Delegates and Events

Function Pointers in C#

Contents

- What are delegates?
- What are they good for?
- Anonymous functions
- Passing Parameters
- Publisher-Subscriber Pattern
- Conclusion

What are Delegates?

- C# Delegates are a type which store a reference to a method
- They are similar to C++ function pointers, but are also type safe and much cleaner to use
- Unlike C++ function pointers, delegates can easily point to member functions
- If a delegate points to a virtual method, it will use the overriding version if available

Delegates

- Delegates are a user defined type, just like a class
  - Therefore, we define our delegate and then we can create an instance of it

- Defining a delegate is easy – it looks like a function signature, prepended with the delegate keyword

```
// Declare a delegate
delegate void TMyDelegate(int i, int j);
```

- A delegate of type TMyDelegate will be able to hold functions which take in two int variables and returns nothing

Delegates

- Delegates can hold both static and member functions, as long as they match the signature of the declared type

```
delegate void TMyDelegate(int, int);

static public void DoSomethingStatic(int i, int j)
{
    Console.WriteLine(i * j);
}
static void Main(string[] args)
{
    // create a variable to store a function
    TMyDelegate d;

    // assign a function to our delegate instance
    d = DoSomethingStatic;

    // somewhere within your program, you can
    // invoke the delegate calling any functions assigned.
    d(10, 20);
}
```

Example

```
class Foo
{
    public void DoSomething(int i, int j)
    { Console.WriteLine(i + j); }
}
```

```
class Program
{
    delegate void TMyDelegate(int, int);

    static void Main(string[] args)
    {

        // Create an Instance of type FOO
        Foo foo = new Foo();

        // create a variable to store a function
        TMyDelegate d;

        // assign a function to our delegate instance
        d = foo.DoSomething;

        // somewhere within your program, you can
        // invoke the delegate calling any method
assigned.
        d(10, 20);
    }
}
```

Delegates

- You can use the += operators to have a delegate point to more than one function at a time
- Functions will get called in the order they are added to the delegate
- Think of this as a list of function pointers within C++

```
class Program
{

    delegate void TMyDelegate(string word);

    static void PrintUpper(string word)
    {  Console.WriteLine(word.ToUpper()); }

    static void PrintLower(string word)
    { Console.WriteLine(word.ToLower()); }

    static void Print(string word)
    { Console.WriteLine(word); }

    static void Main(string[] args)
    {
        // create a variable to store a function
        TMyDelegate del;

        // assign a function to our delegate instance
        del = PrintUpper;
        del += PrintLower;
        del += Print;

        // call our delegate methods
        del("hello WORLD");

        Console.ReadLine();
    }
}
```

What are they good for

- Delegates are fantastic for providing custom functionality
- Lets say you create your own generic container type
  - You can provide a LessThan delegate to enable you to write a sorting method

```
static bool LessThan(string a, string b)
{
    return a.Length < b.Length;
}

static void Main(string[] args)
{
    MyContainerType myContainer<string> = new MyContainerType();
    myContainer.Add("hello");
    myContainer.Add("aaa");
    myContainer.Add("bbbb");

    myContainer.LessThanFunc = LessThan;

    myContainer.Sort();

    Console.ReadLine();
}
```

What are they good for

- They are also fantastic for handling events

```
static void HandleButtonClick(Button btn)
{
    Console.WriteLine(btn.m_name);
}

static void Main(string[] args)
{
    Button btn1 = new Button("btn1");
    Button btn2 = new Button("btn2");

    btn1.OnClick = HandleButtonClick;
    btn2.OnClick = HandleButtonClick;

    btn1.Update();
    btn2.Update();
}
```

```
class Button
{
    public delegate void TButtonEvent(Button btn);
    public TButtonEvent OnClick;

    public string m_name;

    public Button(string name)
    { m_name = name; }

    public void Update()
    {
        if( /*Mouse Clicks on button and */  OnClick !=
null)
        {
            // call the onclick delegate...
            OnClick(this);
        }
    }
}
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Passing a delegate as an argument

- Delegates can be passed around through function parameters, just like any other type of variable

```
delegate void TWordFunc(string word);

static void PrintWordUpper(string word)
{ Console.WriteLine(word.ToUpper()); }

static void PrintWordLower(string word)
{ Console.WriteLine(word.ToLower()); }

static void DoSomethingWithWord(TWordFunc func, string word)
{
    func( word );
}

static void Main(string[] args)
{

    DoSomethingWithWord( PrintWordUpper, "hello" );
    DoSomethingWithWord( PrintWordLower, "HELLO" );

