C# Basics and Events

236703 - Object-oriented Programming

Updated: January 2016.
using System;

public class MyClass {
    public static void Main() {
        Console.WriteLine("Hello, World!");
    }
}

C:\> csc hello.cs
... ...
C:\> hello.exe
Genealogy

• Designer: Anders Hejlsberg (Microsoft)
  – Designer of Turbo Pascal, Visual J++, Delphi (Borland)

• C Dynasty: Play on Words
  – C++ - increment C by one, C# - the musical note half tone above C

• Yet another curly bracket programming language
  – Grouping: {}
  – Terministic camp: statements terminated by ";"
  – C operators: ++ % != += & & ^, >>, ?: ...
  – C like control:
    • if () ... else ...
    • for (...) ... break ..., while (...) ... continue ..., do ... while (....)
    • switch (...) ... case ... default (no fall through, works on strings)
Design Principals

All the Good Things: Simplicity, General Purpose, Portability, Object Oriented

• Programmer Protection:
  – Strong Nominative Typing
  – Array Bounds Checking
  – Garbage Collection
  – Check against using uninitialized variables

• Evolutionary: dramatic changes in each language version
  – Learn from Java mistakes (no checked exceptions, since Anders Hejlsberg doesn't know yet how to do these right)

• Differences From Java?
  – Developed by Microsoft
  – Runs on CLR "Common Language Runtime"
  – Compiles to the CIL "Common Intermediate Language"
  – Support for "unsafe" features, including pointers.
  – Lambda expressions, extension methods...
Object Oriented Purity

- Global Variables? No
  - All variables are defined in functions/classes
- Global Routines? No
  - All routines (functions) are defined in classes
- Non OO Types? No
  - Even primitive types belong in the OO hierarchy
- OO Control Flow? No
  - If, while, for, ... are imperative statements
- Pre-processor? Yes
  - Only conditional compilation and compiler directives

1.Equals(2) is legal
Value/Reference Semantics

• Value Types
  – Simple types: char, int, float, ...
  – Enum types
    ```
    public enum Color {Red, Blue, Green}
    ```
  – Struct types
    ```
    public struct Point { public int x, y; }
    ```

• Reference Types
  – Classes, Interfaces, Delegates
  – Nullable Value Types
    ```
    char? c_null = eof() ? null : getchar();
    if (c_null == null) { ... }
    char c = c_null ?? ' '
    ```
Inheritance Hierarchy

• Classes:
  – Single Inheritance with common root: System.Object
  – Unified type system: includes all built-in types (except void)
    • System.ValueType: base class of all value types
    • System.Enum: base class of all enum types
    • System.Array: base class of all arrays
• Unextendable classes: denoted by the keyword sealed
• Static classes: denoted by the keyword static
  – No non-static members
  – Must inherit form System.Object
• Interfaces: multiple inheritance hierarchy
  – May be implemented by classes and structs
• Structs: no inheritance, but may implement interfaces
Inheritance and Binding

• **Method Binding:** static, unless method is declared virtual
  - virtual modifier cannot go with any of static, abstract, private or override modifiers.
  - Properties can be virtual

• **Overriding:** inheritance is strict by default.
  - Overriding methods must be declared as such with override keyword
  - Cannot override non-virtual functions
  - Use new modifier to indicate hiding

• **Sealing:** use sealed keyword to indicate that an overriding function cannot be overridden further
  - No point in sealing a "plain" virtual function.
No co-variance of return type

```java
class Animal { }
class Fish : Animal { }

abstract class Enclosure {
    public abstract Animal Contents();
}

class Aquarium : Enclosure {
    public override Fish Contents() {
        return new Fish();
    }
}

Compilation error: 
Aquarium.Contents(): return type must be 'Animal' to match overridden member 'Enclosure.Contents()'
```
Array Covariance

• Suppose B is a subtype of A: then an array of B is a subtype of an array of A
  – Like Java, unlike C++
• Runtime type checking: in assignments to array elements
• No Array Covariance of Value types
  – why?
• Method arguments and return type are no-variant

```java
class A{}
class B: A {}

void f(A[] a) {
    a[0] = new A(); //OK?
}

// Legal assignment
A[] x = new B[4];

// Legal call
f(x);

// Illegal assignment
object a[] = new int[5];
```
static class Utils {
  // Will dump any reference type array thanks to array co-variance
  public static void Print(string title, object[] array) {
    if (array.Length == 0) return;
    Console.WriteLine(title + ":");
    foreach (object item in array) {
      Console.WriteLine("  " + item);
    }
    Console.WriteLine();
  }
}
Properties

