C#

A Programmer's Introduction

Class Logistics

Prerequisites

Target Audience

Class scope

Practical / hands-on

Prerequisites

Minimum 6 months or 1 semester programming experience in any language

- No previous OO experience is assumed but it is helpful

Working knowledge of computer networking and Internet fundamentals

No qualms about using the command line

Target Audience

This class is both lecture and hands-on!

Developers who ...

- learn C#
- get start with Microsoft .NET
- don't mind getting their hands dirty

Class Scope

C# fundamentals...

- How to do the things you already know how to do in C#

Object Oriented programming in C#

- C# combines the features of C++ with the memory managed environment of JAVA

Class Scope

Unique features of C#

- Language support in C# for things that many other "mainstream" languages lack

A few advanced C# topics

- C# makes some things that were previously reserved for wizards much more accessible to mere mortals

Practical / Hands-on

The only way to learn how to code is to do it

Your responsibilities as a student:

- Try all of the sample code… this means actually compiling and running it even if you know what will happen
- Complete all hands-on assignments

Summary

You must have programming experience

You should not be afraid to use command lines and get your hands dirty

Your performance will be evaluated by your ability to write solid code

Virtual Machine Execution Environment

C# and JAVA target VMs (CLI and JVM)

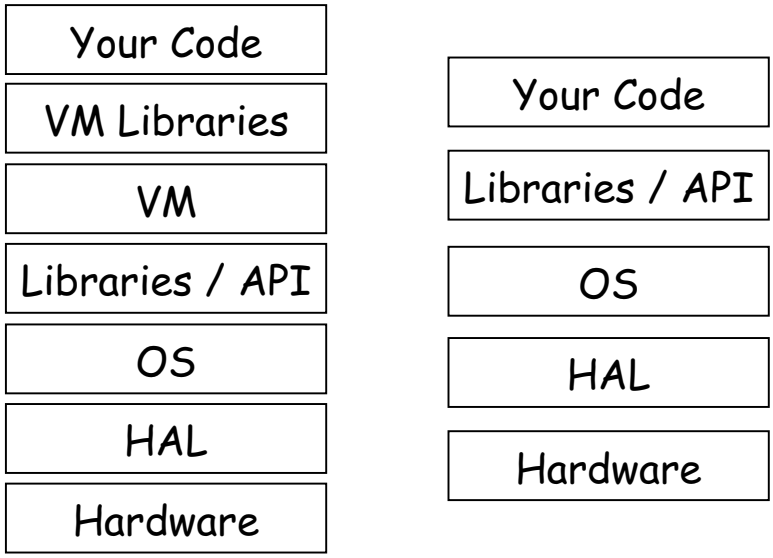
VM is a simulated computer, complete with instruction set and assembly language

Source code compiles down to VM assembly

Assembly executes in simulation

Simulator executes on top of real hardware

VM EE vs. Native Env.



VMs - The Good

Less likely to bring down the system

- You can crash your VM (potentially) but in theory this is just a process on your OS

Provable (mathematically) correctness

- Not completely true... but still better than others because of limited assembly

VMs - The Bad

Performance penalty

- Putting a "layer" between your app and the OS (and ultimately the hardware)

Difficult to access hardware

- Unless a specific package is provided to do so (e.g. DirectX)

VMs - The Ugly

Unmanageable distribution

- The VM execution environment is HUGE (64 kb app becomes 200 MB nightmare)
- Differences between various EEs can (and often do) become show-stopping issues
- Sun has admitted that it CANNOT use JAVA internally because of this problem

Why use VM EEs?

According to the marketing departments...

- JAVA to support uniform development and deployment across many platforms
- Microsoft .NET / C# to support distributed applications developed in a multi-language environment

In Reality

JAVA

- Client-side JAVA failed
- JAVA still fractionated (J2ME, J2SE…)

Microsoft .NET

- Who develops apps in 2 languages?
- C# has more complete access to CLR

But, but, but, why?

C# and JAVA are popular because they do a better job preventing developers from slicing off their own fingers

- Memory managed environment
- "No possibility" of buffer overflow
- No manual pointer arithmetic

Summary

Virtual machine environments...

- Slow things down
- Keep you sane

C# (CLR/CLI) logistics

To develop in a VM EE you need...

- Tools (compiler, etc.)
- Libraries
- Execution engine (simulator)

Typically put together in a "SDK" package

Windows 2000 / XP

The "favored child", .NET CLR

Download and install Microsoft .NET SDK

- http://msdn.microsoft.com/net
- make sure you get SDK, not the runtime

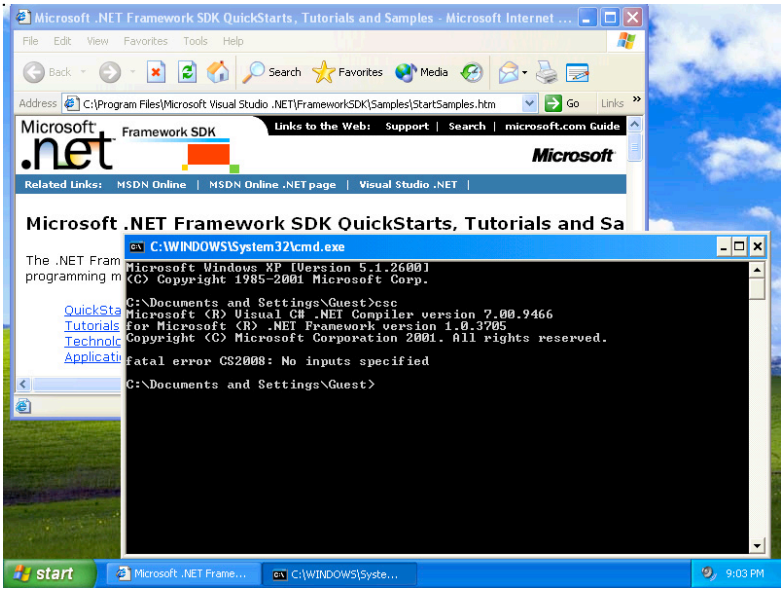
Have $ to burn? Buy Visual Studio .NET

Windows 2000 / XP

Links to HTML documentation are added to the start menu

Access tools from command line

- Start :: Run :: cmd [ENTER]
- csc is the name of the compiler
- Type executable name to run program



FreeBSD, MacOS X

Use the "Rotor" shared-source CLI

http://msdn.microsoft.com/net/sscli

Rotor download is in source form

Need to have development tools (C++)

Missing some stuff found in CLR

VB compiler, System.Data (SQL access)...

Documentation is a SEPARATE download

Building Rotor

Download and extract the tarball

Run the env.sh (or env.csh) script to setup your environment

Run the buildall script to build the SSCLI

~2MLOC! Takes 35 min on 1 GHz PPC G4

Details have been posted online

Using Rotor

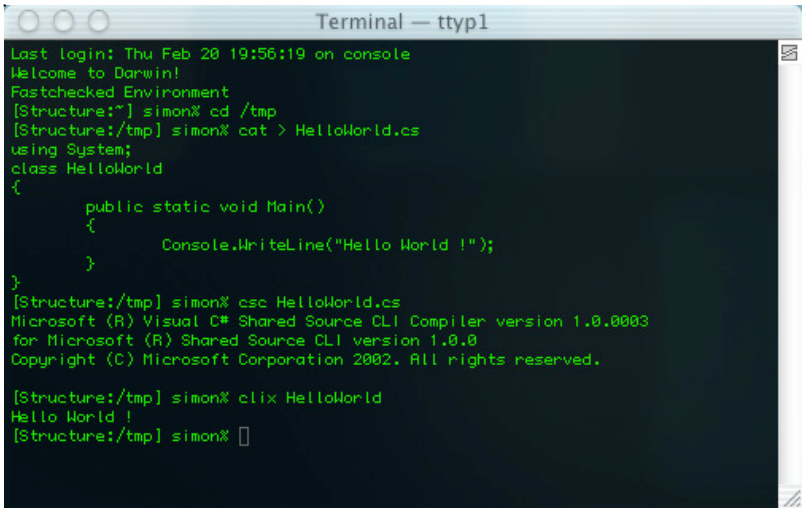
Setup environment by running the env script

You may want to incorporate into login

Use "csc" to compile C# source files

Use "clix" to execute compiled assemblies

Rotor on MacOS X



Windows 95, 98, ME

You're screwed

- Upgrade your OS to 2000 or XP
- Get VMware or Bochs to run 2000, XP or FreeBSD under emulation
- Get a friend to give you remote access to a machine that can run either CLR or SSCLI

Linux, other UNIXes

Mono Project - http://www.go-mono.com

- Binaries available in RPM forma
- use mcs to compile
- use mono to exec with JIT, mint without

Summary

C# is available in many forms

- The MS CLR for Windows XP, 2K
- The SSCLI for FreeBSD & MacOS X
- The Mono project for Linux, Solaris, etc.

If you are still running 95 or 98, get with the program

C# Program Structure

Primary container - class

Point of entry - Main

External references - using

Packaging - namespace

Class Definition

Most C# code is placed inside a class

No concept of global variables / methods

```
class HelloWorld
{
   ....
}
```

Point of Entry

Main is where a C# program starts

Must be declared with static keyword

```
class HelloWorld
{
  static void Main() {
     ...
  }
}
```


HelloWorld in C#

```
class HelloWorld
{
  public static void Main()
  {
    System.Console.WriteLine("Hello World!");
  }
}
```

Using Namespaces

The using keyword imports namespaces

- Do not have to specify the fully qualified path of an imported namespace
- Lots of built-in functionality in various CLI namespaces

HelloWorld Redux

```
using System;

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

Custom NameSpaces

```
using System;

namespace MyPersonalNameSpace {
  class HelloWorld {
     public static void Main() {
        Console.WriteLine("Hello World !");
     }
  }
}
```


Multiple Classes

Having many classes does not imply having many files

Numerous points of entry (Mains) can be defined

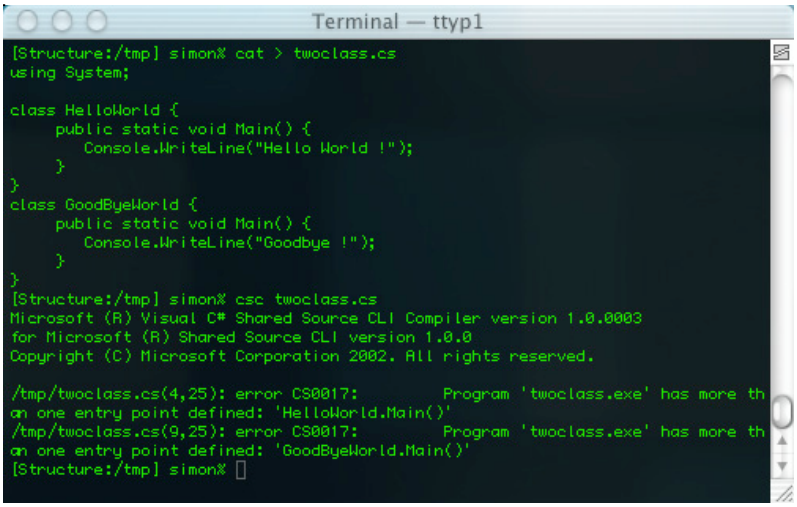
At compile time, the Main for set of all classes being compiled into the same file (assembly) must be chosen

Example: 2 Classes

