Framework 4.5 : Reminder of C#

2- Var Keyword (implicit typing)

C# is strongly typed (not loosely typed) language, ***var*** tells the compiler to infer the variable at initialisation, and if there is any confusion at initialisation we will get a compiling error:

```
var nullVar = null; //will give error since null is used for all types
var nullVarInitialized = (string)null; //this will work since it's initialized
var varString = "John"; //this will work since it's initialized
```

3- Generics

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            var result = new Result<String,int> { success = true, data = "String Data
1", extraData=10 };
            //Console.WriteLine(result.success);
            //Console.WriteLine(result.data);
            //Console.ReadKey();
            Helper.print(result);
            var result2 = new Result<int,string> { success = true, data = 10,
extraData="String Data 2" };
            //Console.WriteLine(result2.success);
            //Console.WriteLine(result2.data);
            Helper.print(result2);

            Console.ReadKey();
        }
    }

    public class Helper {
        public static void print<T,U>(Result<T,U> result) {
            Console.WriteLine(result.success);
            Console.WriteLine(result.data);
            Console.WriteLine(result.extraData);
        }

    }

    public class Result<T,U>{
        public bool success { get; set; }
        public T data { get; set; }
        public U extraData { get; set; }
    }
}
```

4- Attributes

Attribute is a simple class that inherits from a class attribute and used as decorator. It's used to restrict a type, property or method inside the decorated class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            var types = from t in Assembly.GetExecutingAssembly().GetTypes()
                        where t.GetCustomAttributes<SampleAttribute>().Count() > 0
//get the class where we applies the SampleAttribute decorators
                         select t;

              foreach (var t in types){
                  Console.WriteLine(t);
                  foreach (var p in t.GetProperties())
                  {
                      Console.WriteLine(p);
                  }

                  foreach (var m in t.GetMethods())
                  {
                      Console.WriteLine(m);
                  }
                  }

              /*
               * OUTPUT
                ConsoleApplication1.Test

                Int32 IntValue

                Int32 get_IntValue()
                Void set_IntValue(Int32)
                Void Method()
                System.String ToString()
                Boolean Equals(System.Object)
                Int32 GetHashCode()
                System.Type GetType()
               */

              Console.ReadKey();
        }
    }


//[AttributeUsage(AttributeTargets.Class|AttributeTargets.Property|AttributeTargets.Me
thod)]
    [AttributeUsage(AttributeTargets.Class)]
    public class SampleAttribute : Attribute {
        public string Name { get; set; }
```

```
        public int Version { get; set; }

    }

    [Sample(Name = "John", Version = 1)] //<- sugar syntax! or  [SampleAttribute(Name
="John", Version=1)]
    public class Test {
        public int IntValue { get; set; }
        public void Method() {}
    }

    public class NoAttribute { }
}
```

5- Reflexion

Avoid using it because of a slight performance hit.

a - Reflexion

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        /*
         *  OUTPUT
                ConsoleApplication1, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
         *
                Type: Program - BaseType: System.Object
                        Method: ToString - ReturnType: System.String
                        Method: Equals - ReturnType: System.Boolean
                        Method: GetHashCode - ReturnType: System.Int32
                        Method: GetType - ReturnType: System.Type
         *
                Type: Sample - BaseType: System.Object
                        Property: Name - PropertyType: System.String
                        Method: get_Name - ReturnType: System.String
                        Method: set_Name - ReturnType: System.Void
                        Method: MyMethod - ReturnType: System.Void
                        Method: ToString - ReturnType: System.String
                        Method: Equals - ReturnType: System.Boolean
                        Method: GetHashCode - ReturnType: System.Int32
                        Method: GetType - ReturnType: System.Type
                        Field: Age - FieldType: System.Int32
         *
                Property: John
         *
                Hello From MyMethod!
         */
```

```
static void Main(string[] args)
      {
          var assembly = Assembly.GetExecutingAssembly();
          Console.WriteLine(assembly.FullName);

          var types = assembly.GetTypes();
          foreach (var t in types)
          {

              Console.WriteLine("Type: " + t.Name + " - BaseType: " + t.BaseType);

              var props = t.GetProperties();
              foreach (var p in props)
              {
  Console.WriteLine("\tProperty: " + p.Name +" - PropertyType: " + p.PropertyType);
              }

              var methods = t.GetMethods();
              foreach (var m in methods)
              {
 Console.WriteLine("\tMethod: " + m.Name + " - ReturnType: " + m.ReturnType);
              }

              var fields = t.GetFields();
              foreach (var f in fields)
              {
 Console.WriteLine("\tField: " + f.Name + " - FieldType: " + f.FieldType);
              }

