# .NET Programing with C#

# Contents

# Introduction

- This section covers:
  - Introduction to .NET
  - Introduction to Object Oriented Programming (OOP)
  - Visual Studio .NET
  - Simple Program
  - Console IO
  - Namespaces

# What is .NET?

- .NET is Microsoft's newest approach to the development of cross platform applications.
- Platform
  - Hardware independent (Windows, Linux, …)
- Languages
  - languages inherently support Object Oriented Programming (OOP)
  - Large library available – Object Oriented Class Library
- Component model
  - Packaged into Assemblies
- Communications:
  - Simple protocols HTTP, SOAP
- WCF (.NET 3)
  - Flexible creation and consumption of services

# Common Type System (CTS)

- The .NET Common Type System provides:
  - Cross language integration
  - Type Safety
  - Object Oriented model
  - A library of primitive types
- Types are either Value or Reference Types
- CTS supports the categories
  - Structures; Classes; Interfaces; Enumerations and Delegates

# .NET Core

- .NET Core is a new version of .NET to allow Cross Platform development:
  - Can be installed on Windows, Linux and Mac
  - Does not support all ‘sub’ frameworks
  - Command line tools available for working with Core
  - Development environment ‘Visual Studio Code’ available of multiple platforms
    - Lightweight development environment

# .NET Framework (now Core!)

- It consists of a Common Language Runtime (CLR), which allows the executions of code written in many different languages.
- The languages must compile up to Common Intermediate Language (CIL).
- The source code uses the .NET Framework Class Library to provide support for common features such as windowing (graphics), database access, threading and much more.
- Active Server Pages .NET (ASP.NET) provides for development of Web sites allowing Server Side Processing

# .NET

**.NET Framework 4.8**
(Full Framework)
- Web
  - **ASP.NET**

- Desktop
  - WPF
  - Windows Forms

**.NET Core and .NET 8**
- **ASP.NET**
- .NET Native (Windows 10)
- **ASP.NET** (Mac and Linux)

**.NET Core 3.1**
- Desktop
  - WPF
  - Windows Forms
- **Blazor**
  - Server side
  - Client side (to be released)

.NET Common
- Runtime
- Compilers
- NuGet

# .NET Application Architecture

**Browser Presentation Layer**

Blazor

Silverlight

**Presentation Layer**

ASP.NET MVC

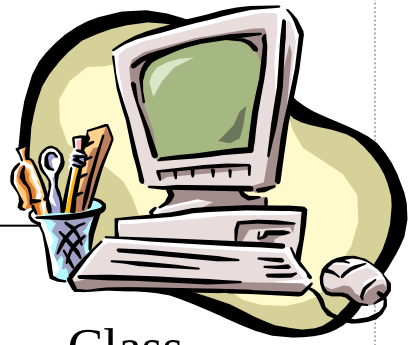
ASP.NET

WPF

WinForms

Windows Store App

Out of Browser Silverlight



Class Library

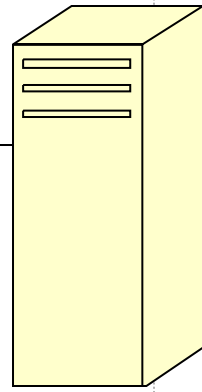
**Business Layer**

WCF

WEB API

LINQ

Class Library



**Data Access Layer**

WCF

WEB API

ADO.NET

Entity Framework

Class Library

# Introduction to OOP

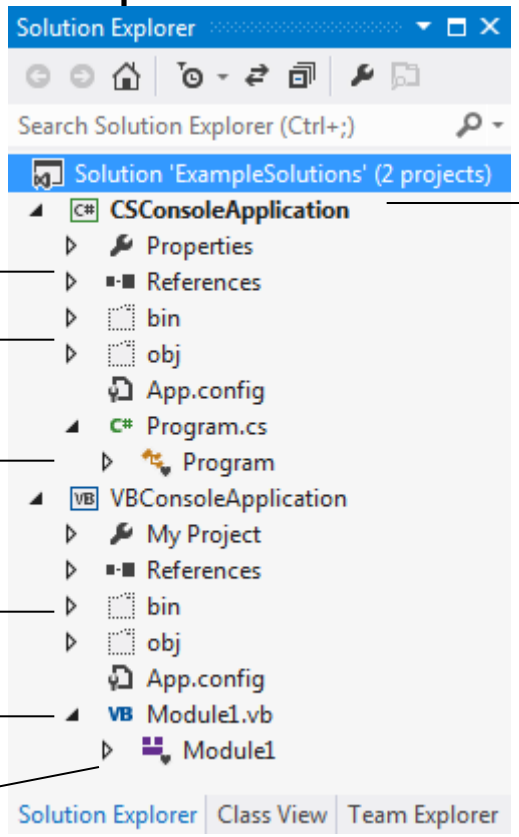
- The .NET languages support Object Oriented Programming
- .NET Framework Class Library makes extensive use of OOP
- Main principles of Object Oriented Programming
  - Data Abstraction
  - Encapsulation - data hiding
  - Inheritance
  - Polymorphism

# Visual Studio .NET

- Visual Studio .NET provides powerful features:
  - IntelliSense
    - list member options - searching now available in VS 2010
    - auto completion
    - parameter information
  - Many Wizard options
  - Syntax highlighting in source editor
- Additional Features:
  - Class Diagrams
  - Class Details
  - Surround with - allows surrounding selected code (C#)
  - Insert Snippet - allows inserting code

# Visual Studio .NET Solution Explorer and Class View

- Solution Explorer and Class View:




