C# Notes

Contents

[C# Types and variables 3](#_Toc51232684)

[Reference Type 3](#_Toc51232685)

[Value Types 3](#_Toc51232686)

[Multi-Threading 4](#_Toc51232687)

[Join and Sleep 4](#_Toc51232688)

[How Threading works? 5](#_Toc51232689)

[Thread Vs Processes 5](#_Toc51232690)

[Thread Usage and Misusage 5](#_Toc51232691)

[Passing Data to the thread 5](#_Toc51232692)

[Thread Naming 6](#_Toc51232693)

[Foreground and Background Threads 6](#_Toc51232694)

[Thread Priority 6](#_Toc51232695)

[Exception handling 6](#_Toc51232696)

[Thread Pooling 7](#_Toc51232697)

[Linked List 8](#_Toc51232698)

[HashSet 8](#_Toc51232699)

[HashTable 8](#_Toc51232700)

[Dictionary 9](#_Toc51232701)

[Sorting 9](#_Toc51232702)

[**Simple Sorts:** 9](#_Toc51232703)

[**Efficient Sorts:** 9](#_Toc51232704)

[**Bubble Sorts and Variant:** 10](#_Toc51232705)

[**Distribution Sorts:** 10](#_Toc51232706)

[Strings 11](#_Toc51232707)

[String vs String Builder 12](#_Toc51232708)

[Writing Clean C# Code 13](#_Toc51232709)

[Poor Naming 13](#_Toc51232710)

[Mysterious Names 13](#_Toc51232711)

[Meaningless Names 13](#_Toc51232712)

[Names with Encoding 13](#_Toc51232713)

[Ambiguous Names 13](#_Toc51232714)

[Noisy Names 13](#_Toc51232715)

[Poor Naming Conventions 14](#_Toc51232716)

[Poor Method Signatures 14](#_Toc51232717)

[Long Parameter List Method 14](#_Toc51232718)

[Output Parameters 14](#_Toc51232719)

[Variable declaration at top 15](#_Toc51232720)

[Magic Numbers 15](#_Toc51232721)

[Nested Conditionals 15](#_Toc51232722)

[Switch Statements 15](#_Toc51232723)

[Duplicate Code 15](#_Toc51232724)

[Comments 16](#_Toc51232725)

[Long methods 16](#_Toc51232726)

[Advanced C# Topics 17](#_Toc51232727)

[Reflection in C# -> System.Reflection 17](#_Toc51232728)

[Late Binding Using Reflection 17](#_Toc51232729)

[Generics 20](#_Toc51232730)

[Generic Type Parameter 20](#_Toc51232731)

[Constraints on type parameter 20](#_Toc51232732)

[Generic Classes 20](#_Toc51232733)

[Generic Interfaces 20](#_Toc51232734)

[Delegates in C#. 21](#_Toc51232735)

[Delegates with named vs Anonymous Methods 22](#_Toc51232736)

[Anonymous Method 22](#_Toc51232737)

[Combining Delegates (Multi Cast Delegate) 22](#_Toc51232738)

[Func and action delegates in C# 22](#_Toc51232739)

[Actions 22](#_Toc51232740)

[Func 23](#_Toc51232741)

[READ MORE: 23](#_Toc51232742)

[Observer Pattern aka Publisher Subscriber Pattern 24](#_Toc51232743)

# Multi-Threading

* It enables parallel execution of code.
* It is in System.Threading namespace.

class ThreadTest {

  static void Main() {

    Thread t = new Thread (WriteY);          *// Kick off a new thread*

    t.Start();                               *// running WriteY()*

*// Simultaneously, do something on the main thread.*

    for (int i = 0; i < 1000; i++) Console.Write ("x");  }

  static void WriteY() {

    for (int i = 0; i < 1000; i++) Console.Write ("y"); } }

* So in above example, main thread creates a new thread and starts executing printing Y and also parallel prints X.
* Thread has **IsAlive** which is set to true once is start to the time it completes.
* CLR assigns thread their own memory stack and they maintain their own copy of local variables.
* Thread share data if they have common reference to the same object instance or the data is shared through static variable.

class ThreadTest

{

  bool done;

  static void Main()

  {

    ThreadTest tt = new ThreadTest();   *// Create a common instance*

    new Thread (tt.Go).Start();

    tt.Go();

  }

*// Note that Go is now an instance method*

  void Go()

  {

     if (!done) { done = true; Console.WriteLine ("Done"); }

  }

}

* Because both threads call **Go()** on the same **ThreadTest** instance, they share the **done** field. This results in "Done" being printed once instead of twice.
* When a thread is blocked it doesn’t consume CPU resource.

## Join and Sleep

* We can wait for another thread to end by calling join.

static void Main()

{

  Thread t = new Thread (Go);

  t.Start();

  t.Join();

  Console.WriteLine ("Thread t has ended!");

}

## How Threading works?

* Threading is managed internally by a thread scheduler. CLR delegates this task to OS.
* It ensures all active thread gets execution time and waiting threads don’t consume CPU.
* **SINGLE PROCESSOR SYSTEM:**  It is handled by time slicing. It rapidly switches the threads execution.
* **MULTI PROCESOR SYSTEM:**  It is handled by time slicing and genuine concurrency. Different threads run simultaneously at different CPUs. There will be time slicing as OS needs to run its own threads as well.
* **Preemption:**  A thread is said to be preempted if execution of thread is stopped by some external factor such as time slicing.

## Thread Vs Processes

* Processes run in parallel and inside process thread also runs in parallel.
* Processes are fully isolated but threads are isolated upto extent as thread may share data as well. Threads share the heap memory.

## Thread Usage and Misusage

* **Maintaining responsive user interface:** Time consuming process can run in parallel, and main Thread can be free to do other things.
* **Making efficient use of otherwise blocked CPU:** If some thread is blocked, meanwhile some other thread may process.
* **Parallel Programing**

## Passing Data to the thread

* WAY 1: Executing a lambda expression which call function with parameter.

static void Main()

{

  Thread t = new Thread ( () => Print ("Hello from t!") );

  t.Start();

}

static void Print (string message)

{

  Console.WriteLine (message);

}

* WAY 2: Wrap entire implementation in multi statement.

new Thread (() =>

{

  Console.WriteLine ("I'm running on another thread!");

  Console.WriteLine ("This is so easy!");

}).Start();

* WAY 3: Using Anonymous Methods

new Thread (delegate()

{

  ...

}).Start();

* WAY 4: Using Thread. Start();

static void Main()

{

  Thread t = new Thread (Print);

  t.Start ("Hello from t!");

}

static void Print (object messageObj)

{

  string message = (string) messageObj;   *// We need to cast here*

  Console.WriteLine (message);

}

## Thread Naming

* Each thread has a name property which can be set for benefit in debugging purpose.
* Thread name can be set only once, if we try to change it later will throw an exception.

Thread.CurrentThread.Name = "main";

    Thread worker = new Thread (Go);

    worker.Name = "worker";

    worker.Start();

    Go();

## Foreground and Background Threads

* By default the explicitly created threads are foreground thread.
* Foreground thread keeps the app alive as long as running, once they stop all background threads are explicitly stopped.
* It is using IsBackground property to set the thread as background or foreground thread.

Thread worker = new Thread ( () => Console.ReadLine() );

    if (args.Length > 0) worker.IsBackground = true;

    worker.Start();

* [Read more <http://www.albahari.com/threading/> ]

## Thread Priority

It is used to decide how much computation time it will get relative to other active threads.

enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }

It is relevant only when multiple active threads are there at same time.

We should be careful while elevating the priority of a thread as it may lead to resource starvation for the other threads.

## Exception handling

* A thread has an independent execution path. Consider the below code, the exception on thread will never be caught so the program will close.

public static void Main() {

  try {

    new Thread (Go).Start(); }

  catch (Exception ex) {

*// We'll never get here!*

    Console.WriteLine ("Exception!"); } }

static void Go() { throw null; }   *// Throws a NullReferenceException*

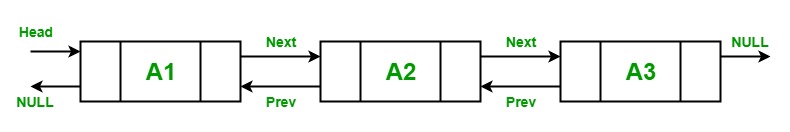
* So we need an exception handler at each thread entry method.

## Thread Pooling

* On thread start, a few hundred microseconds are spent organizing such things as a fresh private local variable stack. Each thread also consumes (by default) around 1 MB of memory.
* The thread pool cuts these overheads by sharing and recycling threads, allowing multi-threading to be applied at a granular level.
* To create a thread pool, we can choose any of the following methods.
  + Task Parallel Library
  + By Calling Thread.QueueUserWorkItem
  + Via async delegates
  + Via Background worker

# Events in C#

# Linked List

* Linear data structure which stores data at non-continuous memory location
* Elements are linked to each other using pointers
* Each node contains a data field and a reference to the next node.
* In C# it is generic and is doubly linked list so it contains reference to forward and backward list.
* It is dynamic and grows as per requirement
* 

# HashSet

* Unordered Collection of unique elements.
* Uses hash table for storage
* Performance is much better than that of list.
* It supports Set Operations such as intersection, union and difference.
* It can dynamic collection and is generic.

Creating a HashSet

HashSet<Type\_of\_hashset> Hashset\_name = new HashSet<Type\_of\_hashset>();

Adding an element

Using .Add() method.

