**Encapsulation 🡪** This means having a boundary around an object to separate external implementation with internal implementation.

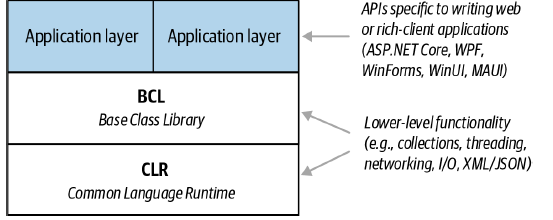
**Unified Typing System** 🡪 All objects in C# be it primitive or user defined ones share a common functionality from *Type.*

**Interface** 🡪 An interface only describes the behavior and it should separate between specification and implementation allowing multiple inheritance.

**Type Safety** 🡪 C# is type safe language meaning objects should interact only via defined protocol i.e.. C# prevents us from interacting with *string type* as though we were interacting with *integer type*. C# also supports static typing i.e.. it enforces type safety at compile time. We can’t call a method or function with string parameter when this function only accepts integer parameter.

**Memory Management** 🡪 C# relies on CLR to automatically destroy objects that are no longer referenced by reclaiming them.

**Common Language RunTime** 🡪 For rich client applications libraries in application layer exists in the run time. CLR also provides automatic garbage collection and exception handling. C# compiles code to managed code or IL (intermediate language) code.



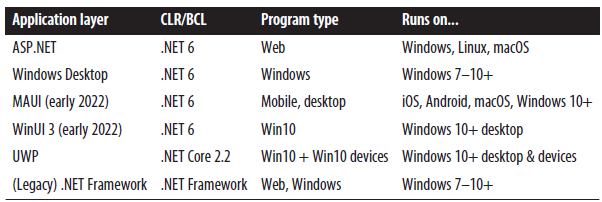
CLR converts such IL code into native code for the machine such as X64 or X86 just before execution. This is called *Just In time Compilation*. To improve start up time for large assemblies or resource constrained devices *Ahead of time compilation* is also available.

The container of managed code is called *assembly*. An assembly can contain metadata that describes *type information.* Having metadata inside the assembly helps other assemblies to directly reference the types in other assemblies without the need for additional files.

A program can query its own metadata and even generate new IL at runtime.

*Base Class Library*: CLR contains set of assemblies called Base Class library that provide functionality like input/output, enumeration, collection, text processing, XML/JSON handling, networking, encryption, interop, concurrency and parallel programming.

Runtime: A runtime is a deployable unit that we can download and install. A runtime contains BCL and optional application layer (that depends on the type of application like web, rich client, mobile etc,.) For a simple console application or non UI library we don’t need an application layer).



.Net 6 🡪 After .Net framework Microsoft started with .Net Core 1, 2 and 3 and didn’t release 4 (to avoid confusion with .NET framework 4.x) and released .Net 5. So, assemblies compiled under .Net core 1, 2, 3 and .Net 5 work comfortably in .Net 6. But assemblies compiled under .Net framework doesn’t are not compatible with .Net 6.

MAUI 🡪 (Multi-platform App UI) is designed for creating mobile apps for iOS, Android as well as cross platform desktop apps for macOS and Windows.

UWP and WinUI 3 🡪 It is designed for writing immerse touch first mobile apps that run on windows 10+ desktop and devices like Xbox, Surface Hub and HoloLens. UWP (Universal Windows Platform) apps are sandboxed and shipped via windows store. UWP is preinstalled on Windows 10.

.Net Framework 🡪 This runtime works on Web based (windows server) and windows based console applications.

**Compilation:** C# compiles the source code and converts to managed code or IL code in an *assembly*. Assembly can either be a console application or library (that can be referenced by other applications). .NET itself is a set of libraries (as well as runtime environment)

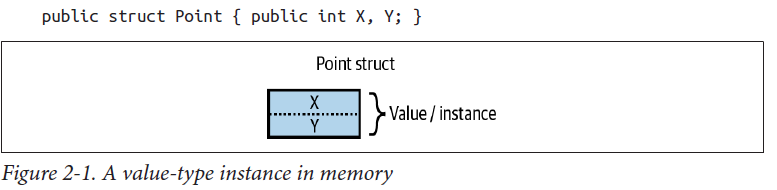
Unlike .Net Framework, .Net 6 assemblies doesn’t have .exe extension. The exe that we see after building the .Net 6 application is platform specific native loader responsible for starting our application’s .dll assembly.

.Net 6 allows us to create self contained deployment that includes loader, assemblies and .Net runtime – all in single .exe file.

**Instance versus Static Members 🡪** The data members and function members that operate on that instance of the type are called *instance* members. Data members that don’t operate on the instance of the type are called *static* members.

**C# Types 🡪** Value Types, Reference Types, Generic Types, Pointer Types.

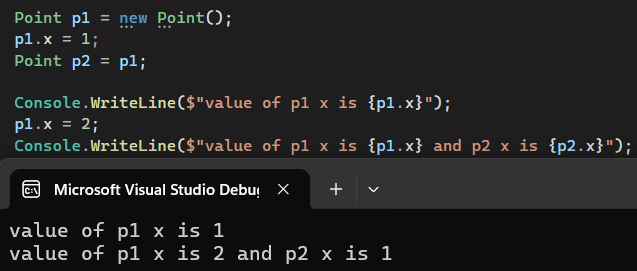
Value Types 🡪 Built in numeric types, char type and Boolean types. The content of a *value type* variable or constant is simply a value. We can define a custom value type by using *struct*.