References to Assemblies

Output

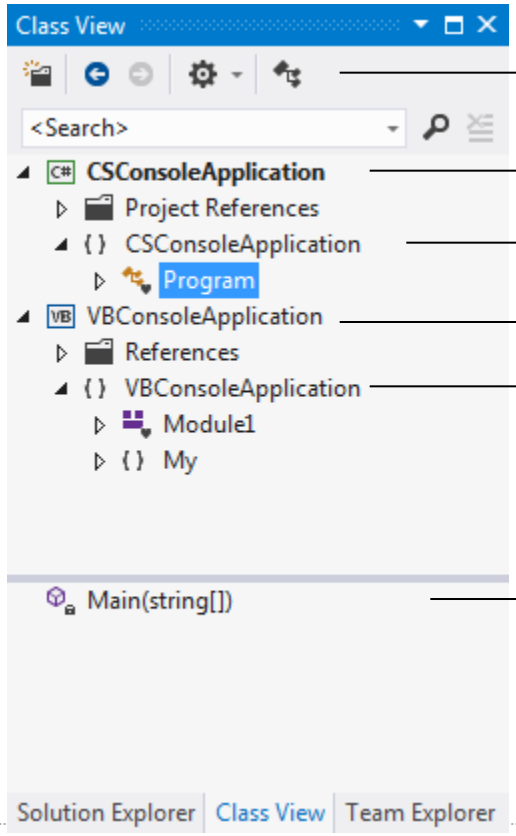
Source File

Output

Source File

Members

Project




ClassView Settings

Project

Namespace

Project

Namespace

Members

# Simple Program

```
using System;

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

Open **System** namespace

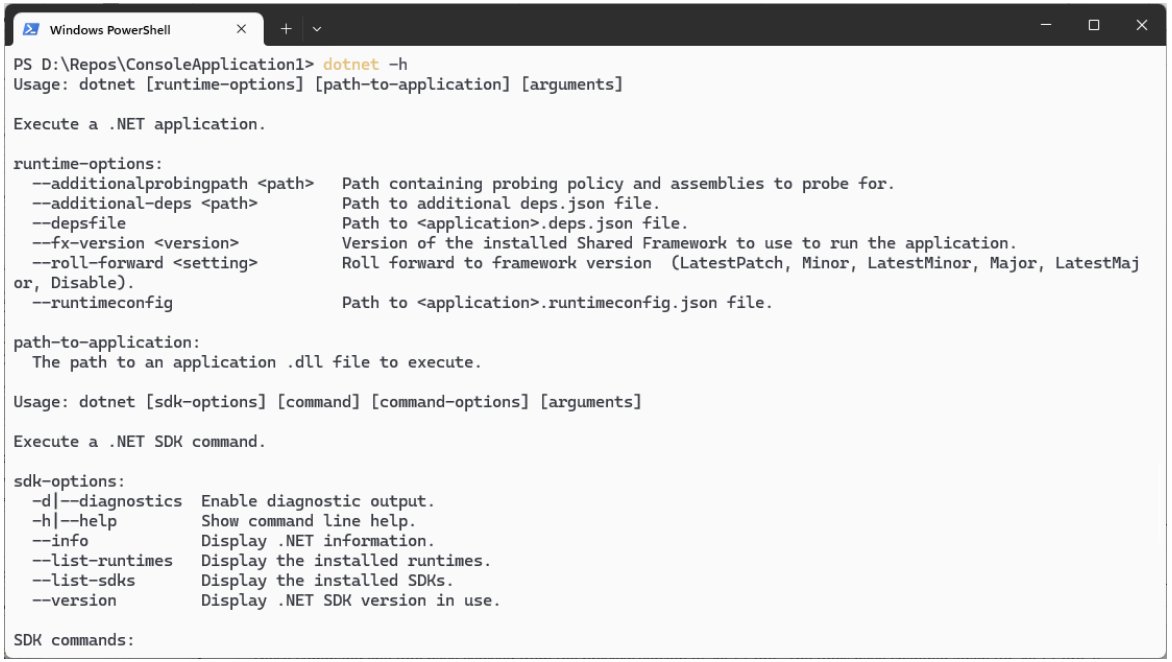
"Main" is the entry-point to the application

Name being used from the System namespace

Write string and append newline character

# DotNet Command Line

- Install .NET Core and then view help:




- Create project using:

>dotnet new console

Creates Console Project

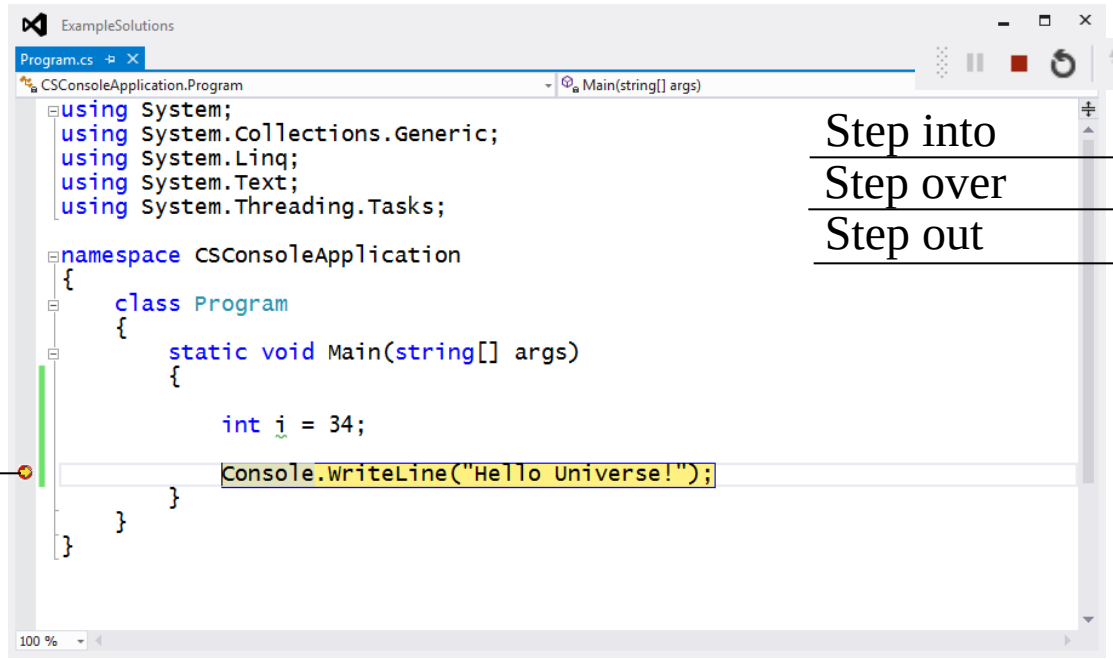
>dotnet new mvc --auth individual

