Task 1- Create a delegate for functions: Input Parameters (int,int), Return: void

Solution:

```
public delegate void MyDelegate(int a,int b);
```

Task 2- Create a delegate for functions: Input Parameters (), Return: float

Solution:

```
public delegate float MyDelegate();
```

Task 3- What is meant by following delegate?

```
public delegate void MyDelegate(Customer c);
```

Solution:

It means that it can hold address of a function which has return type as “void” and which takes a customer object.

Task 4- For following delegate, how will you provide it a function?

```
public delegate void MyDelegate(int a,int b);
```

Solution:

Let say you have following function (which can be hold by above delegate)

```
void Sum(int a, int b)
{
    //Some Logic
}
```

Now you can use following approaches to store this function “address” in delegate

```
MyDelegate func = new MyDelegate(Sum);
```

OR

```
MyDelegate func1 = Sum;
```

Then you may use following syntax to execute the “referenced” function.

```
func(10, 15);
func1(10, 20);
```

Task 5: Can a delegate object hold more than 1 function references?

Solution:

Yes it can. We call such delegate as “Multicast Delegate”

Delegate:

```
public delegate void MyDelegate(int a, int b);
```

Functions:

```
void Sum(int a, int b)
{
    //Some Logic
}

0 references
void DummyFunc(int a, int b)
{
    //Some Logic
}
```

How to add multiple functions in a delegate:

```
MyDelegate func = Sum;
func += DummyFunc;

func(10, 15);
```

Now here you have two “functions” in “func” variable. If you will call “func”, it will call first “Sum” function and then “DummyFunc”.

Here if you want to “already” added function reference from delegate object.

```
func -= DummyFunc;
```

Task 6: What is anonymous function? How do we create it?

Solution:

A function without a name is called anonymous function. You create it using “delegate” keyword. You don’t provide any return type it. In following code snippet, we are creating an anonymous function and assigning it to our delegate object.

```
MyDelegate func2 = delegate(int x, int y)
{
    //Some Logic
};
```

Task 7: How do we create a function using “Lambda” expression?

Solution:

Our Delegate

```
delegate int TestDelegate(int a);
```

In following code snippet “=>” is lambda expression. Input parameters are on left side and function body is on right side.

```
TestDelegate func1 = (x => x * 2);
```

It is equivalent to

```
TestDelegate func2 = delegate(int x)
{
    return x * 2;
};
```

If input parameters are more or body has more statements

```
MyDelegate func3 = (x, y) =>
{
    //Some Logic
};
```

Task 8: Can we use "delegate" reference variable as function parameter?

Answer: Delegate type is internally a class type. So yes, you can use it as function parameter etc. Check following example.

Here "**WriteString**" is a delegate type. "Subscribe" function receives a "function reference of type WriteString" and store it in "writer" object. "Write" function just call the "writer" function which would be holding "references" of functions. "MyStringWriter" class is unaware about the "logic" of functions being passed to "Subscribe". It knows only about "writer" object.

```
public delegate void WriteString(String s);
2 references
public class MyStringWriter
{
    WriteString writer = null;
    1 reference
    public void Subscribe(WriteString f)
    {
        if (writer == null)
            writer = f;
        else
            writer += f;
    }
    1 reference
    public void Write(String s)
    {
        if (writer != null)
        {
            writer(s);
        }
    }
}//end of StringWriter
```

In following code snipped, we are using above class. "ConsoleWriter" & "FileWriter" are two functions and their signatures match with "WriteString" delegate. So these functions can be passed to "Subscribe" function as parameter.

```
public class Test2
{
    1 reference
    void ConsoleWriter(String s)
    {
        Console.WriteLine(s);
    }
    1 reference
    void FileWriter(String s)
    {
        using (StreamWriter objStream = new StreamWriter("D:\\test.txt"))
        {
            objStream.Write(s);
            objStream.Close();
        }
    }
    0 references
    public void PerformTest()
    {
        MyStringWriter obj1 = new MyStringWriter();
        obj1.Subscribe(ConsoleWriter);
        obj1.Subscribe(FileWriter);
        obj1.Write("Hello World");
    }
}
```

Task 9: What is Func<> delegate & Action<> delegate templates?

Answer: To ease your work, Func<> delegate is provided. Instead of creating a delegate first to hold a function reference, you can directly use “Func<>” delegate.

So let say we want to use “Func<>” type to store address of following type of function.

```
String Sum(int a, int b)
{
    return "Hello World";
}
```

Here is how to use “Func<>” delegate. Here last parameter is always “return” type and other parameters are “input” parameters.

```
Func<int, int, String> f = Sum;
```

Note: if you are function has return type as “void”, you can’t use “Func<>” delegate. In that case you may use “Action<>” delegate. For example, here is a sample function

```
void Sum(int a, int b)
{
    //Some Logic
}
```

Here is how to use “Action<>” delegate. No need to mention “void” as return type.

```
Action<int, int> f2 = Sum;
```

Useful Links

- LINQ
  - Go through this link
    - http://stackoverflow.com/questions/16322/learning-about-linq
  - 101 Samples
    - https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b
    - http://linq101.nilzorblog.com/linq101-lambda.php
    - http://linqsamples.com/
- Delegates
  - http://www.tutorialspoint.com/csharp/csharp_delegates.htm
  - http://stackoverflow.com/questions/2019402/when-why-to-use-delegates

Extra for Reading

- IEnumerator
    - Supports a simple iteration over a non-generic collection.
    - IEnumerator is the base interface for all non-generic enumerators.
    - The **foreach** statement of the C# language hides the complexity of the enumerators. Therefore, using **foreach** is recommended instead of directly manipulating the enumerator.
    - An enumerator remains valid as long as the collection remains unchanged.
    - For the generic version of this interface see IEnumerator<T>
    - https://msdn.microsoft.com/en-us/library/system.collections.ienumerator(v=vs.100).aspx

- IEnumerable
    - Exposes an enumerator, which supports a simple iteration over a non-generic collection.
    - IEnumerable is the base interface for all non-generic collections that can be enumerated. For the generic version of this interface see System.Collections.Generic.IEnumerable<T>
    - It is a best practice to implement IEnumerable and IEnumerator on your collection classes to enable the **foreach**
    - However implementing IEnumerable is not required but "GetEnumerator" method should be provided.
- IQueryable
    - Provides functionality to evaluate queries against a specific data source wherein the type of the data is not specified.
    - The IQueryable interface inherits the IEnumerable interface so that if it represents a query, the results of that query can be enumerated.

Enumeration causes the expression tree associated with an IQueryable object to be executed. The definition of "executing an expression tree" is specific to a query provider. For example, it may involve translating the expression tree to an appropriate query language for the underlying data source.