inside of a task.
* Closure variables are *actually* implemented as fields within the compiler generated class and the code inside and outside the lambda actually references those shared fields.
* Use [https://sharplab.io](https://sharplab.io/) and create a simple action in the method that modifies a local variable (within the method but outside of the action) to see the generated closure class.

# Local Functions

<https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7#local-functions>

* Introduced in C# 7.0
* Local functions are methods that are declared inside the context of another method
* This means the local function may only be called from within the context of the method in which it’s declared
  + This is useful in certain circumstances where even declaring a method private wouldn’t limit its usage enough. See document above for examples of this scenario with public iterator methods and public async methods.

# Expression-bodied Members

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members>

* Provides a concise syntax for providing member implementations when very little code is required
* Uses “arrow”/”lambda operator” syntax (ex: member => expression;)
* As of C# 7.0, the following member types support expression-bodied members: methods, properties, constructors, finalizers, indexers

# Object and Collection Initializers

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers>

* Object Initializers
  + Allows instantiation and member assignments in a single statement
  + Works with any accessible fields or properties
    - var person = new Person { Name = “Bob”, Age = 52 }
* Collection Initializers
  + Allows initialization of collections that implement IEnumerable and have Add instance methods
  + List<Person> people = new List<People> { new Person { Name = “Bob”, Age = 52 }, new Person { Name = “Amy”, Age = 41 }};

# Anonymous Types

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/anonymous-types>

* Allows creation of a type in code that is “anonymous” (not named – the compiler generates a name, but it can’t be used in code) and cannot be accessed anywhere else within the code
* Only contains a set of read-only properties, no methods, events, delegates, etc.
* Useful for creating objects that are only used in one specific context and don’t need a formal type definition
* Use new operator with an object initializer (types inferred by the compiler):
  + var somePerson = new { Name = “Bob”, Age = 52 }

# Task Parallel Library (TPL)

## Key Concepts

* **Concurrency:** Doing more than one thing at once (logically, not necessarily simultaneously). Ex: multiple applications running simultaneously on a single-core processor.
* **Parallelism:** Doing more than one thing at once (physically, hardware - multi-processor/core required)
* **Thread:** A basic unit of CPU utilization; a set of work to be executed
* **Multithreaded:** Multiple execution contexts (not necessarily, but usually, concurrently)
* **Synchronous:** Code is run sequentially, line by line, each waiting for the previous to complete
* **Asynchronous:** Not having to wait - later code can run before previous code completes. Improves responsiveness. Very useful for slow-running tasks, ex: retrieving file from disk
* **Thread Pool:** A collection of threads waiting for tasks to execute

## Why use asynchronous or parallel programming?

* Asynchronous
  + Web APIs and I/O-bound work
    - Synchronous requests block even when waiting on I/O-bound work (file system, database, web request, etc.)
    - Only a limited number of threads are available in the thread pool so new requests must wait while others finish when no threads are available
    - Asynchronous APIs increase vertical scalability by improving resource utilization
      * Ex: a single server running the same API could handle more simultaneous requests by utilizing asynchrony
      * Threads can be returned to the thread pool while asynchronously waiting for I/O-bound tasks to complete, freeing them up to handle more requests
  + Responsive User Interface
    - Many UI frameworks require certain work (typically UI updates) to be done on the UI thread. If the code to do the update needs to perform some slow operation, say reading from a database, synchronous code would block the UI thread, making the application appear unresponsive (not a good experience).
    - If the slow operations could be run asynchronously, the UI thread would be freed up and stay responsive.
* Parallel
  + Performance: divide and conquer
    - Complex problems may often be broken up into parts that may be executed simultaneously and aggregated, which can greatly increase performance.

## TPL

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>

“The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details. By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.”

## Async and await keywords

* **async modifier**
  + Enables the await keyword to be used inside a method
  + Compiler generates a state-machine for this method
  + asynchronous methods may return one of the following types:
    - **void**
      * not advisable unless no other option (event handlers)
      * can’t be awaited, so hard to handle exceptions -> difficult to test
        + should try to avoid throwing exceptions in these methods (use try-catch blocks)
      * no way to provide status to caller
    - **Task, Task<T>**
      * Abstraction that represents a single operation that returns nothing (Task) or a value of type T (Task<T>) that usually executes asynchronously.
      * Represents the execution of the async method (not the result)
      * Status, IsCanceled, IsCompleted, IsFaulted properties may be used to determine state
      * Tasks are managed by the state machine generated by the compiled async method
        + Implements System.Runtime.CompilerServices.IAsyncStateMachine