[**Remove(T)**](https://www.geeksforgeeks.org/c-remove-the-specified-element-from-a-hashset/)**:**This method is used to remove the specified element from a HashSet object.

[**RemoveWhere(Predicate)**](https://www.geeksforgeeks.org/c-remove-elements-from-a-hashset-with-conditions-defined-by-the-predicate/)**:** This method is used to remove all elements that match the conditions defined by the specified predicate from a HashSet collection.

[**Clear**](https://www.geeksforgeeks.org/c-remove-all-elements-from-a-hashset/)**:** This method is used to remove all elements from a HashSet object.

Union of hash sets

myhash1.UnionWith(myhash2);

Intersection of hash set

myhash1.IntersectWith(myhash2);

Set difference using hash set

myhash1.ExceptWith(myhash2);

# HashTable

* Collection of key pair value which is arranged based on hash code of the key
* Uses hash table for storage
* It has optimized lookup as it looks up hash code of the key
* Key cannot be null but value can be
* We can store elements of same or different types
* Duplicate keys are not allowed

Creating a new hash table

Hashtable hashtable\_name = new Hashtable()

* [**Clear**](https://www.geeksforgeeks.org/c-remove-all-elements-from-the-hashtable/)**:**This method is used to remove all the objects from the hashtable.
* [**Remove**](https://www.geeksforgeeks.org/c-remove-the-element-with-the-specified-key-from-the-hashtable/)**:**This method is used to remove the element with the specified key from the hashtable.
* [**Contains**](https://www.geeksforgeeks.org/c-check-whether-a-hashtable-contains-a-specific-key-or-not/)**:**This method is used to check whether the Hashtable contains a specific key.
* [**ContainsKey**](https://www.geeksforgeeks.org/c-check-if-the-hashtable-contains-a-specific-key/)**:**This method is also used to check whether the Hashtable contains a specific key.
* [**ContainsValue**](https://www.geeksforgeeks.org/c-check-if-the-hashtable-contains-a-specific-value/)**:**This method is used to check whether the Hashtable contains a specific value.

# Dictionary

* It is used to store key pair values and works like hash table but it is generic
* It is dynamic in nature
* Key cannot be null but value can be null
* Key must be unique otherwise if we try to store duplicate keys we get exception

# Sorting

### **Simple Sorts:**

* Selection Sort program in C# ([Solution](https://www.csharpstar.com/c-program-to-perform-selection-sort/))
  + Selection Sort is a sorting algorithm that finds the minimum value in the array for each iteration of the loop. Then this minimum value is swapped with the current array element. This procedure is followed until the array is sorted.
* Insertion Sort program in C# ([Solution](https://www.csharpstar.com/csharp-program-to-perform-insertion-sort/))

### **Efficient Sorts:**

* Heap Sort program in C# ([Solution](https://www.csharpstar.com/heap-sort-csharp-program/))
* Merge Sort program in C# ([Solution](https://www.csharpstar.com/merge-sort-csharp-program/))

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

* Quick Sort program in C# ([Solution](https://www.csharpstar.com/csharp-program-quick-sort/))

### **Bubble Sorts and Variant:**

* Bubble Sort program  in C# ([Solution](https://www.csharpstar.com/csharp-program-to-perform-bubble-sort/))
* Shell Sort program in C# ([Solution](https://www.csharpstar.com/shell-sort-csharp-program/))
* Comb Sort program in C# ([Solution](https://www.csharpstar.com/comb-sort-program-csharp/))

### **Distribution Sorts:**

* Bucket Sort program in C# ([Solution](https://www.csharpstar.com/csharp-program-to-perform-bucket-sort/))
* Radix Sort program in C# ([Solution](https://www.csharpstar.com/csharp-program-radix-sort/))

# Strings

**Iterating a String –** Can be done using for or for each loop. For loop gives more flexibility.

 for (int i = 0; i < strOriginal.Length; i++)

    {

        MessageBox.Show(strOriginal[i].ToString());

    }

foreach (char c in strOriginal)

    {

        MessageBox.Show(c.ToString());

    }

**Split a string-** This can be done using .Spit() method this takes a delimiter

char[] delim = {' '};

    string[] strArr = strOriginal.Split(delim);

    foreach (string s in strArr)

    {

        MessageBox.Show(s);

    }

**Extract Substring of a string –** Using .Substring Method

**Create a string array**

 string[] strArr = new string[3] { "string 1", "string 2", "string 3"};

    // Omit Size of Array

 string[] strArr1 = new string[] { "string 1", "string 2", "string 3" };

    // Omit new keyword

 string[] strArr2 = {"string 1", "string 2", "string 3"};

string[,] strArr3 = new string[2, 2] { { "string 1", "string 2" }, { "string 3", "string 4" } };

    // Omit Size of Array

    string[,] strArr4 = new string[,] { { "string 1", "string 2" }, { "string 3", "string 4" } };

    // Omit new keyword

    string[,] strArr5 = { { "string 1", "string 2" }, { "string 3", "string 4" } };

**Reverse a string**  using Microsoft.VisualBasic.Strings.StrReverse(input)

**Compare String** You can use the String.Compare() to compare two strings. The third parameter is a Boolean parameter that determines if the search is case sensitive(false) or not(true).

**Convert a string to byte array**

byte[] b = Encoding.Unicode.GetBytes(strOriginal);

**Convert byte[] to string**

strModified = Encoding.Unicode.GetString(b);

char[] chArr = strOriginal.ToCharArray();

strModified = new String(chArr);

**Test if string is null or empty**

bool check = String.IsNullOrEmpty(strOriginal);

**Convert the Case of a String** – The String class contains methods to convert a string to lower and upper cases. However, it lacks a method to convert a string to Proper Case/Title Case. Hence we will use the ‘TextInfo’ class to do the same.

C#

    System.Globalization.CultureInfo cultureInfo = System.Threading.Thread.CurrentThread.CurrentCulture;

    System.Globalization.TextInfo textInfo = cultureInfo.TextInfo;

    // Lower Case

    MessageBox.Show(textInfo.ToLower(strOriginal));

    // Upper Case

    MessageBox.Show(textInfo.ToUpper(strOriginal));

    // Proper Case

    MessageBox.Show(textInfo.ToTitleCase(strOriginal));

# String vs String Builder

StringBuilder is used to represent a mutable string of characters. Mutable means the string which can be changed. So String objects are immutable but StringBuilder is the mutable string type. It will not create a new modified instance of the current string object but do the modifications in the existing string object.

**NEED OF STRING BUILDER**

As stated above that the String class objects are immutable which means that if the user will modify any string object it will result into the creation of a new string object. It makes the use of string costly. So when the user needs the repetitive operations on the string then the need of StringBuilder come into existence. It provides the optimized way to deal with the repetitive and multiple string manipulation operations.

# Writing Clean C# Code

**Clean Code -** clean, maintainable, reveals the intent and reduces the guess work & reads like a story.

**Code Smells –** Poorly written code

**Refactoring –** Changing structure of code without chaining the functionality. We can use ReSharper and get benefited from it.

## Poor Naming

One of the most common code smells.

### Mysterious Names

It includes mysterious names like variable named dr1, od, x or Button1\_Click (); etc.

The reader should now go somewhere else to understand the implementation of your code.

### Meaningless Names

When function is more than 10 lines of code it is doing more than 1 thing and it’s difficult to come up with meaningful name.

### Names with Encoding

Int iMaxRequest -> this is in Hungarian notation where variable is prefixed with data type. Or m\_objCollection