```
using System;

class HelloWorld {
     public static void Main() {
        Console.WriteLine("Hello World !");
     }
}
class GoodByeWorld {
     public static void Main() {
        Console.WriteLine("Goodbye !");
     }
}
```


Two entry points!



CSC Options

/main:<classname>

Specifies which entry point to use

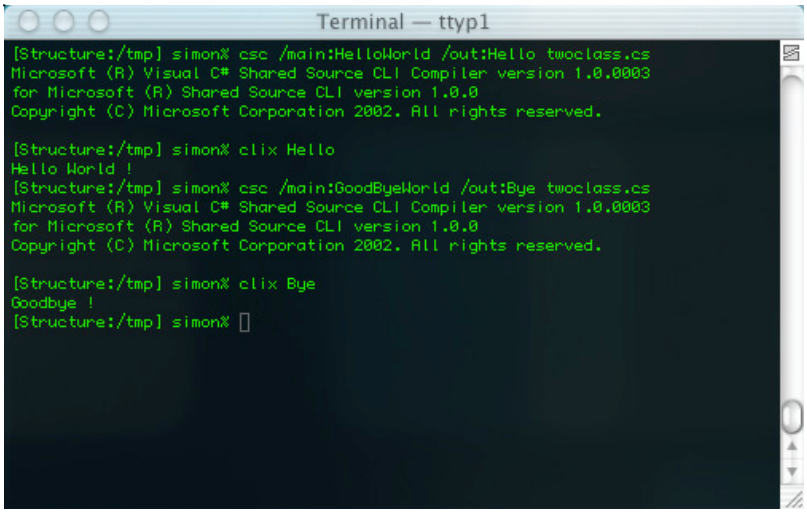
/out:<filename>

Specifies output filename

/target:[exe|library|module|winexe]

Specifies output file type

Entry Points Resolved



More CSC Options

/r:<filename>

Link in (reference) some other assembly

/lib:<directory>, [<dir2>]

Specify location of files being referenced

Compiling and JIT

csc: static compilation of high level language (C#) into intermediate language (IL)

True simulation of IL would be too slow

Compile IL into native assembly "just-in-time" before executing

Dynamic JIT

Compiling lots of IL just before executing would cause huge startup delays

Do not compile IL until it is needed

- Typically on a function by function basis

Cache native code so subsequent calls will be very fast

NGEN - Precompilation

ngen myassembly.exe

Creates native image from an assembly (result of C# compilation)

Installs native image into the native cache on local machine

Decreases startup time

Want to see it?

Use the `ildasm` tool

ildasm myfile.exe

- Extracts the IL (virtual machine assembly) in an assembly
- GUI under windows
- Just dumps to console under SSCLI

Summary

Classes are the fundamental atom

Namespaces help keep you organized

To compile, use the `csc` tool

To disassemble, use the `ildasm` tool

Assemblies

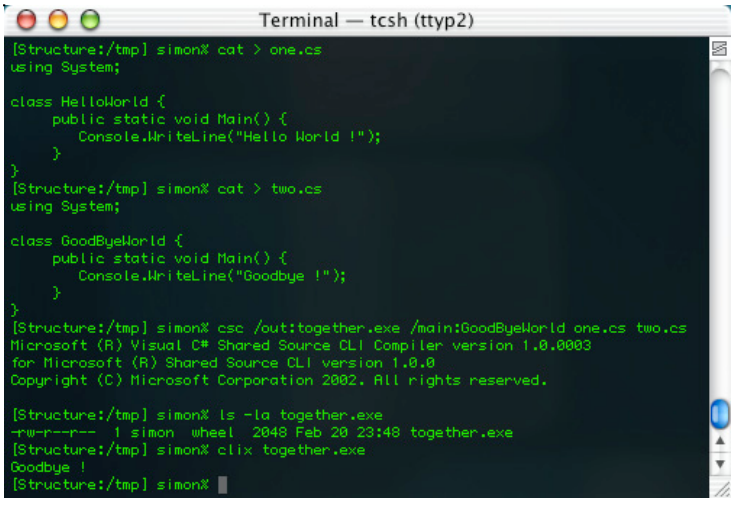
A logical grouping of types (classes)

Includes special meta-data, manifest and possibly a digital signature

Only a single point of entry is allowed

csc /r:<assem file> to resolve references between assemblies during compilation

2 classes saved in 2 source files compiled into 1 assembly



More on Assemblies

Encapsulate notion of DLLs or SOs, plus…

- Cross language integration
- Multi-module capability (span many files)
- "Robust" 4 part naming system (replaces the GUID insanity of COM)
- Code signing and key management tools are built into the environment

Cross-language Assemblies

Create an assembly in VB or JScript

- Call classes from C# or vice-versa

Allows people familiar or tied to different languages to build one application

Potentially allows exploitation of the power of particular languages

VB Example

```
Class Greeting
   Shared Function SayHello as String
      Return “Hello World!”
   End Function
End Class
```

```
class Hello {
  static void Main() {
    Console.WriteLine(Greeting.SayHello());
  }
}
```


Compiling

Compile the VB class using VB compiler

vbc myVBfile.vbs

Compile C# class with a reference to the result of the VB compilation

csc /r:myVBfile.exe myCSfile.cs

Language Variability

Languages differ in small but important ways

- C# has signed and unsigned integers
- VB only has signed integers

Interoperable code should stick with things that are supported by all languages

- Sometimes called CLS compliant code

Multi-Module Assemblies

Initial CSC execution:

```
csc /t:module first.cs
```

creates file called `first.netmodule`

Final generation of assembly:

```
csc /addmodule:first.netmodule
/t:library second.cs
```

Assembly Naming

Robust naming is an essential part of an shared object / dynamic library system

- Name - typically filename w/o extension
- Version - major.minor.build.revision
- CultureInfo - Specifies localization
- PublicKey - Hash of key to ID publisher

Specifying Assembly Names

Special attributes are used to assign names

```
using System.Reflection;

[assembly: AssemblyVersion("1.2.3.4") ]
[assembly: AssemblyCulture("en-US") ]
[assembly: AssemblyKeyFile("acmecorp.snk") ]

public class BlahBlahBlah { ....
```

Default Naming of Version

0.0.0.0 if nothing specified

1.0.0.0 if 1 is specified

1.2.d.s if 1.2 is specified

- d is # of days since Feb 2000
- s # of seconds since midnight

Digitally Signed Assemblies

Use sn.exe to create and manage keypairs

Use the AssemblyKeyFile attribute to associate a key to do the signature

AssemblyDelaySign can be used to specify a key but not sign until later

- Useful if developer's aren't allowed to sign code "for the company"

Public (Asymmetric) Key Cryptography

Each individual developer creates a key pair

- One key is held secret and kept in a private place only accessible by developer
  - Many systems use symmetric crypto (passphrase) to protect key as well
- Second key is published in a public forum

Digital Signatures

Keys are "symmetric"

- Encrypt with one, decrypt with the other
- Encrypting with secret key means anybody can decrypt it
  - Makes the payload tamper-proof
- In practice, only "sign" a "hash" of the original document (assembly file)

Regular Example

```
sn -k keyring.snk    (first time only)
csc /t:library myfile.cs
    [assembly: AssemblyKeyFile("keyring.snk") ]
    [assembly: AssemblyDelaySign(false)
```


Delay Sign Example

```
sn -k keyring.snk    (first time only)
sn -p keyring.snk pubkey.snk
csc /t:library myfile.cs
    [assembly: AssemblyKeyFile("pubkey.snk") ]
    [assembly: AssemblyDelaySign(true)
sn -R mylib.dll keyring.snk
```

Loading an Assembly

Assembly.LoadFrom(url)

- Load an assembly from a URL

Assembly.Load(name)

- Load an assembly using resolved and four part file name
- Policies can be applied to change the final decision of the resolver

Example

```
public class Test {
   public void FirstExample() {
      string url = "file://c:/spam/blah.dll";
      Assembly a = Assembly.LoadFrom(url);
   }
   public void SecondExample() {
      string name = "blah, Version=2.0.4.0,
                     Culture=neutral,
           PublicKeyToken=28651165f17c6189";
      Assembly a = Assembly.LoadFrom(url);
   }
}
```

Assembly Policies

Requested version can be mapped to a newer or older version

Resolver can be configured on application and system wide basis

Policies are applied in order

- Application : Publisher : Machine
- In other words, machine can override any of the others

Policy Configuration

XML files define the policy

- Application: myfile.exe.config
- Publisher: might be anywhere...
- Machine: machine.config (found somewhere in the runtime distribution)

File Format

```
<?xml version="1.0" />

<runtime>
 <asm:assemblyBinding>
 <asm:dependentAssembly>
  <asm:assemblyIdentity name="Test.Balh"
        publicKeyToken="28651165f17c6189" />
  <asm:bindingRedirect oldVersion="1.2.3.4"
                       newVersion="1.3.0.0" />
 </asm:dependentAssembly>
 </asm:assemblyBinding>
</runtime>
```


Global Assembly Cache

Assemblies available from anywhere

Download cache for over-web downloads

Only contains signed assemblies & pub keys

In theory, it requires admin privileges to edit/delete entries

Resolver always consults GAC first

DLL Hell has become GAC Purgatory

CLI treats same-named types as distinct if them come from different assemblies

- Types from v2 cannot be passed where types defined in v1 are expected

One copy of global/static vars per version

- Use a non-versioned assembly for globals

Summary

Assemblies...

- Logical grouping of types
- Language independent
- Can be strongly named (digitally signed)
- DLL Hell has become GAC Purgatory

C#

Language Fundamentals

Comments

3 kinds of comments

C style /* */

C++ style //

JAVA style (javadoc)
but the token is ///

Docs are in XML!

```
/// <summary>
/// Test app </summary>
public class Test {
  public static void Main() {
     // single line comment
     int i = 0;
     int j = 2;
     /* this is a multiple
        line comment */
     do.something();
     do.something.else(i,j);
  }
}
```

C# Types

Value types

- Stack allocated
  - Fast access, small in size
- Primitive (e.g., int) and aggregate (struct)

Reference types

- Managed heap memory access

Basic Types

Fixed point

- Not just for integers

Floating point

Pointers (references) to aggregate types

- No pointer arithmetic (at least, for now...)

Fixed point value types

sbyte	signed	no suffix
byte	8 bits	no suffix
short	16 bits	no suffix
ushort	unsigned	no suffix
int	32 bits	no suffix
uint	unsigned	U
long	64 bits	L
ulong	unsigned	u or l

Floating point value types

float	32 bits	f or F
double	64 bits	no suffix
decimal	128 bits	m or M

Other types

bool	1 bit	true false
char	16 bits	'A' '\x0041'
string	variable	reference type!

Assignment

```
int i = 0;
long l = 1283758L;

float f = 3.14159265F;
double d = 3.14156265;

bool b = true;
string s = "Hello World";
```

Pointer to Types

arrays - block allocated homogenous types

value type - struct

- Stack allocated - very high performance
- needs to remain small in size

reference type - class

- Storage for payload is heap allocated
- Pointer to heap is allocated on the stack

Arrays

Efficient storage / access for groups of homogeneous types

Default values are 0 for numeric types and false for bools

Use Length property to get size

```
int[] a = new int[5];
a[0] = 15;
a[4] = 16;

int[] b = {0, 2, -1};

int[] c = new int[3]
                {0, 2, -1};

int len = c.Length;
```

n-dimensional Arrays

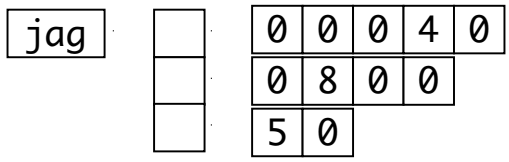
```
int[,] a = new int[5,2];
a[1,1] = 25;

int[,,] b = new int[3,3,3];
b[2,2,2] = -128;

int[,] i = { {1,0,0}, {0,1,0}, {0,0,1} };
int len1 = i.Length;
int len2 = i[1].Length;
int len3 = i.getLength(0);
int len4 = i.getLength(1);
```


Jagged Arrays

```
int[][] jag = new int[3][];
jag[0] = new int[5];
jag[1] = new int[4];
jag[2] = new int[2];
jag[0][3] = 4;
jag[1][1] = 8;
jag[2][0] = 5;
```



Ordering Arrays

```
myarray.Reverse();
```

Reverses order of elements in myarray

```
System.Array.Sort(myarray);
```

Sorts myarray into ascending order

Searching Arrays

```
System.Array.IndexOf(arr, 5);
System.Array.LastIndexOf(arr, 5);
System.Array.BinarySearch(arr, 5);
```

Returns index of '5' in arr

Negative # returned if '5' not in arr

Numeric Conversion

Automatic type conversion

- from smaller to larger type
- e.g., int -> long

Cast is required when something may be lost

```
double d = 3.14159265;
float f = (float) d;
```


Structs

Aggregation and storage of heterogeneous types

Default values are 0 for numeric types and false for bools

Can incorporate methods (functions)

```
// definition
struct Tuple {
   public int x;
   public int y;
}

// creation
Tuple ms;

//usage
ms.x = 10;
ms.y = 20;
```

Struct Methods

C-like syntax for specifying method definition

return_type method (param_type name, ...)

Use void for methods that do not return anything

```
// definition
struct Tuple {
   public int x;
   public int y;
   public int sum() {
      return x + y;
   }
}
// usage
Tuple t;
t.x = 5;
t.y = 12;
int s = t.sum();
```


Struct Constructors

All structs have a default constructor (initializes everything to zero or false)

You can make your own constructor as well for custom initialization

```
// definition
struct Tuple {
  public int x;
  public int y;
  public Tuple(int i) {
     x = y = i;
  }
}

// usage
Tuple t = new Tuple(5);
Console.WriteLine(t.x);
```

Boxing and Unboxing

- Unified type system
  - All types (even value types!) are objects
- Wrap value types up in "boxes"
  - boxing is automatic (implicit)
  - unboxing requires a cast to resolve type ambiguity

Example

```
using System;

public class Test {
  public static void Main() {
     int j = 42;
     object o = j;      // boxing
     int k = (int) o;   // unboxing
     Console.WriteLine("{0} : {1}", j, k);
  }
}
```

Details

Values are copied into wrapper objects

Changes to original value will not affect the boxed copy!

Useful in preserving value type that would normally be blown away with stack

Incurs some runtime overhead due to heap allocation and reclamation

Bizarre code... but it works!

