Krishna Nigalye

Software Developer

Learning C-Sharp

Delegates

WHAT ARE DELEGATES?

- Delegates are **function pointers** in C# which are type safe.
- It is a *reference type* who signature is same as the method you will be passing to it as a parameter.

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

This is how we define a delegate. When a Compiler sees this delegate, it converts it into a class named '*AdditionDelegate* which inherits from a class called *MulticastDelegate* which is a built in .NET type.

- When delegate is called, under the hood 'Invoke' method of the delegate is called.
- Delegates can be passed to methods as an argument and can be returned from methods as a return type.

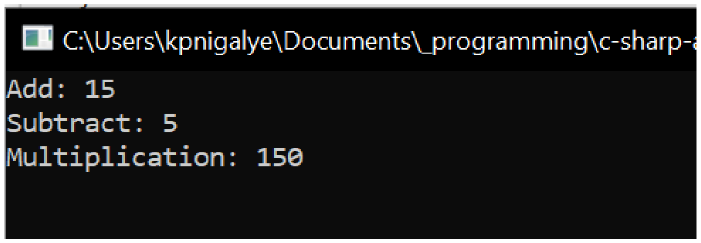
How to use Delegates?

Any method whose has the same signature (return type and parameters) can be invoked using this delegate. Consider following two methods which has the same method signature as our delegate so they can be invoked by the delegate.

```csharp
public static void Add(int a, int b)
{
    int c = a + b;
    Console.WriteLine($"Add: {c}");
}

public static void Substract(int a, int b)
{
    int c = a - b;
    Console.WriteLine($"Subtract: {c}");
}
```

The following code shows evolution and different ways of using delegates.

```csharp
// Traditional way of calling delegate using 'Invoke' function
ArithmaticDelegate operation1 = new ArithmaticDelegate(Add);
operation1.Invoke(5, 10);

// Assign method directly to delegate and pass arguments to delegate
ArithmaticDelegate operation2 = Substract;
operation2(15, 10);

// Using Lamda Expression
ArithmaticDelegate operation3 = (a, b) =>
{
    int c = a * b;
    Console.WriteLine($"Multiplication: {c}");
};
operation3(15, 10);
```



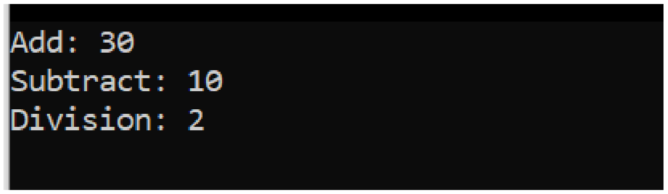
C:\Users\kpnigalye\Documents_programming\c-sharp-

```
Add: 15
Subtract: 5
Multiplication: 150
```

Multicast Delegates

- A delegate that can point to more than one method is a **Multicast Delegate** like this. You can add a function to the invocation list by using a '+' sign.

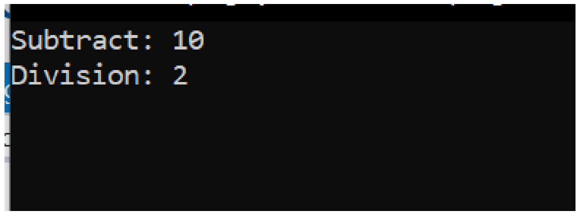
```
ArithmaticDelegate op1 = Add;
ArithmaticDelegate op2 = Substract;

ArithmaticDelegate operation = op1 + op2;

// Lamda Expression
operation += (a, b) =>
{
    int c = a / b;
    Console.WriteLine($"Division: {c}");
};
operation(20, 10);
```



```
Add: 30
Subtract: 10
Division: 2
```

You can remove a function to the invocation list by using a '-' sign.

```
ArithmaticDelegate op1 = Add;
ArithmaticDelegate op2 = Substract;

ArithmaticDelegate operation = op1 + op2;

// Lamda Expression
operation += (a, b) =>
{
    int c = a / b;
    Console.WriteLine($"Division: {c}");
};

operation -= op1;   // deletes op1

operation(20, 10);
```



```
Subtract: 10
Division: 2
```

- If a delegate returns a value, then the last assigned target method's value will be return when a multicast delegate called.