### Ambiguous Names

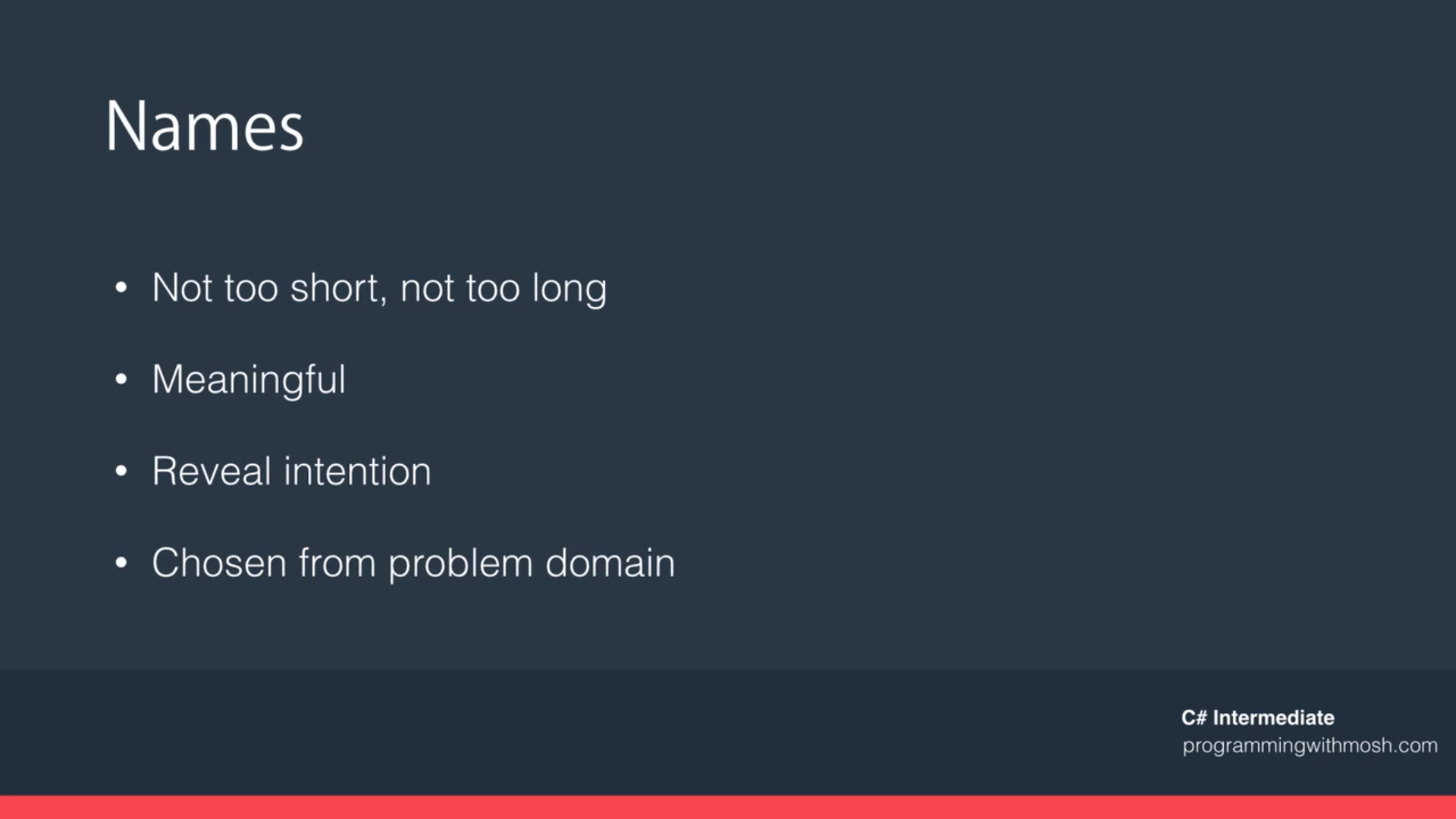
Eg bool MultiSelect() {}

### Noisy Names

Customer theCutomer -> can be renamed to just customer

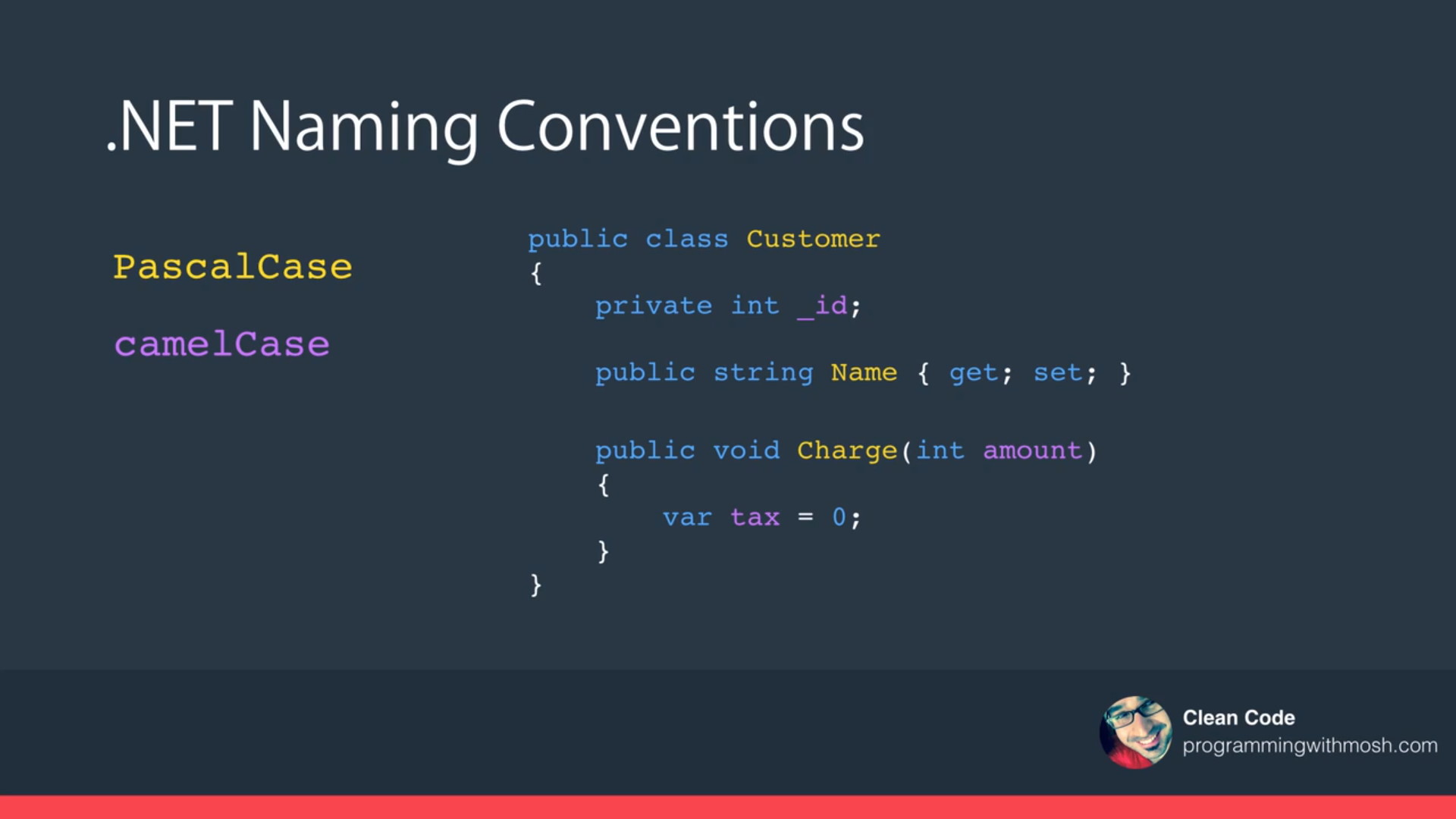
List<Customer> lifeofApprovedCustomers -> can be renamed to Approved Customer

**KEEP IN MIND**



## Poor Naming Conventions

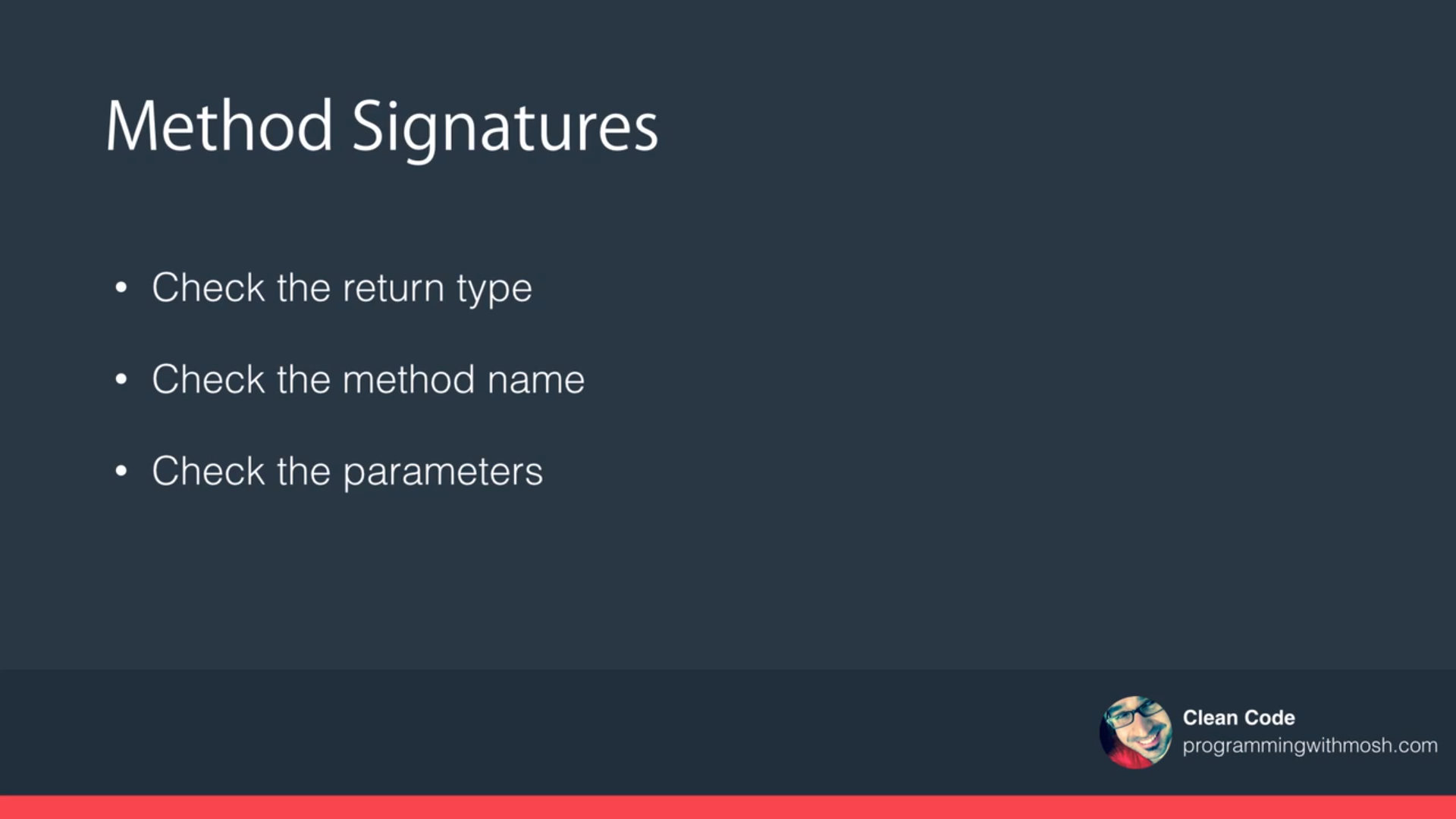
In .net we have PascalCase and camelCase naming convention. Follow the below convention and private fields needs to be prefixed with \_ and your code base should look like it is written by a single person and not 50 persons over 10 year.