<https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.iasyncstatemachine>

* + - * + See example state machine here: <https://sharplab.io/> (make method asynchronous to see the difference)
      * Result property
        + Synchronously waits for the Task to complete, then returns the result. To avoid blocking threads, this property should only be accessed after the Task has asynchronously completed.
        + If the task is in the Faulted state, accessing the result property will cause a wrapping **Aggregate Exception** to be thrown.
    - Types with accessible **GetAwaiter** methods (C# 7+)
      * Must implement System.Runtime.CompilerServices.ICriticalNotifyCompletion
      * Tasks are classes, which have overhead; this supports generalized return types which allows returning value types (lightweight)
* **await operator**
  + Tells the compiler that the async method can’t continue until the awaited asynchronous process is complete (think “asynchronous wait”)
  + Returns control to the caller of the async method (potentially all the way up until the thread is freed)
  + When awaiting a task of type TResult, the expression is of type TResult
  + Behind the scenes, a callback is installed on the task using a **continuation**.
    - The callback resumes the asynchronous method at the point of suspension.
    - If the awaited operation completed successfully, a TResult is returned
    - If the awaited operation was canceled, an OperationCanceledException is thrown
    - If the awaited operation faulted, the exception that caused the fault is thrown (if multiple exceptions occurred, the task’s Exception property will contain them all wrapped in an **Aggregate Exception**)
* A method that is not marked with the async modifier cannot use await
* When an async method doesn’t contain an await operator, it will be executed synchronously.

## Asynchronous Method Execution

* When an asynchronous method is called, it executes the code body synchronously until reaching the first await expression on an awaitable instance that is **incomplete** (if it is complete, execution continues synchronously), at which point method execution suspends, and the invocation returns to the caller with:
  + If the method return type is not void, a Task or Task<TResult> (of the return type of the method) to represent the ongoing work
* When suspension occurs, the synchronization context is captured to be used when the awaited instance completes (the method will be resumed from the same context):
  + If a synchronization context was associated with the thread that was executing the method before suspension, the method will resume on that synchronization context.
  + Otherwise, the task scheduler used at the time of suspension (typically TaskScheduler.Default which targets the thread pool) will be used to determine how to run or schedule the remaining work. The default scheduler typically continues the method on same thread that completed the awaited operation.
  + For application level code (especially with a UI), this is almost always what you want. For library code, this is never what you want (see **Task.ConfigureAwait**)
* If a return statement or the end of the method body is reached, the task enters the ran to completion state
* If an uncaught exception causes exiting the method body, the task enters the faulted or canceled state (see **Cancellation**)

### Task.ConfigureAwait

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.configureawait>

* ConfigureAwait accepts a single Boolean parameter: continueOnCapturedContext
  + true (default): capture context and post continuation back to captured context/scheduler
  + false: don’t capture context and, if possible, continue executing where the await task completes
* **Performance:** avoids unnecessary thread marshalling (moving to another thread) which leads to better performance
* **Deadlock:** ConfigureAwait(false) in library code can avoid deadlock in certain scenarios in which poorly written code attempts to synchronously wait for asynchronous library to finish while the code it’s waiting for is attempting to run a continuation in the same context (which is blocked by the wait).
* **Note:** your mileage may vary on different platforms. For example, ASP.NET Core does not have a synchronization context, so configure await will have no effect.
* **Guidance:** Use default behavior in UI scenarios in which a specific context is required to do certain work, and use ConfigureAwait(false) in shared library code in which the context in which the continuation runs is of no concern.
* More info on the Synchronization Context: <https://msdn.microsoft.com/en-us/magazine/gg598924.aspx>

### Task.Yield

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.yield>

* Can be used to force an asynchronous method to run asynchronously (ex: if all awaited items are complete)
* Awaiting Task.Yield() will yield back to the current context and will schedule the remaining method code according to the normal rules (see **Asynchronous Method Execution**)

## Tasks and async

* Supports Task-based Asynchronous Pattern (TAP) – best practice today
* Older asynchronous patterns are still valid but no longer preferred:
  + Event-based Asynchronous Pattern (EAP) – generally have async method names and associated competed events that may be subscribed to.
  + Asynchronous Programming Model (APM) – *BeginOperation/EndOperation* methods

## Task Completion Source

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1>

* Generic class that allows completion of a thread-less task and control of its state
* Useful for hardware tasks (I/O) etc. that don’t need a thread
* Can also be used in code to await events (See UWP example)
* Task property returns Task<T>
* Use **TaskCompetionSource<object>** for plain tasks (no output)
* Task status is initially waiting for activation
* Task will never enter the running state (thread-less) -> will transition directly to final state
* Complete with Try/Set Result, Exception, Canceled
  + Setting the exception(s) results in the Task’s Exception property being set to the provided exceptions wrapped in an aggregate exception
  + Single-use – may not set final state again after it has been set

## Thread-based Tasks

### Execution Model

* Thread is dedicated to task until it completes (code block exited – by completing or throwing an exception)

### Creating tasks

* A task may be instantiated directly, but execution will not begin until its Start method is called.
* **Task.Factory.StartNew** static method can be used to enqueue the work to be started on the thread pool
  + Accepts a delegate that will be run
  + Returns a Task representing the ongoing work
* **Task.Run** should be thought of as a quick way to start a Task with the following default options supplied: CancellationToken.None, TaskCreationOptions.DenyChildAttach, TaskScheduler.Default
  + <https://blogs.msdn.microsoft.com/pfxteam/2011/10/24/task-run-vs-task-factory-startnew/>
* The code supplied could itself be asynchronous by supplying an asynchronous delegate (use async key word before lambda)
  + Returns a Task<Task> or Task<Task<T>> (representing the outer (Task.Run) and inner (async delegate) work)

#### TaskCreationOptions

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions>

Specifies flags that control optional behavior for the creation and execution of tasks.