          }


          /*
           * Manipulation of Reflexion
           *
             OUTPUT-> Property: John
           */
          var sample = new Sample { Name = "John", Age = 25 };
          //'typeof(Sample)' is a compile time operation whereas 'sample.GetType()'
          //a runtime operation we don't know for sure which
          //type we will get!!!
          var sampleType = typeof(Sample); // sample.GetType();
          var nameProperty = sampleType.GetProperty("Name");
          Console.WriteLine("Property: " + nameProperty.GetValue(sample));

          var myMethod = sampleType.GetMethod("MyMethod");

          //we need to specify the object which we run the method on ('sample') and
          //the params if any
          myMethod.Invoke(sample, null);

          Console.ReadKey();
      }
  }
```

```
public class Sample {
        /*
          Sugar Syntax : C# creates behind the scene!!!
          Method: get Name - System.String
          Method: set_Name - System.Void
         */
        public string Name { get; set; }
        public int Age;

        public void MyMethod(){
            Console.WriteLine("Hello From MyMethod!");
        }
    }


}
```

b - Reflexion

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    /*
        Sample
        MyMethod
     */
    class Program
    {
        static void Main(string[] args)
        {
            var assembly = Assembly.GetExecutingAssembly();
            var types = assembly.GetTypes().Where(t =>
t.GetCustomAttributes<MyClassAttribute>().Count() > 0);

            foreach (var type in types)
            {
                Console.WriteLine(type.Name);
                var methodTypes = type.GetMethods().Where(t =>
t.GetCustomAttributes<MyMethodAttribute>().Count() > 0);
                foreach (var m in methodTypes)
                {
                    Console.WriteLine(m.Name);
                }
            }
            Console.ReadKey();
        }
    }

     [MyClass]
     public class Sample {
        /*
          Sugar Syntax : C# creates behind the scene!!!
          Method: get_Name - System.String
          Method: set_Name - System.Void
         */
```

```
        public string Name { get; set; }
        public int Age;
        [MyMethod]
        public void MyMethod(){
            Console.WriteLine("Hello From MyMethod!");
        }
        public void NoAttributeMethod() { }
    }
    [AttributeUsage(AttributeTargets.Class)]
    public class MyClassAttribute : Attribute {
    }
    [AttributeUsage(AttributeTargets.Method)]
    public class MyMethodAttribute : Attribute {
    }
}
```

6- Delegates

Delegate is class that encapsulates functionalities or a method and treat it as 1st class object and pass it around as argument or as return type or you can chain multiple methods together or remove them from the chain and execute them all in a sequence **(Chaining)** whenever you like.
The compiler creates behind the scenes the class delegate and we need to create a delegate with the same signature as the method which encapsulates it.

a - Delegates

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    //keyword 'delegate' + method/func return type + method/func argument type
    delegate void MySimpleDelegate();
    delegate void MyDelegate(string Name);

    class Program
    {
        static void SayHello()
        {
            Console.WriteLine("Hey There!");
        }

        static void SayHelloWithParams( string Name)
        {
            Console.WriteLine("Hey There, {0}!", Name);
        }

        static void Main(string[] args)
        {
            //0 - Calling a func
            SayHello();
```

```
            //1- traditional way of using delegate:  a class which encapsulates a
            //function
            MySimpleDelegate simpleDel = new MySimpleDelegate(SayHello);
            simpleDel.Invoke();

            //2-Sugar or shorthand syntax
            MySimpleDelegate sugarSyntaxDel = SayHello;
            sugarSyntaxDel();

            //3- Passing a function as delegate
            Test(SayHello);

            //4- Sugar or shorthand syntax
            MyDelegate sugarSyntaxDelWithParams = SayHelloWithParams;
            sugarSyntaxDelWithParams("Kejeiri");

            //5- Delegate with params, passing a function with params
            TestWithParams(SayHelloWithParams);