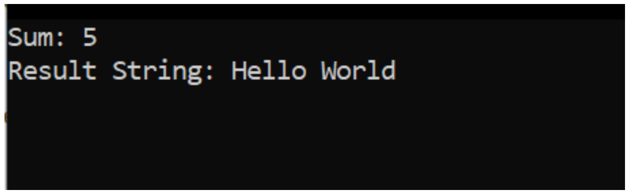
Generic Delegates

A generic delegate can be defined the same way as a delegate but using generic type parameters or return type. The generic type must be specified when you set a target method.

```
public delegate T AddDel<T>(T a, T b);
```

You can see the use of generic delegates below.

```
AddDel<double> sum = (a,b) =>
{
    return a + b;
};
Console.WriteLine($"Sum: {sum(3.5, 1.5)}");

AddDel<string> concat = (str1, str2) =>
{
    return str1 + str2;
};
Console.WriteLine($"Result String: {concat("Hello ", "World")}");
```

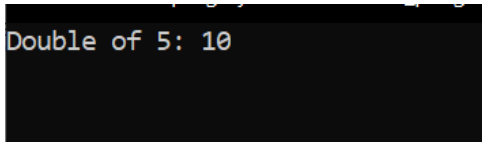


```
Sum: 5
Result String: Hello World
```

ANONYMOUS METHODS

- As the name suggests, anonymous method is a method without a name.
- It can be defined using 'delegate' keyword and can be assigned to variable of delegate type.

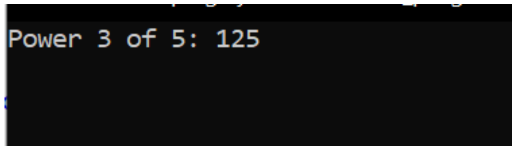
```
OperateDel operate = delegate (int a)
{
    Console.WriteLine($"Double of {a}: {a * 2}");
};

operate(5);
```



```
Double of 5: 10
```

- Anonymous methods can use a variable defined outside its scope.

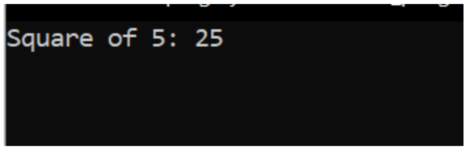
```
int p = 3;
OperateDel power = delegate (int a)
{
    Console.WriteLine($"Power {p} of {a}: {Math.Pow(a, p)}");
};
power(5);
```



```
Power 3 of 5: 125
```

- Anonymous methods can be defined using lambda expressions as well. **Lambda Expression** is a shorter way of representing anonymous method using some special syntax.

```
OperateDel operate = a => Console.WriteLine($"Square of {a}: {a * a}");
operate(5);
```



```
Square of 5: 25
```

- Both Anonymous methods and Lambda Expressions can also be passed to a method that accepts the delegate as a parameter.
- Anonymous methods can also be used as event handlers.

```
saveButton.Click += delegate(Object o, EventArgs e)
{
    System.Windows.Forms.MessageBox.Show("Save Successfully!");
};
```

Predefined Delegate Types

- There are 3 in-built delegate types namely **Func**, **Action** and **Predicate**.
- They are generic and in-built delegate type defined in System namespace.
- They can be used with Target methods, anonymous methods as well as lambda expression.

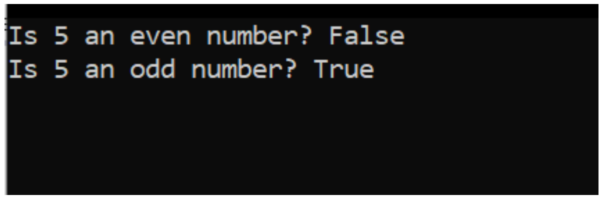
1. Func Delegate

- It always expects a return value.
- The last parameter is considered as an out parameter.
- They do not support ref or out keywords.

Let's consider following example of Func delegate. There is a func delegate which takes an int parameter as input and return bool as output.

You can see that; we have used target method as well as lambda expression in this code sample.

```
public static bool CheckIfEven(int number)
{
    return number % 2 == 0;
}

int n = 5;
Func<int, bool> func = CheckIfEven;

Console.WriteLine($"Is {n} an even number? {func(n)}");

func = (numb) => numb % 2 != 0;
Console.WriteLine($"Is {n} an odd number? {func(n)}");
```



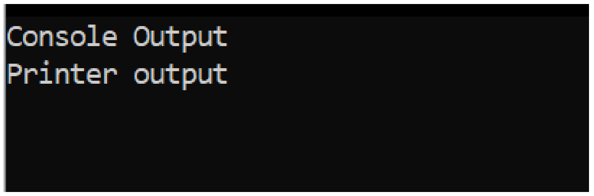
```
Is 5 an even number? False
Is 5 an odd number? True
```

2. Action Delegate

- This delegate does not return anything. In other words, in can be used with methods whose return type is void.