* **None** – Specifies that the default behavior should be used
* **PreferFairness** – A hint to a TaskScheduler to schedule a task in as fair a manner as possible, meaning that tasks scheduled sooner will be more likely to be run sooner, and tasks scheduled later will be more likely to be run later
* **LongRunning** – Specifies that a task will be a long-running, coarse-grained operation involving fewer, larger components than fine-grained systems. It provides a hint to the TaskScheduler that oversubscription (creation of more threads than the available number of hardware threads) may be warranted
* **AttachedToParent** – Specifies that a task is attached to a parent in the task hierarchy
* **DenyChildAttach** – Specifies that any child task that attempts to execute as an attached child task (that is, it is created with the AttachedToParent option) will not be able to attach to the parent task and will execute instead as a detached child task
* **HideScheduler** – Prevents the ambient scheduler from being seen as the current scheduler in the created task. This means that operations like StartNew or ContinueWith that are performed in the created task will see Default as the current scheduler**.**
* **RunContinuationsAsynchronously** – Forces continuations added to the current task to be executed asynchronously

## Continuations

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.continuewith>

* **Task.ContinueWith** instance method may be used to create a continuation.
* This is simply another task that runs asynchronously when the supplied task completes
* It accepts a delegate with the original task (antecedent) as a parameter so that it may respond to the result/status of the original task.
* By default, continuations are guaranteed to run (event if the antecedent task faulted), but this behavior may be adjusted by supplying **TaskContinuationOptions**.

### TaskContinuationOptions

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcontinuationoptions>

Enumeration whose flags values specify the behavior of a continuation task.

* Status-dependent execution options
  + **None** – The continuation runs asynchronously when the antecedent task completes, regardless of the antecedent's final Status property value
  + **NotOnFaulted** – Continuation task should not be scheduled if its antecedent threw an unhandled exception
  + **NotOnRanToCompletion** – Continuation task should not be scheduled if its antecedent ran to completion
  + **OnlyOnCanceled** – Continuation should be scheduled only if its antecedent was canceled
  + **OnlyOnFaulted** – Continuation task should be scheduled only if its antecedent threw an unhandled exception
  + **OnlyOnRanToCompletion** – Continuation should be scheduled only if its antecedent ran to completion
* Task scheduler options
  + **PreferFairness** – A hint to a TaskScheduler to schedule task in the order in which they were scheduled, so that tasks scheduled sooner are more likely to run sooner, and tasks scheduled later are more likely to run later
  + **LongRunning** – Continuation will be a long-running, course-grained operation. It provides a hint to the TaskScheduler that oversubscription may be warranted
  + **HideScheduler** – Tasks created by the continuation by calling methods such as Run(Action) or ContinueWith(Action<Task>) see the default scheduler (Default) rather than the scheduler on which this continuation is running as the current scheduler. See <https://devblogs.microsoft.com/pfxteam/new-taskcreationoptions-and-taskcontinuationoptions-in-net-4-5/>
* Task hierarchy options
  + **AttachedToParent** – Continuation, if it is a child task, is attached to a parent in the task hierarchy
  + **DenyChildAttach** – Any child task (that is, any nested inner task created by this continuation) that is created with the AttachedToParent option and attempts to execute as an attached child task will not be able to attach to the parent task and will execute instead as a detached child task
* Execution
  + **LazyCancellation** – In the case of continuation cancellation, prevents completion of the continuation until the antecedent has completed. See <https://devblogs.microsoft.com/pfxteam/new-taskcreationoptions-and-taskcontinuationoptions-in-net-4-5/>
  + **RunContinuationsAsynchronously** – Specifies that the continuation task should be run asynchronously
  + **ExecuteSynchronously** – Specifies that the continuation task should be executed synchronously. With this option specified, the continuation runs on the same thread that causes the antecedent task to transition into its final state. If the antecedent is already complete when the continuation is created, the continuation will run on the thread that creates the continuation. If the antecedent's CancellationTokenSource is disposed in a finally block (Finally in Visual Basic), a continuation with this option will run in that finally block. Only very short-running continuations should be executed synchronously.

## Composite Tasks

The TPL provides methods that can be used to compose multiple tasks. These methods themselves return a Task that represents execution of the supplied tasks.

### Task.WhenAll

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenall>

Creates a task that will complete when all the supplied tasks have completed. The returned task’s final status will be set according to the following criteria:

* **Faulted**: if any of the supplied tasks completes in the faulted state
* **Canceled**: if none of the supplied tasks completes in the faulted state and at least one was canceled
* **RanToCompletion**: if none of the supplied tasks is faulted or canceled

### Task.WhenAny

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany>

Creates a task that will complete when any of the supplied tasks have completed. The returned task’s (Task<Task<T>>) final status (when any of the supplied tasks completes) will always be RanToCompletion and the returned task’s result will be that of the first of the supplied tasks to complete.