            //6- Using a function to create a delegate
            MyDelegate DelWithParams = GiveMeMyDelegate();
            DelWithParams("Kejeiri: GiveMeMyDelegate");


            Console.ReadKey();
        }

        static void Test(MySimpleDelegate del) {
            del();
        }

        static void TestWithParams(MyDelegate del)
        {
            del("Adam");
        }

        static MyDelegate GiveMeMyDelegate() {
            return new MyDelegate(SayHelloWithParams);
        }
    }
}
```

b - Delegates

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    delegate void Operation(int num);
    class Program
    {
        static void Main(string[] args)
        {
            Operation op= Double;
            executeOperation(2, op); //Or Just  op(2);
            op = Triple;
            executeOperation(2, op); //Or Just  op(2);

            //Chaining!!!
            Console.WriteLine("---- Chaining --- ");
            Operation opChain;
            opChain = Double;
            opChain = opChain + Triple; //OR ...
            opChain += Triple;
            opChain += Double;
            opChain -= Triple;
            opChain -= Double;
            executeOperation(2, opChain); //Or Just  op(2);

            Console.ReadKey();
        }

        static void Double(int num) {
            Console.WriteLine("{0} x 2 = {1}", num, num * 2);
        }
        static void Triple(int num)
        {
            Console.WriteLine("{0} x 3 = {1}", num, num * 3);
        }
        static void executeOperation(int num, Operation op) {
            op(num);
        }
    }
}
```

7 - Anonymous Methods and Lambda Expressions

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    delegate void OperationVoid(int num);
    delegate int OperationFunc(int num);
    class Program
    {
        static void Main(string[] args)
        {
            //0 - Delegate
            OperationVoid op = Double;
            op(2);

            //1 - Anonymous Method :allow us to skip to delcare "Double" Method and write the
functionality inline...
            //there is no limitation on how many line of code inside anonymous method/func,
however if it grows the code will become unreadable
            //keep 3 lines max!

            OperationVoid AnonymousMethod = delegate(int num){
                Console.WriteLine("{0} x 2 = {1}", num, num * 2);
                Console.WriteLine("{0} x 3 = {1}", num, num * 3);
            };
            AnonymousMethod(2);

            //2 - Anonymous function
            OperationFunc opFunc = delegate(int num)
            {
                return num * 2;
            };
            int IntValue = opFunc(2);
            Console.WriteLine("Anonymous function Result: {0}",IntValue);


            /*
             *  Parenthesis are required in case of multiple params and the type could also be
infered
             */
            //OperationVoid LAMDA = (int num) => { Console.WriteLine("LAMDA: {0} x 2 = {1}",
num, num * 2); };
            OperationVoid LAMDA =  num => { Console.WriteLine("LAMDA: {0} x 2 = {1}", num, num
* 2); };
            LAMDA(2);


            /*
             * to skip the declaration of delegate we could also use generic delegates such as
:
             * Action<> (doesn't return value) OR Func<> (retrun value) both  are built-in
delegate
             */

            Action<int> opAction = num => { Console.WriteLine("Action: {0} x 2 = {1}", num, num
* 2); };
            opAction(2);

            Func<int, int> opFunction = num => { return num * 2; };
             Console.WriteLine("func: {0} ", opFunction(2));
             Console.ReadKey();
        }
```

```
        static void Double(int num) {
            Console.WriteLine("{0} x 2 = {1}", num, num * 2);
        }
    }
}
```

8 - Events

An event is a way for an object to subscribe to an event happening in other object and do some logic around that.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    public delegate void ChimeEventHandler(object sender, ClockTowerEventArgs e);
   class Program
    {
        static void Main(string[] args)
        {
            var tower = new ClockTower();
            var john = new Person("John", tower);
            tower.Chime +=(s, e)=> {
                Console.WriteLine("{0} : heard the clock chime.", john.GetName());
                switch (e.message)
                {
                    case 6: Console.WriteLine("{0} is wakking up!", john.GetName());
                        break;
                    case 17: Console.WriteLine("{0} is going home!", john.GetName());
                        break;
                }
            };
            tower.ChimeAtSixAm();
            tower.ChimeAtFivePm();
            Console.ReadKey();
        }
    }

    public class ClockTower
    {
        public event ChimeEventHandler Chime;

        public void ChimeAtSixAm()
        {
            var chimeMessage = new ClockTowerEventArgs(6);
            Chime(this, chimeMessage);
        }

        public void ChimeAtFivePm()
        {
            var chimeMessage = new ClockTowerEventArgs(17);
            Chime(this, chimeMessage);
        }
    }
    public class Person
    {
        string _name;
        ClockTower _tower;
```

```
        public Person(string Name, ClockTower Tower)
        {
            this._name = Name;
            this._tower = Tower;
        }

       public string GetName()
        {
            return this._name;
        }
    }

    public class ClockTowerEventArgs : EventArgs {

        public int message { get; set; }
        public ClockTowerEventArgs(int Message){
            this.message = Message;
        }
    }
}
```