## Poor Method Signatures

Usually Boolean flags as method input parameter is a code smell as based on this flag method behavior may change.

**KEEP IN MIND**



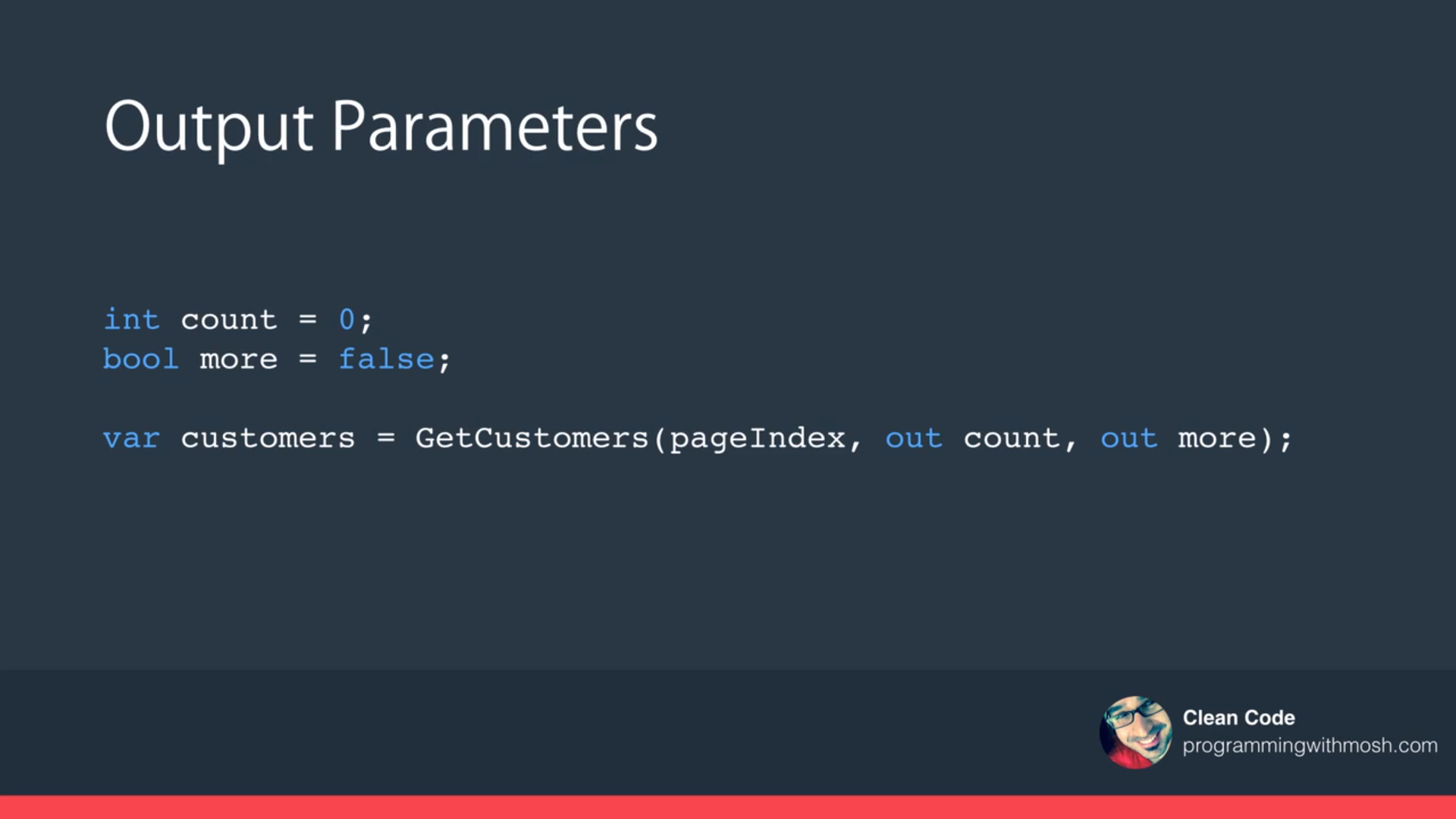
## Long Parameter List Method

Example if you have 2 dates input to\_date and from\_date then this can be encapsulated in a DateRange class parameter which reduces param list by 1.

Another example if you are take multiple input to filter out and get reservations then you can encapsulate in in reservations Query parameter.

Best practice is to limit the parameter to 3 for this we can either try to remove or encapsulate logically related parameters into a class

## Output Parameters



Here we need to unnecessarily initialize 2 variables and we get data out of method from parameter.

So we can refactor this to return a type of tuple. But we need to eliminate tuples are they are another code smell as they have item1… which doesn’t reveal the intent of code. So we can create a new class and put it in.

At all-time avoid out param and return from object instead

## Variable declaration at top

We should declare the variable where we need them not at top. This was done previously where early c compilers which didn’t understand variables declared in middle. As if we declare all variable at once the code reader will have to spend more brain power to read and understand all and as clean coders we should avoid it.

If we have a variable which is used at couple of place in code block so we should keep it at top.

**Always declare variable close to where they are used.**

## Magic Numbers

**Use Constants or Enums**

For this you can make it a constant and give it a proper readable name.

A better way to solve this problem is to create a enumeration if we have multiple relatable const. Also creating an enumeration make it reusable at other places. For naming enum we use singular form. And if it is method parameter then change the signature of method to take enumeration instead of casting enumeration inside method.

## Nested Conditionals

It makes programs hard to read, understand, test and modify.

To avoid this we can use the following strategy

1. Use Ternary operators. But don’t abuse it by applying more than once in a single statement
2. Try to combine nested blocks by using logical operators. But use this technique in moderation.
3. Use early exist technique

## Switch Statements

Replace them with polymorphic dispatch

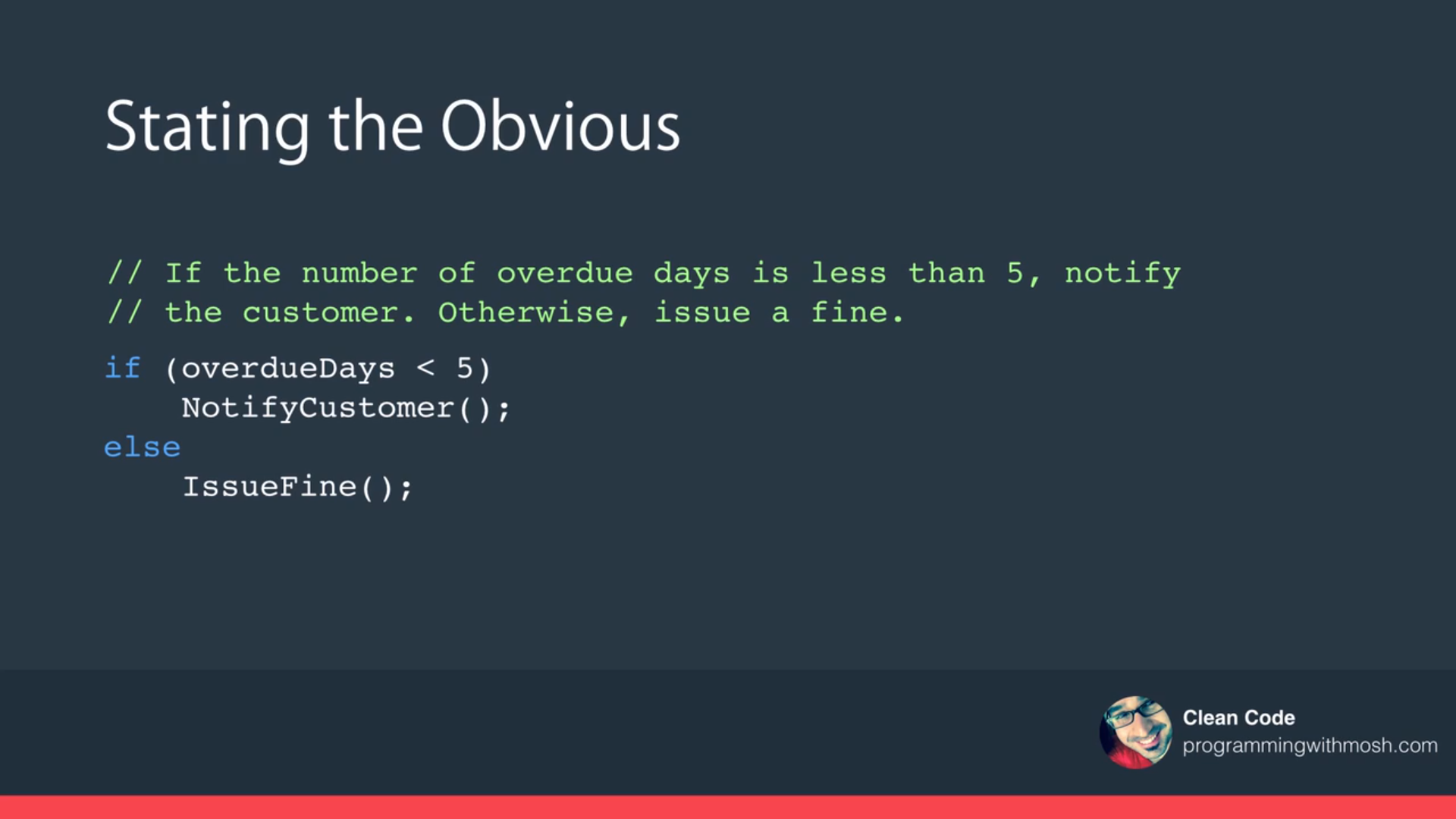
Use push member down refactoring.