## Task.Delay

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay>

Returns a Task that completes after the specified time span.

## Task.FromResult

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromresult>

Returns a Task<TResult> that has completed successfully with the specified result.

## Task.FromCanceled

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromcanceled>

Returns a Task or Task<TResult> that has completed due to cancellation with the specified cancellation token (cancellation must be requested on the supplied token or an ArgumentOutOfRangeException will be thrown).

## Task.FromException

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromexception>

Returns a Task or Task<TResult> that has completed with a specified exception.

## Child (Nested) Tasks

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/attached-and-detached-child-tasks>

A child task is a task instance that is created within the user delegate of another (parent) task. A child task can either be attached or detached from its parent. A detached child task executes independently of its parent. An attached child task is created with the **TaskCreationOptions**.AttachedToParent option whose parent does not explicitly or by default prohibit it from being attached.

Attached child tasks cause the following behaviors that are not present in detached child tasks:

* Parent tasks waits for child tasks to complete
* Parent tasks propagates exceptions thrown by child tasks
* Parent task’s status depends on child tasks’ statuses

The default behavior of detached child tasks is recommended for most scenarios.

## Errors

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library>

* TaskCompetionSource.SetException may be used to set an exception on a thread-less task
  + Pass in a single exception or IEnumerable<Exception>
  + TaskCompletionSource wraps passed exceptions as inner exceptions inside an AggregateException and places the associated task in the faulted state
* Unhandled exceptions thrown by code running inside of a task are propagated back to the calling thread when the static or instance Task.Wait methods are used or the Result is accessed.
  + These may be caught as usual using a try-catch block
  + The TPL wraps the thrown exception(s) in an **Aggregate Exception** before throwing.
* Awaiting a Task in a faulted state will cause the original exception to be thrown (single task) or one of the original exceptions (composite task with multiple exceptions), but it cannot be known which of the exceptions will be thrown.

### Aggregate Exceptions

<https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception>

When a single task or a composite task fails, the underlying exception(s) may be retrieved using the InnerException or InnerExceptions (read-only collection) properties, respectively, of the wrapping Aggregate Exception. In the case of multiple exceptions, the underlying exceptions may be iterated through to handle each, or the **AggregateException.Handle** method may be utilized.

If a task has child tasks with unhandled exceptions. The aggregate exception of the child tasks will be propogated to the parent task, which will wrap the aggregate exception in its own aggregate exception. To avoid needing to traverse a tree of nested aggregate exceptions, you may use the **AggregateException.Flatten** method documented here: <https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception.flatten>

#### Handle Method

<https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception.handle>

Iterates the inner exceptions of an aggregate exception and executes the provided function for each. For each exception, the function returns true to indicate that the exception was handled and false otherwise. If all inner exceptions are handled, execution continues; otherwise a new Aggregate Exception is thrown, containing each of the unhandled exceptions.

### Unobserved Exceptions

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler.unobservedtaskexception>

Unobserved task exceptions (like any unobserved exceptions) trigger exception escalation policy, which prior to .NET Framework version 4.5, would terminate the process. Now an UnobservedTaskException is raised, but the exception is handled by the runtime instead of terminating the process. The behavior may be reverted via configuration if process termination is desired.

## Cancellation

### Cancellation Model

* Consistent mechanism for cancelling plurality of long-running-task implementations in .NET
  + Synchronous and asynchronous
  + Thread-based and threadless work
* Chainable – single cancellation request that can be propagated to sub-operations
* Optional
  + Non-coercive – method’s implementor decides *whether* to implement cancellation
  + Asynchronous – method’s implementor decides *if/when* to honor cancellation
    - Cancellation may never occur if past point of no return.
    - Time may elapse before a successful cancellation completes so the method can do cleanup, etc.

### CancellationToken (Target)

* Single-use – a cancellation token may be cancelled once (cannot be reset once canceled)
* May be shared (passed down to sub-operations)
* Recipients may discover requested cancellation and act
  + Polling – Synchronous work may do this by polling at various stages or in loops, etc.
  + Notification – a callback may be registered to be notified when cancellation occurs
    - CancellationToken.Register returns a CancellationTokenRegistration which may be used to unregister callbacks as phases complete.
  + WaitHandle – can signal when cancellation is requested
* Cancellation cannot be requested/initiated via the token.
* ValueType – copied each time passed down, but each references the same CancellationTokenSource (CTS). Considered equal if they reference the same CTS.

