# Delegates, Events and Lambda Expressions

CSIS 3540

Client Server Systems

Class 04

©Michael Hrybyk and others
NOT TO BE REDISTRIBUTED

# Topics

- Delegates (only the simple version!)
- Generic Delegates
- Action and Func
- handlers and Events
- Lambda expressions

# Delegates

- Delegates are “pointers” to functions
  - Akin to those in C++
- A delegate “function” is any method with a signature consisting of parameters and a return value.
- Note: We could create a new BinaryOp with ANY class method that has two int parameters returning an int.
- See SimpleDelegate (partial code below)

```
// This delegate can point to any method,
   // taking two integers and returning an integer.
   public delegate int BinaryOp( int x, int y );

   public delegate int UnaryOp(int x);

// create operators classes to be used to assign to delegates
// note the methods of each class are unary or binary
   const int operand1 = 5;
   const int operand2 = 12;

   SimpleMathOperators simpleMathOps = new SimpleMathOperators();
   OtherMathOperators anotherMathOp = new OtherMathOperators();

           // Create a BinaryOp delegate object that
           // "points to" SimpleMathOperators.Add().

   BinaryOp addTheseTogether = new BinaryOp(simpleMathOps.Add);
   Console.WriteLine($"Add: {operand1} and {operand2} is {addTheseTogether(operand1, operand2)}");
```

# Generic Delegates

- Generics can be used with delegates as well
- See GenericDelegate - shows use of <string> and <int>

```
public delegate void MyGenericDelegate<T>( T arg );
static void Main( string[] args )
{
    WriteLine("***** Generic Delegates *****\n");

    // Register targets.
    // Note how the arg to MyGenericDelegate constructor is a function.

    MyGenericDelegate<string> stringFunction =
      new MyGenericDelegate<string>(MakeUpperCase);

    stringFunction("Some string data");

    stringFunction = new MyGenericDelegate<string>(MakeLowerCase);
    stringFunction("SOME MORE string DATA");


    MyGenericDelegate<int> integerFunction =
      new MyGenericDelegate<int>(IncrementInteger);

    integerFunction(9);

    integerFunction = new MyGenericDelegate<int>(DecrementInteger);
    integerFunction(201);

    Console.ReadLine();
}

static void MakeUpperCase( string arg )
{
    WriteLine($"MakeUpperCase: arg in uppercase is: {arg.ToUpper()}");
}

static void MakeLowerCase(string arg)
{
    WriteLine($"MakeLowerCasee: arg in lowercase is: {arg.ToLower()}");
}
static void IncrementInteger( int arg )
{
    WriteLine($"IncrementInteger: {arg}++ is: {++arg}");
}

static void DecrementInteger(int arg)
{
    WriteLine($"DecrementInteger: {arg}-- is: {--arg}");
}
```

# Action and Func

- Action is a delegate (pointer) to a method, that takes zero, one or more input parameters, **but does not return anything**.
- Func is a delegate (pointer) to a method, that takes zero, one or more input parameters, and **returns a value** (or reference).
  - The last argument is the return value
- Both can use the Invoke() method
- See ActionAndFuncDelegates sample code.

```
Action<string, ConsoleColor, int> actionTarget =
    new Action<string, ConsoleColor, int>(DisplayMessage);

Action<string, ConsoleColor, int> actionTarget2 = DisplayMessage;

actionTarget("Action Message!", ConsoleColor.Yellow, 5);

actionTarget = DisplayDifferentMessage;
actionTarget2 = DisplayMessage;

actionTarget("A Different Action Message!", ConsoleColor.Green, 5);

actionTarget2("Second Action Message", ConsoleColor.Red, 3);


Func<int, int, int> funcTarget = new Func<int, int, int>(Add);
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = new Func<int, int, string>(SumToString);
string sum = funcTarget2(90, 300);
Console.WriteLine($"SumToString 90 + 300 {sum}");

Func<int, int, int> funcTarget3 = Add;
int result2 = funcTarget3.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result2);

funcTarget3 = Subtract;
Console.WriteLine($"50 - 17 = {funcTarget3(50, 17)}");

Func<int, int, string> funcTarget4 = SumToString;
string sum2 = funcTarget4(80, 200);
Console.WriteLine($"SumToString 80 + 200 {sum2}");
```

```
static int Add(int x, int y)
{
        return x + y;
}

static string SumToString(int x, int y)
{
        return (x + y).ToString();
}

static int Subtract(int x, int y)
{
        return x - y;
}
```

