C# in 75 Minutes

Perry Kivolowitz

New console application

static void Main(string[] args)
{
    Console.WriteLine("Hello World.");
}

Intellisense

Intellisense

Launching / Start Page
Hello World

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Breakpoints / single stepping

- F9 toggles breakpoints
- F5 start debugging
- F10 step over
- F11 step into
- Shift F5 kills debugging

Watching / rerunning

Breakpoint indicator indicates next line to be executed
Hovering over variable shows its value

Types

- object **ALL types derive from object**
  - int
  - string `s = "hello world"`;
  - char `c = 'A';`
  - float `f = 3.0f;`
  - bool `b = true;`
  - enums eg: `DayOfWeek.Friday`
  - user defined classes and structs
  - many more

Object: base class of all types

- Even value types like int and float derive from Object:
  - `int x = 1;`
  - `x.ToString() → "1"`
  - `1.ToString() → "1"`
- Object implements other methods like:
  - `object.Equals(other)`
  - `object.GetType()`
  - Not used in this course
Strong typing

- **Pro**
  - Catch / prevent errors at compile time
- **Con**
  - Verbosity

Type conversion

- **Implicit**
  - Obvious relationship exists
  - No loss of information
- **Explicit**
  - Like a cast in C or C++
- **Type conversion**
  - `System.Convert.To_________()`
  - Use this extensively for ADO.NET work.

Strings

- Can be indexed: `char c = s[2];`
- Usual escape sequences
  - `\" \n \"` etc.
- Precede with `@` to make a literal string
  - `@"C:\temp\foo"` is the same as
  - `"C:\temp\foo"`
- Well defined logical operators like `=, >, etc.`
- See also StringBuilder class

Equivalent of (s)printf

- `string System.Format()`
- Uses positional notation:
  - `System.Format("Hi {0} {1}.", fName, lName);`
- Formatting for specific types available
  - Left for the reader
Arrays, ArrayLists, Generics

- **Arrays**: strongly typed, fixed length
  - int[] i = {1, 2, 4};
  - int[] i = int[3];
- **ArrayList**: loosely typed, variable length
  - Recommend use of Generics instead
- **Generics**: strongly typed, variable length
  - Like C++ templates

Generics

- **Strongly typed collections**
- **Variable length**
  - Includes List<>, LinkedList<>, Queue<>, etc.
- **Enable by**
  - "using System.Collections.Generic;"

Generics

```csharp
public void sum()
{
    List<string> flavors = new List<string>();
    flavors.Add("Vanilla");
    flavors.Add("Chocolate");
    if (flavors.Count > 0) {
        if (flavors.Contains("Chocolate")) {
            flavors.Insert(0, "Strawberry");
            flavors.Add(1);
        }
    }
}
```

Classes

- Analogs in the real world
- At the heart of OOP
- An encapsulation of related functions and data
- A class is a blueprint for all things of that type
- Instance of a class is a thing, an *object*

Classes

```csharp
public class AClass
{
}

AClass aClass = new AClass();
```

Classes

- Have members:
  - Methods – functions in other languages
  - Data
    - Variables – member data as in other languages
    - Attributes – functions that behave like data
- Members have protection levels
  - Public – visible outside class
  - Private – hidden outside class
  - Protected – visible inside “derived” classes only

Classes
Constructors

• Constructors are special methods
  – same name as class
  – no return type
  – used for initialization of a class instance
  – may be “overloaded”

```java
public class AClass
{
    private int count = 0;
    AClass(int newCount)
    {
        count = newCount;
    }
}
```

Uses Default Constructor

• Members get their default or otherwise initialized values

Methods

• Must exist in a surrounding class or struct
• “Global” methods can be fudged
• Return values:
  – void for no return value
  ```java
  public void foo() {}
  ```
  – Specify type to return a specific type
  ```java
  public int foo(){ }
  ```

Parameters

• All parameters passed by value by default
• To pass by reference use ref keyword (demo)
• To return more than one result, use out keyword (demo)
• Keywords must be present in both method definition and invocation

Member variables

• Typically are not “public”
• Public variables
  – break “data encapsulation”
  – cause loss of “control” of the class
  – easier for the lazy or hurried
• Use attributes for public faces to internal variables
Member attributes

• “Functions” that behave like variables
• Use to provide controlled access to variables
• Can be used to make read-only variables
• Implemented via get and set syntax

Static versus instance

• All individuals of a class (instances) share certain traits – but have individual copies
  – Rexx and Fido are Dogs but have different names
  – Name is a trait shared by all instances of the class Dog but each instance of Dog has its own copy
  – This is the “default”
• To have all instances share the same member, make it “static”

