**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)

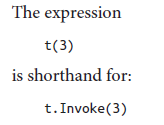
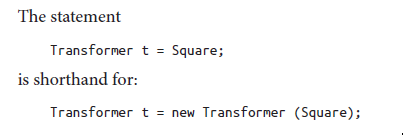
The above delegate takes int parameter and returns an int type. So it can point to any method that takes one int parameter and returns int type like this one 🡪

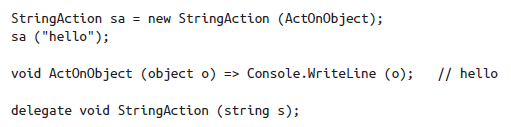
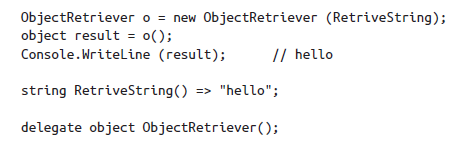
int square (int x) => x \* x;

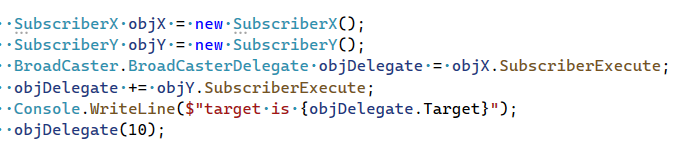
Transformer objDelegate = square ; // assigning a method to delegate variable creates delegate instance

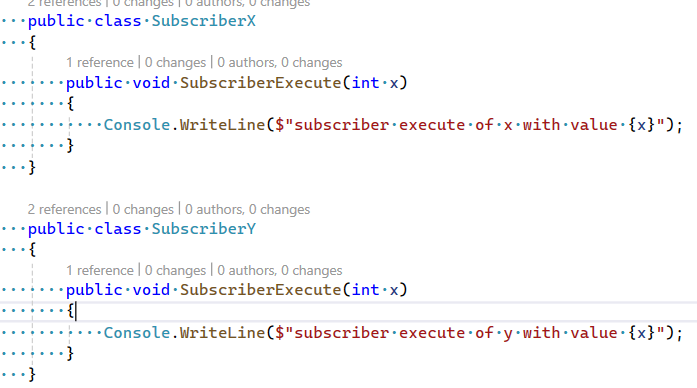
objDelegate(3); // invokes the instance.

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:

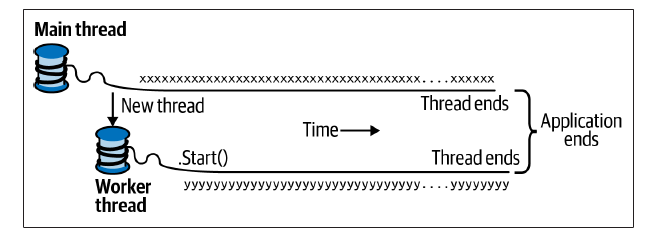
* Replace other subscribers by reassigning the ‘objDelegate’
* Clear all subscribers by assigning objDelegate to null
* Broadcast to all subscribers by invoking the delegate

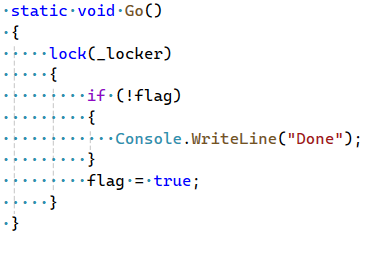
**Events:**

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

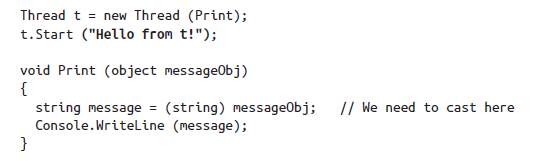
**Concurrency**:

* 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.

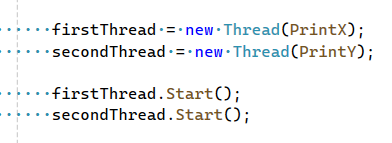


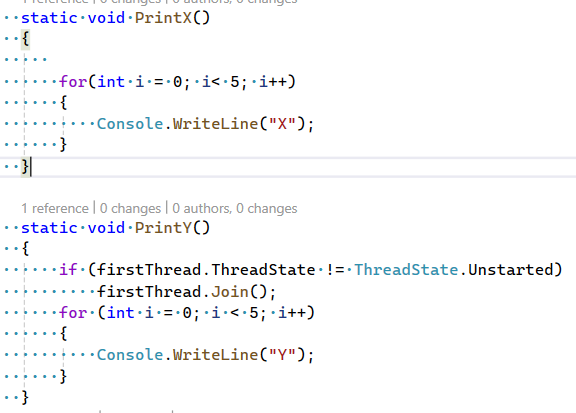
* Static fields can be shared between all the threads in the application domain.
* When threads share data it is possible to have either errors or no thread safety code.
* **Locks:** When more than one thread encounters the lock (which is a reference type object) , one thread waits or blocks until the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