```
using System;

public class Test {
  public static void Main() {
     int j = 25;
     Console.WriteLine(j.ToString());
     Console.WriteLine(3.ToString());
  }
}
```

Summary

C# has all the usual simple types

Arrays are dealt with in the usual way

- Jagged and n-D arrays are possible

Programmer has a choice of aggregate types

- Classes are heap allocated
- Structs are stack allocated

C# Operators

Operators are the atom of processing

- Mathematical
- Logical
- Type conversion

Mathematical overflow can be explicitly controlled in C#

Basic Operators

Infix	+ - * / %
Pre/Postfix	++ --
Binary	<< >> & | ~ ^

Operators in Action

```
int i = 5;
int j = 12;

int p = i + j;

int q = 12;
q *= p;

int r = i ^ j;
```

	8	4	2	1
5	0	1	0	1
6	0	1	1	0
&	0	1	0	0
|	0	1	1	1
^	0	0	1	1

Bit Shifting Rules

Left shift

high order dropped, low order zero pad

Right shift

uint/ulong, drop low, high zero pad

int/long, drop low, high zero pad if non-negative, one pad if negative

Pre vs. Postfix

```
public class PrePostfixTest {
  public static void Main() {
    int i = 5;
    System.Console.WriteLine(i);
    System.Console.WriteLine(++i);
    System.Console.WriteLine(i);
    System.Console.WriteLine(i++);
    System.Console.WriteLine(i);
  }
}
```

Other Operators

Relational

```
==   !=   <   <=   >   >=
```

Logical

```
&   |   ^   &&   ||
```

Conditional

```
? :
```


Type Operators

`typeof`

Returns type of the object

`is`

See if x "is of type" y

`as`

Infix type conversion (treat x "as type" y)

Overflow

Use checked/unchecked to control reporting

Constant expressions cause compiler errors

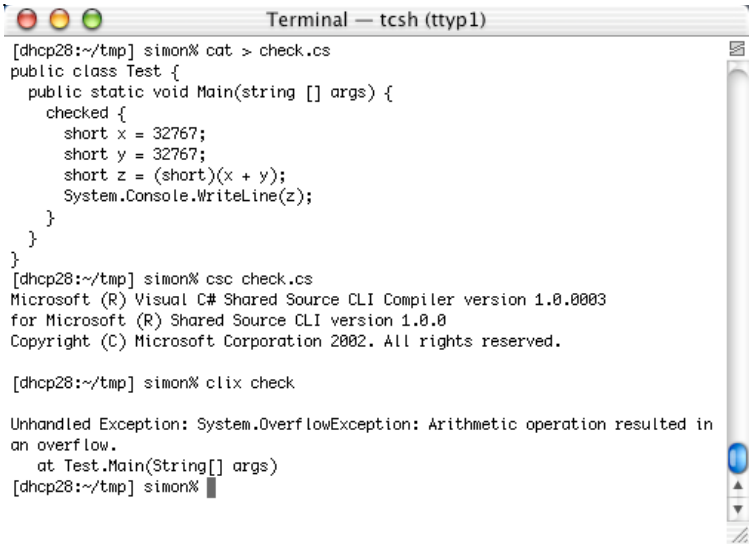
Run-time overflow throws exceptions

Compiler options used to set defaults

Checked Example

```
public class Test {
  public static void Main(string [] args) {
    checked {
      short x = 32767;
      short y = 32767;
      short z = (short)(x + y);
      System.Console.WriteLine(z);
    }
  }
}
```

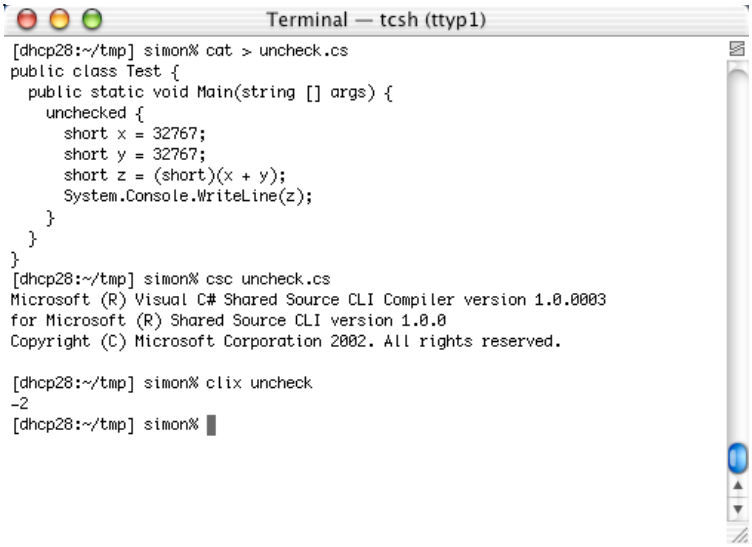
Checked Example



Unchecked Example

```
public class Test {
  public static void Main(string [] args) {
    unchecked {
      short x = 32767;
      short y = 32767;
      short z = (short)(x + y);
      System.Console.WriteLine(z);
    }
  }
}
```

Unchecked Example



Default Behavior

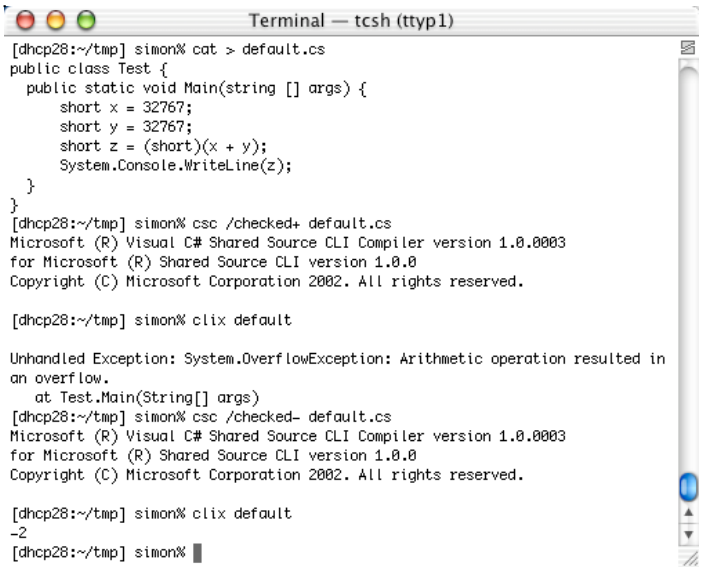
csc's /checked option controls the default

- default applies to code not contained within checked/unchecked block

csc /checked+ to enable checking

csc /checked- to disable checking

Example



Summary

C# has all the usual operators

- Also has some neat stuff for dealing with types

Numerical overflow is can be explicitly controlled by the developer

Flow of Control in C#

Selection

- if switch

Iteration

- while do for foreach

Jump

- break continue goto

If

Attention C programmers!

- Must evaluate an expression of type bool
- if (value) is NOT if (value == 0)

```
int value = 0;

if (value == 0) {
   // do something
} else if (value == 1) {
   // do something else
} else {
   // drastic action
}
```

Switch

All cases must end with either a break or a goto

Exception: empty case

You can use string variables with the switch statement

```
string test = "FORM";
switch(test) {
  case "BR":
  case "UL":
     Console.WriteLine();
     break;
  case "TD":
     Some.Stuff();
     goto "TABLE";
  case "TABLE":
     Some.More.Stuff();
     break;
  default:
     Major.Problem();
     break;
}
```


While

Loop continues to run while the condition is true

Similar behavior to most other languages

```
bool flag = true;
while (flag) {
   Do.Some.Stuff();
   if (success) {
      flag = false;
   }
}
```

Do / While

Similar to while loop

Guaranteed to run at least once

Behaves similar to other languages

```
bool invalidInput = true;

do {
  getSomeInput();
  if (checkInput()) {
     invalidInput = false;
  }
} while (invalidInput);
```


For

Iterate a set number of times

Creation of local variable (scoped inside of loop) is permitted

```
for (int i=0; i<10; i++) {
  Console.WriteLine(i);
}
```

Foreach

Most for loops iterate over an array or array-like object

"ArrayOutOfBounds Exception" is very common

Similar construct in PERL and VisualBasic

```
ArrayList a = ...

for (int i=0; i<a.Count; i++)
{
  MyObj tmp = (MyObj)a[i];
  Console.WriteLine(tmp);
}

foreach (MyObj tmp in a) {
  Console.WriteLin(tmp);
}
```


Foreach

Automatic type conversion

Loop variable is

- scoped to loop
- read only

```
Hashtable h = new Hashtable();

h.Add("Fred", "Flintstone");
h.Add("Barney", "Rubble");
h.Add("Wilma", "Flintstone");
h.Add("Betty", "Rubble");

foreach(string f in h.Keys) {
  Console.WriteLine("{0} {1}",
                    f, h[f]);
}
```

Break

Ends iteration or jumps out of the current switch statement

```
ArrayList a = ...

foreach (string tmp in a) {
  Console.WriteLin(tmp);
  if (tmp == “done”) break;
}

for(int i=0; i<5; i++ ) {
 for(int j=0; j<5; j++) {
  Console.WriteLine("{0}{1}",
                          i, j);
  if (j==2) break;
 }
 if (i==3) break;
}
```


Continue

Short-circuits current interation and moves execution onto the next iteration

```
ArrayList a = ...

foreach (string tmp in a) {
  if (tmp == “NOPRINT”) {
     continue;
  }
  Console.WriteLin(tmp);
}
```

Goto

Typical "bad" uses of goto are prohibited in C#

Only recommended use is in switch statements

```
string test = "FORM";
switch(test) {
  case "BR":
  case "UL":
     Console.WriteLine();
     break;
  case "TD":
     Some.Stuff();
     goto "TABLE";
  case "TABLE":
     Some.More.Stuff();
     break;
  default:
     Major.Problem();
     break;
}
```


Control Flow Summary

Conditionals

- If, Switch

Loops

- While, Do While, For, Foreach

Jump

- Break, Continue, Goto

Basic I/O

Command line arguments

Console I/O

- Console.ReadLine / Console.WriteLine

File I/O

- FileStream
- StreamReader / StreamWriter

Command Line Args

Passed in on console command line when the program to be executed is specified

Examples:

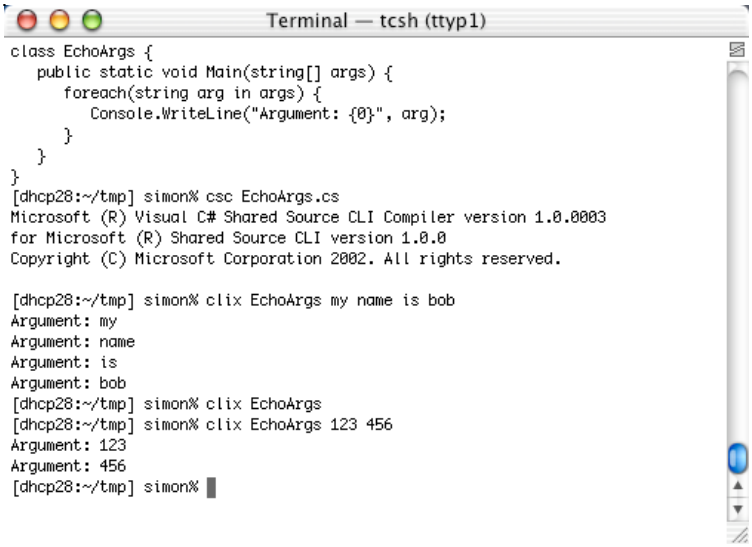
- Windows: HelloWorld.exe ABC 123 test
- UNIX: clix HelloWorld.exe ABC 123 test

Reading Arguments

```
using System;

class EchoArgs {
  public static void Main(string[] args) {
    foreach(string arg in args) {
      Console.WriteLine("arg: {0}", arg);
    }
  }
}
```


Example



Console I/O

Standard way to make a program interactive

Console.WriteLine can take lots of differents kinds of parameters including formatting strings

Console.ReadLine is a blocking call that returns a string

Console Output

```
int i = 3;
double d = 5.2;

Console.WriteLine(i);
Console.WriteLine(d);

// use a formatting string for
// stylized output
Console.WriteLine("i: {0}  d: {1}",
                                i,  d);
```

Formatting Numbers

General Form

{[0-9]:[CDEFGNPRX]nn}

First number represents argument number

Second represents output format (C is currency, D is decimal, E is exponential…)

Third represents precision

Example Formats

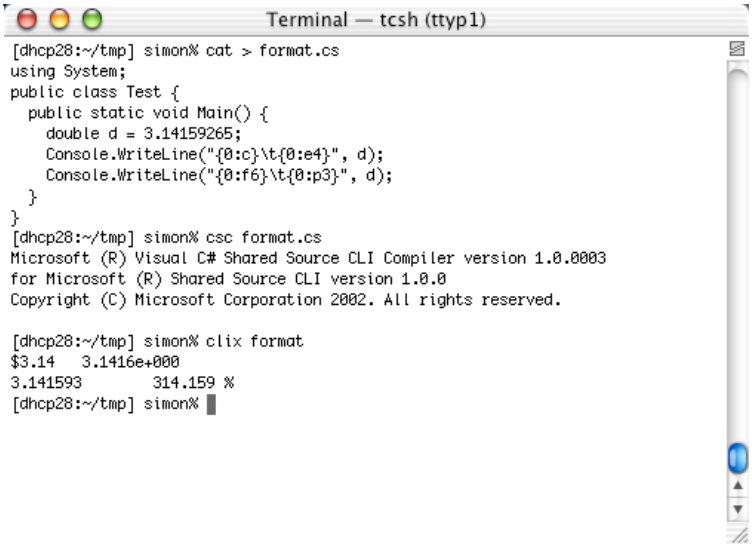
{0:c} - display zeroth arg as currency

{0:e4} - display in engineering notation with up to four places beyond the decimal point

{0:g6} - display in general (most compact) form with up to places beyond decimal point

{0:p3) - display as a percentage with up to three places beyond the decimal point

Example



Console Input