The assignment of a value type instance always copies the instance. For example:

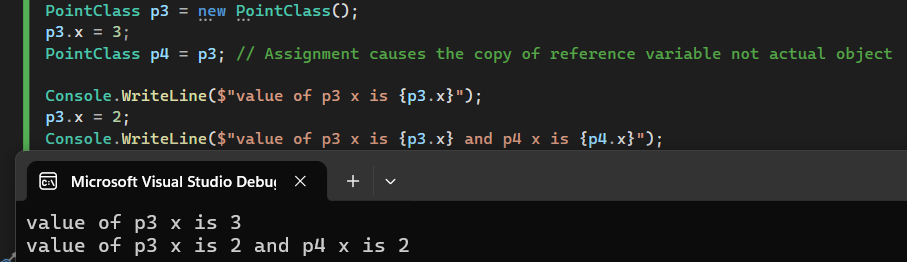
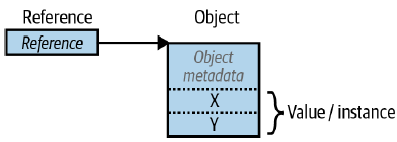
Point P1 = new Point();

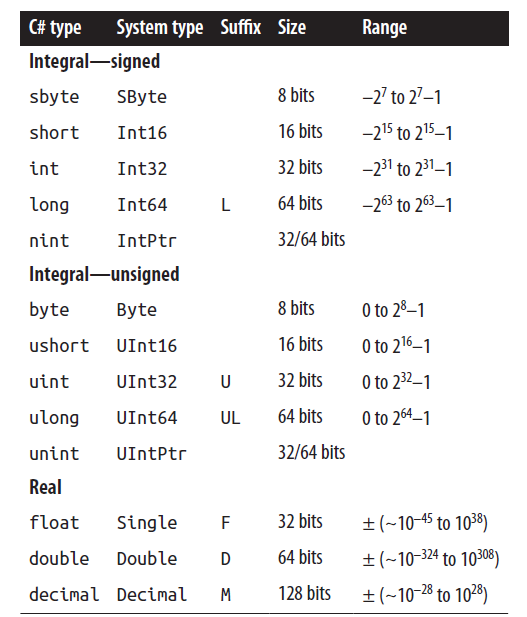
Point P2 = P1;



Reference Types 🡪 Delegate, Interface, class, string, arrays.

A reference type has two parts: object (in heap) and reference variable (in stack). The content of the reference variable the *reference* to the actual object.





**Memory Allocation**:

struct A { byte b; long l; }

Here’s a step-by-step explanation:

1. **Field Sizes**:
   * byte b: A byte is 1 byte.
   * long l: A long is 8 bytes.
2. **Alignment and Padding**:
   * Structs are typically aligned to the size of their largest field, which in this case is long (8 bytes).
   * To ensure proper alignment, the runtime may insert padding bytes between fields.

For the struct A:

* + The byte b will be stored at the start of the struct.
  + After the byte, there will be 7 bytes of padding added to align the long field properly on an 8-byte boundary.

1. **Memory Layout**:
   * byte b takes 1 byte.
   * 7 bytes of padding are added after byte b.
   * long l takes 8 bytes.

So, the total size would be:

* + 1 byte (for b) + 7 bytes (padding) + 8 bytes (for l) = 16 bytes.

**Summary**: The struct A will be allocated 16 bytes of memory. This includes padding to ensure proper alignment of the long field.

Class A { byte b; long l; }

Here’s a breakdown of the memory considerations:

1. **Field Sizes**:
   * byte b: 1 byte
   * long l: 8 bytes
2. **Object Overhead**:
   * Classes in C# include overhead for the object header, which typically includes information such as type information and synchronization data. This overhead is usually 8 or 16 bytes, depending on the platform (32-bit or 64-bit).
3. **Alignment and Padding**:
   * Memory for fields in a class is also aligned to the largest field, which is 8 bytes for long in this case.
   * The byte b will be placed first, followed by padding to align the long field properly.
   * After byte b, there will be 7 bytes of padding before the long l.
4. **Memory Layout**:
   * Object header overhead: 8 or 16 bytes
   * byte b: 1 byte
   * Padding: 7 bytes
   * long l: 8 bytes

Total memory required for fields:

* + byte b + padding + long l = 1 byte + 7 bytes + 8 bytes = 16 bytes.

So, the size of the fields alone is 16 bytes, but the actual memory allocated for the class object will be larger due to the object header.

**Summary**: On a 64-bit platform, the memory allocated for an instance of class A will typically be 24 bytes:

* 16 bytes for the fields (including padding),
* 8 bytes for the object header (assuming typical 64-bit alignment).

On a 32-bit platform, the memory might be slightly different, often 16 bytes for fields plus 8 bytes for the object header, totalling 24 bytes as well. However, the actual size may vary slightly depending on the runtime implementation and optimizations.

**Type Inference**: By default, the compiler *infers* a numeric literal to be either double or an integral type. If a literal contains decimal point or exponential symbol (E) it is double otherwise it is either int, uint, long and ulong.

Decimal type needs base 10 arithmetic and high precision and hence it is preferred for financial calculations.

 Without letter F this line won’t compile as 4.5 will be inferred by C# as double and it thinks we are trying to assign a double value to float value. 

Same goes with decimal literal value it should end up with letter M.

**Strings and Characters**

A char in C# can store a single character in Unicode format that represents 2 bytes (UTF-16).

