Delegates and Events
First assignment due *today*

Second assignment will be released
  ▪ due in one week
Review

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

- Delegates
- Anonymous methods
- Events
- Treat functions as “first-class” objects
- Pass functions to other functions
  - OCaml
    - ```
    map (fun x -> x*x) [1; 2; 3] ;;
    ```
  - 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
  - Is a type that references a method
  - 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
delegate int Foo(ref int x);
static int Increment(ref int x)
{
    return x++; 
}
static int Decrement(ref int x)
{
    return x--; 
}
Foo f1 = new Foo(Increment);
f1 += Decrement;
int x = 10;
Console.WriteLine(f1(ref x));
Console.WriteLine(x);
Delegates Usage Pattern

- Declared like a function
- Instantiated like a reference type
  - Takes a method parameter in constructor
- Modified with +, -, +=, -=
  - Can add multiple instances of a method
  - Removes the last instance of the method in the list
- Called like a function
  - Invoking a delegate that has not been assigned a list of methods causes an exception
delegate int Foo(int x);

List<int> Map(Foo f, List<int> list)
{
    List<int> result = new List<int>();
    foreach (int element in list)
    {
        result.Add(f(element));
    }
    return result;
}
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
  - Variables captured by anonymous method
    - Outer variables
    - e.g. y in the previous example
Local variables declared outside the scope of an anonymous method
- Captured & remain for the lifetime of the delegate
- Outer variables values are captured once per delegate
Event-based programming

- Events are raised by run-time
  - Indirectly through external actions, function calls
- Client code registers *event handles* to be invoked
  - Also called *callbacks*
  - Allows asynchronous computation
- e.g. GUI programming
Events in C#

- In C#
  - Events – special delegates
  - Event handlers – functions
- Created from delegates using `event` keyword
  - 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 Pattern

- 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

- **Example**
  - when want to control the space used for storage
  - or use to control accessibility