```
// Output a prompt without a
// newline to make it look right
Console.Write(”Enter number: “);

// ReadLine returns string
string s = Console.ReadLine();

// Parse the string using
// Convert to make numbers
int i = Convert.ToInt32(s);
double d = Convert.ToDouble(s);
```

File I/O

File input and output is useful for reading and writing from persistent storage

Can be emulated using < and > on most command shells

Two step process

- First create a handle onto a file
- Then create I/O streams on the handle

File Output

```
using System.IO;
class FileTest {
  public static void Main() {
    FileStream f;
    f = FileStream("myfile.txt",
                   FileMode.Create);
    StreamWriter w = new StreamWriter(f);
    w.WriteLine("{0} {1}", "test", 42);
    w.Close();
    f.Close();
  }
}
```

File Input

```
using System.IO;
class FileTest {
  public static void Main() {
    FileStream f;
    f = FileStream("myfile.txt",
                   FileMode.Open);
    StreamReader r = new StreamReader(f);
    string input = r.ReadLine();
    r.Close();
    f.Close();
  }
}
```


Basic I/O Summary

Command line arguments

- Passed in as string array to Main

Console I/O

- System.Console.ReadLine and WriteLine

File I/O

- FileStream combined with Reader/Writer

More on I/O

Object oriented filesystem access

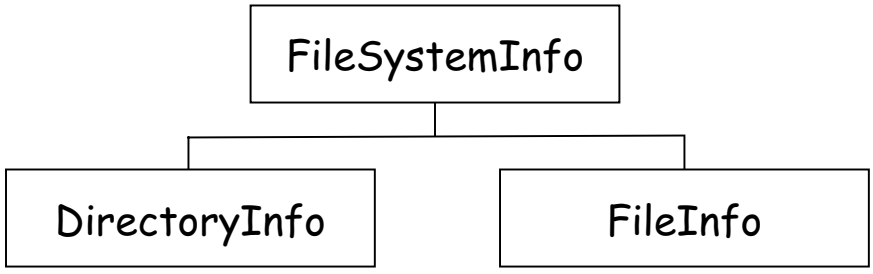
Procedural access to the filesystem

Basic socket I/O for network programming

Filesystem Access

Get file or directory (meta) information

Not really for modifying file contents



FileSystemInfo

Cannot be used directly

Contains things common to files and folders

- Name
- CreationTime
- LastAccessTime
- Delete

DirectoryInfo

Everything in FileSystemInfo, plus

- CreateSubdirectory
- GetFiles (returns FileInfo array)
- GetDirectories (returns DirectoryInfo[])
- Parent

FileInfo

Everything in FileSystemInfo, plus

- Length
- Directory (returns DirectoryInfo)
- CopyTo
- MoveTo

Directory Listing Example

```
using System.IO;
public class List {
  public static void Main(string[] args) {
    DumpList(args[0]);
  }
  public static void DumpList(string s) {
    DirectoryInfo d = new DirectoryInfo(s);
    System.Console.WriteLine(d.FullName);
    FileInfo[] files = d.GetFiles();
    foreach(FileInfo f in files)
      System.Console.WriteLine(f.Name);
    DirectoryInfo[] dirs = d.GetDirectories();
    foreach(DirectoryInfo d2 in dirs)
      DumpList(d2.FullName);
```

Searching for Files and Directories

GetFiles and GetDirectories can take a string pattern as an argument

Only entries matching the pattern will be returned

Often used for looking up a single file

```
DirectoryInfo d = new DirectoryInfo(s);
FileInfo[] fa = d.GetFiles("myfile.txt");
```


Filesystem Utility Classes

Utility classes contain functions to retrieve filesystem info

- Path
- Directory
- File

No need to instantiate objects, etc.

Path

```
string path =
    @"C:\WINDOWS\system32\mscoree.dll";
string a = Path.GetDirectoryName(path);
        // C:\WINDOWS\system32
string b = Path.GetFileName(path);
        // mscoreee.dll
string d = Path.GetExtension(path);
        // .dll
string e = Path.GetPathRoot(path);
        // C:\
```


Directory

```
// retrieve all drives on ssytem
string[] l = Directory.GetLogicalDrives();
foreach (string a in l) {
   // see if disk is in drive
   if (Directory.Exists(a)) {
      // get files on the root of the drive
      string[] f = Directory.GetFiles(a);
   }
}
```

File

```
// check to see if a file exists
bool b = File.Exists(filename);

// remove a file
File.Delete(filename);

// copy or move a file
File.Copy(source, destination);
File.Move(source, destination);
```


Object Oriented vs. Procedural Access

Procedural Access

- Permission checks on each call
- More convenient if doing one thing

Object Oriented

- Lots of permission checks at instantiation
- Not efficient if only doing one thing

Socket I/O

- "Low-level" network access
  - Lots of "plumbing code" to deal with
  - Remoting is far easier to manage
- Useful in certain circumstances
  - Integrating with legacy network systems
  - May give higher performance than RPC

URL Retriever

```
using System.IO;
using System.Net.Sockets;
public class GetWeb {
  public static void Main(string[] args) {
    TcpClient c = new TcpClient();
    c.Connect(args[0], 80);
    Stream s = c.GetStream();
    StreamWriter sw = new StreamWriter(s);
    sw.WriteLine("GET /");
    sw.Flush();
    StreamReader sr = new StreamReader(s);
    string buf = sr.ReadToEnd();
    System.Console.WriteLine(buf);
  }
```

Server Shell

```
public class Server {
  public static void Main(string[] args) {
    int p = Convert.ToInt32(args[0]);
    TcpListener l = new TcpListener(p);
    // the following line blocks
    Socket s = l.Accept();
    // only get here when client connected
    DealWithClient(s);
  }
}
```


Dealing with the Client

```
public class Server {
  public void DealWithClient(Socket s) {
    NetworkStream ns = new NetworkStream(s);
    StreamReader sr = new StreamReader(ns);
    StreamWriter sw = new StreamWriter(ns);
    while(<we're not done criteria>) {
       string input = sr.ReadLine()
       sw.WriteLine(output);
    }
  }
}
```

Summary

Two ways to access filesystem metadata

- OO: FileSystemInfo, DirectoryInfo
- Procedural: Path, Directory, File

Sockets provide low level network access

- TCPClient, Socket, NetworkStream

C#

Object Oriented Programming

What is an Object?

Collection of information and functionality

Three basic kinds

- real-world manifestation (e.g., employee)
- virtual meaning (e.g., window on screen)
- convenient abstraction (e.g. display list)

Inheritance

Extend functionality of an object

- base class -> derived class
- Vehicle -> Car -> Honda

No need to re-implement existing information storage and functionality

Often touted and widely overused

Inheritance Problems

Derived class "is-a" base class

- Base always more general than derived
- Real world problems aren't this simple

Author must code with inheritance in mind

- Changing base can break derived class

Containment

Put objects inside other objects as "state"

- Allows for aggregation in languages that do not support multiple inheritance

Switch to inheritance only if is-a relationship truly applies

Polymorphism

Developers can pass around references to base class rather than derived class

- Useful for writing "generic" code
- Eases partitioning of tasks
- Usually comes in handy with interfaces

Encapsulation

Reduce direct visibility of information

Force developer to use functionality rather than modifying information directly

Guarantee compatibility with future versions

Summary

Object - information and functionality combined into one package

Key concepts:

- Inheritance and Polymorphism
- Containment and Encapsulation

Simple Class Definition

```
class SimpleClass {
   // information
   public int first = 0;
   public float second = 0;
   // functionality
   public double getSum() {
      double output = first + second;
      return output;
   }
}
```

Using a Class

A class is a template … it must be instantiated before it becomes useful

The new keyword creates a new instance

```
class UseSimpleClass {
  public static void Main() {
    SimpleClass SC = new SimpleClass();
    Console.WriteLine(SC.getSum());
  }
}
```


Arrays of Classes

Reference types initialize to null

```
SimpleClass[] arr;
arr = new SimpleClass[2];
arr[0] = new SimpleClass(12, 2.07);
arr[1] = new SimpleClass(-1, 15.2);
```

Constructors

Function with the same name as the class

Typically used to initialize values

Special variable `this` is often used here

- `this` refers to the current instance
- `this` is available in all member functions

Constructor Example

```
class SimpleClass {
   int first = 0;
   float second = 0;
   public SimpleClass(int f, float s) {
      this.first = f;
      this.second = s;
   }
   ...
}
```

Read Only Fields

- const prefix
  - Compile-time specified values
- readonly prefix
  - Value specified as an initializer or in the constructor

Example

```
public class ReadOnlyDemo {
  public const float pi = 3.14f;
  public static readonly Color Red;
  public static readonly Color Green =
                      new Color(0,255,0);
  public ReadOnlyDemo() {
     Red = new Color(255,0,0);
  }
}
```

Member Functions

Functions with a name other than class name

Must return something (or void)

Accessed by using the . (dot) after instance

```
myInstance.myFunction()
```


Simple Example

```
class SimpleClass {
   int first = 0;
   float second = 0;
   public int getFirst() {
      return first;
   }
   public double getSecond() {
      return second;
   }
}
```

Ref and Out

Function can only return one value

Two function calls to get the two values

C# allows for call-by-reference parameters

- put ref prefix on parameter
- use out prefix for output only

Ref and Out Example

```
class SimpleClass {
   int first = 0;
   float second = 0;
   public void getData(ref int f, out float s)
   {
      f = this.first;
      s = this.second;
   }
}
```

Calling with Ref / Out

```
class CallerClass {
   public static void Main() {
     int i = 0;  // need to initialize
     float f;    // unless we use “out”
     SimpleClass SC;
     SC = new SimpleClass(10, 3.14159f);
     SC.getData(ref i, out f);
   }
}
```


Overloading

Functions can have same name

Need to have different parameters

Particularly common for constructors

```
class SimpleClass {
  public doStuff(int i) { ... }
  public doStuff(float f) { ... }
}
```

Variable Length Parameter Lists

Methods can be setup to accept an arbitrary number of parameters

Console.WriteLine() is a good example

Add the params prefix to a parameter array to change the way the compiler looks up the functions

Example

```
public class ParamsDemo {
   public int sum(params int[] args) {
      int sum = 0;
      foreach(int i in args) {
         sum += i;
      }
      return sum;
   }
   public static void Main() {
      ParamsDemo PD = new ParamsDemo();
      Console.WriteLine(PD.sum(3,3,3));
   }
}
```

Caveats

There is serious overhead to this

- An array is automatically created and destroyed behind the scenes

Not all languages support this

Recommendation: create overloads with one, two and three parameters as well as array

Operator Overloading

Make classes behave like primitives when used with +, -, *, ++, etc.

Allow classes to be compared logically (e.g., by contents)

Example

```
class C {
  public double r = 0;
  public double i = 0;
  public static bool operator==(C c1, C c2) {
    if ((c1.r == c2.r) && (c1.i == c2.i)) {
       return true;
    } else {
       return false;
    }
  }
}
```


Another Example

```
class C {
  public double r = 0;
  public double i = 0;
  public static C operator*(C c1, C c2) {
     C c = new C();
     c.r = c1.r * c2.r - c1.i * c2.i;
     c.i = c1.r * c2.i + c2.r * c1.i;
     return c;
  }
}
```

Usage

```
class Test {
  public static void Main() {
    C me = new C();
    C me2 = new C();
    me.r = 1;  me.i = 2;
    me2.r = 1; me2.r = 3;
    if (me == m2) { // do something }
    C me3 = me * me2;
  }
}
```


Accessibility

Hide parts of classes or entire classes from the developer using the class

Extremely useful in code distribution and reuse scenarios to prevent unpredictable behavior

Can be applied to classes and members

Valid Modifiers

public - anybody can access it

private - only this class can access it

protected - this class and it's derivatives

internal - everybody in this assembly

internal is often combined with protected

Plausible Example

```
public class AccessDemo {
   internal protected int i;
   public int getValue()
      return i;
   }
   private void init() {
      i = 42;
   }
}
```

Combining Modifiers

All modifiers must be satisfied

The public member is automatically reduced to internal accessibility

```
interal class AccessDemo {
   public int i;
   internal int j;
   protected int k;
}
```


Nested Classes

Useful for creating private helper classes

Prevents pollution of the namespace

Helps developers realize when a class is completely not important to understand except if you need extreme detail

Stronger protection than an `internal` class

Example

