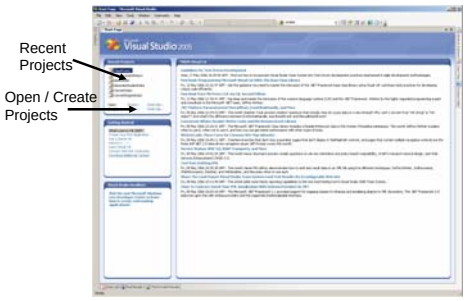
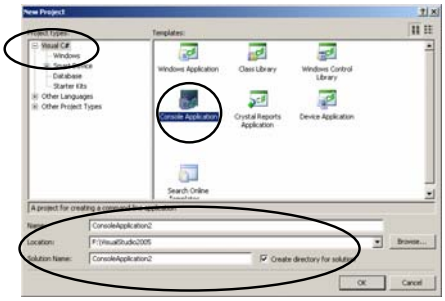
C# in 75 Minutes

Perry Kivolowitz

Launching / Start Page



New console application



Intellisense

```
static void Main(string[] args)
{
    Con
}
```

class
CLSCompliantAttribute
Comparer<>
Comparison<>
Console
ConsoleCancelEventArgs
ConsoleCancelEventHandler
ConsoleColor
ConsoleKey
ConsoleKeyInfo

class System.Console
Represents the standard input, output, and

Intellisense

```
static void Main(string[] args)
{
    Console.Wri
}
```

SetWindowPosition
SetWindowSize
Title
TreatControlCAsInput
WindowHeight
WindowLeft
WindowTop
WindowWidth
Write
WriteLine

void Console.WriteLine(string format, params
Writes the text representation of the specifie

Exceptions:
System.FormatException
System.IO.IOException
System.ArgumentNullException

Intellisense

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

11 of 19 void Console.WriteLine (**string value**)
value: The value to write.

Hello World

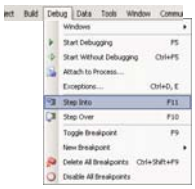
```
using System;
using System.Collections.Generic;
using System.Text;

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

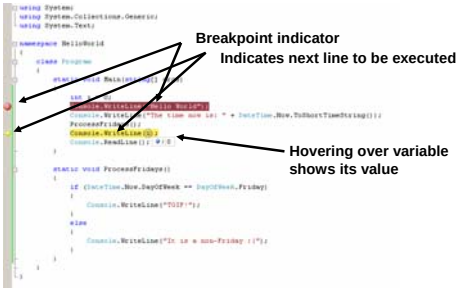
Hello World

Breakpoints / single stepping

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



Watching / rerunning



Types

- object — ***ALL*** *types derive from object*
  - int — *int i = 0;*
  - string — *string s = "hello world";*
  - char — *char c = 'A';*
  - float — *float f = 3.0f;*
  - bool — *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

Strong typing

```
class Class1
{
    public void foo()
    {
        int i = 0;
        float f = i;                        // implicit - works
        float g = ((float) i) / 1.0f;       // explicit - works
        string s = i;                       // implicit - fails
        string t = (string)i;               // explicit - fails
        string u = i.ToString();            // all types have ToString()
        string v = System.Convert.ToString(i);  // generalized type conversion
        char c = "c";                       // fails
        char d = 'c';                       // works
        float h = 3.0;                      // fails
        float j = 3.0f;                     // works
    }
```

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

- string s = “some string”;
- string s += “ and some other string”;
- s.Length – read only attribute
- s.Trim() – returns string without leading / trailing white space
- Many other members to the string class
  - Split(), SubString(), etc.

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

```
public void fum()
{
    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

```
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

Constructors

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

Constructors

```
public class AClass
{
}
```

Uses Default Constructor

- Members get their default or otherwise initialized values

Constructors

```
public class AClass
{
    private int count = 0;

    AClass(int newCount)
    {
        count = newCount;
    }
}
```

Methods

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

Member attributes

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

        set
        {
            updateCounter++;
            thing = value;
        }
    }
```

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”

Static versus instance

```
public class Mammal
{
    protected string name = "";

    public Mammal(string name)
    {
        this.name = name;
    }
}

public class Dog : Mammal
{
    public Dog(string name) : base(name)
    {
    }
}

public class Foo
{
    public Dog Fido = new Dog("Fido");
    public Dog Rexx = new Dog("Rexx");
}
```

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 foo()
{
    Copy("a", "b", true);
    Copy("a", "b", DeleteAfterCopy.Yes);
}
```

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 Mammal
{
    public string name = "";
    static public readonly string status = "OK";

    public Mammal(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

Overloading methods

```
class Program
{
    static void Main(string[] args)
    {
        LookUpPerson(
    }    1 of 2  void Program.LookUpPerson(string socialSecurityNumber)

    static private void LookUpPerson(string firstName, string lastName)
    {
        Console.WriteLine("Looking up by first and last name.");
    }

    static private void LookUpPerson(string socialSecurityNumber)
    {
        Console.WriteLine("Looking up by social security number.");
    }
}
```

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

- Some “classes” in the real world are specializations of other “classes”
- All dogs are mammals
- All cats are mammals too
- Dogs and cats share certain traits, these could be implemented in the mammal class
- Dogs and cats derive from mammals
- They “inherit” common traits and “specialize” from there

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

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

    public Mammal(string name)
    {
        this.name = name;
    }
}

public class Dog : Mammal
{
    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

- try { } encloses section of code where a particular exception could occur
- catch (type ex) { } catches an exception of the type specified and handles it
- Catch most specific to most general
- finally { } encloses code you want run no matter what happens (exception or not). Good for clean up

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)