- Property: a field implemented with methods
- Varieties: read only, write only, read-write
- Contextual keywords: `get`, `set`, `value` (also add and del for events)
  - Provide specific meaning in the code (not otherwise reserved)

```csharp
public struct Window {
    public int n_read = 0;
    private string title;
    public string Title { // read-write property
        get { // property getter method
            n_read++;
            return title;
        }
        set { // property setter method
            if (title == value) // implicit parameter
                return;
            title = value;
            redraw();
        }
    }
}

Window w = new Window("Initial Title");
Console.WriteLine(w.Title); // increment n_read
w.Title = "My Title"; // redraw
```
public struct HarryPotterBook {
    static private int count;

    // static constructor
    static HarryPotterBook() {
        count = she_wrote_it() ? 7 : 6;
    }

    // read-only static property
    static public int Count {
        get { return count; }
    }

    ...
}
Delegates

- A type that references a method (similar to C++ function pointer)
- Supply type safety: only a method that matches* the delegate's signature (return type and parameters) can be assigned to the delegate
- Allow methods to be used as first class objects (e.g. passed as parameters)
- Multicast (multiple methods can be assigned to a single delegate)
- Delegates reference to a method, and thus associated with an instance (with the exception of static methods).
- Support anonymous methods (later in this lesson)
Using Delegates (1/2)

- There are three steps in defining and using delegates: declaration, assignment, and invocation
- Classical Example: when a button is pressed, listeners should be notified with "callback functions"
- Declaring a delegate:

  [modifiers] delegate return_type identifier([parameter_types])

  //Declare a delegate type
  public delegate void EventHandler();

  public partial class Window {
    //Declare a delegate variable
    public EventHandler OnClick;
  }
Using Delegates (2/2)

• Assigning a delegate to a variable
  – Assign a delegate using = operator
  – Add a delegate using += operator
    • has a default implementation, but the programmer can provide his own add.
  – Remove a delegate using -= operator
    • has a default implementation, but the programmer can provide his own remove.

• Invoking a Delegate

```csharp
public class Painter {
    public Painter(Window w) {
        //Assign a delegate
        w.OnClick += Paint;
    }
    public void Paint() { ... }
}
```

```csharp
public partial class Window {
    void ButtonClick() {
        if (null != OnClick)
            OnClick(); //Invoke all delegates
    }
}
```
• Consider the following code samples:

```java
public class Beeper {
    public Beeper(Window w) {
        w.OnClick = Beep;
    }
    public void Beep() { ... }
}

public class Frame {
    private Window _w;
    public Frame(Window w) { _w = w; }
    private void KeyStroke() {
        _w.OnClick();
    }
}

class Program {
    static void Main(string[] args) {
        Window w = new Window();
        new Painter(w);
        new Beeper(w);
        w.ButtonClick();
    }
}
```
The event keyword

- The event keyword can be associated with a field of type delegate
- An event can only appear on the lhs of += or -= except when used from the defining class
  - Excluding derived classes as well
  - The defining class treats it like any other delegate (assign it with =, invoke it, etc.)
  - Event are like properties, which have two actions: add, remove.
- Events may appear in interfaces!

Our Example: **royal social conduct**

If King is happy: notify all men in the kingdom
If King is ill: notify noblemen only.
• Name: every man has a name
• Title: every man has a title, based on his status and name
  – Realized by method ToString()
• Service: every man can be set to service any number of kings.
  – Method service(King k) in class Man
Delegate Notify

• Type definition: A reference to a method (to be called when a king changes his state)

```csharp
namespace Kingdom {
    delegate object Notify(King k);
    ...
}
```

• Simple (uninteresting) use:

```csharp
void foo(King k){
    Notify bar; // A delegate variable of type Notify
    bar = FunctionDescriptor;
    ...
    object o = bar(k); // Invoke FunctionDescriptor
}
```

• A more interesting use:

```csharp
class Foo {
    event Notify baz; // An event of type Notify
    public void foo(King k) {
        baz += FunctionDescriptor;
        ...
        baz(k); // Invoke FunctionDescriptor
    }
}
```
namespace Kingdom {
...
enum State { Happy, Ill };
...
class King : Man {
    public State state;
    public King(String name): base(name){state = State.Ill;}
    sealed override public string ToString() {
        return "His Majesty, King " + Name;
    }
    // events for those interested in the King's being.
    public event Notify OnKingIllness, OnKingHappiness;

    //returns last listener's return value
    public object BeIll() {
        state = State.Ill;
        return OnKingIllness!= null ? OnKingIllness(this): null;
    }
    //returns last listener's return value
    public object BeHappy() {
        state = State.Happy;
        return OnKingHappiness!= null ? OnKingHappiness(this): null;
    }
}
}
namespace Kingdom {
    abstract class Man {
        protected readonly String Name;
        public Man(String name) { Name = name; }
        abstract override public String ToString();