```
public class Parser {
   Token[] tokens;
   private class Token {
      string name;
   }
}
```


Private Constructors

Prevents constructor from being run

Useful in numerous cases

- function only classes (e.g., System.Math)
- singleton classes (only one should exist)

Summary

Define objects with the `class` keyword

Use objects with the new keyword

Classes contain methods and fields

Class creation is controlled via constructors

Visibility of elements within a class is controlled with `public`, `private`, etc.

Inheritance

Base class contains general information and functionality

Derived class inherits all information and functionality from the base class

Derived class defined by the : (colon)

Inheritance Example

```
class Other: SimpleClass {
   int j;
   public Other(int i, float f, int j)
            : base(i, f) {
      this.j = j;
   }
   public new double getSum() {
      double val = j + first + second;
      return val;
   }
}
```


New and Base

new - override functions

Replace functions with new functions of the same name in derived class

base - call base constructions

Used in constructor declaration to call base class constructor

Binding

What if you have a reference of type SimpleClass to an instance of type Other

- Derived class overrides method in base
- Polymorphic reference to derived class is of the type of the base class

What happens when getSum is called?

Static Binding

The type of the reference defines the method being called

Type of reference is defined at compile time (hence term static)

This is the default behavior in C#

Dynamic Binding

Method being called depends on the type of the object

Type of the object is unknown until runtime

Use the virtual and override keywords to implement this

Example

```
public class SimpleClass {
   virtual public double getSum() {
      return first + second;
   }
}
public class Other : SimpleClass {
   override public double getSum() {
      return first + second + j;
   }
}
```

Virtual Functions in Arrays of Classes

Heterogeneous types can be used in an array while maintaining the right semantics

```
SimpleClass[] arr;
arr = new SimpleClass[5];
arr[0] = new SimpleClass(12, 2.07);
arr[1] = new Other(-1, 15.2, 5);

double d1 = arr[0].getSum();
double d2 = arr[1].getSum();
```


Abstract Classes

Force developers to "override" methods

Abstract classes cannot be instantiated

Abstract functions are used to define names without providing implementations

- Keeps code consistent
- Names of functions are well defined

Example

```
abstract class SimpleClass {
   abstract public double getSum();
}

public class Other : SimpleClass {
   override public double getSum() {
      return first + second + j;
   }
}
```


Sealed Classes

Basically the opposite of abstract classes

Sealed classes cannot be used as a base class to derive from

Useful to prevent breaking of strict requirements on how a class is to behave

Summary

Inheritance allows...

- objects to expand on other objects
- objects to be treated as other objects

Often abused, should use containment most of the time

The Static Keyword

Associates a field or member with the class rather than an instance of a class

```
class SimpleClass {
  public static int i;
  public static void doStuff() {
     // do some generic stuff
  }
}
```

Static Fields

All instances "share" this variable

Simple way to do shared memory

Closest thing to a "global variable"

Can only be accessed using the class name

Different from C++ / JAVA

Prevents code readability problems

Example

```
public class StaticTest {
   public static const double PI = 3.1415927;
   internal protected static int cnt = 0;
   public int getCnt() { return cnt; }
   public void incCnt() { cnt++; }
   public static void Main() {
      StaticTest A = new StaticTest();
      StaticTest B = new StaticTest();
      StaticTest C = new StaticTest();
      A.incCnt();   B.incCnt();
      Console.Writeline("{0} {1}",
              C.getCnt(), StaticTest.PI);
   }
}
```

Static Functions

Useful for providing utility or generic functions

Must never access non-static fields

Must be called using the class name

Example

```
public class Geometry {
  public static const double PI = 3.1415927;
  public static double
                Circumference(double radius) {
    return (radius * Geometry.PI * 2);
  }
  private Geometry() { }
  public static void Main() {
    Console.WriteLine(Geometry.Circumference(2);
  }
}
```

Static Constructor

Special case of a static function

Called before first instance of a class is created

Useful for doing one time initialization

e.g., setup pooled database connection

Cannot have any parameters

Example

```
public class StaticTest {
  interal protected static Connection DBConn;
  public static const
            DBURL = “odbc:localhost:NorthWind”;
  public static StaticTest() {
    DBConn = new Connection(StaticTest.DBURL);
  }
  public static void Main() {
    StaticTest ST = new StaticTest();
  }
}
```

Summary

Static keyword

- Generally means "global" or "one-of"
- Fields - global variables
- Functions - callable without instantiation
- Constructor - global initialization

Interfaces

Essentially abstract classes w/o fields

Defines a contract between two developers

Can be multiply-inherited

Often used with polymorphism to allow for runtime configuration of implementation

Example

```
interface TwoWayPager {
  void sendMessage(string msg);
  string[] getMessages();
  void setTime(Date updatedTime);
}
public class RIMBlackBerry: TwoWayPager {
   ....
}
public class MotorolaT900: TwoWayPager {
   ....
}
```


Using Interfaces

```
public class PagingSystem {
  public static void Main() {
    TwoWayPager[] pagers;
    pagers = new TwoWayPager[MAX_PAGERS];
    pagers[0] = new RIMBlackBerry();
    pagers[1] = new MotorolaT900();
    ...
  }
}
```

Checking Compatibility

```
public class PagingSystem {
  public static void Main() {
    ArrayList arr = new ArrayList(MAX);
    arr.Add(new ElCheapoPager());
    arr.Add(new MotorolaT900());
    foreach(Object o in arr) {
       if (o is TwoWayPager) {
          .....
       }
    }
  }
}
```


Alternative Method

More efficient (one cast per object that is valid instead of two) but less readable

```
foreach(Object o in arr) {
  TwoWayPager TWP = o as TwoWayPager;
  if (TWP != null) {
     ....
  }
}
```

Multiple Inheritance

```
interface IFoo {
   void doFoo();
}
interface IBar {
   void doBar();
}
class Test: IFoo, IBar {
   public void doFoo() { }
   public void doBar() { }
}
```


Explicit Implementation

What if two interfaces define contracts for a functions with identical signatures?

Prefix implementations with the interface name

Neither can be used directly by a type of the implementing class (requires cast to type of the interface)

Example

```
interface IFoo {
   void doStuff();
}
interface IBar {
   void doStuff();
}
class Test: IFoo, IBar {
   void IFoo.doStuff() { }
   void IBar.doStuff() { }
}
```


Combining Interfaces

Interfaces can inherit from other interfaces

Implementations must have functions for all contracts defined in all interfaces in the inheritance hierarchy

Example

```
interface IFoo {
   void doFoo();
}
interface IBar {
   void doBar();
}
interface IFooBar: IFoo, IBar {
   void doComboMove();
}
```


Summary

Interfaces

- Contract between caller and implementer
- Multiple implementation allowed
- Provides polymorphism

C#

Deeper into Language Details

Exceptions

Method for dealing with error conditions

Without exceptions, we use return codes

```
int errno = someFunction();
```

errno can be ignored

The meaning of errno may differ between functions

trying and catching

Surround code with potential error conditions in a try block

Follow the try block with catch blocks for each exception that might occur

Example

```
static int Zero = 0;
public static void Main() {
  try {
    int j = 42 / Zero
  } catch (Exception e) {
    Console.WriteLine(”e: “ + e.Message);
  }
}
```

More Details

Use multiple catch blocks to deal with different exceptions

Catch blocks must be listed from most specific to most generic

e.g., DivideByZeroException must be caught before Exception

Another Example

```
static int Zero = 0;
public static void Main() {
  try {
    int j = 42 / Zero
  } catch (DivideByZeroException e) {
    Console.Writeline(”Div: {0}, e);
  } catch (Exception e) {
    Console.WriteLine(”Ex: {0}“, e);
  }
}
```

Catching Exceptions

Exceptions can be caught anywhere in the call stack

Sometimes the best thing to do is to not catch the exception

Allows for other developers to deal with the exception in whatever way they want to

Example

```
class ExceptionTest {
  public SomeFunction() {
    // problem might occur over here
  }
  public static void Main() {
    try {
       SomeFunction();
    } catch(Exception e) {
       ...
    }
  }
}
```

Re-thrown Exceptions

Sometimes we need local exception handling

e.g., cleanup value-types

We also want to notify caller of a problem

Handle the exception, then re-throw

Potentially pass some extra info

Example

```
class ExceptionTest {
  public SomeFunction() {
    try { ... }
    catch(SomeException e) {
       // do local cleanup
       throw;
    }
  }
  public static void Main() {
    SomeFunction();
  }
}
```

Another Example

```
class ExceptionTest {
  public SomeFunction() {
    try { ... }
    catch(SomeException e) {
       // do local cleanup
       string msg = "problem occurred.";
       throw (new SomeException(msg, e));
    }
  }
  ....
}
```


Common Exceptions

C# library has lots of built-in exceptions

IndexOutOfRangeException

OutOfMemoryException

IOException

- FileNotFoundException
- FileLoadException

Custom Exceptions

The name of an exception "conveys" what went wrong

Sometimes the pre-defined exceptions are not good enough

Creating Exceptions

Create a class

- Derive from Exception, or some class that derives from Exception

Implement three constructors

- No params
- One string param (message)
- One string and one Exception

Example

```
class MyException: Exception {
   public MyException() { }
   public MyException(string message) :
                          base(message) {
   }
   public MyException(string message,
                       Exception inner) :
                    base(message, inner) {
   }
}
```


finally

Sometimes an exception occurs and you need to clean something up

- close open files
- commit or rollback database changes

An exception will probably prevent normal cleanup code from executing

Example

```
FileStream f;
f = new FileStream(”file”, FileMode.Open);
try {
   StreamReader t = new StreamReader(f);
   while((line = t.ReadeLine()) != null)
   {
      string line = t.ReadLine();
      int sum = Convert.ToInt32(line);
   }
} finally {
   f.close();
}
```


Summary

Exceptions provide…

- error reporting infrastructure
- propagation of reporting up the call stack
- guaranteed execution of cleanup code

XML Documentation

Language level support for documentation

Helps prevent synchronization problems between API documentation and actual API

Documentation is inline in source code!

Similar to JAVADOC, but more flexible because the result is XML, not HTML

Procedure

Use /// comments to put inline API documentation into the source code

Run `csc /doc:myfile.xml myfile.cs` to generate XML file from source code

Obtain an XSL file to transform the XML into something human readable (e.g., XHTML, PDF, Word document, etc.)

Example Source Code

```
using System;
///<remarks> Responsible for initialization and cleanup of web
/// service.  Does all sorts of other neat stuff as well. </remarks>
class StockTickerMain
{
   ///<summary> Keeps copy of client domain. </summary>
   String strDomainRequesting;

   ///<summary> Initialize StockTicker </summary>
   ///<return> Returns non-zero value if init fails. </return>
   ///<param name="cnfFileName"> Path of config file. </param>
    public int Init(String cnfFileName) {
      return 0;
    }
}
```


Recognized Tags

Tags in the /// blocks are defined in a special XML DTD unique to C#

Primary tags for semantic associations

e.g., <remarks>, <summary>, ...

Support tags for controlling look and feel

e.g. <code>, <list>, <item>, ...

Summary and Remarks

<summary> is used for providing information about a member (e.g. field or function)

<remarks> is used like <summary> but for an actual type definition (e.g. class or struct)

You can define <summary> for a class and <remarks> for a field but you should not do this by convention

Example and Exception

Put <example> around a demonstration

- Typically also put <code> (discussed later in support tags) around the actual source

<exception cref="SomeException"> is used to discuss any exceptions that might occur

- Use one <exception> block for each exception that is likely to be thrown

Param and Returns

<param name="someparam"> used to talk about a particular passed parameter

One for each parameter per function

<returns> describes what is returned by the function

One for each function

Seealso and Include

<seealso cref="SomeMember"> provides a link to something the user should "see also"

<include file="somefile.xml" path="xpath/"> is used to pull in an external XML file

C and Code

Use <c> and </c> for inline code

- Basically used to change to fixed font

Use <code> and </code> around large blocks

- Also used for fixed font change
- Typically implies paragraph separation

Lists

<list> and </list> describe list context

<listheader> is for the header

<item> is for each member of the list

Members (both <listheader> and <item>) can optionally have <term> and <description>

Text Blocks

<para> and </para> define paragraphs

<paramref>ParameterName</paramref> is used to link to a parameter description inside a paragraph

<see cref="SomeTypeOrMember" /> provides links inside of a paragraph

Resulting XML file