### CancellationTokenSource (Trigger)

* Implements IDisposable (wrap in using or dispose)
* May be used to request cancellation on related CancellationToken

### Responding to Cancellation Requests (Opting-in)

* When the CancellationTokenSource.Cancel method is called, the cancellable operation (user delegate) that received its token may respond in one of two ways:
  + Simply returning from the delegate – the task will transition to the TaskStatus.RanToCompletion state (no indication it was successfully canceled).
  + Throw an OperationCanceledException and include the CancellationToken (this can be accomplished by calling the token’s ThrowIfCancellationRequested method). The task will then be transitioned to the Canceled state which the calling code could use to verify successful cancellation.
    - Note: the supplied CancellationToken must have cancellation requested, or the TPL will put the task in the Faulted (not Cancelled) state

### Tasks and Cancellation

* Thread-based Tasks (Task.Run, Task.Factory.StartNew, etc.) have intrinsic cancellation support that we get for free (delegate need not opt-in)
  + If you pass a cancellation token to a thread-based task and it is cancelled before the task starts to run, the TPL will not run the task and put it into the canceled state immediately.
* Responding to Cancellation Requests (Opting-in)
  + When the CancellationTokenSource.Cancel method is called, the cancellable operation (user delegate) that received its token may respond in one of two ways:
    - Simply returning from the delegate – the task will transition to the TaskStatus.RanToCompletion state (no indication it was successfully canceled).
    - Throw an OperationCanceledException and include the CancellationToken (this can be accomplished by calling the token’s ThrowIfCancellationRequested method). The task will then be transitioned to the Canceled state which the calling code could use to verify successful cancellation.
  + If an OperationCanceledException is thrown from within the task’s body, the TPL checks to see if the exception’s token matches the task’s token and that cancellation was requested
    - If cancellation wasn’t requested, there are no CancellationTokens supplied, or they don’t match the task will enter the faulted state (as with any other exception).
    - If cancellation was requested and tokens match, the task will enter the canceled state.
* Tasks in the canceled state will throw a TaskCanceledException (derives from OperationCanceledException) if they are waited or the result is read.
* When a task enters the canceled state, that is its final state so any continuations will then run just as if it completed successfully (continuations don’t pick up antecedent canceled state)
  + If continuation cancelation is desired in that case, just pass the same token to the continuation as the original task.
* Task.WhenAll
  + If all tasks succeed, the composite task succeeds
  + If at least one task faults, the composite task enters faulted state
  + If no tasks fault, but at least on is canceled, the composite task enters canceled state

### Linked Tokens

Allows supporting multiple cancellation tokens that may be used to cancel all work if any single token is cancelled.

* CancellationTokenSource.CreateLinkedTokenSource creates a linked cancellation token source from multiple cancellation tokens.

### Cancellation and Deadlock

* Rules for callbacks (CancellationToken.Register):
  + Must complete work quickly – token source’s cancel method won’t return until all the registered callbacks have returned. Anything more than trivial amount of work is probably a bad design.
  + Be careful with CancellationTokenRegistration.Dispose method
    - Dispose method guarantees registration is not currently, and will never, run, which means dispose must wait for the callback to complete if it has already been called when disposal initiated.
    - Must be careful to not dispose registration while lock held in callback. Ex: Thread1 acquires lock -> Thread2 calls cancel -> cancellation callback waiting to acquire lock -> Thread1 disposes of cancellation registration -> Dispose will never return as it’s waiting for callback that will never return -> deadlock
  + Don’t use SynchronizationContext, etc. to push work onto a different thread from callback (if you need the cancellation to run on a particular thread, you can request that when registering the callback).

## Synchronous Execution

### Wait and Result

### Task.WaitAll and Task.WaitAny

## Designing Your TPL Code

### Asynchronous vs. Multithreaded

* Thread-based tasks should generally only be used for compute-bound work.
  + Threads and the scheduler that manages them are limited resources
  + Using thread-based tasks for IO-bound work (reading from disk, waiting for an http response, etc.) needlessly multiplies threads that are just waiting. This is especially troublesome on server-side applications where many requests could amplify the problem by creating many “waiting” threads.

# Multithreading

## Thread Synchronization

### Synchronization Primitives

<https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives>

### Interlocked

### Monitor (Exclusive Lock)

#### Lock Statement

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement>

### Semaphore

### SemaphoreSlim

### Memory Barrier

### Auto Reset Event

# Language Integrated Query (LINQ)

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>

* Provides uniform, strongly-typed query functionality across various data sources (ex: objects in memory, relational databases, XML documents, file system, JSON documents, etc.)
  + These characteristics of LINQ make it quite powerful and appealing
  + Uniform: no need to learn an ever-increasing number of APIs for disparate data sources
  + Strongly-typed: allows for IntelliSense assistance and compile-time validity checking of queries reduces debug cycles
* Implemented as a set of **extension methods** on **IEnumerable<T>**

## Syntax

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/query-syntax-and-method-syntax-in-linq>

LINQ supports two syntaxes for creating queries: *query expression syntax* and *method syntax*. Not every LINQ operator is available using query expression syntax, so sometimes the method syntax is required (ex. Count, Take, Skip).