Member attributes

```java
class Program
{
    private int accessCounter = 0;
    private int updateCounter = 0;
    private int thing
    {
        get
        {
            accessCounter++;
            return thing;
        }
        set
        {
            updateCounter++;
            thing = value;
        }
    }
}
```

Static versus instance

```java
public class Rexx
{
    protected string name = "";
    public Rexx(string name)
    {
        this.name = name;
    }
}
```

```java
public class Dog : Rexx
{
    public Dog(string name) : base(name)
    {
    }
}
```

Static versus instance

```java
public class Rexx;
{
    protected string name = "";
    public Rexx(string name)
    {
        this.name = name;
    }
}
```

```java
public class Dog : Rexx
{
    public Dog(string name) : base(name)
    {
    }
}
```

Static versus instance

• A trait that is present in all instances of a class and physically shared by all instances is called a static trait
  – Can be methods or variables
  – Must be fully named using enclosing class

Other qualifiers

• const
  – Compile time constant
• readonly
  – May be initialized at compile time or in a constructor
• Neither can be changed after its value has been initialized
Control structures

• Same as other languages such as C, C++
  – if, then, else
  – while
  – do
  – conditional operator
  – for
  – switch
  – continue, break
• foreach not found in C or C++

Operators

• All the usual operators are provided
• The usual order of precedence holds

Enums

• Strongly typed enumerations
  – Not interchangeable with integers as in C, C++
  – Intellisense makes good use of them
• Makes code more readable and maintainable

Enums

private enum DeleteAfterCopy
    { 
    Yes,
    No
    }

private void Copy(string from, string to, bool deleteAfterCopy)
{ 
    private void Copy(string from, string to, DeleteAfterCopy deleteAfterCopy)
    { 
    private void func()
    { Copy("a", "b", true);
    Copy("c", "d", DeleteAfterCopy.Test); 
    }

Back to classes

• You have seen “this”
  – Reference members of the current instance
  – Cannot be used to reference static members
  • Use class name instead
  – Typically used for disambiguating a member variable from a method parameter of the same name

Classes

public class Manual
{ 
    public string name = "";
    static public readonly string status = "OK";

    public Manual(string name)
    { this.name = name;
    }
}
Overloading methods

- Used to provide alternate method signatures with the same name
  - Which method is called depends upon parameters
  - Assisted by Intellisense

Operator overloading

- Operators are implemented as methods
- Since methods can be overloaded, it follows that operators can be overloaded
- Particularly useful for user defined types such as classes
- (demo)

Inheritance

- Creates specializations of a class
- The “is a” relationship (versus “has a”)
  - Dog “is a” mammal
  - Cat “is a” mammal
  - Dog “has a” name
- C# supports single inheritance
- Multiple “interfaces” can be inherited
  - Interfaces not covered in this course

Inheritance

```csharp
public class Manual
{
    public string name = "";
    static public readonly string status = "OK";
    public Manual(string name)
    {
        this.name = name;
    }
}

public class Dog : Manual
{
    public Dog(string name) : base(name)
    {
    }
}
```
### Virtual methods
- Mark method you wish to be able to override with the keyword “virtual”
- Mark overriding methods with keyword “override”
- If you want to specifically hide a super classes method or variable, mark the subclass’ attribute with keyword “new”
- Use “base.method()” syntax to call the named method in the base class
- Demo

### Polymorphism
- A subclass can be used anywhere a super class is expected because the subclass has everything the super class (plus some other stuff)
- If Bichon is a Dog and Dog is a Mammal, then Bichon is a Mammal
- If a subclass specializes a method present in the super class, which method is called when? (demo)

### Abstract Methods / Classes
- Makes a contract for the API but does not provide implementation
- Use keyword “abstract” to mark a method you will force a subclass to implement
- If any method in class is “abstract” then class must be “abstract”
- Abstract classes cannot be instantiated
- Demo

### Strings
- A vital type in web application development
- All strings are Unicode (multibyte)
- Can be compared with logical operators
- Have many methods for handling
- Use StringBuilder if a lot of concatenation is to be performed
- (demo)

### Exceptions
- Greatly simplifies code by allowing an assumption of success
- Provides uniform structure to error handling
- All exceptions derive from System.Exception
- Implemented with
  - try, catch, throw, and finally.
Exceptions

- throw causes an exception to be raised with the specified type and contents
- If throw called with no type, then previously thrown exception is re-thrown

Exceptions

- All exceptions derive from System.Exception
- Unhandled exceptions percolate upward until they are caught or the program dies
- (demo)