9 – Extension Methods

```
 Use extension if you cannot add methods/function to the class in the usual way,
 more often the native classes...
 Step to follow to add extension:
 1- Create a  public static class such as Extensions
 2- Create a  public static Method or function, first param should be preceded by
    this keyword and type of class you should operates on
    (e.g. : public static void SayHello(this Person person))
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var john = new Person() { Name = "John", Age = 32 };
            var Sally = new Person() { Name = "Sally", Age = 33 };
            john.SayHello();
            Sally.SayHello();
            john.SayHelloTo(Sally);
            Sally.SayHelloTo(john);
            Console.ReadKey();
        }
    }

     public static class Extensions
     {
         public static void SayHello(this Person person) {
             Console.WriteLine("{0} says hello", person.Name);
         }
         public static void SayHelloTo(this Person person, Person personTo) {
             Console.WriteLine("{0} says hello to {1}", person.Name, personTo.Name);
         }
     }
     public class Person {
         public string Name { get; set; }
         public int Age { get; set; }
     }
}
```

10 – LINQ

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var Peoples = new List<Person>() {
                new Person() {FirstName="John", LastName="Doe", Age = 25},
                new Person() {FirstName="Jane", LastName="Doe", Age = 27},
                new Person() {FirstName="John", LastName="Williams", Age = 30},
                new Person() {FirstName="Samantha", LastName="Williams", Age = 35},
                new Person() {FirstName="Bob", LastName="Walters", Age = 36}
            };

            var result = from p in Peoples
                         orderby p.LastName descending
                         group p by p.LastName;
            foreach (var res in result)
            {
                Console.WriteLine("{0} - {1}", res.Key, res.Count());
                foreach (var p in res)
                {
                    Console.WriteLine("\t\t {0} , {1}", p.FirstName, p.LastName);
                }
            }
            Console.ReadKey();
        }
    }
     public class Person {
         public string FirstName { get; set; }
         public string LastName { get; set; }
         public int Age { get; set; }
     }
}
```

11 - Anonymous Types

Allow us to trim down all unneeded properties. Anonymous types are immutable (anonymous properties are ***read only***).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var Peoples = new List<Person> {
            new Person(){FirstName="John", LastName="Doe", Age=25},
            new Person(){FirstName="Jane", LastName="Doe", Age=29},
            new Person(){FirstName="Bob", LastName="Williams", Age=19},
            new Person(){FirstName="Walker", LastName="Williams", Age=35},
            new Person(){FirstName="Jenny", LastName="Doe", Age=40},
            new Person(){FirstName="Harry", LastName="Harper", Age=38},
            };
            var result = from p in Peoples
                         where p.LastName == "Doe"
                         select p;
            foreach (var res in result)
            {
                Console.WriteLine("{0} - {1}", res.LastName, res.FirstName);
            }

            Console.WriteLine("\n---------    Anonymous    ---------\n");
            var resultAnonymous = from p in Peoples
                                  where p.LastName == "Doe"
                                  select new {LName= p.LastName, FName= p.FirstName};


            foreach (var res in resultAnonymous)
            {
                Console.WriteLine("{0} - {1}", res.LName, res.FName);
                //Error: Property or indexer 'AnonymousType#1.LName'
                //cannot be assigned to -- it is read only
                //res.LName = "NotPossible";
            }
            Console.ReadKey();
        }
    }
     public class Person {
         public string FirstName { get; set; }
         public string LastName { get; set; }
         public int Age { get; set; }

         public string MyProperty1 { get; set; }
         public int MyProperty2 { get; set; }
         public int MyProperty3 { get; set; }
     }

}
```

12 - The dynamic Keyword and Late Binding

Use python: dynamic keyword

```
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var pythonRuntime = PythonRuntime.CreateRuntime();
            dynamic pythonFile = pythonRuntime.UseFile("file.py");
            pythonFile.SayHelloToPython();
            Console.ReadKey();
        }
    }

}
```

file.py

```
import sys;
 def SayHelloToPython():
      print "Hello there C#"
      print "Nice to finally chat"
```

dynamic allow us to bypass the compiler and everything is checked at runtime.

