Delegates and Events

Mingsheng Hong
CS 215, Spring 2008

Announcements

- First assignment due *today*
- Second assignment released
  - due in one week

Review

- Function parameters: ref, out, params
- Iterators
- Advanced C# Types
  - Nullable types
  - Partial types
  - Generics

Roadmap for Today’s Lecture

- Delegates
- Anonymous methods
- Events

Motivation – Function Pointers

- Treat functions as first-class objects
  - Scheme: `(map myfunc somelist)`
  - C/C++:

    -
      ```
      typedef int (*fptr)(int);
      int apply(fptr f, int var) {
           return f(var);
      }
      int F(int var) { … }
      ```

    -
      ```
      fptr f = F;
      apply(f, 10);//same as F(10)
      ```
  - Java
    - no equivalent way to get function pointers
    - use inner classes (or interfaces) that contain methods

Delegates

- An objectified function
  - behaves like C/C++ style function pointer
  - inherits from `System.Delegate`
  - sealed implicitly
- eg. `delegate int Func(int x)`
  - defines a new type `Func`: takes `int`, returns `int`
  - declared like a function with an extra keyword
    - Contrast C syntax: `typedef int (*fptr)(int);`
  - stores a *list* of methods to call

Delegates – Example

```
delegate int Func(ref int x);
int Increment(ref int x) { return x++; }
int Decrement(ref int x) { return x--; }
Func F1 = new Func(Increment);
F1 += Decrement;
x = 10;
Console.WriteLine(F1(ref x));
Console.WriteLine(x);
```

- Delegate calls methods in order
  - ref values updated between calls
  - return value is the value of the last call

Delegates – Usage Patterns

- Declared like a function
- Instantiated like a reference type
  - takes a method parameter in the constructor
- Modified with +, -, +=, -=
  - can add more than one instance of a method
  - removes the last instance of the method in the list
- Called like a function
  - Invoking a delegate with an empty list of methods causes an error (exception)

Functional Programming Example

- Allows code written in a more functional style

```
delegate int Func(int x);
List<int> Map(Func d, List<int> l) {
      List<int> newL = new List<int>();
     foreach(int i in l) {
         newL.Add(d(i));
     }
     return newL;
}
```

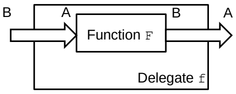
Instance Delegates

- `this` pointer captured by instance delegates
  - `delegate int Func(int x);`
  - `Class A {`
    ```
     public int F(int val) { … }
    }
    ```
  - `A a = new A();`
    ```
    Func f = new Func(a.F);
    f(10); //same as calling a.F(10)
    ```

Covariance & Contravariance

- Flexibility when matching method signatures with delegate types
  - **Covariance**: if the return type of a method derives from that of the delegate
  - **Contravariance:** if the type of a method parameter is a base class of the delegate parameter type
- E.g. let `B` derive from `A`

```
delegate A Func(B b);
B F(A a) {…}
Func f = new Func(F);
```



Anonymous Methods

```
//f is a delegate
int y = 10;
f += delegate(int x) { return x + y; }
```

- Creates a method and adds it to delegate
  - treated the same as other methods
  - good for one-time, short delegates
- Variables captured by anonymous method
  - **outer variables**
  - like `y` in the above example

Semantics of Outer Variables

```
using System;
delegate void D();
class Test {
   static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
                int x = i * 2 + 1;
                result[i]= delegate{Console.WriteLine(x);};
        }
        return result;
   }

   static void Main() {
        foreach (D d in F()) d();
   }
}
```

Semantics of Outer Variables

-
```
static D[] F() {
     D[] result = new D[3];
     int x;
     for (int i = 0; i < 3; i++) {
             x = i * 2 + 1;
             result[i] = delegate { Console.WriteLine(x); };
     }
     return result;
}
```
- First returns 1,3,5. Second returns 5,5,5
  - Outer variables are captured with *locations*
  - Not given values at delegate creation time
- Can communicate through outer variables

Events – Motivation

- Event-based programming
  - Events are raised by run-time
    - Indirectly through external actions, function calls
  - Client code registers *event handlers* to be invoked
    - Also called *callbacks*
    - Allows asynchronous computation
  - E.g. GUI programming
- Event-based programming in C#
  - Events – special delegates
  - Event handlers – functions

Events – Example

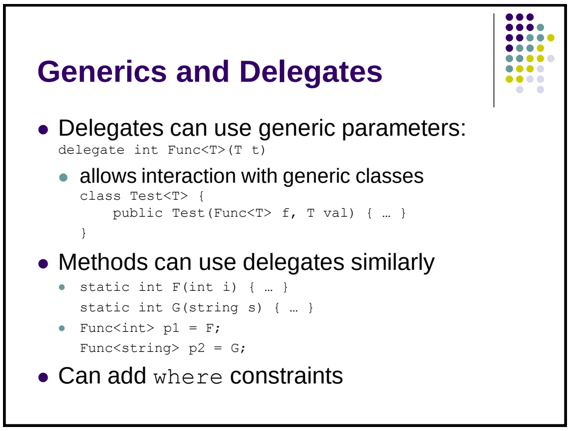
- Created from delegates using `event`
  - Declares a class member, enabling the class to *raise* events (i.e., to invoke the event delegate)
  -
```
public delegate void EventHandler(object source,
EventArgs e);
class Room {
    public event EventHandler Enter;
    public void RegisterGuest(object source,
EventArgs e) { … }
    public static void Main(string[] args) {
        Enter += new EventHandler(RegisterGuest);
        if (Enter != null) {
            Enter(this, new EventArgs());
        }
    }
}
```

Events – Usage Patterns

- `Enter` is an object of type delegate
  - when event is “raised” each delegate called
  - C# allows any delegate to be attached to an event
- Differences from regular delegates
  - delegates cannot be defined in interfaces; events can
  - can only raise an event in its defining class
  - outside, can only do += and -=
    - public/private: define accessibility of += and -=
- To raise events from outside
  - normally with methods: eg. `Button.OnClick`

Events – Accessors

- `add` and `remove` accessors
  - Like `get` and `set` for properties and indexers
  - Invoked by +=, -= operations
  - can be explicitly defined for events
  - normally generated by compiler
- For example
  - when want to control the space used for storage
  - or use to control accessibility



Generics and Delegates

- Delegates can use generic parameters:
  `delegate int Func<T>(T t)`
  - allows interaction with generic classes

```
class Test<T> {
     public Test(Func<T> f, T val) { … }
}
```

- Methods can use delegates similarly

  -
    ```
    static int F(int i) { … }
    static int G(string s) { … }
    ```

  -
    ```
    Func<int> p1 = F;
    Func<string> p2 = G;
    ```
- Can add `where` constraints