```
<?xml version="1.0"?>
<doc>
  <assembly> <name>stockdemo</name> </assembly>
  <members>
    <member name="T:StockTickerMain">
      <remarks> Responsible for initialization and cleanup of web
      service. Does all sorts of other neat stuff as well. </remarks>
    </member>
    <member name="F:StockTickerMain.strDomainRequesting">
       <summary>  Keeps copy of client domain. </summary>
    </member>
    <member name="M:StockTickerMain.Init(System.String)">
       <summary> Initialize StockTicker </summary>
       <return> Returns non-zero value if init fails. </return>
       <param name="cnfFileName"> Path of config file. </param>
    </member>
  </members>
</doc>
```

Displaying Results

Raw XML is flexible but not readable

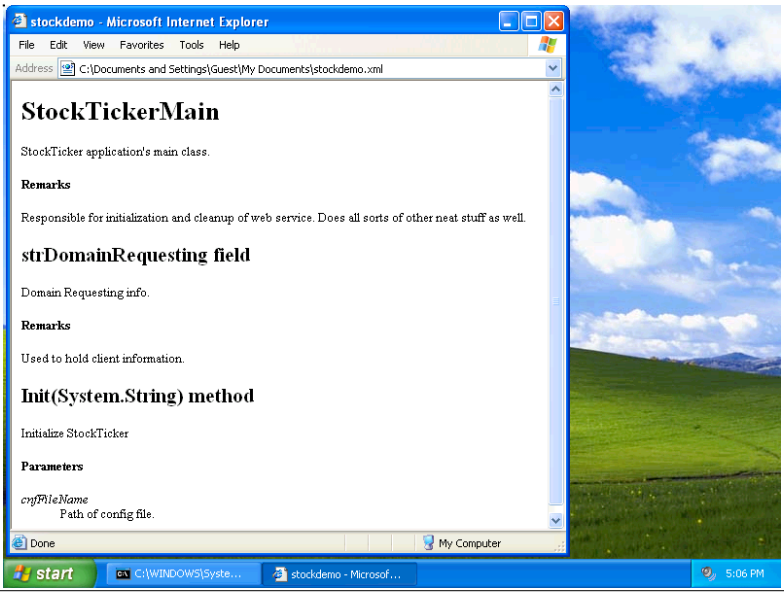
Apply an XSL to get XHTML

Add the folowing line right below header

```
<?xml-stylesheet href="doc.xsl" type="text/xsl"?>
```

Sample doc.xsl is available on MSDN website and mirrored on the class website

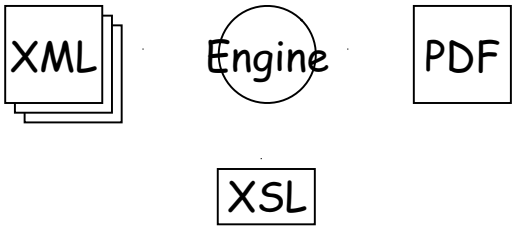
Final Result



More Complex XSLT

Using XSL transforms to get PDF or Word Documents is a little harder

- Need to use an XSLT engine
- Included with most XML libs (e.g. DOM)



Summary

XML documentation support provides...

- Inline API documentation with your code
- Flexible presentation via XSL transforms

Properties

Typical paradigm used to protect state vars

private state variable myField

public functions getMyField, setMyField

Example

```
class MyClass {
   private int someValue = 0;
   public int getSomeValue() {
       return someValue;
   }
   public void setSomeValue(int value) {
      this.someValue = value;
   }
}
```

Potential Problems

Naming convention is not enforced

- Developer can easily make a mistake
- Beginners may not fully understand this convention and may screw it up

A lot of extra typing

Accessors

Create a private state variable

Declare a public variable but put curly braces on the end of it

- In the curly braces, put two code blocks prefixed by `get` and `set`
- The identifier `value` is used in the code blocks to represent user input

Example

```
class Test  {
  private string name;
  public string Name {
    get {
       return name;
    }
    set {
       name = value;
    }
  }
}
```


Using the Property

```
class UseTest  {
  public static void main() {
     Test t = new Test();
     t.Name = “Fred”;
     Console.WriteLine(t.Name);
  }
}
```

Getters and Setters

Readonly properties do not have a set block

Writeonly properties do not have a get block

The type of `value` is always the same as the type of the property

Two Usage scenarios

Initialize values

- Declare values to be zero or negative one
- Load actual values from database or file

Side-effects to actions

- Fire some kind of event or function when a value is read or written to

Initialization

```
class Test  {
  private int z = -1;
  public int Z {
    get {
       if (z == -1) {
          z = loadValue();
       }
       return z;
    }
  }
}
```


Side-effects

```
class Test  {
  private int z = -1;
  public int Z {
    set {
       z = value;
       updateTotals();
    }
  }
}
```

Another Side-effect

```
class Test  {
  private int z = -1;
  public Decimal Total {
    get {
      Decimal out = quantity * price;
      if (quantity >= 10) {
          out *= 0.9m
      }
      return out;
    }
  }
}
```


Static Properties

Return a fresh new instance of a class each time it's requested

```
class Test  {
   public static Color Red {
      get {
          return(new Color(255,0,0));
      }
   }
}
```

Abstract Properties

Force a person deriving from your class to implement a particular property/accessor

```
public abstract class Test  {
   public abstract string Name {
      get;
   }
}
```


Properties Summary

Language-level support for encapsulated data with public access via methods

Keeps the code you write clean and in sync

Indexers

Sometimes we want to access an object like an array

- Triangle object contains three Point objs
- Might be useful to do Tri[0], Tri[1] and Tri[2] to reference points

Implementation similar to the Property paradigm

Example

```
class CardinalArray  {
  private int[] z;
  public CardinalArray(int size) {
     z = new int[size];
  }
  public int this[int i] {
    get {  return z[i-1];   }
    set {  z[i-1] = value;  }
  }
}
```

Usage

```
class Test  {
  static void Main() {
    CardinalArray CA;
    CA = new CardinalArray(5);
    CA[3] = 25;
    CA[5] = 100; // 1 to 5 is valid
  }
}
```


String Indexors

Sometimes it is useful to use strings to index into an object

Similar in functionality to a hashtable

Example

```
class CardinalArray  {
  private int[] z;
  int col(string name) {
     // do some mapping here
     // e.g., use switch
  }
  public int this[string name] {
    get {  return z[col(name)];   }
    set {  z[col(name)] = value;  }
  }
}
```


Multi-Dimensional Indexors

Sometimes we want many dimensions of indexing, similar to multi-dimensional array

Put a comma inside the square brackets to denote dimensions

Example

```
class Board  {
  private int[,] z = new int[8,8];
  int row(string r) {
     // convert r to an integer
     // e.g., A is 0, B is 1 ...
  }
  public int this[string r, int c] {
    get {  return z[row(r), c-1];   }
    set {  z[row(r), c-1] = value;  }
  }
}
```


Indexors Summary

Allow developers to access aggregate types in unusual ways

- As an array
- As a hashtable

Enumerators

If we can access an object as an array, we probably want to loop over it

In particular, we want foreach access to it

IEnumerable

Have your class implement `IEnumerable`

You must provide a function that matches `public IEnumerator GetEnumerator()`

The body of `GetEnumerator()` should instantiate a object that implements the `IEnumerator` interface

IEnumerable Example

```
class Test: IEnumerable  {
   public int this[int index] {
     get { .... }
   }
   public IEnumerator getEnumerator() {
     return (new MyEnumerator(this));
   }
}
```


IEnumerator Example

```
class MyEnumerator: IEnumerator  {
   Test t;
   int index;
   internal MyEnumerator(Test t) {
      this.t = t;
   }
   // need to implement 3 functions
}
```

IEnumerator Example

```
class MyEnumerator: IEnumerator  {
  public bool MoveNext() {
     bool output;
     index++;
     if (index >= t.size)
         output = false;
     else output = true;
     return output;
  }
}
```


IEnumerator Example

```
class MyEnumerator: IEnumerator  {
  public object Current {
     get {
         return(t[index]);
     }
  }
  public void Reset() {
     index = -1;
  }
}
```

Potential Problems

Enumerator may depend on things like database connections that need to be cleaned up

Lots of costly casts and conversions

Type-safeness enforced at run-time... hard to detect problems at compile time

Summary

Enumerators provide language level support for looping over elements of an object

Particularly powerful when combined with indexors and foreach loops

Enumerations

Map a set of identifiers (known at compile time) to a set of values

The values themselves aren't important

Convenient way to access values

Makes code more readable

Example

```
public class Draw {
  public enum LineStyle {
    Solid,
    Dotted,
    DotDash
  }
}
```

Using the Enumeration

```
public class Draw {
  public void DrawLine(LineStyle l) {
     switch(l) {
        case LineStyle.Solid:
        // draw here
        break;
        ....
     }
  }
}
```


Enumeration Base Types

Enumerations are always implemented as some kind of fixed point type

- The default is int

You might want to override this if

- more entries than can be held by an int
- want smaller storage size

Example

```
public class Test {
   public enum SmallEnum : byte {
      A,
      B,
      C,
      D
}
```


Initialization

Sometimes you want to control the actual value of the enumerations

Must be careful to always have a zero value

Example

```
public class Test {
   public enum Enum {
      A = 0,
      B = 1,
      C = 2,
      D = 4,
      E = 8
   }
}
```


System.Enum

Numerous utility functions for dealing with enumerations can be found in System.Enum

- GetNames
- GetType
- IsDefined

Summary

Enumerations provide...

- Syntactic sugar for associating lists with numerical values
- Developer can optionally control the mapping between list items and values

C#

Advanced Topics

AppDomains

AppDomain -> Process :: CLR/CLI -> OS/HW

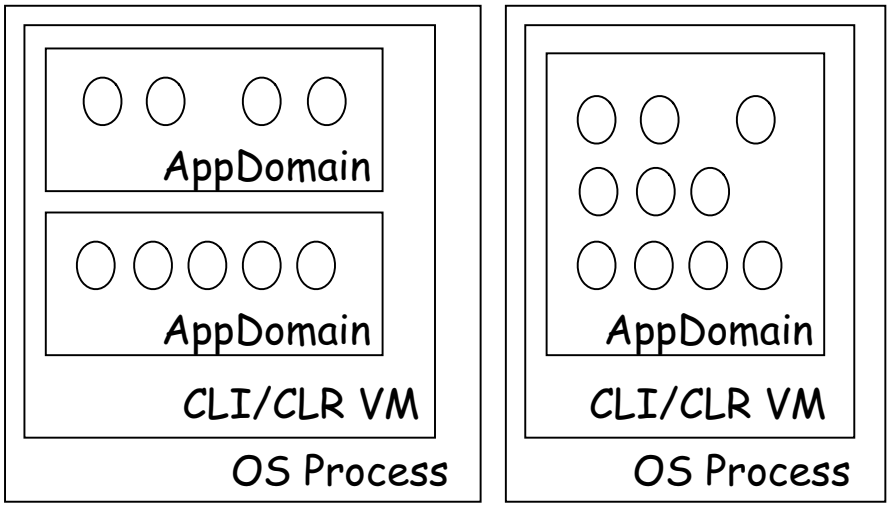
1 OS process can host 1 CLR/CLI VM

Each VM can host many AppDomains

AppDomains are "cheaper" than processes

Context switch is much faster

Objects, AppDomains and Processes



AppDomain API

System.AppDomain Class

- create new AppDomains
- stop / unload AppDomains
- execute EXE assemblies in an AppDomain
- read/modify environment variables

Usage Scenarios

Multi-user application hosting

- Data security / isolation
- Even statics aren't shared!

Five nines servers

- Dynamic code refresh

Simple Example

```
using System;
public class AppDomainTest {
  public static int Main(string[] args) {
    AppDomain a = null;
    a = AppDomain.CreateDomain("friend");
    int r = a.ExecuteAssembly("me.exe",
                              null, args);
    AppDomain.Unload(a);
    return r;
  }
}
```


AppDomain Meta-info

```
using System;
public class App {
  public static int Main() {
    AppDomain a = AppDomain.CurrentDomain;
    Console.WriteLine(a.FriendlyName);
    Assembly asm =
          Assembly.GetExecutingAssembly();
    Console.Writeline(asm.Location);
  }
}
```

Sandboxing a Type

```
using System;
public class AppDomainTest {
  public static int Main(string[] args) {
    AppDomain a = null;
    a = AppDomain.CreateDomain("friend");
    MyClass MC = (MyClass)
         a.CreateInstanceAndUnwrap(
               "myassembly", "MyClass");
    MC.doSomething();
  }
}
```


Summary

AppDomains provide…

- CLR/CLI version of a OS process
- Low overhead context switching
- Dynamic loading of entire applications
- Sandboxing of types

Interop

Not all code can be managed by CLI/CLR

- Vendors often provide COM DLL binaries
- Integrating with a legacy system
- Native code is often faster
- Leverage Windows specific technologies

Methodology

Make sure DLL is in the accessible PATH

Create a C# wrapper class that declares signatures of methods to be called

- Use DllImport attribute to "link" the DLL

Call the function as if it were a C# function

Example

```
public class MyWrapper {

  [ DllImport(”kernel32.dll”) ]
  public extern static void Sleep(uint msec);

  [ DllImport(”user32.dll”) ]
  public extern static uint MessageBox(
    int hwnd, string m, string c, uint flags);

}
```


Passing Structures

Many APIs/SDKs define structures to be passed as params to functions calls

- The function then populates the struct

We’re not in control of memory anymore!

- Replicate the structure in C#
- Mark the structure for sequential layout (in memory)

Example