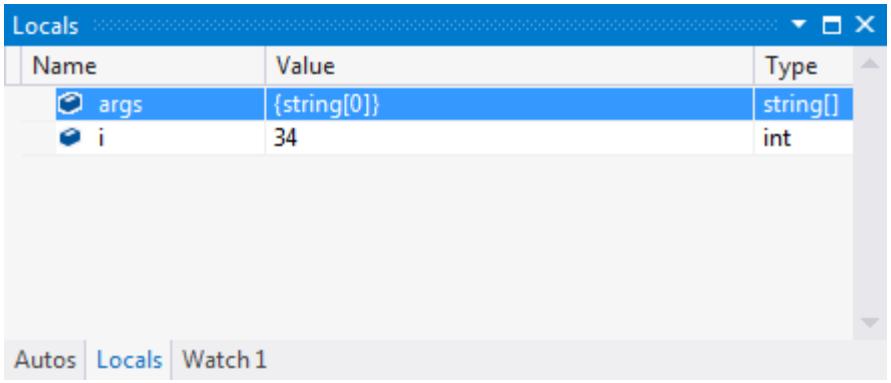
Creates MVC Project with Authentication

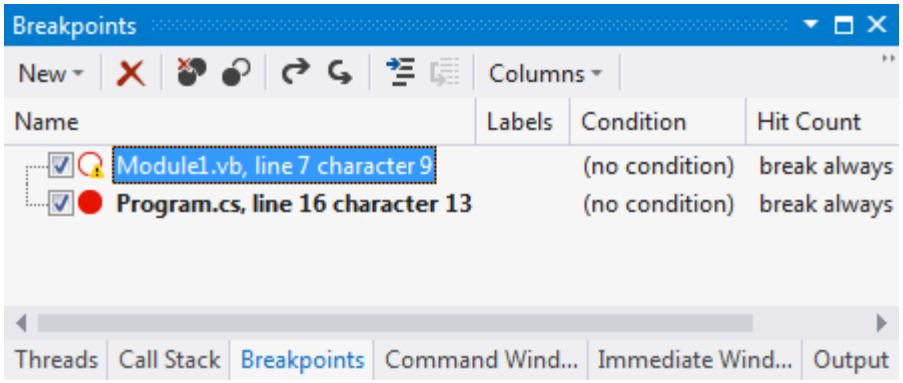
# Debugging

Step through code in Break mode and view variables

Breakpoint set by clicking

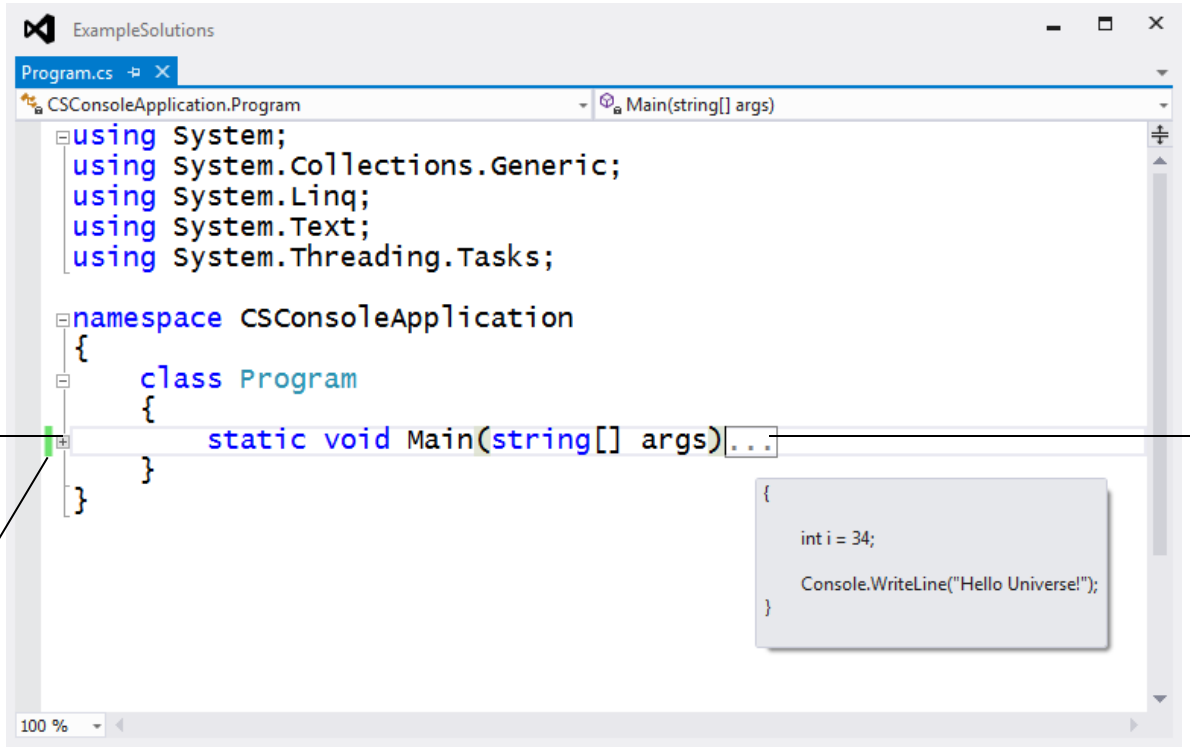


# Code Outlining

- Code outlining allows code to be folded away:




Click to fold and unfold

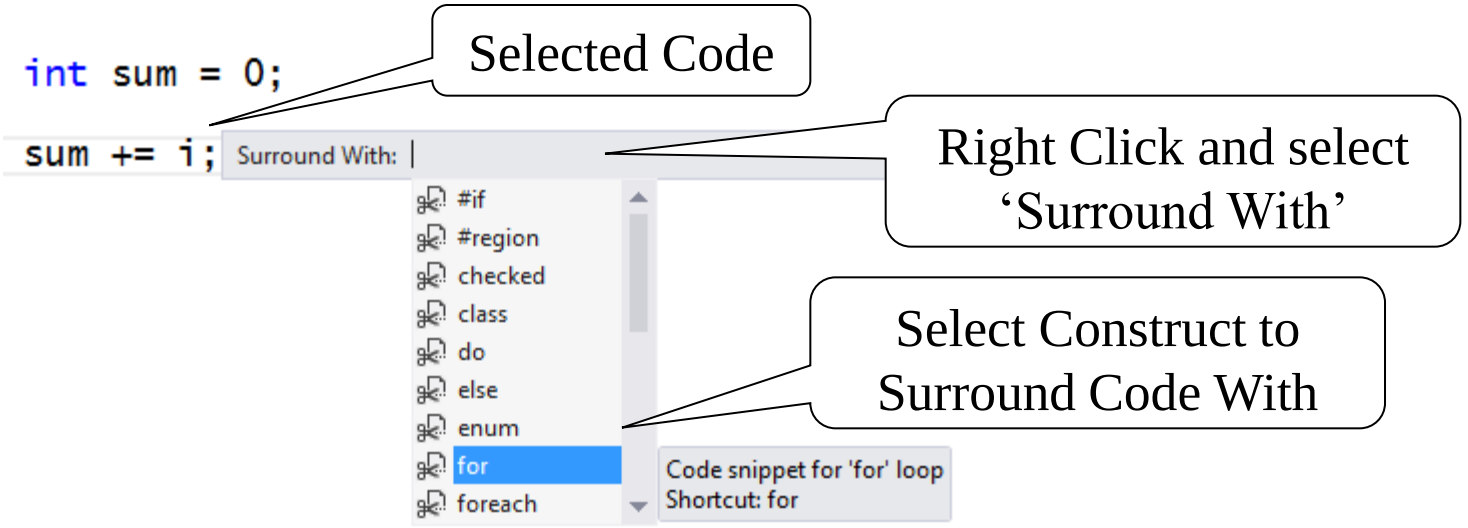
Colour code indicates edited lines

Hang cursor to view code

Yellow vertical line indicates edited line, whilst green saved to file.

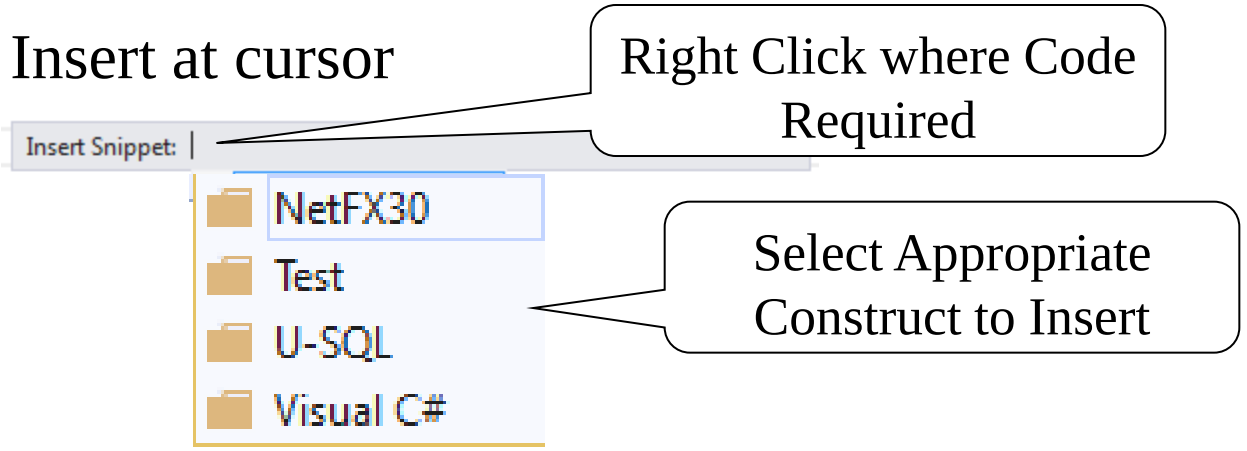
# Surround With Options

- Allows surrounding selected code with another code construct