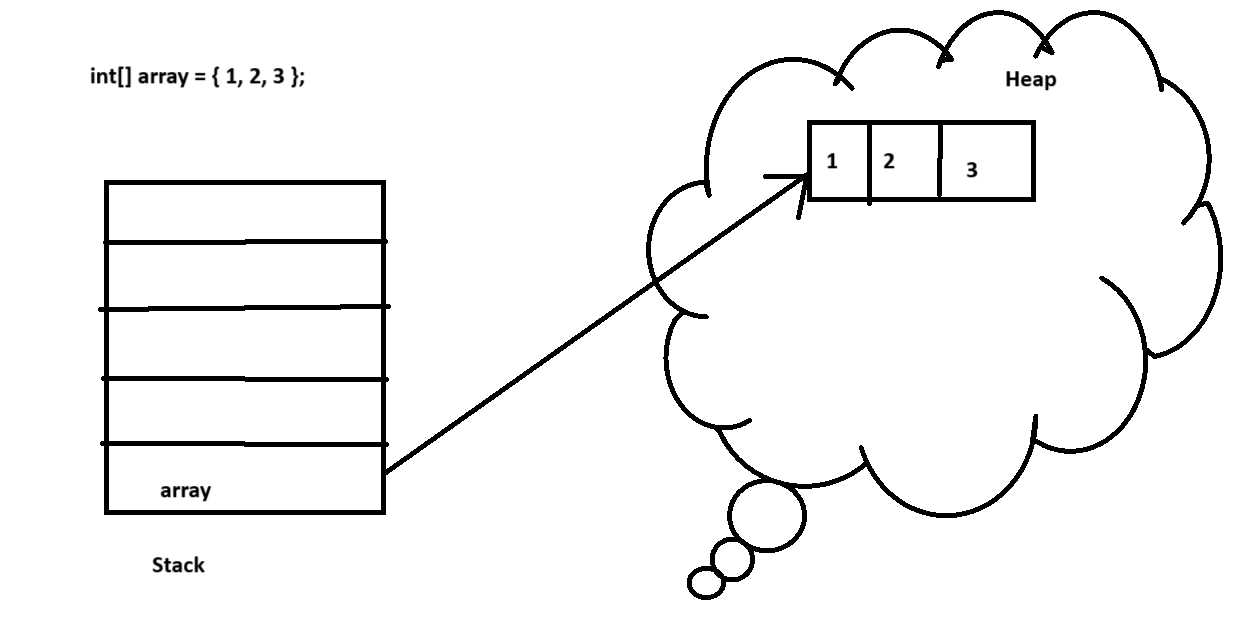
A string is sequence of Unicode characters that represents an immutable (unmodifiable) sequence of Unicode characters. Strings are reference type but their equality operators however follow value-type semantics.

**Arrays**

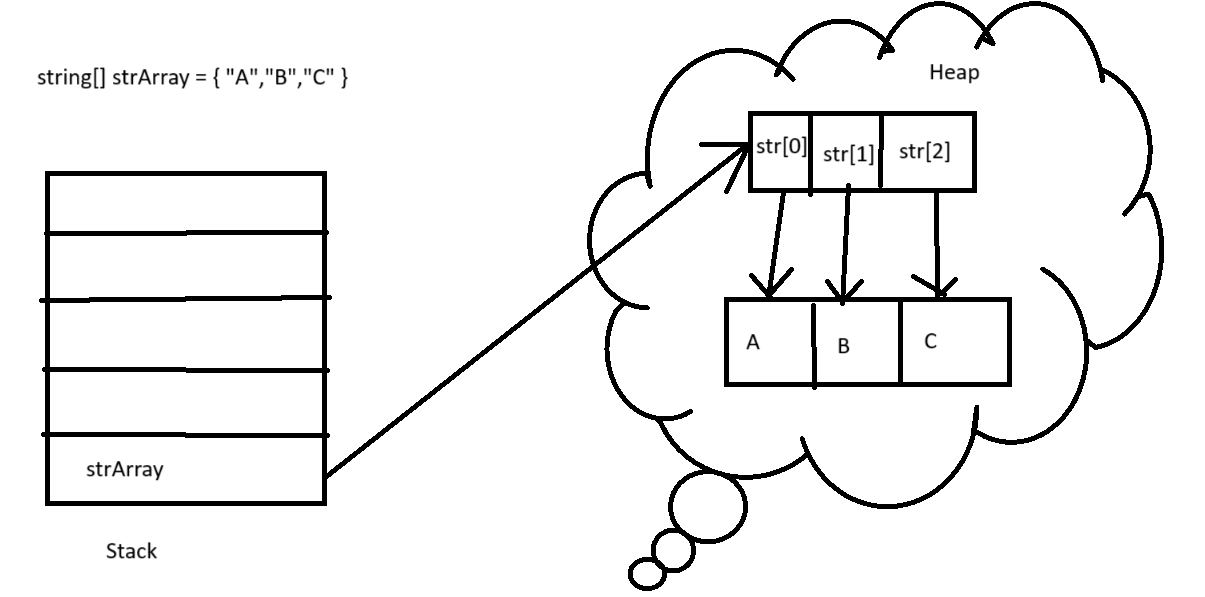
An array represents fixed number of elements of particular type. Elements in the array are stored in contiguous block of memory for high efficient access. Creating an array always pre initializes the elements with default values. For example: int[] a = new int[1000]; would allocate default values of 0 for 1000 integers in contiguous block of memory.

An array initialization expression let’s us declare and initialize the single statement: char[] a = {‘a’, ’b’, ’c’}

Value Type 🡪 As array is reference type and when it is allocated with value types then reference variable will be stored in stack and points to ‘actual’ array in the heap. The ‘actual’ array in the heap contains actual values.



Reference Type 🡪 When an array of reference types are allocated in an array then the ‘actual’ array in the heap contains references that point to actual objects in the heap.



Indices 🡪 They let us access the array from the end like

char[] vowels = new char[] {'a','e','i','o','u'};

char lastElement = vowels [**^1**]; // 'u'

char secondToLast = vowels [**^2**]; // 'o'

^0 equals the length of the array so vowels[^0] throws error.