        // Every man is happy when a King he serves is happy.
        virtual public void Serve(King k) {
            k.OnKingHappiness += Happy;
        }

        public Man Happy(Man m) {
            Console.WriteLine(this + " is happy to hear that " +
                             m + " is happy.");
            return this;
        }
    }
}
Delegates and Events

• Delegate type definition:

```csharp
namespace Kingdom {
    delegate object Notify(King k);
    ...
}
```

• Event definition:

```csharp
namespace Kingdom {
    class King : Man {
        public event Notify OnKingIllness, OnKingHappiness;
        ...
    }
}
```

Why were `OnKingIllness`, `OnKingHappiness` declared as events?
• Actual Delegate: a method for notification

```csharp
namespace Kingdom {
    abstract class Man {
        public Man Happy(Man m) {
            Console.WriteLine(this + " is happy to hear that " + m + " is happy.");
            return this;
        }
    }
}
```

• Register a delegate: with += operator

```csharp
namespace Kingdom {
    abstract class Man {
        virtual public void Serve(King k) {
            k.OnKingHappiness += Happy;
        }
    }
}
```
namespace Kingdom {...
    sealed class Citizen : Man {
    public Citizen(string name) : base(name) { }
    override public string ToString() {
        return "Citizen " + Name;
    }
    }
    }
    class Nobleman : Man {
    public Nobleman(string name) : base(name) { }
    override public string ToString() {
        return "Nobleman " + Name;
    }
    public Nobleman Ill(Man m) {
        Console.WriteLine("this + " + m + " is ill.");
        return this;
    }
    override public void Serve(King k) {
        base.Serve(k);
        k.OnKingIllness += Ill;
    }
    }
    ...}
```csharp
static class Life {
    static void Main() {
        King R = new King("Richard");
        King G = new King("George");
        Citizen a = new Citizen("Al");
        Citizen b = new Citizen("Bob");
        Nobleman v = new Nobleman("Virgil");
        a.Serve(R);
        b.Serve(R); b.Serve(G);
        v.Serve(R);
        G.serve(R);
        R.beIll();
        R.beHappy();
        Console.WriteLine("----");
        G.beIll();
        G.beHappy();
        Console.WriteLine("----");
    }
}
```
**Scenario**

1. King Richard becomes ill  
2. King Richard becomes happy  
3. King George becomes ill  
4. King George becomes happy

*Remember, every man is happy when a king he serves is happy, but only a Noblemen is to be notified when his king is ill.*

Noblemen Virgil is sorry to hear that His Majesty, King Richard is ill.  
Citizen Al is happy to hear that His Majesty, King Richard is happy.  
Citizen Bob is happy to hear that His Majesty, King Richard is happy.  
Noblemen Virgil is happy to hear that His Majesty, King Richard is happy.  
His Majesty, King George is happy to hear that His Majesty, King Richard is happy.

---

Citizen Bob is happy to hear that His Majesty, King George is happy.

---
Delegate Variance

• Delegate definition:
  `delegate object Notify(King k);`

• Actual delegates:
  `public Man Happy(Man m) {...}`
  `public Nobleman Ill(Man m){...}`

• Variance:
  – Return Type: Co-Variant
  – Arguments Type: Contra-Variant
  – Static / Non-static: both static and non-static actuals may be used.

• Compatibility Rules: **Allow a specific listener to subscribe to a more general event distribution channel**
  – It is OK to pass an actual `King` argument to the listener `Happy(Man m)`
  – Event triggering expects an `object`, and is type safe even if a `Man` or a `Nobleman` is returned.
using System;
using System.Reflection;

delegate object MyDelegateType(Type o);

public static class DelegateTypeHierarchy {
    private static string DelFunc(Type t) { as before... }
    private static void Ancestory(object o) { as before... }
    public static void Main() {
        Ancestory(new MyDelegateType(DelFunc));
    }
}

MyDelegateType inherits from System.MulticastDelegate
System.MulticastDelegate inherits from System.Delegate
System.Delegate inherits from System.Object
System.Object inherits from null
Extension methods

• Extension methods are a special kind of static method
• Extension methods allow "adding" methods to existing types without modifying the original type

```csharp
public static class Extension {
    public static bool IsNullOrEmpty(this string s) {
        return (s == null || s.Trim().Length == 0);
    }
    public static bool IsLengthEquals(this string s, int length) {
        return (s != null && s.Length == length);
    }
}
```

```csharp
static void Main(string[] args) {
    string newTextString = null;
    if (newTextString.IsNullOrEmpty()) { /* Do Something1 */ } 
    if (newTextString.IsLengthEquals(3)) { /* Do Something2 */ }
}
```
Rules for the extension methods

• The first parameter specifies which type the method operates on
  – the parameter is preceded by `this` modifier

• Extension methods can be used only when the namespace in which the methods are defined is explicitly imported with a `using` directive.