```
[ StructLayout(LayoutKind.Sequential) ]
public struct TRACKER_DATA {
  public double x;
  public double y;
  public double z;
  public double theta;
  public double rho;
  public double phi;
  public byte pressure;
}
```


Example Continued

```
public class MyWrapper {
  [ DllImport("isense.dll") ]
  public extern static uint init();
  [ DllImport("isense.dll") ]
  public extern static uint
          readtracker(TRACKER_DATA d, uint h);
  public static void Main() {
    int handle = init();
    TRACKER_DATA t;
    readtracker(t, handle);
    Console.WriteLine("{0}:{1}",t.x,t.y);
  }
}
```

Batch Conversion

`tlbimp mylibrary.dll`

- Locates mylibrary.dll in the path
- Creates MyLibrary.dll assembly
- Automatically creates all the signatures
- May get confused if library is complex

Neat Things to Try

Every MS application is just a COM object

- Create an instance of Internet Explorer
- Display an MS Word document

Go the other way

- Interop allows managed code to be called from unmanaged code

IE Example

run the command `tlbimp shdocvw.dll`

create a C# source, file, add the following:

```
using SHDocVw;
public class IE {
  public static void Main() {
      InternetExplorer ie = new InternetExplorer();
      IWebBrowserApp iwba = (IWebBrowserApp) ie;
      iwba.Visible = true;
      iwba.GoHome();
  }
}
```


IE Example

To compile:

```
csc /r:SHDocVw.dll ie.cs
```

To cause IE to go to a particular page:

```
ie.Navigate(url, ref obj, ref
obj, ref obj, ref obj);
```

the obj can be a null object in the code

Interop Summary

Managed <-> Unmanaged code bridge

C# needs to call DLLs for various reasons...

- Windows specific technologies
- Vendor provided binaries
- Performance
- Legacy system integration

Garbage Collection

You are not in control

GC chooses...

- Who goes away
- What order they go away

GC will not cleanup unmanaged (aka "unsafe") objects

Object Destruction

You want cleanup stuff to happen when an object goes away

- This is doable with finalizers

You want the ability to make an object go away (and hence, cleanup) on demand

- This is not quite so easy

Sometimes a Problem

Resource contention may occur

Particularly problematic:

- Database connections
- File handles
- Graphics objects and display lists

Simple Case

Have something happen when garbage collection occurs

Override the Finalize method

Cannot be done directly to prevent users from doing stupid things

Use special C# syntax: ~ClassName()

Finalize Example

```
public class MyThing {
   SomeThingSpecial STS;

   public MyThing() {
      STS.open();
   }

   ~MyThing() {
      STS.close();
   }

}
```

IDisposable

- Allows a user to get rid of an object
- Defines a single method called "Dispose"
  - Allows user to specify what happens when an object goes away
- Preserves GC's right to make object go away

Example

```
public class MyThing: IDisposable {
   SomeThingSpecial STS;
   protected virtual void Cleanup() {
      STS.close();
   }
   public void Dispose() {
      Cleanup();
      GC.SuppressFinalize(this);
    }
   ~MyThing() { Cleanup(); }
}
```

Temporary Allocation of Reference Types

Sometimes you know something isn't going to live very long

You want to make sure it dies quickly

You can use a try/finally block for this

The using block is a nicer way to do it

The Try / Finally Way

```
public class Test {
  public static void Main() {
    FileStream fs = null;
    try {
      fs = new FileStream("a.txt", ...);
      fs.Write(blah, 8, ...);
    } finally {
      if (fs != null) {
         ((IDisposable)fs).Dispose();
      }
    }
  }
}
```

The Using Statement

```
public class Test {
  public static void Main() {
    using (FileStream fs =
         new FileStream("a.txt", ...)) {
       fs.Write(blah, 8, ...);
    }
  }
}
```


Summary

Garbage collection can be "controlled"

Finalizers and the `IDispose` interface

`using` keyword

Unsafe Code

C# has pointer arithmetic

- must use unsafe markers

Use for special cases, like...

- Advanced Interop / COM structures with pointers in them
- Performance critical code

Unsafe Markers

Designed specifically to prevent misuse

- Will not execute in secure environment

unsafe keyword added to function definition

```
public unsafe void Copy( ... ) {
```

compile with the /unsafe option

```
csc /unsafe MyFile.cs
```

Pointers

Use C-like syntex

int* i - i is a pointer to an integer

*i - dereferences the pointer i

&i - gets the address of i

Pointer Usage

```
public class Test {
  public static unsafe void me() {

    int i = 42;
    int* iptr = &i;

    int[] arr = { 3, 8, 22 };
    int* aptr = &arr[0];
    int* aptr2 = arr;

  }
}
```

GC Woes

Garbage collector can reorder memory at any time

This means pointers can magically start pointing to the wrong place!

Use the "fixed" keyword to prevent GC from moving things around

Example

```
public class Test {
  public static unsafe void ArrayCopy(
             int* src, int* dest, int cnt) {
     for (int i = 0; i < cnt; i++) {
        *src = *dest;  src++;  dest++;
     }
  }
  public static unsafe void caller(
             int[] src, int[] dest) {
     fixed(int* a = src, b = dest) {
        ArrayCopy(a, b, src.Length);
     }
  }
}
```

Summary

Unsafe code blocks provide...

- Ability to do pointer arithmetic
- High-performance execution

Make sure GC doesn't ruin the day by using the fixed keyword when apprpriate

Delegates

A contract between caller and implementer

Similar to interfaces, but different...

- Only specify single function
- Created at runtime

Very similar to "pointers to functions" found in many other languages

Typical Uses

Callback

- Notify a caller that a process completed

Dynamic behavior changes

- Specifying a different sorting algorithm

A World Without Delegates

We can do similar things without delegates

- Using classes alone
- Using interfaces

Both of these are quite clunky

Classes - Caller

```
class Slave {
  public void setMaster(Master m) { this.m = m; }
  public void DoWork() {
     Console.WriteLine(”Slave: work began”);
     if (m != null) m.WorkStarted();
     Console.WriteLine(”Slave: work done”);
     if (m != null) {
        int eval = m.WorkCompleted();
        Console.WriteLine(”Worker eval {0}”, eval);
     }
  }
  private Master m;
}
```


Classes - Target

```
class Master {
  public void WorkStarted() {
    Console.WriteLine(”Work faster!”);
  }
  public int WorkCompleted() {
    Console.WriteLine(”You are too slow!”);
    return 2;
  }
}
```

Classes - Usage

```
class Plantation {
  public static void Main () {
     Master m = new Master();
     Slave s = new Slave();
     s.setMaster(m);
     s.DoWork();
  }
}
```


With Interfaces

Slightly better, more flexible

Does not require implementer to use a particular type

Does not require having access to complete implementation ahead of time

Callback Interface

```
interface IMaster {
   void WorkStarted();
   int WorkCompleted();
}
```


Interfaces - Caller

```
class Slave {
  public void setMaster(IMaster m) { this.m = m; }
  public void DoWork() {
     Console.WriteLine(”Slave: work began”);
     if (m != null) m.WorkStarted();
     Console.WriteLine(”Slave: work done”);
     if (m != null) {
        int eval = m.WorkCompleted();
        Console.WriteLine(”Worker eval {0}”, eval);
     }
  }
  private IMaster m;
}
```

Interfaces - Target

```
class Master : IMaster {
  public void WorkStarted() {
    Console.WriteLine(”Work faster!”);
  }
  public int WorkCompleted() {
    Console.WriteLine(”You are too slow!”);
    return 2;
  }
}
```


Interfaces - Usage

```
class Plantation {
  public static void Main () {
     IMaster m = new Master();
     Slave s = new Slave();
     s.setMaster(m);  // polymorphism
     s.DoWork();
  }
}
```

Delegate Advantages

Enforce single method signature (not names)

... like a tiny interface with one method

Integration w/o source code or stubs/skels

Support for multiple call targets

Delegate Definitions

```
namespace SlaveSociety {

  delegate void WorkStarted();
  delegate int WorkCompleted();

}
```

Delegates - Caller

```
class Slave {
  public void DoWork() {
     Console.WriteLine(”Slave: work began”);
     if (started != null) strtd();
     Console.WriteLine(”Slave: work done”);
     if (completed != null) {
        int eval = cmpltd();
        Console.WriteLine(”Worker eval {0}”, eval);
     }
  }
  public WorkStarted strtd;
  public WorkCompleted cmpltd;
}
```


Delegates - Target

```
class Master {
  public void WorkStarted() {
    Console.WriteLine(”Work faster!”);
  }
  public int WorkCompleted() {
    Console.WriteLine(”You are too slow!”);
    return 2;
  }
}
```

Delegates - Usage

```
class Plantation {
  public static void Main () {
     Master m = new Master();
     Slave s = new Slave();
     s.strtd = new WorkStarted(m.WorkStarted);
     s.cmpltd = new WorkCompleted(m.WorkCompleted);
     s.DoWork();
  }
}
```


Multiple Targets

Delegates have automatic built-in support for multiple targets

Registered targets invoked sequentially

Manual iteration supported (needed for retrieving output) but not required

Example - Setup

```
class Plantation {
  public static void Main () {
     Master m = new Master();
     Master m2 = new Master();
     Slave s = new Slave();
     s.strtd = new WorkStarted(m.WorkStarted);
     s.strtd += new WorkStarted(m2.WorkStarted);
     s.cmpltd = new WorkCompleted(m.WorkCompleted);
     s.DoWork();
  }
}
```


Example - Iteration

```
class Slave {
  public void DoWork() {
     Console.WriteLine("Slave: work done");
     if (cmpltd != null) {
        foreach(WorkCompleted wc
              in completed.GetInvocationList() ) {
          Console.WriteLine("eval is {0}", wc());
        }
     }
  }
  public WorkCompleted cmpltd;
}
```

Delegate Summary

Essentially a pointer to a function with some value-added extras

- Strict enforcement of signature
- Multiple receivers can be associated with a given delegate

Events

Delegates with some extra features

- Public registration / deregistration
- Private implementation
- Limits who can fire the event / delegate to the client

Custom Add/Remove

The += and -= operators to add and remove events can be customized

Useful for controlling access to who can receive events

Example

```
public class Test {
  public event int SomeEvent {
       add {
          ...
       }
       remove {
          ...
       }
  }
}
```

Summary

Events are delegates with some extra syntactic sugar for adding and removing receivers

Asynchronous Execution

A single processor can appear to run many tasks at the same time

Useful for I/O handling, computationally intensive tasks or prioritizing operations

Incurs some amount of overheard due to context switching

Using Delegates

Create a delegate

Call BeginInvoke

IAsyncResult returned to get status, etc.

Call EndInvoke to harvest results and exceptions generated

Example

```
NameSpace Pi {
  public delegate double CalcPi(int precision);
  public class App {
    public static void Main() {
      CalcPi calcPi = new CalcPi(App.findPi);
      calcPi.BeginInvoke(42, null, null);
    }
    public static double findPi(int precision) {
      // task that takes a long time
      return 3.1415927;
    }
  }
}
```

Completion

Three ways to deal with completion

- Poll
- Block for some amount of time
- Set a callback

EndInvoke must be used in all cases to get return values and out parameters

Polling

```
NameSpace Pi {
  public delegate double CalcPi(int precision);
  public class App {
    public static void Main() {
      CalcPi calcPi = new CalcPi(App.findPi);
      IAsyncResult ar;
      ar = calcPi.BeginInvoke(42, null, null);
      while (!ar.IsCompleted) { Thread.Sleep(50); }
      double pi = calcPi.EndInvoke(ar);
    }
  }
}
```

Waiting w/o Poll

```
NameSpace Pi {
  public delegate double CalcPi(int precision);
  public class App {
    public static void Main() {
      CalcPi calcPi = new CalcPi(App.findPi);
      IAsyncResult ar;
      ar = calcPi.BeginInvoke(42, null, null);
      if (ar.AsyncWaitHandle.WaitOne(500)==true) {
        double pi = calcPi.EndInvoke(ar);
      } else {
        Console.WriteLine(”operation timed out”);
      }
    }
```


Callbacks

```
NameSpace Pi {
  public delegate double CalcPi(int precision);
  public class App {
    public static void Main() {
      CalcPi calcPi = new CalcPi(App.findPi);
      calcPi.BeginInvoke(42, new
             AsyncCallback(OnPiComplete), null);
    }
    static void OnPiComplete(IAsyncResult iar) {
       CalcPi calcPi = (CalcPi)ar.AsyncDelegate;
       double result = calcPi.EndInvoke(iar);
       Console.WriteLine(result);
    }
```

Timers

Very popular use of asynch execution

Have a callback function executed at a regular interval

Example

```
using System.Threading;
public class App {
  public static void Main() {
     Timer t = new Timer(1500);
     t.Elapsed += new ElapsedEventHandler(f);
     t.Start();
  }
  static void f(object arg, ElapsedEventArgs eea) {
     Console.WriteLine(”Hello again!”);
  }
}
```

Asynchronous I/O

Read / Write normally block

Simple workaround is to use threads

- Problem is that this is not scalable

Better way is to use asynchronous I/O

Simple Paradigm

Create AsyncCallback

Attach a function to AsychCallback

- Call EndRead / EndWrite in there

Call BeginRead / BeginWrite

- Pass in the AsychCallback as an argument

Example