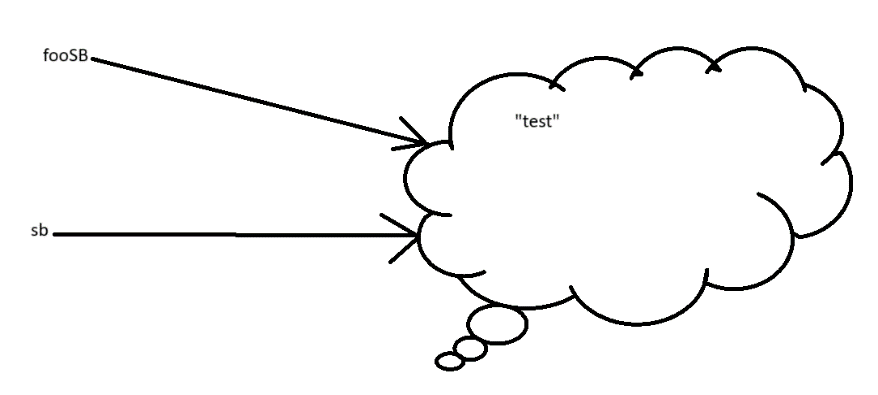
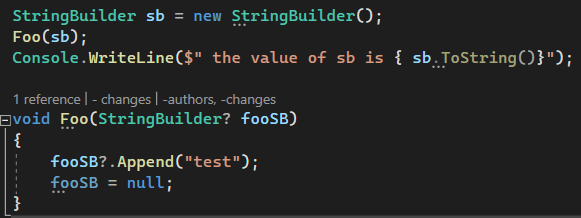
**Rectangular arrays** are declared using commas to separate each dimension. The

following declares a rectangular two-dimensional array for which the dimensions

are 3 by 3:

int[,] matrix = new int[3,3];

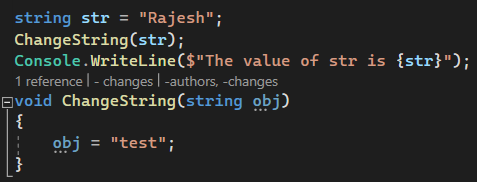
**Passing arguments by value**: In the below code, the value of sb will be “test” because “sb” will be sent as a copy to ‘Foo’ method.



Inside the method the reference ‘fooSB’ points to the same object pointed by ‘sb’. So, ‘fooSB’ updates the object with “test” and “sb” will be “test”.

Passing a value type or reference type only passes the copy of the value.

In below example , the value of “str” will be printed as “Rajesh” not “test”. The value of str remains "Rajesh" because strings are immutable, and when you pass a string to a method, only a copy of the reference is passed. Changing the local parameter obj to point to a new string does not alter the original string or the original reference. This behavior demonstrates that the original variable str remains unaffected by the reassignment within the method.



**“Out” Modifier:** When you pass an argument by reference, you alias the storage location of an existing variable rather than create a new storage location**.** An *out* argument is like a *ref* argument it is also passed by reference except for the following:

• It need not be assigned before going into the function.

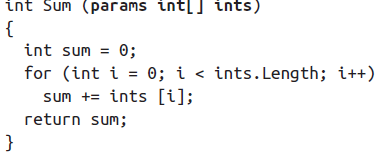
• It must be assigned before it comes *out* of the function.

When calling methods with multiple out parameters, sometimes you’re not interested in receiving values from all the parameters. In such cases, you can “discard”the ones in which you’re uninterested by using an underscore:

Split ("Stevie Ray Vaughan", out string a, **out \_**); // Discard 2nd param

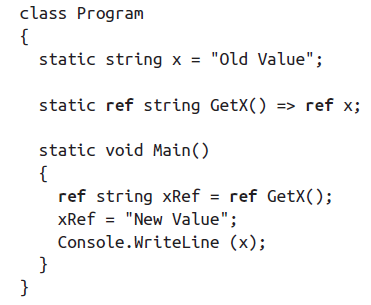
In this case, the compiler treats the underscore as a special symbol, called a *discard*.

**“In” modifier:** An *in* parameter is similar to *ref* parameter in that method shouldn’t modify the value else compile time error will be generated. This modifier is most useful when passing large value data types to the method because it allows compiler to avoid the overhead of copying the argument prior to passing it.

**“Params modifier”:** This modifier when applied to last parameter of the method can accept any number of arguments of a particular type. 

**Optional Parameters:** These can’t have ref or out. These parameters must appear at the end after other mandatory parameters for a method.

**Named Arguments**: Rather than identifying an argument by position, you can identify an argument by name: Foo (**x:1, y:2**).

**Ref Return**: 

**Var:** If the compiler is able to identify the type from the initialization expression that we can use ‘var’ keyword. Implicitly typed variables are of static typed. 

**Target-Typed new Expressions:** From C# 9 we can use ‘new’ keyword to create objects by not specifying the type name if the compiler is able to unambiguously infer it.

**Operators**:

**1. Arithmetic Operators**

These operators perform basic mathematical operations:

* **Addition (+)**: Adds two operands. **int sum = 5 + 3; // 8**
* **Subtraction (-)**: Subtracts the second operand from the first. **int difference = 5 - 3; // 2**
* **Multiplication (\*)**: Multiplies two operands. **int product = 5 \* 3; // 15**
* **Division (/)**: Divides the first operand by the second. **int quotient = 6 / 3; // 2**
* **Modulus (%)**: Returns the remainder of the division of the first operand by the second. **int remainder = 5 % 3; // 2**

**2. Relational (Comparison) Operators**

These operators compare two operands and return a Boolean result (true or false):

* **Equal to (==)**: Checks if two operands are equal. **bool isEqual = (5 == 3); // false**
* **Not equal to (!=)**: Checks if two operands are not equal. **bool isNotEqual = (5 != 3); // true**
* **Greater than (>)**: Checks if the first operand is greater than the second. **bool isGreater = (5 > 3); // true**
* **Less than (<)**: Checks if the first operand is less than the second. **bool isLess = (5 < 3); // false**
* **Greater than or equal to (>=)**: Checks if the first operand is greater than or equal to the second**. bool isGreaterOrEqual = (5 >= 3); // true**
* **Less than or equal to (<=)**: Checks if the first operand is less than or equal to the second. **bool isLessOrEqual = (5 <= 3); // false**

**3. Logical Operators**

These operators are used for combining boolean expressions:

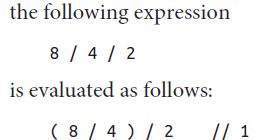
* **Logical AND (&&)**: Returns true if both operands are true. **bool result = (5 > 3) && (2 < 4); // true**
* **Logical OR (||)**: Returns true if at least one of the operands is true. **bool result = (5 > 3) || (2 > 4); // true**
* **Logical XOR (^)**: Returns true if exactly one of the operands is true. **bool result = (5 > 3) ^ (2 > 4); // true**
* **Logical NOT (!)**: Negates the boolean value of the operand. **bool result = !(5 > 3); // false**

**4. Bitwise Operators**

These operators perform operations on the binary representations of integer types:

* **Bitwise AND (&)**: Performs a bitwise AND operation. **int result = 5 & 3;** // 1 (binary: 0101 & 0011 = 0001)
* **Bitwise OR (|)**: Performs a bitwise OR operation. **int result = 5 | 3;** // 7 (binary: 0101 | 0011 = 0111)
* **Bitwise XOR (^)**: Performs a bitwise XOR operation. **int result = 5 ^ 3;** // 6 (binary: 0101 ^ 0011 = 0110)
* **Bitwise complement (~)**: Inverts all the bits of the operand. **int result = ~5;** // -6 (binary: ~0101 = 1010, in two's complement form)
* **Left shift (<<)**: Shifts bits of the first operand to the left by the number of positions specified by the second operand. **int result = 5 << 1;** // 10 (binary: 0101 << 1 = 1010)
* **Right shift (>>)**: Shifts bits of the first operand to the right by the number of positions specified by the second operand.

**int result = 5 >> 1**; // 2 (binary: 0101 >> 1 = 0010)

**Operator precedence:** *Binary operators* (except for assignment, lambda, and null-coalescing operators) are *left-associative*; in other words, they are evaluated from left to right. 

The *assignment operators* as well as the lambda, null-coalescing, and conditional operators are *right-associative.*

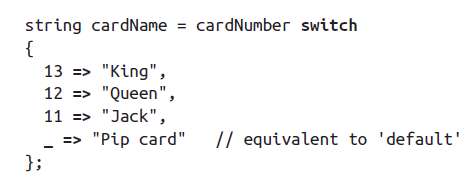
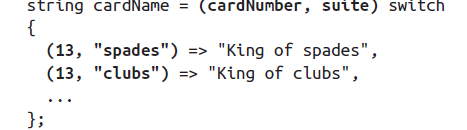


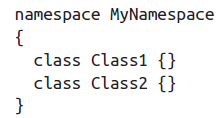
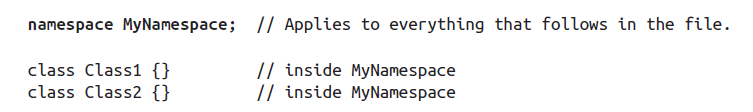
**Null Operators**

**Null Coalescing Operator:** The ?? operator is called Null Coalescing operator. It means “If the operator to the left of ?? is non null then give it to me otherwise give me another value”.

**Null Coalescing Assignment Operator:** The ??= operator says that “if operand to the left is null then assign the right operand to the left operand” myVariable ??= someDefault; This is equivalent to: *if (myVariable == null) myVariable = someDefault;* This operator is useful in implementing lazy loading.

**Null Conditional Operator:** The ?. operator is the *null-conditional* or Elvis operator. It allows you to call methods and access members just like the standard dot operator except that if the operand on the left is null, the expression evaluates to null instead of throwing a NullReferenceException

**Switch Expression: ** ****

**File Scoped Namespaces:** ****this can be written like this : 

**Delegates:** Using Delegates functions can be passed as values.

* A delegate is an object that points to a method and knows to call that method.
* It has a return type and parameter types.
  + Example : delegate int Transformer(int x)

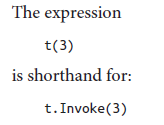
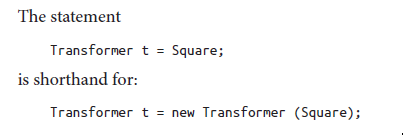
The above delegate takes int parameter and returns an int type. So it can point to any method that takes one int parameter and returns int type like this one 🡪

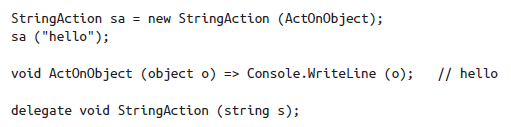
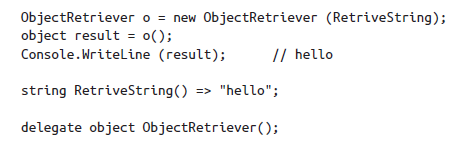
int square (int x) => x \* x;

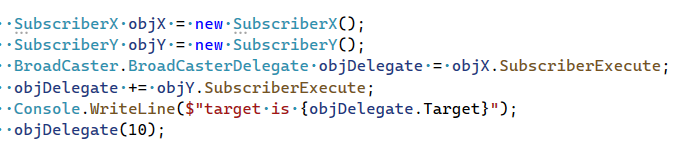
Transformer objDelegate = square ; // assigning a method to delegate variable creates delegate instance

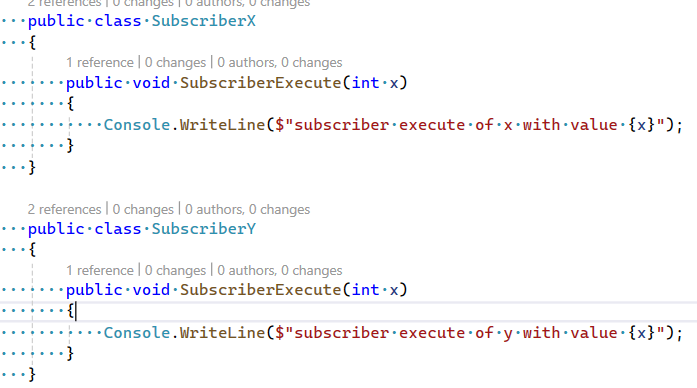
objDelegate(3); // invokes the instance.

