**Delegates:**

* 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)

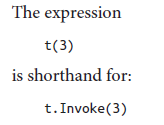
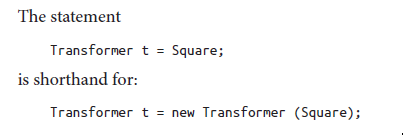
The above delegate takes int parameter and returns an int type. So it can point to any method that takes one int parameter and returns int type like this one 🡪

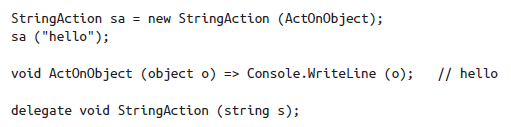
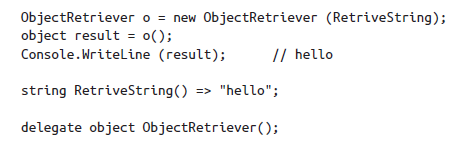
int square (int x) => x \* x;

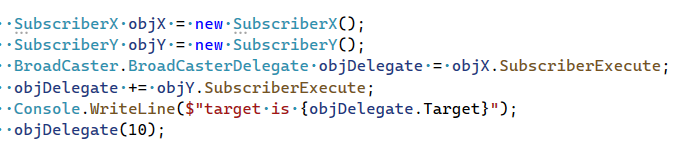
Transformer objDelegate = square ; // assigning a method to delegate variable creates delegate instance

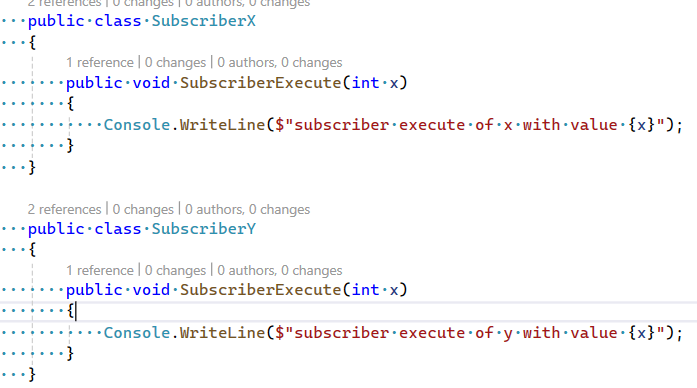
objDelegate(3); // invokes the instance.

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:

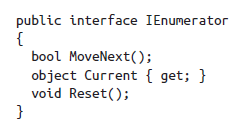
* Replace other subscribers by reassigning the ‘objDelegate’
* Clear all subscribers by assigning objDelegate to null
* Broadcast to all subscribers by invoking the delegate

**Events:**

* System.EventArgs 🡪 A base class that contains no members except ‘Empty’ that coneys information about an event.

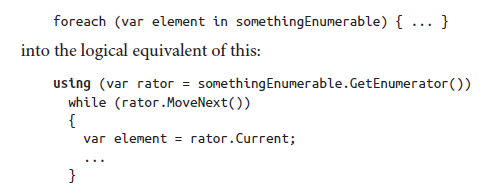
**Collections**:

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



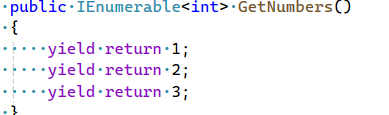
* MoveNext() 🡪 moves the cursor to the next element in the collection , it returns false if the collection is empty.
* Current 🡪 it returns the element at current position (usually cast from object to more specific type)
* Reset() 🡪 It moves the cursor to the initial position (start position) allowing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

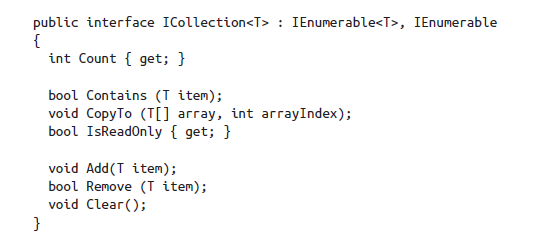


The method GetNumbers() returns 1,2 and 3 one at a time.

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.