ViewBag in MVC allow us to add dynamically properties because it uses the ***ExpandoObject()***

```
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            dynamic test = new ExpandoObject();
            test.Name = "John";
            test.Age = 25;
            Console.WriteLine("Name {0} and Age {1}", test.Name, test.Age);
            Console.ReadKey();
        }
    }

}
```

13 Optional Parameters

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            //Optional parameters must appear after all required parameters
            //Optional parameters need to be specified in order of the method signature :
            //Argument 2: cannot convert from 'int' to 'string'
            PrintData("John");
            PrintData("Sally", "Williams",35);

            //use this if you don't mind the order
            PrintData(age:35, lastName:"Doe", firstName:"John");
            PrintData("John", age:35);

            Console.ReadKey();
        }
        private static void PrintData(string firstName, string lastName = null, int age = 0)
        {
            Console.WriteLine("{0} {1} is {2} year old.",firstName, lastName, age);
        }
    }
}
```

14 - Task Parallel Library

Run process on multicore. Running parallel task on a single core will have an extra cost, it cause a lot of context switching trying to execute the processes as parallel as possible ➔ slowing down of performance. Avoid using the old Threading library (a lot of overhead!) and use this new MS approach.

a- traditional way :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var t1 = new Task(() => DoSomeImportantWork(1, 1500));
            t1.Start();
            var t2 = new Task(() => DoSomeImportantWork(2, 3000));
            t2.Start();
            var t3 = new Task(() => DoSomeImportantWork(3, 1000)); //1 second
            t3.Start();
            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }
        private static void DoSomeImportantWork(int id, int sleepTime)
        {
            Console.WriteLine("{0} is begining", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed", id);
        }
```

```
    }}
```

b- Using Task.Factory :

No need to call task.start()!

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var t1 = Task.Factory.StartNew(() => DoSomeImportantWork(1, 1500));
            var t2 = Task.Factory.StartNew(() => DoSomeImportantWork(2, 3000));
            var t3 = Task.Factory.StartNew(() => DoSomeImportantWork(3, 1000));
            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }
        private static void DoSomeImportantWork(int id, int sleepTime)
        {
            Console.WriteLine("{0} is begining", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed", id);
        }
    }}
```

c – Chaining of task using continueWith :

Instead of waiting when a task is completed and start a new one MS has introduced ***continueWith*** method. We could chain as much as required on single task/thread using this method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var t1 = Task.Factory.StartNew(() => DoSomeImportantWork(1, 1500))
                                 .ContinueWith((prevTask) => DoSomeMoreImportantWork(1,3000));
            var t2 = Task.Factory.StartNew(() => DoSomeImportantWork(2, 3000))
                         .ContinueWith((prevTask) => DoSomeMoreImportantWork(2,2000));
            var t3 = Task.Factory.StartNew(() => DoSomeImportantWork(3, 1000))
                         .ContinueWith((prevTask) => DoSomeMoreImportantWork(3, 500));
            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }
        private static void DoSomeImportantWork(int id, int sleepTime)
        {
            Console.WriteLine("{0} is begining", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed", id);
        }
        private static void DoSomeMoreImportantWork(int id, int sleepTime)
        {
            Console.WriteLine("{0} is begining more work", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed more work", id);
        }
    }}
```

d – Wait for a task to complete :