A delegate instance acts as a delegate by calling the target method. In this way the caller is loosely coupled from the target method which the caller wants to call.



* *Contravariance*: When we are passing more specific types as argument than asked for to the parameters of the method. A delegate can have more specific parameter types than it’s target method. 
* *Covariance*: When we get more specific return type than we asked for. A delegate’s target method can return more specific return type than described by the delegate. 
* Broadcaster and Subscriber pattern:

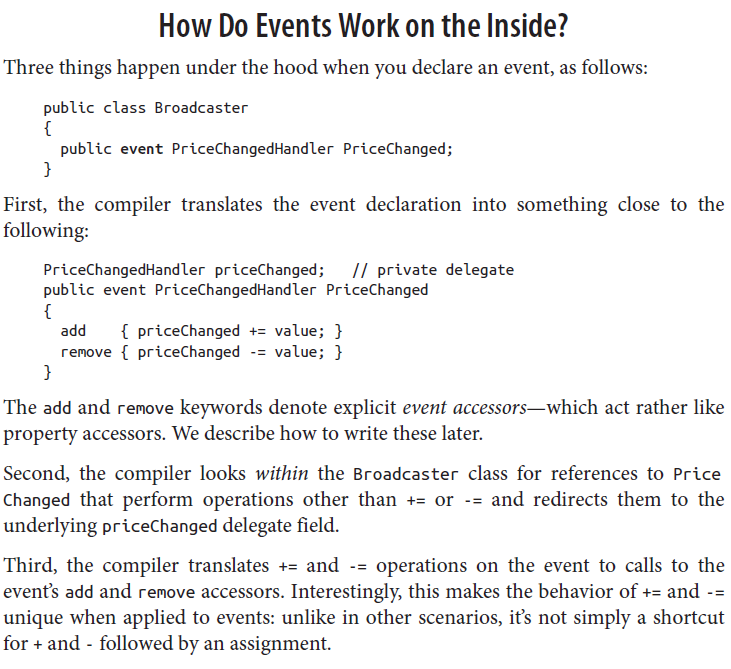




The problem with above code is subscribers can do the following:

* Replace other subscribers by reassigning the ‘objDelegate’
* Clear all subscribers by assigning objDelegate to null
* Broadcast to all subscribers by invoking the delegate

**Events:**

* An event is a construct that exposes subset of delegate features. The main purpose of events is to prevent subscribers from interfering with one another.
* If we add ‘event’ word before instantiating a delegate it becomes an event.
* System.EventArgs 🡪 A base class that contains no members except ‘Empty’ that coneys information about an event.

**Anonymous Methods**:

* These begin with key word ‘delegate’ 

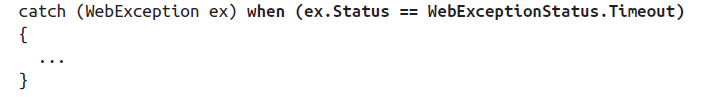
**Try/Catch/Finally:**

* A black and white text

  Description automatically generated

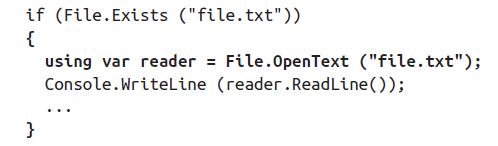
A white background with black text

Description automatically generated

**ExceptionFilters 🡪** A catch clause with a when clause 

This means that when WebException is thrown and if the status of the exception is ‘timeout’ only then webexception will be caught.

Using Declaration 🡪 If we omit brackets and statement block following a using statement (C# 8 +) it becomes *using declaration*.



In this case the resource is disposed when the execution comes out the *if* block.

**Return vs yield return** **vs yield break:**

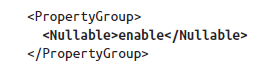
* return statement expresses 🡪 Here’s the value you asked me to return from this method
* yield return expresses 🡪 Here’s the next element you asked me to yield from the enumerator. On each “yield return” control is transferred back to the caller maintaining the callee’s state till the end of enumeration.
* yield break 🡪 It says to exit the enumeration at the moment it spots this statement without completing it.

**Nullable** **Value Types:**

* Nullable value types work particularly well with ?? operator. This is called ‘Null Coalescing operator’, it will return the first non null value.

**Nullable Reference Types**:

* The Nullable reference types enforce non-nullability with the purpose of avoiding NullReferenceException.



**Extension Methods:**

* These allow an existing type to be extended with new methods to extend the functionality. An extension method is a static method of a static class, where the *this* modifier is applied to the first parameter.
* Any compatible instance method takes precedence over extension method.
* If two extension methods have same signature then any extension method could used as an ordinary static method to avoid ambiguity.
* If two extension methods have same signature then the one with more specific arguments will take precedence.

**Anonymous Type:** var sometype = new { Name=”Rajesh” , Age = 39 }, when the compiler sees this line it automatically generates below code :

internal class AnonymouslyGeneratedTypeName

{

private string name; // Actual field name is irrelevant

private int age; // Actual field name is irrelevant

public AnonymousGeneratedTypeName (string name, int age)

{

this.name = name; this.age = age;

}

**}**

public string Name { get { return name; } }

public int Age { get { return age; } }

// The Equals and GetHashCode methods are overridden (see Chapter 6).

// The ToString method is also overridden.

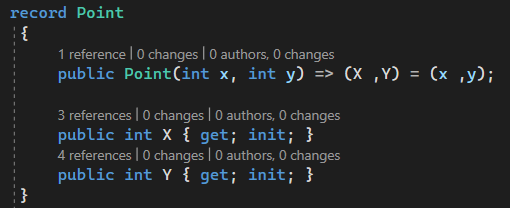
}

...