  - Wide range of constructs can be used
    - Detailed later



# Code Snippet

- Allows predefined Code Snippet to be inserted
  - Wide range of code elements can be inserted
    - Insert at cursor

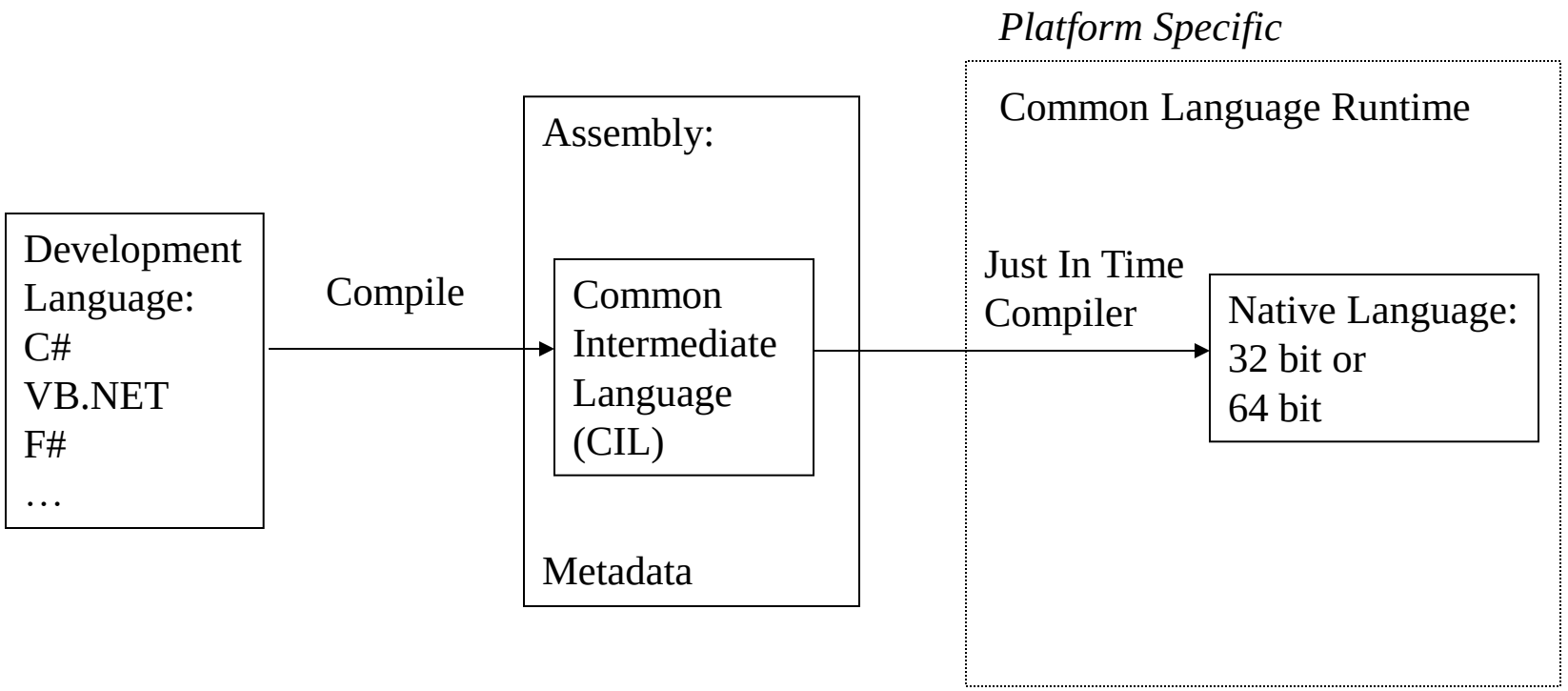



    - Type name of snippet and tab

# Visual Studio – Solution!

- Visual Studio .NET can associate a number of Projects into a *Solution*
  - Some previous versions used the term Workspace
- Most Projects compile to an Assembly
- Assemblies can be:
  - Application - .exe
  - Library - .dll

# .NET Code and Assemblies



# Output to the Console

- Output to the console is achieve by use of System.Console
- WriteLine() outputs string and appends newline character