In the code below, we can see the use of Action delegate with an anonymous method and lambda expression.

```
Action<string> action = delegate (string message)
{
    Console.WriteLine(message);
};

action("Console Output");

action = (msg) => Console.WriteLine(msg);
action("Printer output");
```

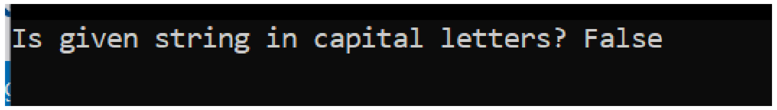


```
Console Output
Printer output
```

3. Predicate Delegate

- This delegate does not specify a return type but as the name suggests it always return **Boolean** as a result.

```
Predicate<string> predicate = (string str) => str.Equals(str.ToUpper());
Console.WriteLine($"Is given string in capital letters? {predicate("Hello World")}");
```



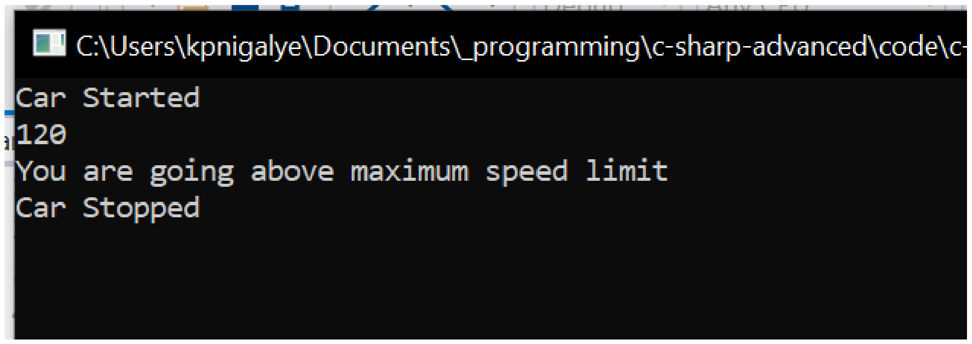
```
Is given string in capital letters? False
```

Practical Example

- Check out the following code which prints alerts based on change in car speeds.

```
public class Car
{
    public int MaxSpeedLimit { get; set; } = 100;
    public int MinSpeedLimit { get; set; } = 40;

    public void StartCar()
    {
        Console.WriteLine("Car Started");
    }

    public void StopCar()
    {
        Console.WriteLine("Car Stopped");
    }

    public void ChangeSpeed(int currentSpeed)
    {
        if (currentSpeed > MaxSpeedLimit)
        {
            Console.WriteLine("You are going above maximum speed limit");
        }
        else if (currentSpeed < MaxSpeedLimit)
        {
            Console.WriteLine("You are going below minimum speed limit");
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Car car = new Car();
        car.StartCar();

        int speed = Convert.ToInt32(Console.ReadLine());
        car.ChangeSpeed(speed);

        car.StopCar();

        Console.ReadLine();
    }
}
```


```
C:\Users\kpnigalye\Documents\_programming\c-sharp-advanced\code\c-
Car Started
120
You are going above maximum speed limit
Car Stopped
```

There is nothing wrong with this code but what we want to add more alerts in future may be alerts based on change in tyre pressure or engine manipulation etc. It’s not a good idea to keep on adding console alerts everywhere. Instead we can use delegates to print our alerts.

Here is how we define a delegate

```
public delegate void CarAlert(string message);
```

This delegate will accept any method which ***accepts a string*** parameter and whose ***return type is void***.

So, let’s define a method to print car alerts for speed.

```
static void SpeedChangeAlert(string message)
{
    Console.WriteLine(message);
}
```

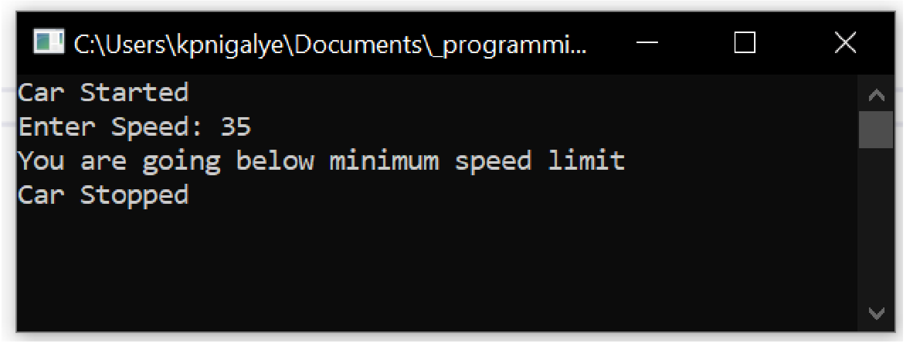
We have to make changes in our *ChangeSpeed()* method as well. Check out the code below.

```
public void ChangeSpeed(int currentSpeed, CarAlert carAlert)
{
    if (currentSpeed > MaxSpeedLimit)
    {
        carAlert("You are going above maximum speed limit");
    }
    else if (currentSpeed < MaxSpeedLimit)
    {
        carAlert("You are going below minimum speed limit");
    }
}
```

In Main method, we have pass function whose signature is same as our delegate type in our case ‘*SpeedChangeAlert*’.

```
car.ChangeSpeed(speed, SpeedChangeAlert);
```

Here is the output of the code.



```
Car Started
Enter Speed: 35
You are going below minimum speed limit
Car Stopped
```

As we can see, code still works the same way and we can use this delegate to print any type of errors.
This is a very simple example and it’s just for demo purpose only.