```
public class ClientHandler {
  private AsynchCallback AC;
  public ClientHandler(Stream s) {
    AC = new AsyncCallback(OnReadComplete);
    s.BeginRead(buf, 0, buf.Length, AC, null);
    // free to do whatever you want here
    // because BeginRead doesn't block
  }
  private void OnReadComplete(IAsyncResult ar) {
    ns.EndRead(ar);
  }
}
```


Data Hazards

Concurrent programming means more than one thing can be accessing data at once

- Parallel read is generally not a problem
- Parallel updates generally is a big problem

Tools at Your Disposal

Monitor class - access to the SyncBlock, easiest way to get the job done

Interlocked class - quick/easy for primitives

ReadWriterLock - better performance, programmer specifies readers/writers

Mutex - most general solution

Monitor and SyncBlock

Each object has a special on-demand allocated memory block for lock state

Programmers agree on the object used

Access via static methods in Monitor class

- Enter/Exist for single-threaded access
- Pulse/Wait for resource-driven designs

Example

```
public class MonitorTest {
  public void ThreadSafeOperation() {
     Monitor.Enter(this);
     try {
        // Do the special thing here
     } finally {
        Monitor.Exit(this);
     }
  }
}
```


Shorthand Version

```
public class MonitorTest {
  public void ThreadSafeOperation() {
     lock(this) {
        // lock keyword is a shorthand version
        // of the code in previous example...
        // do the special something here
     }
  }
}
```

Protecting Statics

```
public class StaticThingie {
  private static int a = 0;
  private static int b = 1;
  public void ThreadSafeOperation() {
     lock(typeof(StaticThingie)) {
        a = b;
        b++;
     }
  }
}
```


Be careful!

Simple DoS... external agent calls Monitor.Enter(refToYourObject) without calling Monitor.Exit();

Subsequent calls to Monitor.Enter() block

Always lock on private members!

May need to add sentinel locking members

Example

```
public class MonitorTest {
  private object sync = new object();
  public void ThreadSafeOperation() {
     lock(sync) { ... }
  }
  public void AnotherThreadSafeOp() {
     lock(sync) { ... }
  }
}
```


What If...

Need to grab lock and check if a resource is available...

Might end up forgetting to release lock if resource is not yet ready

Solution: Monitor.Pulse and Monitor.Wait

Example

```
public class MonitorTest {
  public void Producer() {
    lock(this) {
      produceTheGoods();
      Monitor.Pulse(this);
    }
  }
  public void Consumer() {
    lock(this) {
      while(<notready>) { Monitor.Wait(this) }
      consumeTheGoods();
    }
  }
}
```


InterLocked

Atomic operations on primitives are often needed…

```
lock(someObj) { i++; }
```

InterLocked class provides this in shorthand and some additional features

InterLocked Details

Some of the functions available…

Increment/Decrement - change and store a fixed point value as an atomic operation

CompareExchange - check for equality and if so, replaces one of the params

Exchange - sets a value as an atomic operation

Example

```
public Class InterLockedTest {
  private int i = 0;
  public void MoveUp() {
    // increments i atomically
    Interlocked.Increment(ref i);
  }
  public void CheckAndSetValue(int j, int k) {
    // results in atomic version of this code:
    //     if (i == k) { i = j; }
    Interlocked.CompareExchange(ref i, j, k);
  }
}
```

ReaderWriterLock

Reading is generally not a problem unless somebody is writing at the same time

- Previous methods have bad performance

ReaderWriterLock to the rescue

- reader can get lock if 0 writers
- writers get lock if 0 readers & 0 writers
- programmer classifies readers/writers

Example

```
public Class RWL {
  prviate ReaderWriterLock rw;
  public RWL { rw = new ReaderWriterLock() }
  public void Read() {
    rw.AcquireReaderLock(Timeout.Infinite);
    // do reading here
    rw.ReleaseReaderlock();
  }
  public void Write() {
    rw.AcquireWriterLock(Timeout.Infinite);
    // do writing here
    rw.ReleaseWriterlock();
  }
}
```

Mutex

What if you have to wait for multiple locks?

Always need to grab in the same order!

- Deadlock will result otherwise

Mutex class hides these problems from you

- Can also be used for interprocess synchronization

Example

```
public Class MutexTest {
  private Mutex myLock = new Mutex();
  public void DoSomething(Some otherGuy) {
    Mutex[] locks = { myLock, otherGuy.lock };
    WaitHandle.WaitAll(locks);
    // do my special stuff here
    foreach (Mutex m in locks) {
      m.ReleaseMutex();
    }
  }
}
```

Summary

Asynchronous execution allows multiple program threads to run in "parallel"

- Good for I/O blocking, but true parallelism requires MP hardware

Data hazards can occur

- Many synchronization tools are available to address this issue

Reflection

Allows anyone to see all information about any type (barring security)

- Information is extensible via attributes
- Information is never optional

Provides support for numerous processes

- Runtime services (remoting, serialization)
- Development tools (code gen, XML docs)

Example

```
public static void DumpObj(Object obj) {
  Type t = obj.GetType();
  Console.WriteLine("Type is {0}", t.Name);
  foreach(FieldInfo f in t.GetFields()) {
    Console.WriteLine("  {0} {1}",
                      f.Name, f.FieldType);
  }
}
```


System.Type

All objects/values are instances of types

Discover the type with GetType method

Reference type name with typeof keyword

All types inherit from System.Type

2 Ways to Get There

typeof(Test) ==
this.GetType();

typeof(int) ==
this.i.GetType();

typeof(double) ==
this.d.GetType();

typeof(bool) ==
this.b.GetType();

```
public class Test {
  int i;
  double d;
  bool b;
}
```


Walking an Assembly

```
using System.Reflection;

public static void ListAll(String name) {
  Assembly a = Assembly.Load(name);
  foreach(Module m in a.GetModules()) {
    foreach(Type t in m.GetTypes()) {
      foreach(MemberInfo mi in t.GetMembers()) {
        Console.WriteLine("{0}.{1}" t, mi.Name);
      }
    }
  }
}
```

Further Info

Further info can be obtained by methods run on the type object

- FieldInfo (GetField)
- MethodInfo / ConstructorInfo (GetMethod)
- PropertyInfo (GetProperty)

Reflection Summary

Types can be loaded off of disk and...

- Examined (walked)
- Instantiated and used

Very useful for dealing with run-time loaded types (e.g. over the web)

Attributes

User (or system) defined logical association of information with types, functions, etc.

- Change CLI behavior of target
- Give additional information info
  - "Link" a type with a database field
  - Specify the author of a type for docs

Example

```
[assembly: Author("John Doe")]

[ Author("Joe Schmoe")]
class JoesClass {

  [ Obselete("Will be removed in next rev.")]
  public DoesSomeLegacyStuff() {
     // bad code
  }

}
```

Built-in Attributes

Numerous built-in attributes for lots of different things

- DefaultValueAttribute
- SerializableAttribute
- DllImportAttribute
- and lots more

Custom Attributes

Sometimes the built-in ones are not enough

Create a class

- Inherit from System.Attribute
- Add properties for the values you want to store in the attribute
- Add constructors

Example

```
[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: System.Attribute {
  public TestAttribute(string t, string d) {
     this.r = r;
     this.d = d;
  }
  public string Reviewer {
     get { return r; }
  }
  private string r;
  private string d;
}
```


Attribute Reflection

```
public void reflect(string asm, string type) {
  Type at = typeOf(MyAttribute);
  Assembly a = Assembly.Load(asm);
  String s = string.Format("{0}, {1}",
                           type, asm);
  Type t = Type.GetType(s);
  object[] arr =
      t.GetCustomAttributes(at, false);
  foreach(MyAttribute m in arr) {
       ....
  }
}
```

Summary

Attributes provide a method for

- associating user-defined meta-data with members of a type
- the developer to mark classes to behave a particular way during runtime

Serialization

Turn a in-memory object into a stream of bytes or characters

Useful for:

- Persistent storage of objects
- Transporting objects between VMs

Logistics

Object must be marked serializable

- System.Seriablizable attribute
- implement ISeriablizable interface

Need all sorts of namespace imports

- System.Runtime.Serialization;
- System.Runtiem.Serialization.Formatters;

Text vs. Binary

Objects can be turned into either

- Binary streams (very efficient)
- XML documents (human-readable, more compatible)

SoapFormatter and BinaryFormatter both implement the IFormatter interface

Object to Stream

```
[System.Serializable]
public class Payload {
  public int data = 0;
}
public class Test {
  public static void Main() {
    Payload p = new Payload();
    p.data = 42;
    SoapFormatter f = new SoapFormatter();
    Stream so = File.OpenWrite("p.dat");
    f.Serialize(so, me);
    so.Close();
  }
}
```


Stream to Object

```
[System.Serializable]
public class Payload {
  public int data = 0;
}
public class Test {
  public static void Main() {
    SoapFormatter f = new SoapFormatter();
    Stream si = File.OpenRead("p.dat");
    Payload p = (Payload) f.Deserialize(si);
    si.Close();
    System.Console.WriteLine(p.data);
  }
}
```

Summary

Take and object and make it into a stream

Developer has a choice of streams

- Space efficient binary streams
- Platform independent XML documents

Remoting

Objects running in one VM should be able to access (call functions on) objects that are in a different VM

Usually this means the two processes are running on two different computers

Similar to RPC and RMI

Procedure

Instantiate target object (on server)

Setup your channels

Register your object (on server)

Get a reference to object (on client)

Call method on server from the client

Server

```
public class Server {
  public static void Main() {

    Target t = new Target();
    RemotingServices.Marshal(t, "me");

    IChannel c = new HttpChannel(8080);
    ChannelServices.RegisterChannel(c);

    Console.WriteLine("Server ready.  ");
    Console.ReadLine();

  }
```

Client

```
public class Client {
  public static void Main() {
    string url = "http://localhost:8080/me";
    Type to = typeof(Target);
    Target t = (Target)
             RemotingServices.Connect(to, url);
    Console.WriteLine("Connected to server.");
    try {
      string msg = Console.ReadLine();
      t.sendMessage(msg);
    } catch(Exception e) {
      Console.WriteLine("Error " + e.Message);
    }
```


Shared Class

```
public class Target : System.MarshalByRefObject
{

  public void sendMessage(string msg) {
    System.Console.WriteLine(msg);
  }

}
```

Passing Types

Reference types passed via remoting must be serializable or marshal-by-reference

Serializable == pass-by-value

copy of object is made

MarshalByRefObject == pass-by-reference

proxy object is passed

Example

```
public class Counter : MarshalByRefObject {
  int count = 0;
  public void incCount() { count++; }
  public int getCount() { return Count; }
}

[Serializable]
public class Counter {
  int count = 0;
  public void incCount() { count++; }
  public int getCount() { return Count; }
}
```

Details

Lots of imports

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```


More Details

Shared object must be available to both the client and the server

Make a separate assembly for it

```
csc /t:library Shared.cs

csc /r:Shared.dll Server.cs

csc /r:Shared.dll Client.cs
```

Dynamic Configuration

Hardcoding config info is ugly

- port
- registration name
- server IP or DNS name

Example

```
class Client
  static void Main() {
    RemotingConfiguration.Configure(”my.cfg”);
    Type to = typeof(Target);
    Target t = (Target)
      RemotingServices.Connect(to,
      ConfigurationSettings.AppSettings[”url”]);
    try {
      t.sendMessage(c.ReadLine());
    } catch(Exception e) {  ...  }
  }
}
```

Config File Example

```
<configuration>
  <system.runtime.remoting>
     <application><client>
        <wellknown
        type="Target, Target"
        url="http://localhost:8080/me" />
     </client><channels>
        <channel ref="http" port="0" />
     </channels></application>
  </system.runtime.remoting>
</configuration>
```


Revised Server

```
public class Server {
  public static void Main() {

     RemotingConfiguration.Configure(”srv.cfg”);

     Console.WriteLine(”Hit any key to exit”);
     Console.ReadLine();

  }
}
```

Client Activated Types

Each client gets it's own instance of a type

Actually use the "new" keyword

Types are not sandboxed

Use AppDomains to sandbox them

Server

```
public class Server {
  static void Main() {
    ChannelServices.RegisterChannel(
                    new HttpChannel(8080));
    RemotingConfiguration.ApplicationName = "hi";
    RemotingConfiguration.
      RegisterActivatedServiceType(typeof(A));
    Console.WriteLine("Server ready.");
    Console.ReadLine();
  }
}
```

Client

```
public class Client
  static void Main() {
    string url = “http://server.com:8080/me”;
    RemotingConfiguration.
        RegisterActivatedClientType(
        typeof(A), url));
    try {
      A a = new A();
      a.dostuff();
    } catch {
      Console.WriteLine(”server call failed”);
    }
  }
}
```


Remoting Summary

Remote (network) access to any object

Special support for…

- dynamic configuration via files
- client activation