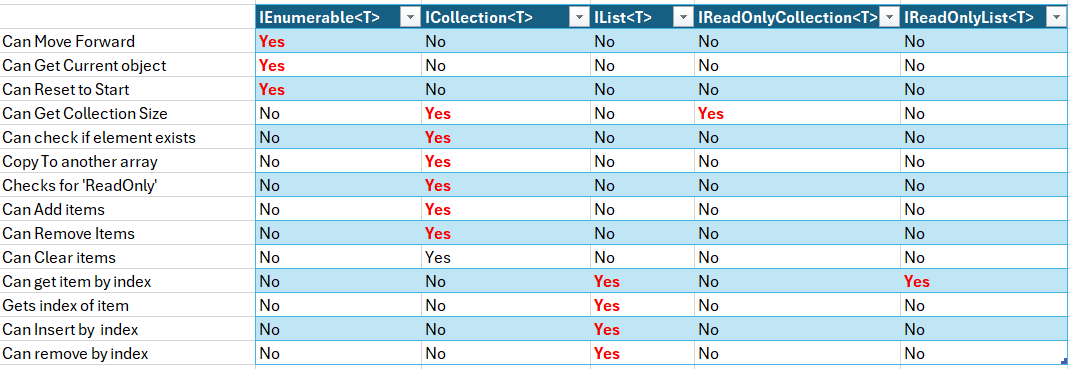
***ICollections and IList:***

* 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

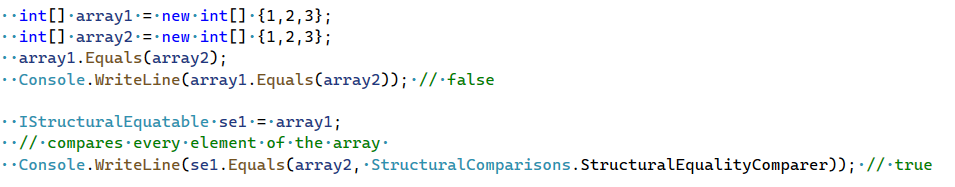
A screen shot of a computer code

Description automatically generated



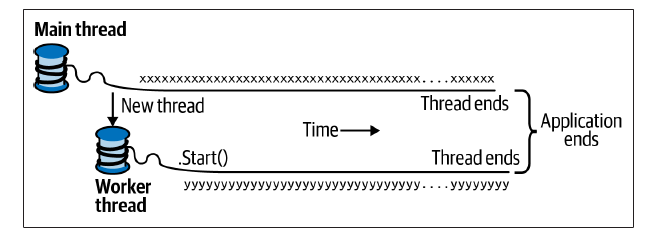
**Arrays**:

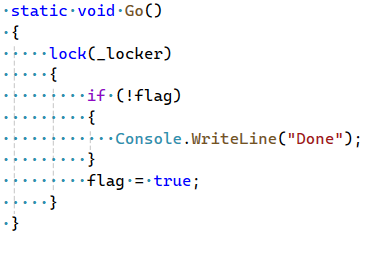
* It’s a fixed length collection i.e.. once created the size of the array cannot be changed.
* Arrays are generally more performant than lists because they provide direct access to elements via index.
* Arrays are memory-efficient since the memory for elements is allocated contiguously. This can lead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



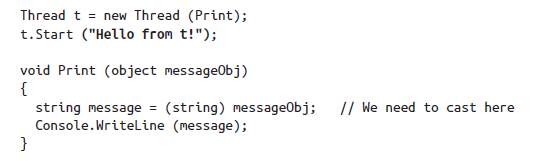
**Concurrency**:

* 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.

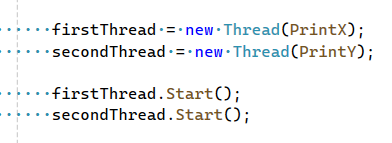


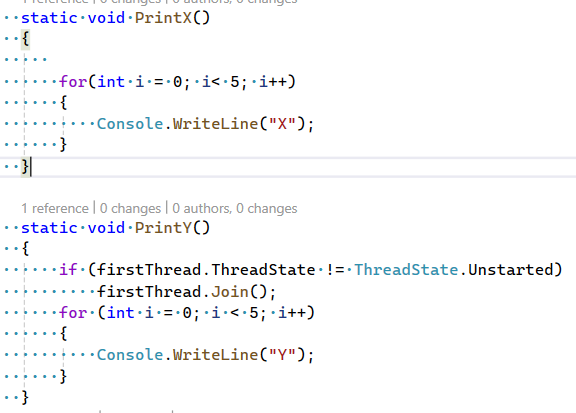
* Static fields can be shared between all the threads in the application domain.
* When threads share data it is possible to have either errors or no thread safety code.
* **Locks:** When more than one thread encounters the lock (which is a reference type object) , one thread waits or blocks until the 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.