### Query Expression Syntax

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/query-keywords>

* Collection of C# keywords that may be composed with variables and parameters to create LINQ queries
  + *from, in, where, select, orderby, descending, ascending, group, into, join, let, on, equals, by*
* Very similar to SQL statements
* Always starts with the *from* clause and ends with a *select* or *group* clause
  + This order is reversed from SQL but it allows for IntelliSense throughout the query since the collection being operated on is known in the first clause
* var groupedDevelopers = from employee in employees

where employee.Type == EmployeeType.Developer

group employee by employee.Level;

### Method Syntax

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable>

* Method syntax refers to using the LINQ extension methods directly
* As each method accepts and often returns an IEnumerable<T>, these methods may be chained to create complex queries
* Many methods accept **functions** which may be supplied using **lambdas** providing logic to be used to filter, group, order, etc.
* var groupedDevelopers = employees.Where(e => e.Type == EmployeeType.Developer)

.GroupBy(e => e.Level);

## Query Execution

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/classification-of-standard-query-operators-by-manner-of-execution>

### Immediate Execution

* Some LINQ operators cause execution of the query to occur immediately. Examples of these are All, Any, Average, Count, ToList, ToArray, etc.

### Deferred Execution

* Many LINQ operators are built using the **yield return** operator which creates deferred (lazy) execution behavior in which the operators will not be executed until the returned IEnumerable<T> is enumerated (with a foreach, ToList, ToArray, etc.)
* The Remarks section of the MSDN documentation for each LINQ method will indicate if the method is implemented using deferred execution with a statement like this: “This method is implemented by using deferred execution. The immediate return value is an object that stores all the information that is required to perform the action. The query represented by this method is not executed until the object is enumerated either by calling its GetEnumerator method directly or by using foreach in Visual C# or For Each in Visual Basic.”

#### Streaming Operators

* LINQ operators that defer execution may be further subdivided into two categories: those that stream results and those that don’t.
* Streaming operators do not need to examine the entire collection to begin yielding results because each item will be included in or excluded from the resulting collection solely based on logic that may be executed on an item-by-item basis. For example:
  + var over60 = employees.Where(e => e.Age > 60);
  + In this query, an employee’s inclusion in the result set only depends on his or her age, so where may be implemented by yield returning employees that meet this criterion one at a time (streaming).
  + Where, Take, Cast, and Distinct are examples of LINQ operators that defer execution and stream their results.
* Other operators require examination of the entire collection before yielding results. For example:
  + var youngestToOldest = employees.OrderBy(e => e.Age);
  + In this query, an employee’s order depends not only on his or her age, but of the ages of all the other employees in the collection, so execution may be deferred, but not streamed.
  + OrderBy, GroupBy, and Join are examples of deferred operators that do not stream their results.

### Implications

#### Performance

* Deferred/streaming execution can result in very performant query execution in certain scenarios:
  + Suppose we have the following query:
    - var top3 = employees.Where(e => e.SomeComplexComputation() > 5.0)

.Take(3);

* + When this query is enumerated, it need only execute until 3 employees have the result of the SomeComplexComputation greater than 5. If say 3 of the first 10 out of a total collection of 10,000 employees met this requirement, the complex calculation would only occur 10 times because both the Where and Take methods utilize streaming execution.
  + Because IEnumerable<T> is a state machine and not necessarily an actual collection, it is possible to create an IEnumerable<T> that is theoretically infinite in length. To perform any sort of query on such a collection (or any sufficiently large collection) would require streaming operators as it would not be possible to process the entire collection before yielding results.
  + Understanding behavior can be quite powerful when trying to make efficient queries because chaining multiple methods that stream their results will result in an entire query that can defer logic execution and memory allocation that might otherwise be required.
  + The resulting data from these queries can also be consumed one item at a time instead of forcing the query to execute on the entire target collection before consumption can begin, which also may be taken advantage of when optimizing performance in user interfaces, etc.
* Deferred/streaming execution can also result in degraded performance if not managed correctly:
  + Suppose we have the following query:
    - var over5 = employees.Where(e => e.SomeComplexComputation() > 5.0);
  + And then we run the following code to display the results:
    - Console.WriteLine(over5.Count());
    - foreach(var x in over5){ Console.WriteLine(x.Name);}
  + In this case, the SomeComplexComputation method would be executed on each item in the collection TWICE since the over5 enumeration would need to execute to assess the count and again for the foreach loop. A more efficient alternative might look like this:
    - var over5Results = over5.ToArray();
    - Console.WriteLine(over5Results.Count()); // or .Length
    - foreach(var x in over5Results){ Console.WriteLine(x.Name);}
* Care should be taken when composing your LINQ queries and the enumeration of the results to ensure enumerations occur only as frequently as required.
* OPERATION ORDER MATTERS!
  + This is primarily true when executing queries on in-memory collections. For example:
  + youngestToOldestMen = employees.OrderBy(e => e.Age)

.Where(e => e.Gender == Gender.Male);

* + If this query were executed on an in-memory collection, the entire collection of employees would be sorted by age and then filtered to include only the men. It would be much more efficient to instead filter first and then sort this collection.

#### Error Handling

* Because the execution of a LINQ query might be deferred until the result is enumerated, care must also be taken when handling error conditions. Consider the following query:
  + var badIdea = employees.Where(e => e.SomeComplexComputation()/0.0 > 5.0);
* The above line of code would not throw an exception. The exception would only be thrown when the query was actually executed due to the enumeration of the badIdea query.
* Ensure appropriate try{} blocks targeting query function logic errors wrap code that will execute the query and not necessarily code that creates the query.