## Duplicate Code

If code is duplicated then if we change at all the places and it makes it hard and also it is violation of principle called DRY ( Don’t Repeat Yourself) .

## Comments

Comments that state the obvious is a code smell as they create noise in the code. Another example is writing version history comments. Comments to clarify the code is a smell instead we should fix the real issue in code. Eg var odr; //object data reader. Commenting dead code. TODO comments can be a good example of what to comment. As if we fixing or implementing something we don’t want to get distracted so we can add a proper TODO comment and then later view all on visual studio.



The ultimate comment for our code should be our code itself. We should strive to write self documented code.

**Don’t write comments, re write your code.**

## Long methods

They are the most popular code smell. A method which is more than 10 lines of code. As human we can keep upto 7 things in our mind at a time. These are hard to understand, change and re-use. We should write method that specialize in one thing. We can apply cohesion principle to separate long methods which means things which are related should be together and things which are not should not be together. We should also apply this to the class level also.

# Default Interface Implementation – C# 8.0

**Before C# 8.0**

* An interface only contains declarations of methods, properties, indexers, and events.
* Interface cannot have access modifiers.
* C# 8.0 allows us to specify implementation for a method in an interface.
* Let’s say we have a fairly large app, we have an interface which is implemented by multiple classes, then we have to go and change all those classes but it can be tedious, so in c# 8.0 we can give a default implementation of an interface method, so we don’t have to implement in all classes, and the classes may to override it.

public interface IBook

{

    void AddBook(string bookName, string autherName);

    void removeBook(string bookName);

    void rateBook(int bookID)

    {

        //default logic here

   Console.WriteLine("\nExecuted the Default implementation in the interface");

    }

}

# Advanced C# Topics

# Reflection in C# -> System.Reflection

* It is used to retrieve metadata on types at runtime. It works with assemblies types & module
* Can be used to:
  + Dynamically create instance of a type
  + Create new types at runtime. Use classes in System.Reflection.Emit
  + For performing late binding and instantiating types in assembly
  + Get type from existing object
  + Invoke methods or access properties and fields of a method
  + Allows to access attributes in code
  + Create a minimal IoC Container [Read]
* GetType()

// Using GetType to obtain type information:

int i = 42;

Type type = i.GetType();

Console.WriteLine(type); -> System.Int32

* Full name of loaded assembly

// Using Reflection to get information of an Assembly:

Assembly info = typeof(int).Assembly;

Console.WriteLine(info);

The output is: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e.

When we write code we produce data and reflection allows to manipulate this data inside code.

## Late Binding Using Reflection

* Binding of methods and object during runtime is late binding or dynamic binding

// C# program to show the Late

// Binding using Reflection

using System;

using System.Reflection;

namespace LateBinding {

class Program {

    // Main Method

    static void Main(string[] args)

    {

        // Declare Instance of class Assembly

        // Call the GetExecutingAssembly method

        // to load the current assembly

**Assembly executing = Assembly.GetExecutingAssembly();**

        // To find the type of the Class Student

**Type studentType = executing.GetType("LateBinding.Student");**

        // Create an Instance of the Student type

**object studentObject = Activator.CreateInstance(studentType);**

        // Store the info of the method in an object

        // of class MethodInfo

**MethodInfo getMethod = studentType.GetMethod("GetDetails");**

        // To store the parameters required

        // by Method GetDetails

        String[] param = new String[2];

        param[0] = "1";

        param[1] = "Lisa";

        // To display the result of the method

        b

        Console.WriteLine("Student Details : ");

        Console.WriteLine("Roll Number - Name \n{0}", det);

    } // end Main

} // end Program

public class Student {

    public String GetDetails(String RollNumber, String Name)

    {

        return RollNumber + " - " + Name;

    }

} // end Student

}

* **EXAMPLE**

public class Customer

{

public Customer() {

//Default constructor

}

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Address { get; set; }

public bool Validate(Customer customerObj)

{

//Code to validate the customer object

return true;

}}

static void Main(string[] args)

        {

**Type type = typeof(Customer);**

Console.WriteLine("Class: " + type.Name);

Console.WriteLine("Namespace: " + type.Namespace);

// **GET PUBLIC PROPERTIES**

**PropertyInfo[] propertyInfo = type.GetProperties();**

 Console.WriteLine("The list of properties of the Customer class are:--");

            foreach (PropertyInfo pInfo in propertyInfo)            {

                Console.WriteLine(pInfo.Name);            }

// **GET ALL PUBLIC CONSTRUCTORS**

**ConstructorInfo[] constructorInfo = type.GetConstructors();**

Console.WriteLine("The Customer class contains the following Constructors:--");

foreach (ConstructorInfo c in constructorInfo) { Console.WriteLine(c); }

// **GET ALL PUBLIC METHODS**

**MethodInfo[] methodInfo = type.GetMethods();**

Console.WriteLine("The methods of the Customer class are:--");

foreach (MethodInfo temp in methodInfo) {

Console.WriteLine(temp.Name); }

        }

# Generics

* Introduced concept of type parameters.
* It allows to define class and methods which defer type specification until they are instantiated

// Declare the generic class.

public class GenericList<T>

{

public void Add(T input) { }

}

* Generic classes ensure type safety and reusability.
* Generic classes may be constrained to work on a particular types

## Generic Type Parameter

* The type parameter is a specific type which client define while instantiating
* For generic parameter naming it should be a descriptive name, if we are using single character name then it should be T
* If we have a descriptive name it should be prefixed with T.

## Constraints on type parameter

* If we don’t specify constraint on types then it can take any types.
* Constraints are specified using where keyword
* Constraints specifies capabilities and expectation of a type parameter
* We can apply constraints to multiple parameter and a multiple constraints on single param

class Base { }

class Test<T, U>

where U : struct

where T : Base, new()

{ }

## Generic Classes

* Used to encapsulate operation which don’t belong to any specific type
* Most common generic classes are collections like list, hash table, stack, queues, etc

## Generic Interfaces

# Delegates in C#.

* Delegates represents reference to a method with particular parameter list and return types
* We can instantiate delegate, associate it to any compatible method and invoke associated method from the delegates instance.
* Used to pass method as arguments to other methods. Due to this we don’t know at compile time which method will be called.
* It is similar to C++ function pointers but in C# they are type safe. Delegates is used to encapsulate a method.
* It can be used to define callbacks to a method.
* Delegates can be changed
* Delegates supports Variance[Read More]
* Anonymous method (Introduced in C# 2.0) & lambda expression(C# 3.0) are compiled delegates.
* A Delegate is constructed by passing name of method or Anonymous function.
* **SINGLE CAST DELEGATE : delegate return-type identifier ([parameters])**
* EG: **Public void delegate SimpleDelegate()** - This can encapsulate any method which doesn’t take an input parameter and doesn’t return anything.
* 3 steps to declare and use delegates:
  + **Declaration**
  + **Instantiation**
  + **Invocation**
* EG:

// **Simple Declaration**

Public void delegate SimpleDelegate ();

Class TestDelegate {

Public static void **MyFunc** ()  
        {  
            Console.WriteLine ("I was called by delegate ...");  
        }  
  
        public static void Main ()  
        {  
            // Instantiation  
            **SimpleDelegate SimpleDelegate = new SimpleDelegate (MyFunc);**

**Or**

**SimpleDelegate simpleDelegate = TestDelegate.MyFunc ();**  
  
            // Invocation  
            **SimpleDelegate ();**  
        }

}

## Delegates with named vs Anonymous Methods

* Delegates encapsulating a named method can either be static or instance method
* Delegate can also have anonymous method or named Function.

**// Instantiate Del by using an anonymous method.**

Del del3 = delegate(string name)

{ Console.WriteLine($"Notification received for: {name}"); };

**// Instantiate Del by using a lambda expression.**

Del del4 = name => { Console.WriteLine($"Notification received for: {name}"); };

## Anonymous Method

* Provides a way to pass on a code block as delegate method parameter. We don’t define the return type it is inferred from the return statement

delegate void NumberChanger(int n);

...

NumberChanger nc = delegate(int x) {

Console.WriteLine("Anonymous Method: {0}", x);

};

nc(10);

## Combining Delegates (Multi Cast Delegate)

* **Multicasting:** C# delegates are multicast which means at one point of time they can point to more than one function.
* It maintains a list of delegates which will be called when we invoke the delegate
* Multiple objects can be assigned to one delegate using + operator
* When multicast delegates are called then delegates are invoked in list in order.
* - is used to remove a delegate
* **using System;**

# Func and action delegates in C#

* Func delegates can return a value whereas Action delegates cannot return a value.
* Func is a delegate which points to a method

## Actions

* Action is a delegate which points to method which can accept parameters but cannot return a value
* Action should be used when delegate return type is void

**SYNTAX**: **Action**<**TParameter**>

**// Sample Code Snippet**

**static** **void** **Main**(**string**[] args)  
        {  
            **Action**<string> action = **new** **Action**<string>(**Display**);  
            action("Hello!!!");  
            **Console**.**Read**();  
        }  
**static** **void** **Display**(**string** message)  
        {  
            **Console**.**WriteLine**(message);  
        }

## Func

SYNTAX : **Func**<**TParameter**, **TOutput**>

**static** **void** **Main**(**string**[] args)  
        {  
            **Func**<**int**, **double**> func = **new** **Func**<**int**, **double**>(**CalculateHra**);  
            **Console**.**WriteLine**(func(50000));  
            **Console**.**Read**();  
        }  
        **static** **double** **CalculateHra**(**int** basic)  
        {  
            **return** (**double**)(basic \* .4);  
        }

# READ MORE:

Making callbacks

Passing callback with func

A function that takes a delegate as a parameter must use a named delegate type; unlike in Objective-C you can't declare an anonymous delegate type inline in the function definition. However, the generics Action<> and Func<> are provided so that you don't have to declare a new type yourself. In the code below I'm assuming the delegate takes a single int as a parameter.