# Action and Func

```
class FunWithActionAndFuncProgram
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Action and Func *****\n");

        Action<string, ConsoleColor, int> actionTarget = DisplayMessage;

        Func<int, int, int> funcTarget = Add;
        int result = funcTarget.Invoke(40, 40);

        actionTarget.Invoke($"Add() 40 + 40 = {result}", ConsoleColor.Red, 4);

        Func<int, int, string> funcTarget2 = SumToString;
        string sum = funcTarget2(90, 300);

        actionTarget.Invoke($"SumToString() 90 + 300 = {sum}", ConsoleColor.Blue, 2);

        Console.ReadLine();
    }

    // This is a target for the Action<> delegate.
    static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)
    {
        // Set color of console text.
        ConsoleColor previous = Console.ForegroundColor;
        Console.ForegroundColor = txtColor;

        for (int i = 0; i < printCount; i++)
        {
            Console.WriteLine(msg);
        }

        // Restore color.
        Console.ForegroundColor = previous;
    }

    // Targets for the Func<> delegate.
    static int Add(int x, int y)
    {
        return x + y;
    }

    static string SumToString(int x, int y)
    {
        return (x + y).ToString();
    }
}
```

# Example using callbacks
## (see CarDelegate and CarDelegateMethodConversion)

Now, consider the following updates, which address the first three points:

```
public class Car
{
  ...
  // 1) Define a delegate type.
  public delegate void CarEngineHandler(string msgForCaller);

  // 2) Define a member variable of this delegate.
  private CarEngineHandler listOfHandlers;

  // 3) Add registration function for the caller.
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers = methodToCall;
  }
}
```

Notice in this example that you define the delegate types directly within the scope of the `Car` class, which is certainly not necessary but does help enforce the idea that the delegate works naturally with this particular class. The delegate type, `CarEngineHandler`, can point to any method taking a single `string` as input and `void` as a return value.

Next, note that you declare a private member variable of your delegate type (named `listOfHandlers`) and a helper function (named `RegisterWithCarEngine()`) that allows the caller to assign a method to the delegate's invocation list.

# Handlers and Events

- Every delegate (generic or otherwise) has an associated handler (or function) that can be assigned to it.
  - Called registration
  - Very messy
- Instead use C# event
  - Can register multiple handlers with += operator
  - De-register using -=
  - This is used by Windows Forms!
  - Avoids the public delegate problem.
    - See **PublicDelegateProblem**, where one can call a delegate directly instead of waiting for an event to fire from registration.

# See FunWithEvents

```
class FunWithEventsProgram
{
    public delegate void HandleMathEvent(int c, int x, int y);

    public static event HandleMathEvent DoSomeMath;
    public static event HandleMathEvent DoSomeMoreMath;

    static void Main(string[] args)
    {
        Random rand = new Random();

        // just one handler for DoSomeMMath
        DoSomeMath += Add;

        // register two handlers for DoSomeMoreMath
        // both will fire when event occurs

        DoSomeMoreMath += Subtract;
        DoSomeMoreMath += RandomAdd;

        // flip a coin, if even DoSomeMath, otherwise DoSomeMoreMath

        for (int i = 0; i < 5; i++)
        {
            int choice = rand.Next(0, 2);
            if (choice == 0)
                DoSomeMath?.Invoke(choice, 2, 3);
            else DoSomeMoreMath?.Invoke(choice, 10, 7);
        }

        Console.ReadLine();
    }

    public static void Add(int c, int x, int y)
    {
        Console.WriteLine("Flip {3} Add: {0} + {1} = {2}", x, y, x + y, c);
    }

    public static void Subtract(int c, int x, int y)
    {
        Console.WriteLine("Flip {3} Subtract: {0} + {1} = {2}", x, y, x - y, c);
    }

    public static void RandomAdd(int c, int x, int y)
    {
        Console.WriteLine("Flip {3} RandomAdd: {0} + {1} = {2}", x, y, x + y + c, c);
    }
```