```
System.Console.WriteLine( "Hello World")
```

- Write() outputs but does not append a newline character, thus the following is equivalent:

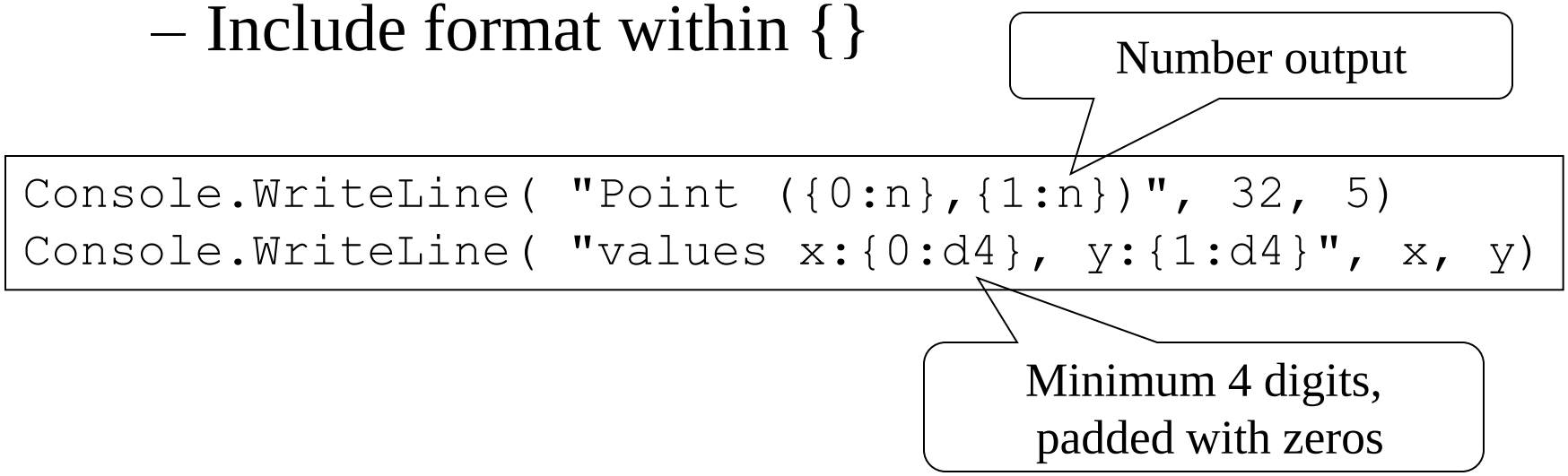
```
Console.Write( "Hello")
Console.WriteLine( " World")
```

- Values can be output using the following syntax:

```
Console.WriteLine( "Point ({0},{1})", 32, 5)
Console.WriteLine( "values x:{0}, y:{1}", x, y)
```

# Formatting Strings

- Output may also be formatted:
  - Include format within {}




  - Wide range of numeric format options
    - c, d, e, f, g, n, p, r and x

# String Interpolation

- C# 6 introduces a new way of formatting strings:

```
double x = 67.789, y = 12.511;

string info = $"{x:f2}, {y:f2}";

Console.WriteLine(info);
```

Variables or expressions can be formatted

```
info = $"{x} + {y} = {x + y}";

Console.WriteLine(info);
```

Simple expressions

# Input from the Console

- Input from the console is achieve by use of System.Console
- ReadLine() reads a string from the Console:

```
name = System.Console.ReadLine()
```

- ReadKey() reads a single character from the console and returns a ConsoleKeyInfo:

```
cki = System.Console.ReadKey( false)
```

Access character using KeyChar property

Intercept

# Namespace Usage

- Namespaces provide a partitioning of the “namespace”
- The same named type may be defined within a number of namespaces
  - Resolved by qualification with namespace
- Namespaces may be nested (produces a hierarchical naming)
- Each project has a Default Namespace (VB.NET - Root Namespace)
  - Defaults to project name
  - Accessed using project properties

# Defining Namespaces

- Single level Namespace

**Class access within a namespace is restricted to *public* and *internal* alternatives (default internal)**

```
namespace ACompany
{
    class AClass
    {
        ...
    }
}
```

- Nested Namespaces

```
namespace ACompany
{
    namespace ALibrary
    {
        class AClass
        {
            ...
        }
    }
}
```

**or**

```
namespace ACompany.ALibrary
{
    class AClass
    {
        ...
    }
}
```

# Using and Imports

- C# 6 introduces the ability to use static/Shared methods without explicit use of type:

```
using static System.Console;
```

```
public static void Main(){
{
        WriteLine( "Hello World!");
}
```

# Introduction - Summary

- This section covered:
  - Introduction to .NET
  - Main principles of OOP
  - Visual Studio .NET
  - Simple Program
  - Console IO
  - Namespaces

# Fundamentals

- This section covers:
  - Comments
  - Data types
  - Declaring Variables
  - Scoping
  - Initialising with Constants
  - Declaring a constant
  - Operators

# Comments

- Three comment styles are possible with C#

```
/* This is a 'C' style comment,
        which may run over
              many lines */
```

```
// Comment extends to the line end!!
// Preferred form!
```

Added Automatically

```
/// <summary>
///
/// </summary>
/// <param name="s"></param>
```

# C# and Case

- C# is a case sensitive language
- C# keywords are lower case
- Variable names, method names may be lower or upper case or mixed case
- By convention
  - variables start with a lower case letter and use camel casing:

```
size
```

```
aLongVariableName
```

  - types, properties and methods start with an upper case letter and use Pascal casing:

```
SomeProperty
```

# Primitive Data Types (Value Types)

There are keywords for basic types, which are synonyms for types defined within System:

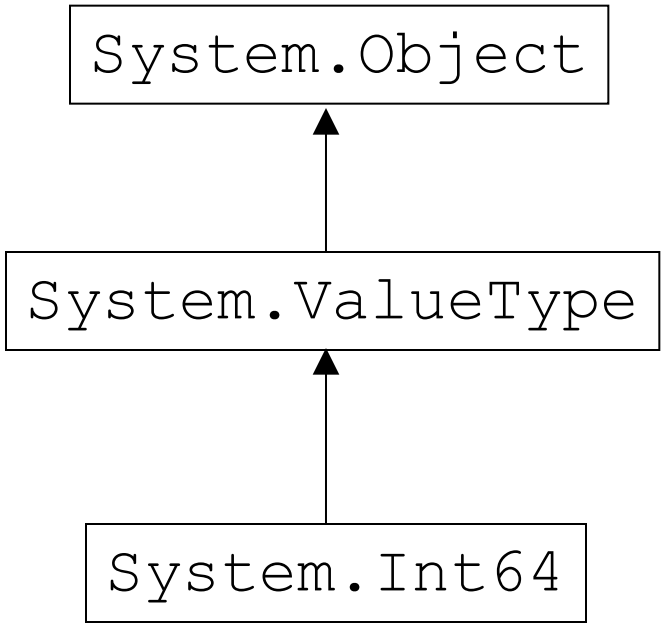
| C# | VB.NET | Full Name |
|---|---|---|
| char | Char | System.Char |
| short | Short | System.Int16 |
| int | Integer | System.Int32 |
| long | Long | System.Int64 |
| float | Single | System.Single |
| double | Double | System.Double |
| bool | Boolean | System.Boolean |

16 bit Unicode Character

- Sizes are platform independent

# Object based Class Library

- Within the .NET Framework every type is derived from System.Object




| C# | VB.NET | Full Name |
|---|---|---|
| object | Object | System.Object |
| string | String | System.String |

Note: *Nullable* types allow the use of Value Types like Reference types (see later…)

# Reference Types and Value Types

| | **Reference** | **Value** |
|---|---|---|
| **Advantage** | long lived | lightweight |
| **Memory Allocation** | CLR heap | threads stack or inline |
| **Allocated by** | new | declaration or new |
| **Deallocation** | garbage collection | stack unwinding or when contain object's memory freed |
| **User defined** | class | struct(ure) |
| **Examples** | StreamWriter | Int32<br>Boolean |

# System.Object Members

- The following methods are supported by Object:

| Methods | Descriptions |
|---|---|
| Equals() | For comparison of objects.<br>For reference types default implementation compares references.<br>Can be overridden in user defined classes to compare values. |
| GetHashCode() | Returns hash code for use in Collections.<br>For reference types default implementation returns a value based upon reference. Can be overridden in user defined classes. |
| GetType() | Returns Type object representing this runtime instance. |
| ToString() | Returns string representation of object.<br>For reference types default implementation returns name of type.<br>Can be overridden in user defined classes. |

# System.Int32

- System.Int32 is representative of the numeric types:
  - Constant fields:

| | | |
|---|---|---|
| MaxValue | - | Maximum value for this type |
| MinValue | - | Minimum value for this type |

  - Static method:

| | | |
|---|---|---|
| Parse() | - | Converts string to number |

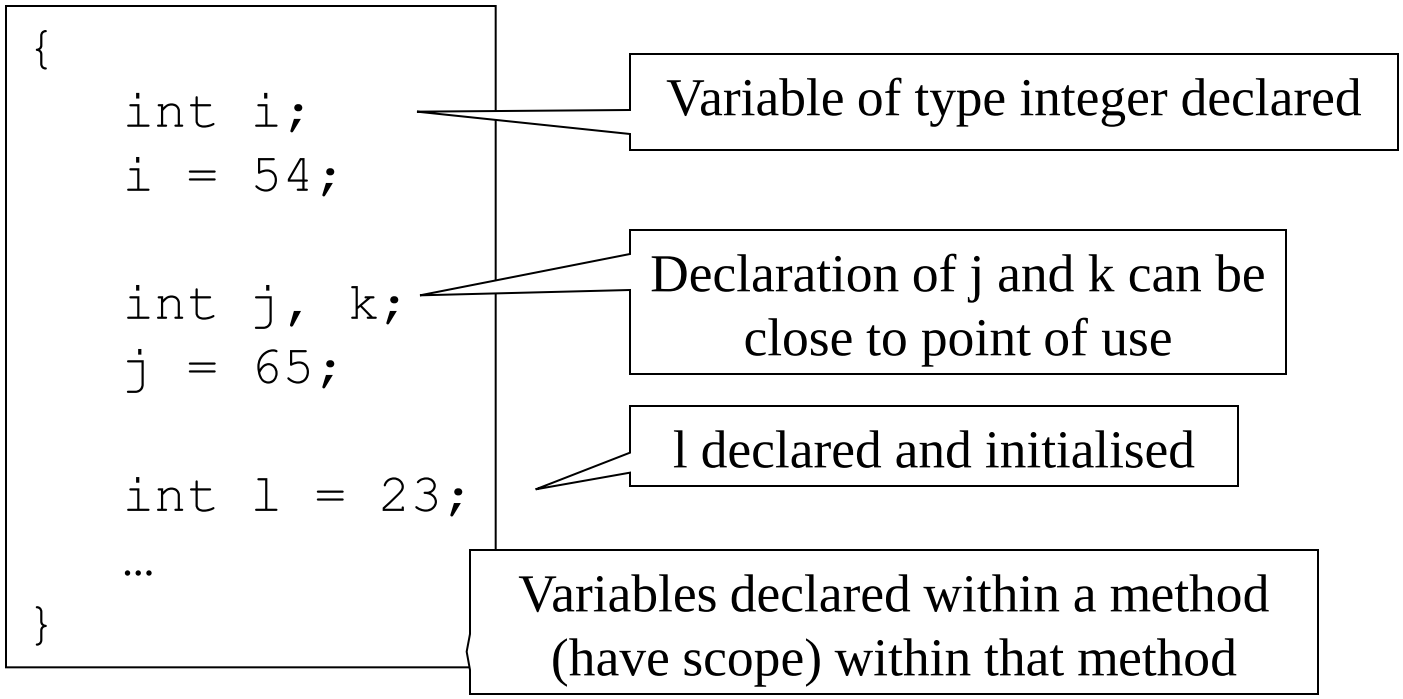
Could throw:
ArgumentNullException
FormatException
OverflowException

| | | |
|---|---|---|
| TryParse(...) | - | Converts string to number |

Returns Boolean indicating success or failure.

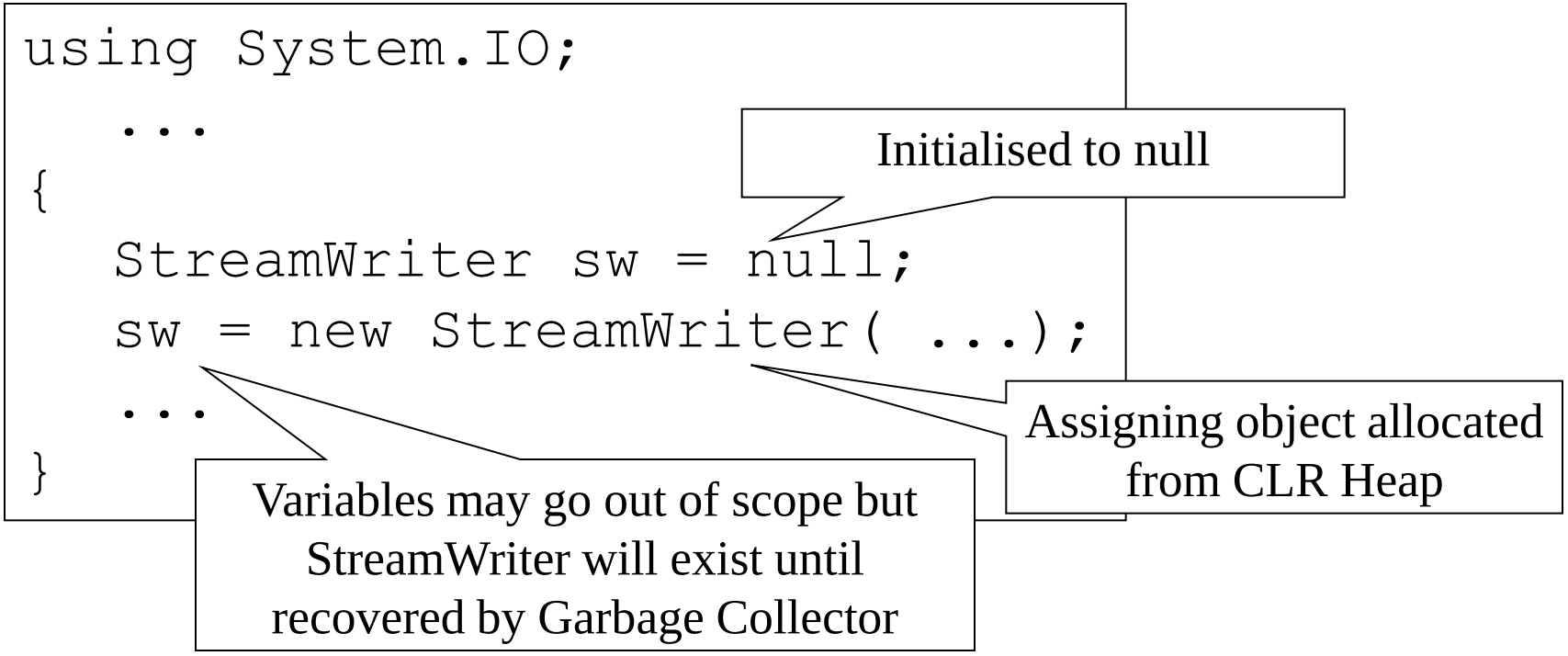
# Declaring variables (Value Types)

- Declarations are statements
  - Declaring a value type creates a value on the stack



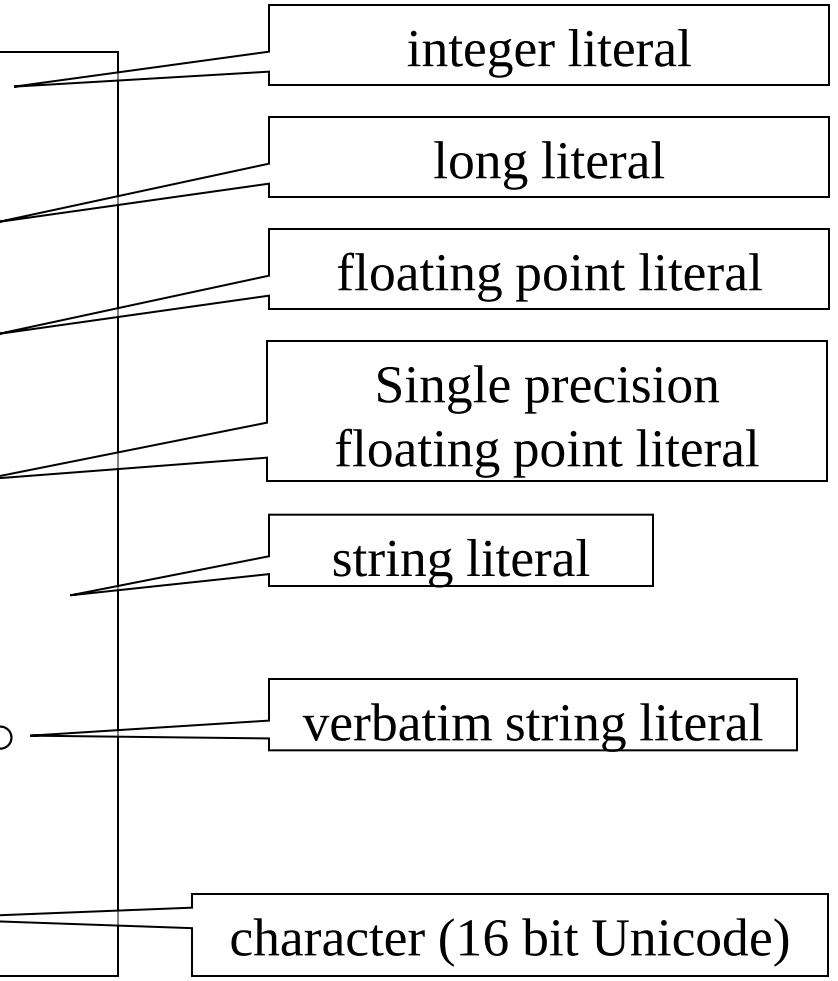
# Declaring variables (Reference Types)

- Declarations are statements
  - Declaring a reference type only allocates a reference



# Initialising with Literals