**void DoSomethingWithCallback(Func<int,Task> callbackDelegate) {**

**Task t = callbackDelegate(42); }**

If this function doesn't actually do anything with the Task object returned (as with the code shown above), you can instead use Action<int> as the delegate type. If you use Action, you can still declare the delegate async (below) but the implicit Task object returned is ignored.

The lambda syntax for calling the above function is straightforward and the syntax you used in the question is correct. Note that the parameter type doesn't need to be specified here since it can be inferred:

DoSomethingWithCallback(async (intParam) => { this.myint = await Int2IntAsync(intParam); });

You can also pass a method or delegate variable, if you wish, instead of using the lambda syntax:

async Task MyInt2Int(int p) { ... }

Func<int,Task> myDelegate;

void OtherMethod()

{

myDelegate = MyInt2Int;

DoSomethingWithCallback(myDelegate); // this ...

DoSomethingWithCallback(MyInt2Int); // ... or this.

}

# Observer Pattern aka Publisher Subscriber Pattern

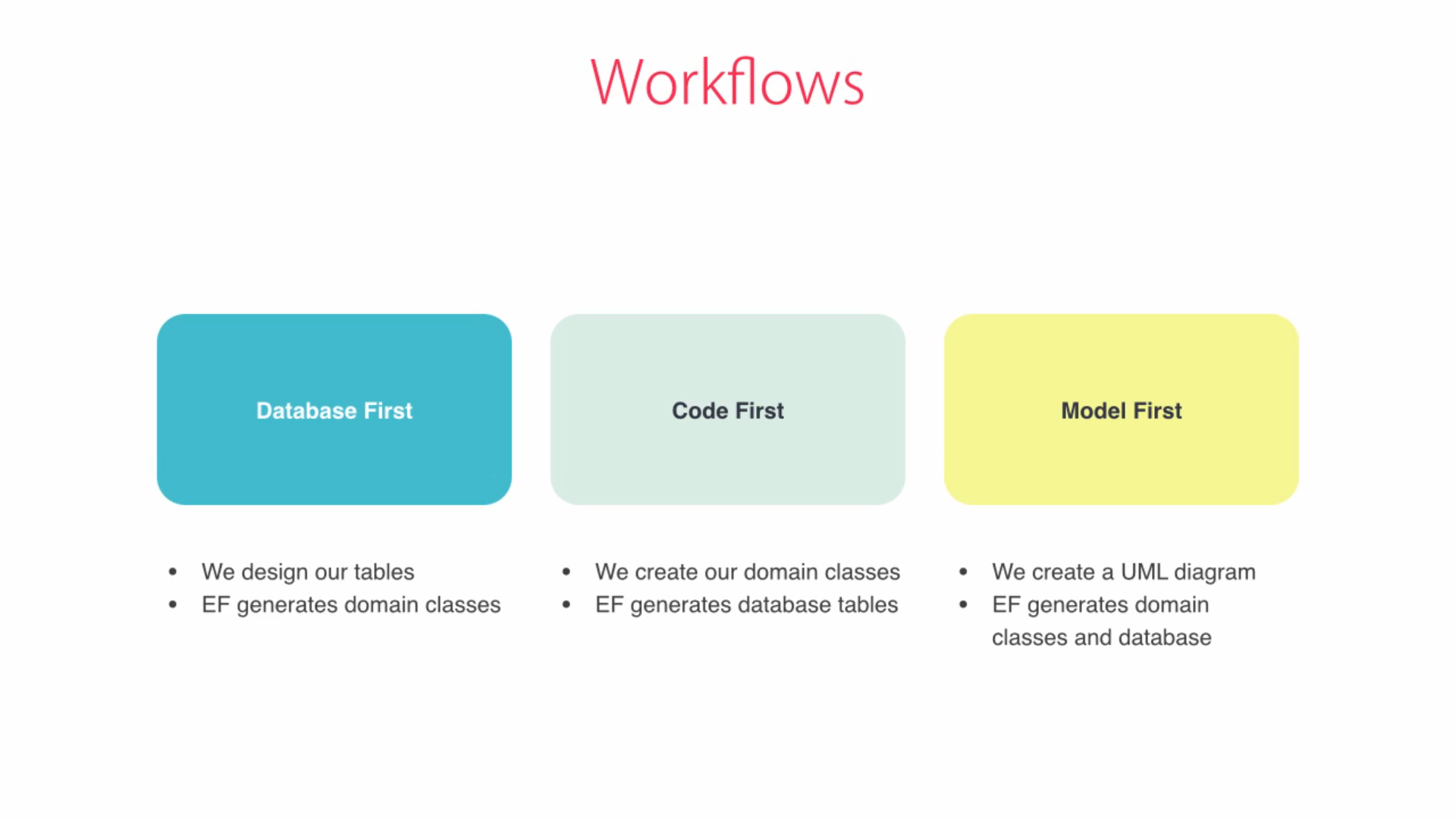
# Entity Framework

# Introduction

* We need a persistence framework to R/W from or to database. We can write our own persistence framework using ADO.NET, which is difficult to write.
* Entity framework is a **persistence framework** which manages connections, and allows R/W to db

## Workflows

* 3 workflows to build a domain model using entity framework
  + **Database First** – we start with db & design our tables then EF design domain models from it
  + **Code First** – we start with domain classes and then EF generates tables
  + **Model First** – with this we use visual designer to model our classes and associations and based on this EF with design domain classes and database tables



# Database First Workflow Quick Overview

* Create a database and create tables in that newly created database
* We can create a change scripts and put it in our repo and use it later if we want to put this data to some other database
* Create a new project in VS – Console App( for demo) & **Packages –** EntityFrameworkCore
* Go to sln explorer and add new item ADO.net Entity Data Model and give it a name, and use ef designer from database and specify the connection to database, you can test connection & OK.
* Now in next window you can select which tables which you require to import and a **.edmx** file is generated.
* Edmx has **all designer generated code**. It has a.tt file which stands for templates, it has a class which derives from dbConnection class. And it adds our tables as DbSet and a DBSet is used to represent a table
* There is a template model code also generated which contains C# classes representing our table.
* We can make change in db and then refresh our edmx file.
* And now you can inject the DbContext class and start using it.

# Code First Workflow Quick Overview

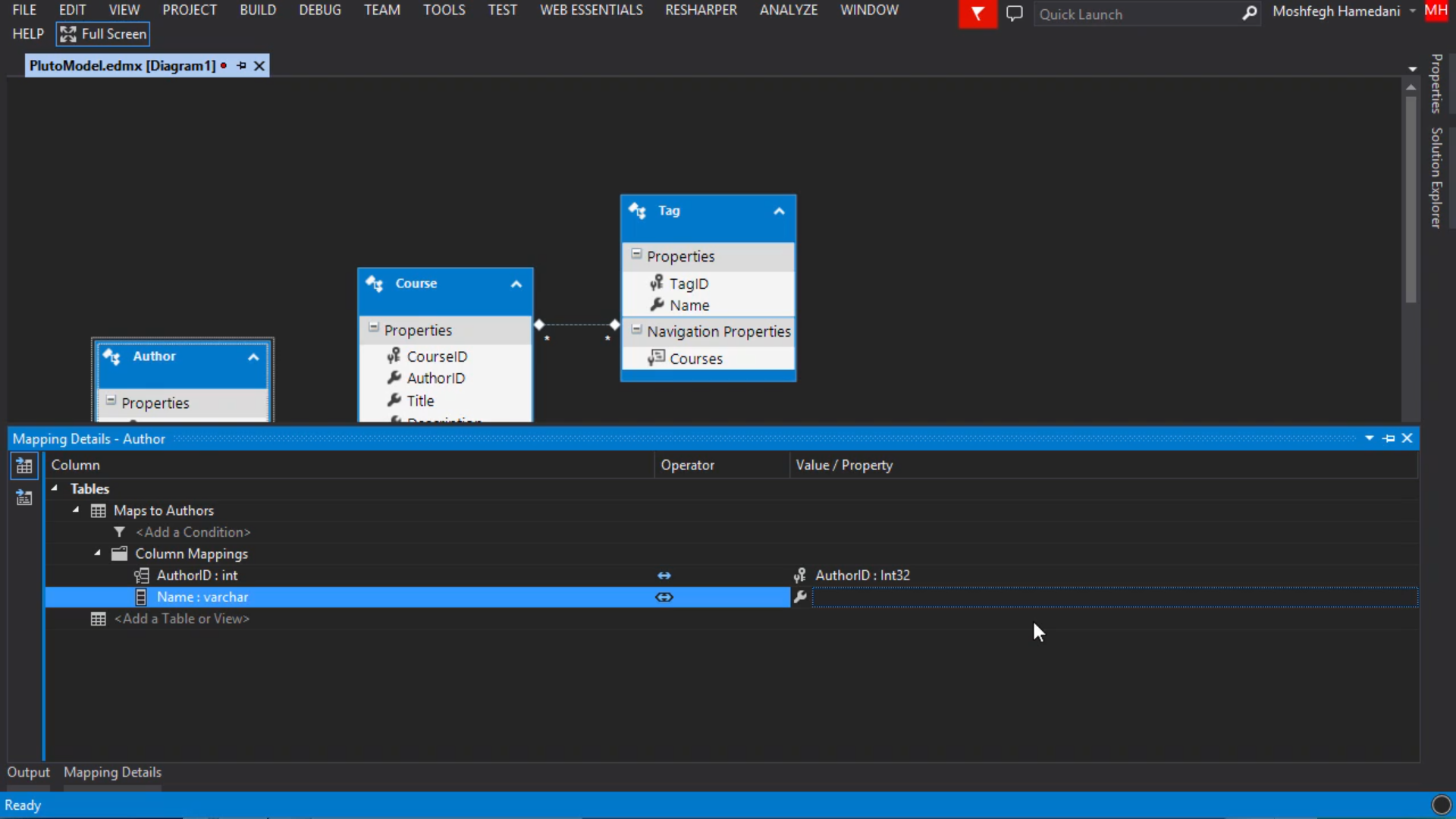
* Create a new console app (for demo) &Install the EntityFrameworkCore package& add an attribute provider name this is required for code first approach
* Now we need to create C# classes that represent our database tables & add a DbContext class as well and derives from DbContext. And now we need to specify the connection string to the database.
* Now we need to enable migrations enable-migrations
* Whenever we make change the models we need to add migrations add-migration <name> once we do this then a new folder is generated called migration and it has a class with 2 method Up() to upgrade db and Down() to downgrade database so using this we can upgrade or downgrade any db
* To run the migration we execute Update-Database