## Operations

### Filtering

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/filtering-data>

Filtering operations restrict the result sequence to elements that satisfy a specified condition.

#### Where

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where>

* Filters a sequence of elements based on a predicate (Func<T,bool> indicating inclusion in result sequence)
* May by used with query expression syntax:
  + where x.Name == “Bob” && x.Age > 65
* May be combined (behaves as if predicates were logically anded)

#### OfType

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.oftype>

* Filters a sequence of elements based on whether it can be cast to type TResult and returns an IEnumerable<TResult>

### Sorting

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/sorting-data>

Sorting operations orders the elements of a sequence based on one or more attributes.

#### OrderBy(Descending)

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderby>

* The OrderBy and OrderByDescending operators sort the returned IOrderedEnumerable<T> using the provided key selector function (Func<T, TKey>) to select the key by which to order the results.
* The IComparer<TKey> may be specified as an argument, otherwise the default comparer for the key type will be used to determine the sort order
* If the selected key does not implement IComparable and no IComparer argument is supplied, the order cannot be determined, and an exception will be throw when the query executes.
* These operators perform a stable sort – if the keys of two elements are equal, the order of the elements is preserved.
* Because IOrderedEnumerable<T> derives from IEnumerable<T>, multiple OrderBy methods may be chained together. If a second OrderBy is supplied after a previous OrderBy or OrderBy/ThenBy sequence, the second introduces a new primary ordering and the first will be ignored when re-ordering with the second (inefficient and unaffective).
* May by used with query expression syntax (ascending is default and need not be specified):
  + orderby [criterion] <ascending/descending>

##### ThenBy(Descening)

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.thenby>

* Accepts and returns an IOrderedEnumerable<T> which means it should only be used in conjunction an OrderBy or OrderByDescding operator but may be layered to provided several cascading order-by levels
* Follows the same rules as OrderBy
* Only executed if previous OrderBy/ThenBy executions yielded equal results (tie breaker)
* May by used with query expression syntax by comma-separating extensions of sort (ascending is default and need not be specified):
  + orderby [criterion1] <ascending/descending>, [criterion2] <ascending/descending>, [criterion3] <ascending/descending>, etc.

#### Reverse

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.reverse>

* Reverses the order of the elements in a sequence
* Not supported in method query syntax

### Quantifier

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/quantifier-operations>

Quantifier operations return a Boolean value indicating whether some or all of the elements in a sequence satisfy a condition.

#### All

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.all>

#### Any

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.any>

#### Contains

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.contains>

### Partitioning

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/partitioning-data>

Partitioning operations divide an input sequence into two sections without rearranging the element, and then return one of the sections.

#### Take

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.take>

* Returns a specified number of contiguous elements from the start of a sequence
* If the specified count exceeds the number of elements in the source, all source elements are returned.

#### TakeLast (.NET Core 3+/.NET Standard 2.1+)

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.takelast>

#### TakeWhile

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.takewhile>

* Returns elements from a sequence as long as the specified condition is true

#### Skip

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skip>

* Bypasses the specified number of elements in a sequence and then returns the remaining elements.
* Returns an empty sequence if the specified count is greater than the length of the underlying sequence.

#### SkipLast (.NET Core 3+/.NET Standard 2.1+)

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skiplast>

#### SkipWhile

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skipwhile>

* Bypasses the elements in a sequence as long as the specified condition is true and then returns the remaining elements.

### Element

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/element-operations>

Element operations return a single, specific element from a sequence.

#### ElementAt

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.elementat>

* Returns the element at the specified zero-based index in the sequence
* Throws an ArgumentOutOfRangeException if less than zero or greater than or equal to the number of elements in the source sequence

#### ElementAtOrDefault

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.elementatordefault>

* Returns the element at the specified zero-based index in the sequence or a default value if the index is out of range

#### First

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.first>

* Returns the first element in the sequence
* If a predicate is supplied, returns the first element in the sequence that satisfies the predicate
* If the sequence/predicated sequence contains no elements, an invalid operation exception is thrown

#### FirstOrDefault

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.firstordefault>

* Returns the first element in the sequence
* If a predicate is supplied, returns the first element in the sequence that satisfies the predicate
* If the sequence/predicated sequence contains no elements, the default value of T is returned

#### Last

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.last>

* Returns the last element in the sequence
* If a predicate is supplied, returns the last element in the sequence that satisfies the predicate
* If the sequence/predicated sequence contains no elements, an invalid operation exception is thrown

#### LastOrDefault

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.lastordefault>

* Returns the last element in the sequence
* If a predicate is supplied, returns the last element in the sequence that satisfies the predicate
* If the sequence/predicated sequence contains no elements, the default value of T is returned

#### Single

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.single>

* Returns the single element in the sequence that contains only one element
* If a predicate is supplied, returns the single element in the sequence that satisfies the predicate
* If the sequence/predicated sequence contains no elements, an invalid operation exception is thrown
* If the sequence/predicated sequence contains more than one element, an invalid operation exception is thrown

#### SingleOrDefault

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.singleordefault>

* Returns the single element in the sequence that contains only one element
* If a predicate is supplied, returns the single element in the sequence that satisfies the predicate
* If the sequence/predicated sequence contains no elements, the default value of T is returned

If the sequence/predicated sequence contains more than one element, an invalid operation exception is thrown

### Projection

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations>

Projection operations transform an object into a new form.