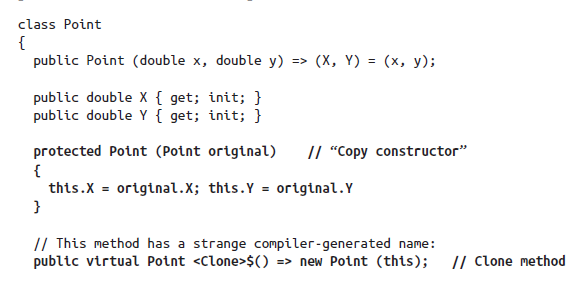
**Records:** These are special type of class or struct that are designed to work with immutable (once created can’t be modified) data. It’s most useful feature is *nondestructive mutation*.

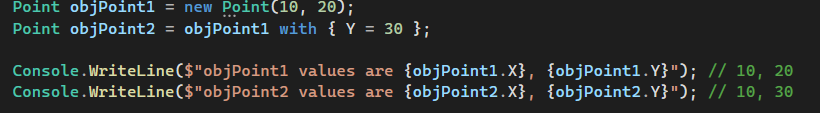
* In order to modify immutable types (whose fields can’t be modified after initialization) we must create a new one and copy over the data along with our modifications.
* In simple cases they eliminate boilerplate code while honoring equality semantics.
* If an object is immutable then it means that it’s identity cannot change and such types can implement structural equality than referential equality.
* **Non Destructive Mutation:** The most important step that compiler does is write a copy constructor and a hidden clone method. This enabled non destructive mutation via the ***with*** keyword.

If we write a record like below :



C# compiler converts it to a class like below :

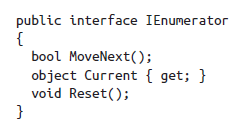


****

Here the copy constructor clones the record and new variables are assigned with init accessors.

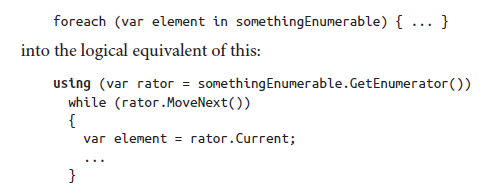
**Collections**:

* IEnumerator 🡪 This interface gives us the ability to traverse through a collection.



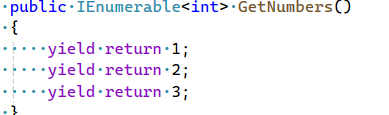
* MoveNext() 🡪 moves the cursor to the next element in the collection , it returns false if the collection is empty.
* Current 🡪 it returns the element at current position (usually cast from object to more specific type)
* Reset() 🡪 It moves the cursor to the initial position (start position) allowing the collection to be enumerated again.

IEnumberable<T> derives from IDisposable which means that enumerators hold references to resources such as database connections and ensure that resources are released when enumeration is complete.



Yield return 🡪

* returns each element of a collection one at a time.
* When a method that contains ‘yield return’ is called it returns an iterator object without actually executing the body of the method.
* The execution of the method is paused when it encounters yield return statement and resumed the next time the iterator is called.

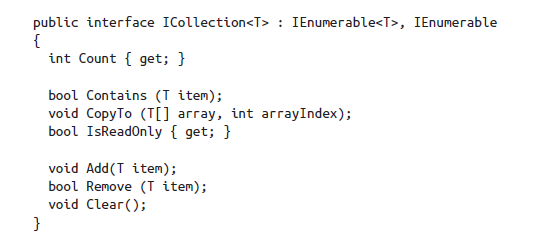


The method GetNumbers() returns 1,2 and 3 one at a time.

When GetNumbers() method is called in foreach loop, it returns an IEnumerable<int>.

The foreach loop calls the iterator, which runs up to first ‘yield return 1’ and yields 1 to the loop.

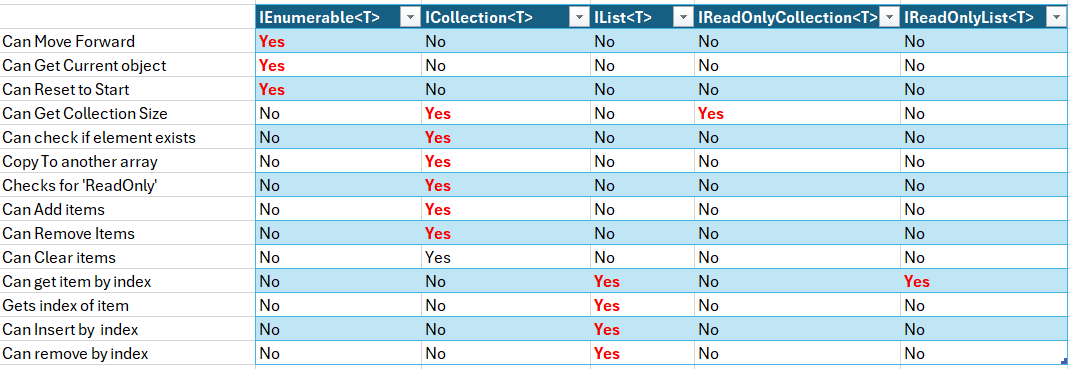
***ICollections and IList:***

* IEnumerable only provides forward only iteration but doesn’t provide getting size of collection, search or modify the collection , access a member by index. These will be provided by ICollection, IList or IDictionary.
*  A screenshot of a computer code