    Console.ReadLine();
}
```

Anonymous and Lambda Functions

- C# provides us with the ability to create a function that is not assigned a function name – instead we can store functions as a delegate variable

- We can do this in 2 ways, using the anonymous function syntax or the lambda expression syntax

- Both methods have slight differences

Anonymous and Lambda Functions

- For the most part, these methods are interchangeable. Read here for the differences
  http://blogs.msdn.com/b/ericlippert/archive/2007/01/10/lambda-expressions-vs-anonymous-methods-part-one.aspx

  In a nutshell, anonymous functions were introduced in C# 2.0 and lambda Expressions were introduced in C# 3.0

```
delegate void TWordFunc(string word);

static void Main(string[] args)
{
    TWordFunc wordFunc1 = delegate(string word)
    {
        Console.WriteLine(word.ToUpper());
    };

    TWordFunc wordFunc2 = delegate(string word)
    {
        Console.WriteLine(word.ToUpper());
    };

    Console.ReadLine();
}
```

Example using anonymous function syntax

```
delegate void TWordFunc(string word);

static void Main(string[] args)
{
    TWordFunc wordFunc1 = (string word) =>
    {
        Console.WriteLine(word.ToUpper());
    };

    TWordFunc wordFunc2 = (string word) =>
    {
        Console.WriteLine(word.ToUpper());
    };

    Console.ReadLine();
}
```

Example using lamda expression syntax

Passing anonymous functions as arguments

- Therefore, we can do this with our previous example

WOW, This started looking like JavaScript!
Anyone used JQuery?

```
delegate void TWordFunc(string word);

static void DoSomethingWithWord(TWordFunc func, string word)
{
    func(word);
}

static void Main(string[] args)
{
    DoSomethingWithWord((string word) =>
    {
        Console.WriteLine(word.ToUpper());

    }, "hello");

    DoSomethingWithWord((string word) =>
    {
        Console.WriteLine(word.ToLower());

    }, "HELLO");

    Console.ReadLine();
}
```

Publisher Subscriber Pattern

- The publisher subscriber design pattern can be implemented easily with delegates
- This pattern is used where events that occur within your application are “published”
- Any object that is “subscribed” to the publisher will receive that type of event

Examples:

A Keyboard Publisher sends notifications about keys that have been pressed

The UI sends notifications when a button is clicked

In an RTS Game, a unit factory may send notifications when a unit has been created. The UI that has subscribed to the event can update its self, and the Sound Manager can listen for this event and play the appropriate sound! Near by enemies may be subscribed to the event and decide to change their target.

TIP: This design pattern is particular useful for keeping your classes decoupled from the rest of your application

Example

```
class ConsoleKeyEventPublisher
{
    public delegate void TKeyPressed(char key);

    // assign an empty lamda func
    private TKeyPressed KeyPressedEvent =
                (char key) => { };

    public void Subscribe(TKeyPressed func)
    {
        KeyPressedEvent += func;
    }

    public void Update()
    {
        if (Console.KeyAvailable)
        {
            char keyPressed =
                    Console.ReadKey(true).KeyChar;

            KeyPressedEvent( keyPressed );
        }
    }
}
```

```
class Program
{
    static bool shouldQuit = false;

    static void MakeBeep(char key)
    {  Console.Beep(key * 50, key * 5); }

    static void ListenForQuitKey(char key)
    { if (key == ' ') shouldQuit = true;  }

    static void Main(string[] args)
    {
        ConsoleKeyEventPublisher consoleKeyEventPublisher =
                new ConsoleKeyEventPublisher();

       consoleKeyEventPublisher.Subscribe(MakeBeep);
       consoleKeyEventPublisher.Subscribe(ListenForQuitKey);

        while (shouldQuit == false)
        {
            consoleKeyEventPublisher.Update();
        }
    }
}
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Summary

- Delegates are a C# type which store a reference to a method
  - The method can then be called instead by calling the delegate
- You can have a delegate point to more than one function at a time by using the += operator
- Anonymous functions and Lambda functions provide a convenient short-hand way of writing out our delegates
- Delegates are mainly used to handle events – the publisher-subscriber pattern is a good example of this

References

- Code Better, 2008, *Back to Basics: Anonymous Methods and Lambda Expressions.*
  - http://codebetter.com/karlseguin/2008/11/27/back-to-basics-delegates-anonymous-methods-and-lambda-expressions/
- Code Project, 2009, *Delegates, Anonymous Methods and Lambda Expressions*
  - http://www.codeproject.com/Articles/47887/C-Delegates-Anonymous-Methods-and-Lambda-Expressio
- Microsoft, 2014, *Delegates (C# Programming Guide)*
  - https://msdn.microsoft.com/en-us/library/ms173171.aspx