#### Select

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select>

#### SelectMany

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany>

### Join

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/join-operations>

Join operations combine elements from two data sources that share a common attribute.

#### Join

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.join>

#### GroupJoin

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupjoin>

### Grouping

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/grouping-data>

Grouping operations put elements into groups based on some common attribute.

#### GroupBy

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby>

#### ToLookup

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.tolookup>

### Generation

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/generation-operations>

Generation operations create a new sequence of values from a source sequence.

#### DefaultIfEmpty

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.defaultifempty>

#### Empty

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.empty>

#### Range

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.range>

#### Repeat

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.repeat>

### Equality

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/equality-operations>

Equality operations return true if each input sequence has the same number of elements and equal corresponding elements.

#### SequenceEqual

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.sequenceequal>

### Set

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/set-operations>

Set operations produce a result set based on the presence or absence of equivalent elements within the same or separate collections (or sets).

#### Distinct

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinct>

#### Except

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.except>

#### Intersect

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.intersect>

#### Union

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.union>

### Concatenation

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/concatenation-operations>

Concatenation operations append one sequence to another.

#### Concat

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.concat>

### Aggregation

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/aggregation-operations>

Aggregation operations compute a single value from a collection of values.

#### Aggregate

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.aggregate>

#### Average

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.average>

#### Count

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.count>

#### LongCount

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.longcount>

#### Max

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.max>

#### Min

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.min>

#### Sum

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.sum>

### Converting

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/converting-data-types>

Conversion operations change the type of the input sequence or elements.

#### AsEnumerable

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.asenumerable>

#### AsQueryable

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.queryable.asqueryable>

#### Cast

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.cast>

#### OfType

See **OfType**

#### ToArray

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.toarray>

#### ToDictionary

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.todictionary>

#### ToList

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.tolist>

#### ToLookup

See **ToLookup**

## LINQ Providers

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/enabling-a-data-source-for-linq-querying1>

When LINQ queries are executed against in-memory collections that implement IEnumerable<T>, it is easy enough to understand how the LINQ extension methods implement query functionality directly, but what about when LINQ queries are run against remote data sources? It would be quite inefficient to pull entire tables from a relational database only to execute some extension methods locally to filter that data down to a few rows-worth of information.

LINQ providers enable LINQ queries to be translated into efficient native queries that may be executed on remote data sources, but there are some things to keep in mind:

* Because all database types may not support corresponding query syntax for all LINQ operators and all expressions (and the permutations thereof), some queries will still be executed (or partially executed) in memory, or worse, may throw an exception if it doesn’t know how to translate your query.
  + Ex: var someQuery = data.Where(d => d.StringProp.Split(‘|’)[2] == “50”);
    - Most databases (and providers) won’t know how to deal with something like this.
    - If you need to execute something like this, the most efficient way would be to do as much remote-able query logic (filtering, sorting, etc. that can be handled by the database) before the non-remote-able a stuff, then force use of the in-memory (IEnumerable-based) LINQ operators by calling an immediate execution method like **ToList**.
  + Not all providers are created equal: even if a database supports operations that should be convertible from LINQ expressions, the provider implementor may not have include support for them.
  + Many LINQ providers provide a way to emit their generated queries so that you can verify their correctness and efficiency.

This translation is accomplished with the help of a set of interfaces and classes deriving from **IQueryable** and **Expression**.

### IQueryable/IQueryable<T>

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable>

* Implements IEnumerable/IEnumerable<T>
  + Like IEnumerable, it represents a data source that may be enumerated
  + Unlike IEnumerable, it does not require execution to occur in memory
* LINQ query methods on IQueryables accept predicates of type **System.Linq.Expression.Expression<Func<T, bool>>** instead of Func<T, bool>. These expressions are no longer invokable directly, but rather expose a data structure (expression tree) that may be translated into a remote query language (like SQL).
* In addition to IEnumerable members, exposes 3 properties:
  + **IQueryProvider Provider** – the query provider associated with the underlying data source
  + **Expression Expression** – expression tree associated with the IQueryable instance
  + **Type ElementType** – type of elements returned when expression tree is executed
* LINQ query methods on IQueryables still offer deferred execution so the same care should be taken when creating and using your queries to avoid redundant database calls, etc.

### Expression/LambdaExpression/Expression<TDelegate>

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.expression-1>

Expression<TDelegate> represents a strongly-typed lambda expression as a data structure in the form of an expression tree.

* May be assigned directly from a lambda expression
  + When this happens, the compiler emits instructions to build an expression tree
  + This expression tree is an in-memory representation of the lambda expression logic
* Not executable directly
  + May be compiled into executable code using one of the compile methods
* This data structure form enables inspection, transformation, and custom processing of expression trees to convert them into remote query operations (like SQL statements).

# Reflection

# Concurrency

# Dynamic

## ExpandoObject

## DynamicObject

# Date and Time