# Database First VS Code First

* Code first has **more control over the database**. Code first allows full versioning of database and can be useful when we are doing db versioning
* Code first has increased productivity as we write more code but it is easy

# Database first workflow in Detail

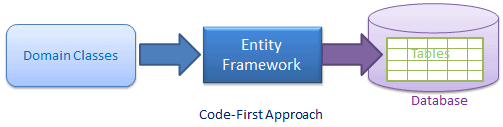
* Here after creating Database we need to add **ADO.NET Entity Data Model &** Select **EF Designer from Database.**
* Every entity has properties and navigation properties
* **Navigation properties** are used to navigate from one entity to the other entities.
* From the visual designer we can remove some particular property binding to make those properties not persist in db like this we can do for calculated fields.



* Right click on model and say update data model. It will have add , refresh and delete**.**
* You can at any time validate you model by right click and select validate**.**

# Code First Workflow in detail

* Here we focus more on creating domain entities rather than designing db so this is better in case of Domain Driven Design



* So we follow the following approach
  + Create/Modify domain classes
  + Configure domain classes using Fluent API or data annotation attribute
  + Create or update database schema using automatic migration
* This requires a context class which derives from DbContext class. Eg

public class SchoolContext: DbContext

{

public SchoolContext(): base() { }

public DbSet<Student> Students { get; set; }

public DbSet<Grade> Grades { get; set; }

}

* And we can use it as following

using (var ctx = new SchoolContext())

{

var stud = new Student() { StudentName = "Bill" };

ctx.Students.Add(stud);

ctx.SaveChanges();

}

* EF API configured PrimaryKeys, ForeignKeys, relationships, column data types etc. from the domain classes without any additional configurations.
* EF 6 infers the One-to-Many relationship using the navigation property by default convention. For many to one or many to many we need to configure it using fluent api or annotations

## Database Initialization in EF 6

* The base constructor in the context class can pass the following values
  + No Parameters – if no parameter is passed then it will create a database with name {NameSpace}.{ContextClassName}
  + Database Name – we can pass db name so then db will be created with that name.

public Context(): base("MySchoolDB")

* + Connection string name – so we can pass the connection string name in the base constructor

public SchoolDBContext() : base("name=SchoolDBConnectionString")

## Database initialization strategies

* **CreateDatabaseIfNotExists –** this is by default initializer. But if we change model class and then run the app it will throw exception
* **DropCreateDatabaseIfModelChanges –**
* **DropCreateDatabaseAlways –** it drops and creates every time we run the app irrespective of if the model has changed or not changed.
* **CustomIntializers**

**We can also turn of initializers. Like for prod env if we don’t want to loose any data.**

## Configuring Domain Classes

* We can use Data Annotation Attributes or Fluent API to configure Domain Classes

### Data Annotation Attributes

* It is simple attribute based configuration which can be applied to domain properties/classes

|  |  |
| --- | --- |
| **Attribute** | Description |
| [Key](https://www.entityframeworktutorial.net/code-first/key-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify a key property in an entity and make the corresponding column a PrimaryKey column in the database. |
| [Timestamp](https://www.entityframeworktutorial.net/code-first/TimeStamp-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify the data type of a corresponding column in the database as rowversion. |
| [ConcurrencyCheck](https://www.entityframeworktutorial.net/code-first/concurrencycheck-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify that the corresponding column should be included in the optimistic concurrency check. |
| [Required](https://www.entityframeworktutorial.net/code-first/required-attribute-dataannotations-in-code-first.aspx) | Can be applied to a property to specify that the corresponding column is a NotNull column in the database. |
| [MinLength](https://www.entityframeworktutorial.net/code-first/maxlength-minlength-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify the minimum string length allowed in the corresponding column in the database. |
| [MaxLength](https://www.entityframeworktutorial.net/code-first/maxlength-minlength-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database. |
| [StringLength](https://www.entityframeworktutorial.net/code-first/stringlength-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database. |
| [Table](https://www.entityframeworktutorial.net/code-first/table-dataannotations-attribute-in-code-first.aspx) | Can be applied to an entity class to configure the corresponding table name and schema in the database. **[Table("TableName")]** |
| [Column](https://www.entityframeworktutorial.net/code-first/column-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to configure the corresponding column name, order and data type in the database.  **[Column("CustomColumnName", TypeName=”DateTime”)]**  **public string StudentName { get; set; }**  type name is option and it corresponds to a type in database |
| [Index](https://www.entityframeworktutorial.net/EntityFramework6/index-attribute-in-code-first.aspx) | Can be applied to a property to configure that the corresponding column should have an Index in the database. (EF 6.1 onwards only) |
| [ForeignKey](https://www.entityframeworktutorial.net/code-first/foreignkey-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to mark it as a foreign key property.  **[ForeignKey(**"Standard"**)]**  **public int StandardRefId { get; set; }**  **public Standard Standard { get; set; }** |
| [NotMapped](https://www.entityframeworktutorial.net/code-first/notmapped-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database. **[NotMapped]** |
| DatabaseGenerated | Can be applied to a property to configure how the underlying database should generate the value for the corresponding column e.g. identity, computed or none.  [DatabaseGenerated(DatabaseGeneratedOption.None)] |
| [InverseProperty](https://www.entityframeworktutorial.net/code-first/inverseproperty-dataannotations-attribute-in-code-first.aspx) | Can be applied to a property to specify the inverse of a navigation property that represents the other end of the same relationship. |
| ComplexType | Marks the class as complex type in EF 6. EF Core 2.0 does not support this attribute. |

## Fluent API Configuration

* Fluent API design forms result by method chaining
* In EF DbModelBuilder acts as fluent api class.
* To write Fluent API Configuration we need to override OnModelCreating Method of DbContext class.

**protected override void OnModelCreating(DbModelBuilder modelBuilder)**

{

//Write Fluent API configurations here

}

* Data Annotation attributes as well as fluent api can be used at same time.
* Fluent API configures the following aspect of a model in Entity Framework 6:
  + Model-wide Configuration: Configures the default Schema, entities to be excluded in mapping, etc.
  + Entity Configuration: Configures entity to table and relationship mappings e.g. PrimaryKey, Index, table name, one-to-one, one-to-many, many-to-many etc.
  + Property Configuration: Configures property to column mappings e.g. column name, nullability, Foreignkey, data type, concurrency column, etc.

# Configuring One to One relationship

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

public virtual StudentAddress Address { get; set; }

}

public class StudentAddress

{

[ForeignKey("Student")]

public int StudentAddressId { get; set; }

public string Address1 { get; set; }

public string Address2 { get; set; }

public string City { get; set; }

public int Zipcode { get; set; }

public string State { get; set; }

public string Country { get; set; }

public virtual Student Student { get; set; }

}

* Both classes have a virtual navigation properties.