# Event Handlers and Anonymous Methods

- CarEvents
  - Example of multiple event handlers
  - If you hit TAB after += VS will fill in the rest for you!
- AnonymousMethods
  - Good examples of using anonymous delegates as event handler functions.
    - No need to declare the method separately.
  - Basis for lambdas!

# Events are important

- All of Windows Forms and other interactive schemes use this
  - Asynchronous
  - Callback
- Steps
  - Start program
  - Draw a screen and REGISTER button event handler
  - Wait
  - User hits button
  - Button event handler fires, executing handler code
- While we are waiting for the button to be clicked, we can be doing other things!
  - Otherwise it is just like the Console

# Lambda Expressions

- More expressive and easier to read version of delegate
- Where delegate spec's a function, lambda makes this easier to declare
- Delegate
  - public delegate int BinaryOp( int x, int y );
  - SimpleMath m = new SimpleMath();
  - BinaryOp b = new BinaryOp(m.Add);
- Lambda
  - (x, y) => (x + y) is the equivalent statement to all of the above, and can be used in any method with a Func argument
  - Args => Result
  - Much simpler
- See SimpleLambdaExpressions and LambdaExpressionsMultipleParameters

```
public delegate int BinaryOp(int x, int y);
static int Add(int x, int y) => x + y;
static void PrintSum(int x, int y)
    => Console.WriteLine("lambda add shorter: {0} + {1} = {2}", x, y, x + y);

static void SimpleLambdaExpression()
{
    BinaryOp b = (x, y) => (x + y);
    Console.WriteLine("lambda add: {0} + {1} = {2}", 2, 3, b(2,3));
    b = (x, y) => ((100 * x) + y);
    Console.WriteLine("lambda 100*x + y: {0} + {1} = {2}", 2, 3, b(2, 3));
    Console.WriteLine("lambda Add(): {0} + {1} = {2}", 2, 3, Add(2, 3));
    PrintSum(2, 3);

}
```

# Using lambdas as arguments in method invocation

- Any method that takes a Func or Predicate as an argument can be expressed by using a lambda.
  - See SimpleLambdaExpressions

```
Console.WriteLine("FindingEvenNumbersUsingLambdas: listing all numbers in FindAll lambda");

// argument to FindAll is a boolean that is the result of a matching function (predicate)

List<int> numbers = list.FindAll((i) =>
{
    Console.Write($"{i} ");
    return ((i % 2) == 0);
});

Console.WriteLine();

DisplayNumberList(numbers, "FindingEvenNumbersUsingLambdas, even numbers:");

int x = numbers.FindIndex(i => i == 8); // silly, could use numbers[8] :-)
Console.WriteLine("index of 8 is {0}", x);

// Example using LINQ. Note LINQ primitives/methods can use lambda args (predicates)

var selection = list
    .Where(i => i > 4)
    .OrderBy(i => i).ToList();

DisplayNumberList(selection, "FindingEvenNumbersUsingLambdas, sorted numbers greater than 4:");
```

# Lambdas are IMPORTANT

- in C# related to LINQ and SQL
- See below – looks like SQL!!

```
IEnumerable<int> selection = list
    .Where(i => i > 4)
    .OrderBy(i => i);

Console.WriteLine("Here are your numbers greater than 4:");
foreach (int s in selection)
{
    Console.WriteLine("{0}", s);
}
```