  Description automatically generated

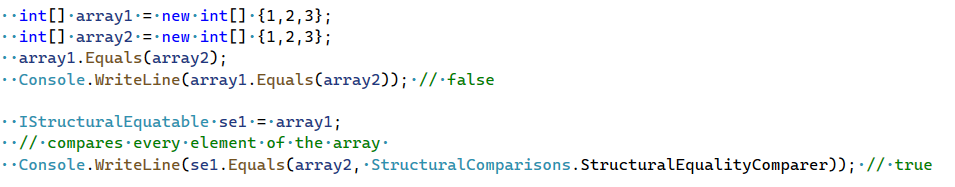
A screen shot of a computer code

Description automatically generated



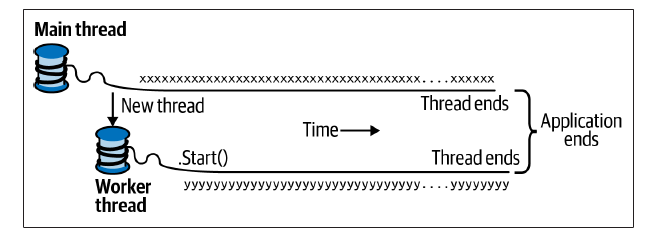
**Arrays**:

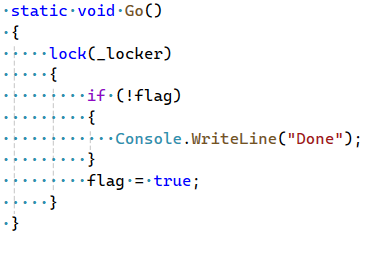
* It’s a fixed length collection i.e.. once created the size of the array cannot be changed.
* Arrays are generally more performant than lists because they provide direct access to elements via index.
* Arrays are memory-efficient since the memory for elements is allocated contiguously. This can lead to better performance.
* Arrays are strongly typed i.e.. the type of elements they store is defined at compile time.



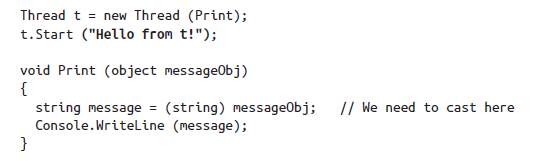
**Concurrency**:

* A thread is an execution path that can proceed to work independently of others.
* With single thread, just one thread runs in the process’s isolated environment where as in multithreaded environment multiple threads in the process sharing resources like memory.

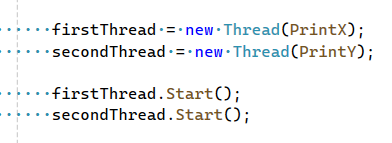


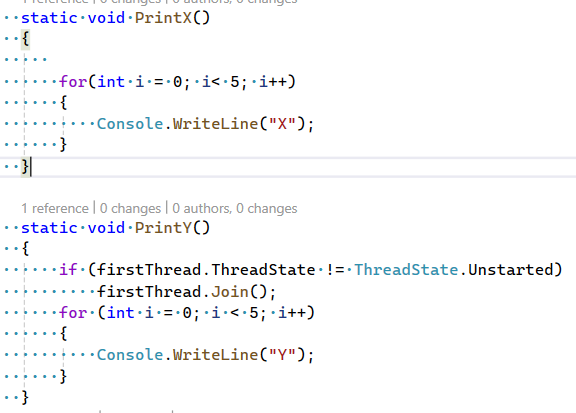
* Static fields can be shared between all the threads in the application domain.
* When threads share data it is possible to have either errors or no thread safety code.
* **Locks:** When more than one thread encounters the lock (which is a reference type object) , one thread waits or blocks until the lock becomes available. “Done” will be printed only once as first thread enters and makes flag true meaning other thread can’t print “Done”.

**Example:** A shared in-memory cache for frequently accessing database objects in an application is best & safe place for using locks without any deadlock happening.

* Passing data to thread: 
* **BackGround threads**:
  + By default threads are foreground threads.
  + Foreground threads keep the application alive as long as anyone of them is running where as background threads do not.
  + After all foreground threads completes, the application ends and any background threads will abruptly terminate.
* **Joins** :
* A thread can wait for another thread to end by calling it’s join method.
* Join is a synchronization method that blocks the calling thread (i.e.. the thread that calls the ‘join’ method for another thread) until the thread whose join method is called has been completed.

Example:





Output:

 Here ‘X’ will be printed first followed by ‘Y’ because when secondThread goes into PrintY() method it is asked to wait for the completion of the firstThread and then join it till then secondThread goes to blocked state

* Sleep 🡪 Thread.Sleep blocks the thread for the time mentioned. Thread.Sleep(0) relinquishes the current time slice immediately voluntarily handling over the CPU to other thread. Thread.Yield() does the same but it is relinquishes only to the threads on the same processor.
* **Thread Pool:**
  + When a new thread is created few microseconds are spent on creating isolated memory stack for the thread.
  + Thread pool cuts this overhead by having a pool of pre-created recyclable threads.
  + Thread pool is quite essential for achieving parallel programming and fine-grained concurrency.
  + Pooled threads are always background threads.
  + *Thread.CurrentThread.IsThreadPoolThread* informs where the current thread is pooled or not.
  + The easiest way to run quickly on a pooled thread is by using this 🡪 Task.Run

Example : Task.Run(() => Console.WriteLine(“Task is running”);

Before .Net 4.0 we used to have :

ThreadPool.QueueUserWorkItem(notused => Console.WriteLine(“Hello World”));

**Tasks:**

* A thread is a low level tool for concurrency but it has few limitations:
  + It can be easy to pass data to a thread but it is difficult to get the “return value” from a thread that we wish to “Join”. And if the operation throws an exception, it is equally painful to catch & propagating the exception back.
  + Secondly we can’t tell a thread to start something else when it’s finished instead of blocking our own thread in the process.
  + To achieve fine asynchronous programming we will need greater reliance on manual synchronization.
  + The *Task* solves this problem. A Task is a high level abstraction – it represents a concurrent operation that might or might not be a thread.
  + Tasks are compositional i.e.. they can be chained by using *continuations.*
  + They can use Thread pool to lessen start up latency and with *TaskCompletionSource*

they can employ a callback approach that avoids threads altogether while waiting on I/O operations.