# Configuring One to Many

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

**public ICollection<Student> Students { get; set; }**

}

* Here a single grade has multiple students associated to it

# Configuring Many to Many

ublic class Student

{

public Student()

{

this.Courses = new HashSet<Course>();

}

public int StudentId { get; set; }

[Required]

public string StudentName { get; set; }

public virtual ICollection<Course> Courses { get; set; }

}

public class Course

{

public Course()

{

this.Students = new HashSet<Student>();

}

public int CourseId { get; set; }

public string CourseName { get; set; }

public virtual ICollection<Student> Students { get; set; }

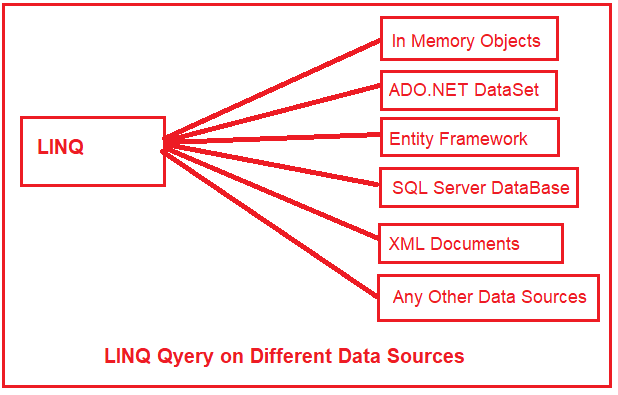
# Code with an existing database

LINQ (Language Integrated Query)

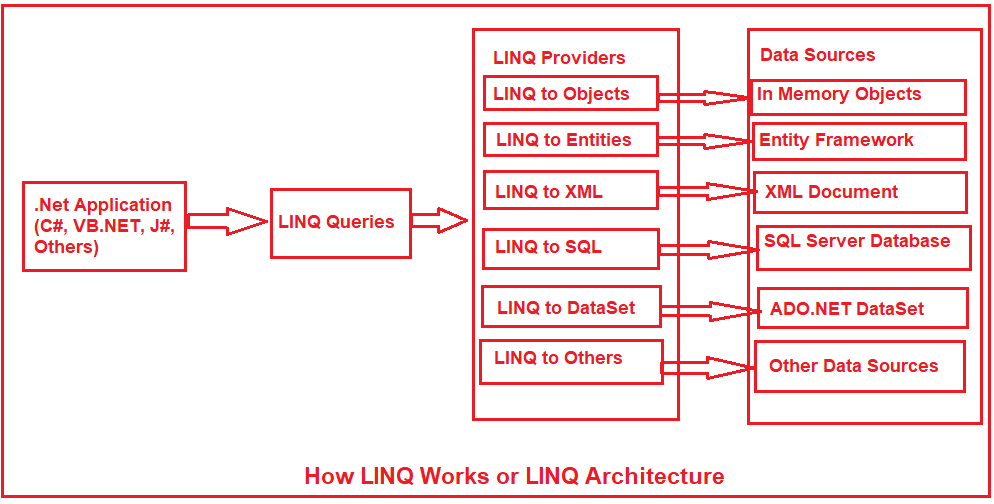
# LINQ

# Overview

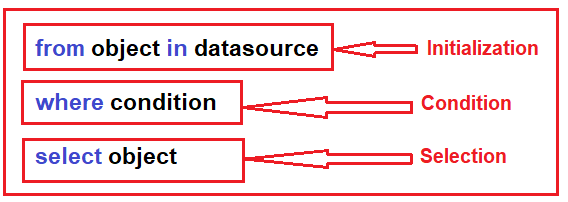
* It provides syntax to allow us **to query data from various sources** like sql server , xml, ADO.NET



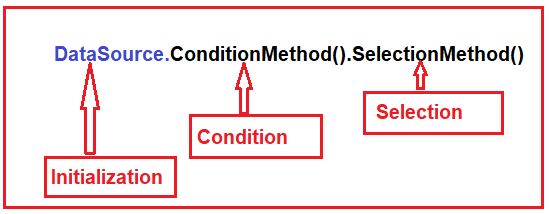
* It provides **uniform programming model** which works with various types of data



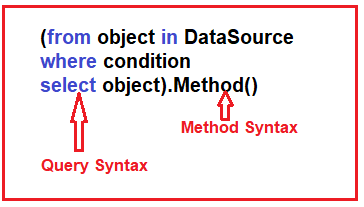
* **LINQ Providers** implement **IQueryable** and **IQueryProvider** interface for various data source.
* Every LINQ query **has initialization , condition and selection**
* We can write queries in 3 ways -> **Query Syntax, Method Syntax , Mixed(Query + Method)**
* Query Syntax :



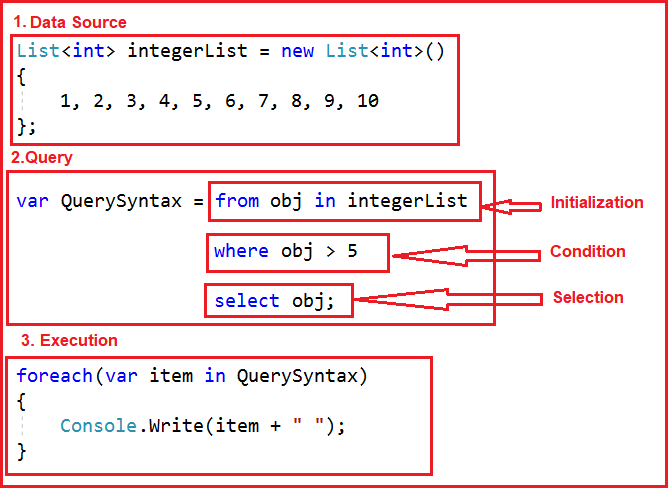
* Method Syntax



* Mixed Syntax



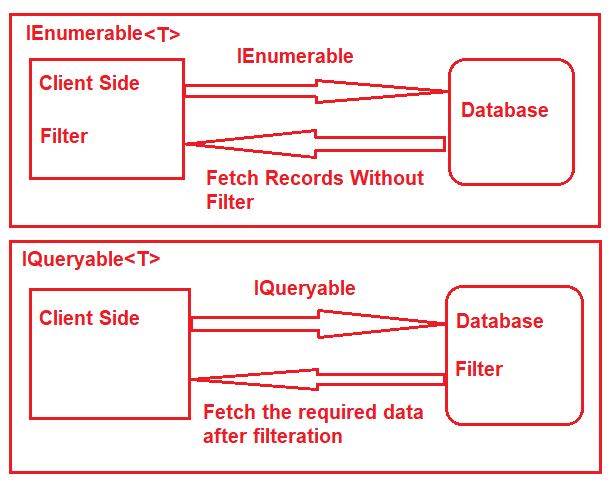
* EXAMPLE



# IEnumerable and IQueryable

* IEnumerable is an interface in **system collection** and is of **iteration design pattern**
* It has method **GetEnumerator**() which returns **IEnumerator**
* IQueryable is an interface in **System.Linq**
* IQueryable is child of IEnumerable and can be stored in an IEnumerable type
* IQueryable has a property called Provider which is of type IQueryProvider
* IQueryProvider is used to create all linq providers

## IEnumerable vs IQueryable



* IEnumerable applies **select statement on server side** (like db) and then applied **filtering when the data is in memory.**
* IEnumerable is suitable for **querying data from in-memory collections** like Array, Lists.
* IEnumerable & IQueryable supports **deferred execution and lazy loading**.
* IQueryable **selects and filters the data on server side itself**
* IQueryable is suitable for **querying data out of memory** like remote databases.

# IEnumerable vs IList

* IList can access an element at a particular position
* It is suitable for querying data in memory.
* IList also **supports add and remove functionality**
* IList can find number of elements in collection without querying all elements in the list.
* IList supports deferred execution

# Var and IEnumerable

* Var is used to declare **implicitly typed local variable** i.e. compiler figures out variable type at compile time. Var is used to create custom type on the fly
* Var **is anonymous type**

# Operators

|  |  |
| --- | --- |
| **Filtering** | Where, OfType |
| **Sorting** | OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse |
| **Grouping** | GroupBy, ToLookup |
| **Join** | GroupJoin, Join |
| **Projection** | Select, SelectMany |
| **Aggregation** | Aggregate, Average, Count, LongCount, Max, Min, Sum |
| **Quantifiers** | All, Any, Contains |
| **Elements** | ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault |
| **Set** | Distinct, Except, Intersect, Union |
| **Partitioning** | Skip, SkipWhile, Take, TakeWhile |
| **Concatenation** | Concat |
| **Equality** | SequenceEqual |
| **Generation** | DefaultEmpty, Empty, Range, Repeat |
| **Conversion** | AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList |

# Expression in LINQ

* The lambda expressions can be assigned to Func or Action types delegates

Example Func<Student, bool> isTeenAger = s => s.Age > 12 && s.Age < 20;

And now we can convert this Func type delegate into expression

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20;

* To invoke an expression we first need to compile it and this can be done as

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20;

//compile Expression using Compile method to invoke it as Delegate

Func<Student, bool> isTeenAger = isTeenAgerExpr.Compile();

//Invoke

bool result = isTeenAger(new Student(){ StudentID = 1, StudentName = "Steve", Age = 20})

# Expression Tree

* It is expressions arranged in a tree like data structure.

Expression.Lambda<Func<Student, bool>>(

Expression.AndAlso(

Expression.GreaterThan(Expression.Property(pe,"Age"), Expression.Constant(12, typeof(int))),

Expression.LessThan(Expression.Property(pe,"Age"), Expression.Constant(20, typeof(int)))),

new[] { pe });

# Deferred and Lazy Operators

Based on the above two types of execution, the LINQ operators are divided into 2 categories. They are as follows:

1. **Deferred or Lazy Operators:**These query operators are used for deferred execution. For example – select, SelectMany, where, Take, Skip, etc. are belongs to Deferred or Lazy Operators category.
2. **Immediate or Greedy Operators:**These query operators are used for immediate execution. For Example – count, average, min, max, First, Last, ToArray, ToList, etc. are belongs to the Immediate or Greedy Operators category.

# LINQ 101 Samples

# Projections

## Select Clause

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numsPlusOne = from n in numbers

select n + 1;

## Selecting a Single Property

List<Product> products = GetProductList();

var productNames = from p in products

select p.ProductName;

## Transform With Select

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

string[] strings = { "zero", "one", "two", "three", "four"};

var textNums = from n in numbers

select strings[n];