• Extension methods cannot access private variables in the type they are extending

• Extension methods can extend a class or interface

• An extension method with the same name and signature as an interface or class method will never be called.
  – At compile time, extension methods always have lower priority than instance methods defined in the type itself
public delegate void PrintDelegate();

public PrintDelegate Create(string message) {
    MethodInfo writelnMethod = typeof(Console).GetMethod("WriteLine", new Type[] { typeof(string) });
    Module currentModule = Assembly.GetExecutingAssembly().GetModules()[0];
    DynamicMethod method = new DynamicMethod("PrintData", typeof(void),
                                                  new Type[0],
                                                  currentModule);
    ILGenerator ilGenerator = method.GetILGenerator();
    ilGenerator.Emit(OpCodes.Ldstr, message);
    ilGenerator.Emit(OpCodes.Call, writelnMethod);
    ilGenerator.Emit(OpCodes.Ret);
    PrintDelegate ret =
        method.CreateDelegate(typeof(PrintDelegate))
                          as PrintDelegate;
    return ret;
}
Anonymous Methods

- Anonymous methods represent a way to pass a code block as a delegate parameter.

```csharp
delegate void Printer(string s);
class AnonymousMethods{
    static void Main() {
        Printer p = delegate(string j) {
            System.Console.WriteLine(j);
        };
        p("The delegate using the anonymous method is called.");

        p = new Printer(AccessMethods.DoWork);
        p("The delegate using the named method is called.");
    }
    static void DoWork(string k) {
        System.Console.WriteLine(k);
    }
}
```

The delegate using the anonymous method is called. The delegate using the named method is called.
delegate void Printer2();

class AnonymousMethods{
    static void Main() {
        string j = "The delegate with local variable is called.";
        Printer2 p2 = delegate() {
            System.Console.WriteLine(j);
        };
        p2();
    }
}

The delegate with local variable is called.
Lambda Expressions

- A lambda expression is an anonymous function, that use the lambda operator `=>`.
  - The left side of the lambda operator specifies the input parameters (if any)
  - The right side holds the expression or statement block

```csharp
delegate void Printer(string s);

class LambdaExpressions{
    static void Main() {
        Printer p = j => System.Console.WriteLine(j);
        p("The delegate using the anonymous method is called.");
    }
}
```

The delegate using the anonymous method is called.
Yield Return Iterator

- "yield" keyword is used to iterate through objects returned by a method.

```csharp
class Team {
    string s1 = "Bryant";
    string s2 = "Curry";
    string s3 = "Irving";
    string s4 = "Casspi";
    public IEnumerable<string> GetPlayers() {
        yield return s1;
        yield return s2;
        yield return s3;
        yield return s4;
    }
    public IEnumerable<string> GetPlayersStartingWith(string s) {
        foreach (string player in GetPlayers())
            if (player.StartsWith(s)) yield return player;
    }
}

static void Main() {
    Team t = new Team();
    foreach (string player in t.GetPlayersStartingWith("C"))
        Console.WriteLine(player);
}
```
More about Yield

- An iterator is a method, get accessor or operator, that return `IEnumerable` or `IEnumerable<T>`. An iterator enables you to support foreach iteration in a class or struct.
- The yield statement can only appear inside an iterator block.
- A yield statement cannot appear in an anonymous method.
- `yield return <expression>`
  - `expression` is evaluated and returned as a value to the enumerator object.
  - `expression` has to be implicitly convertible to the yield type of the iterator.
  - `expression` is lazily evaluated, that is evaluated only when reached.
  - A yield return statement cannot appear inside a catch block.
  - A yield return statement cannot appear inside a try block, unless it has no catch blocks.
- `yield break`
  - The operation unconditionally returns control to the caller of the iterator.
  - A yield break statement cannot be located in a finally block.
class PokerRound {
    public int bet = 100;

    public IEnumerable<string> GetActions() {
        yield return "raise";
        if (bet<=100)
            yield return "check";
        else {
            yield return "fold";
            yield break;
        }
        yield return "FLIP TABLE (ಠ_ಠ)";
    }
}

static void Main() {
    var round = new PokerRound();
    foreach (string action in round.GetActions()) {
        Console.WriteLine(action);
        if (action == "raise")
            round.bet+=10;
    }
}
Summary – C# Unique Features

• Properties: implement fields with functions
  – Conforms to Eiffel Principle of Uniform Reference.

• Delegates: type safe function pointers
  – Events: list of delegates.

• Static classes

• Seamless integration of value semantics
  – Including nullable values.

• Default Strict Inheritance
  – override and new keywords