We could press any key and interrupt the program at any time, but if we want to wait of any specific or all tasks to be completed we need to do this.
WaitAll will block the execution until the specified lists of tasks are completed.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var t1 = Task.Factory.StartNew(() => DoSomeImportantWork(1, 1500))
                                 .ContinueWith((prevTask) => DoSomeMoreImportantWork(1,3000));
            var t2 = Task.Factory.StartNew(() => DoSomeImportantWork(2, 3000))
                         .ContinueWith((prevTask) => DoSomeMoreImportantWork(2,2000));
            var t3 = Task.Factory.StartNew(() => DoSomeImportantWork(3, 1000))
                         .ContinueWith((prevTask) => DoSomeMoreImportantWork(3, 500));

            //meanwhile doing some other logic while waiting for all task to finish
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("Doing some other work");
                Thread.Sleep(250);
                Console.WriteLine("iteration : {0}", i);
            }
            var taskList = new List<Task> { t1, t2, t3 };
            Task.WaitAll(taskList.ToArray());

            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }
        private static void DoSomeImportantWork(int id, int sleepTime)
        {
            Console.WriteLine("{0} is begining", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed", id);
        }
        private static void DoSomeMoreImportantWork(int id, int sleepTime)
        {
            Console.WriteLine("{0} is begining more work", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed more work", id);
        }
    }}
```

e – using Parallel :

Parallel.ForEach and Parallel.For are blocking operations, inside them everything happens in parallel and without sequential order.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            var intList = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 30 };
            Parallel.ForEach(intList,(i) => Console.WriteLine("{0}",i));
            Parallel.For(1,100, (i) => Console.WriteLine("iteration : {0}", i));
            //execution will reach this point when the two parallel loops are completed
            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }
    }}
```

f – Cancellation token :

When we need to interrupt a thread in the middle of its execution because something went wrong with the program we use the cancellation token.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
     class Program
    {
        static void Main(string[] args)
        {
            //most common way to create a cancellation token
            //create a source which will be a reference throughout the code
            //we have to do manually the checking of cancellation signal,
            //there isn't an auto way.
            var source = new CancellationTokenSource();
            try
            {
              var t1 = Task.Factory.StartNew(() => DoSomeImportantWork(1, 1500, source.Token))
                    .ContinueWith((prevTask) => DoSomeMoreImportantWork(1, 500, source.Token));
               source.Cancel();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.GetType());
            }
            Console.WriteLine("Press any key to quit");
            Console.ReadKey();
        }

 private static void DoSomeImportantWork(int id, int sleepTime, CancellationToken token)
        {
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("DoSomeImportantWork: Cancelation requested");
                //we have to handle this exception in our code
                token.ThrowIfCancellationRequested();

            }
            Console.WriteLine("{0} is begining", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed", id);
        }
 private static void DoSomeMoreImportantWork(int id, int sleepTime, CancellationToken token)
        {
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("DoSomeMoreImportantWork: Cancelation requested");
                //we have to handle this exception in our code
                token.ThrowIfCancellationRequested();
            }
            Console.WriteLine("{0} is begining more work", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("{0} is Completed more work", id);
        }
    }}
```

15 Async & Await

Async & Await VS Task library are very similar:

- Task library: it is a library, its functionality/code that built into the framework to allow us to create tasks into async programming without directly to interact with Threads.

- Async & Await is a contruct of C# language that helps us to interact with that parallel library a little bit fluently from the language as opposed to dig deeply into the library its self.

```
public partial class Form1 : Form{
        public Form1(){
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e){

            // BigLongImportantMethod("John"); //blocking!!!

            //Cross-thread operation not valid: Control 'label1' accessed from a
            //thread other than the thread it was created on.
            //to solve this we need : TaskScheduler.FromCurrentSynchronizationContext()
            Task.Factory.StartNew(()=>BigLongImportantMethod("Sally"))
                    //Access the result property from previous task!
                        .ContinueWith(t => label1.Text = t.Result,
                                TaskScheduler.FromCurrentSynchronizationContext());

           //using Async much more simple!
            CallImportantMethodAsync();
            label1.Text = "Waiting...";
        }

        Private async void CallImportantMethodAsync(){
            var result = await BigLongImportantMethodAsync("Kejeiri");
            label1.Text = result;
        }
//execute a TY-a
        private Task<string> BigLongImportantMethodAsync(string Name){
            //do in the backgroud
            return Task.Factory.StartNew(()=> BigLongImportantMethod(Name));
        }
        private string BigLongImportantMethod(string Name){
            Thread.Sleep(3000);
            return "hello " + Name;
        }}
```