```
   int i = 43;

   long l = 534534L;

   double d = 543.4;

   float f = 543.4F;

   string name = "Jim";

   string greet = @"Hello
there";

   char ch = 'a';
```



# Declaring a Constant

- Where a constant value is required it is good practice to declare a constant (avoid magic numbers):

```
const int MAXVALUE = 100;
```

- Constants must be initialised
- Any attempt to change the value of the constant will result in a compiler error

# Casting and Type Conversion

- Some implicit conversions are not possible:

```
string s = "32";
int i;
//i = s;
```

Implicit conversion not allowed

- The following parsing would be necessary:

Parse Method

```
i = int.Parse( s, CultureInfo.CurrentCulture);
```

- Numeric conversion where data could be lost require the use of a cast:

```
long l = 5354;
int i;
i = (int) l;
```

Cast from long to integer

# Arithmetic Operators

- C# has a wide range of operators, including:
  + * / % - ++ --

```
int j = 34;

j = j + 1; //Alternatively    ++j or j++
```

Prefix increment operator returns value after increment

Postfix increment operator returns value before increment

- Mathematics functions are available on the System.Math class

# Assignment Operators

- There is assignment operator: =
- Also there are compound assignment operators including: += -= *= /=

```
int j = 34, k;

k = 4;  // Assignment

j += k; //Same as        j = j + k

j /= k; //Same as        j = j / k
```

# Fundamentals - Summary

- This section has covered, as follows:
  - Comments
  - Data types
  - Declaring Variables
  - Scoping
  - Initialising with Constants
  - Declaring a constant
  - Operators

# Methods

- This Section will give an overview of Methods and their usage
  - Introduction
  - Methods General Form
  - Passing by Reference
  - Uninitialised Reference
  - Method Overloading
  - Default Arguments
  - Boxing

# Methods - General Form

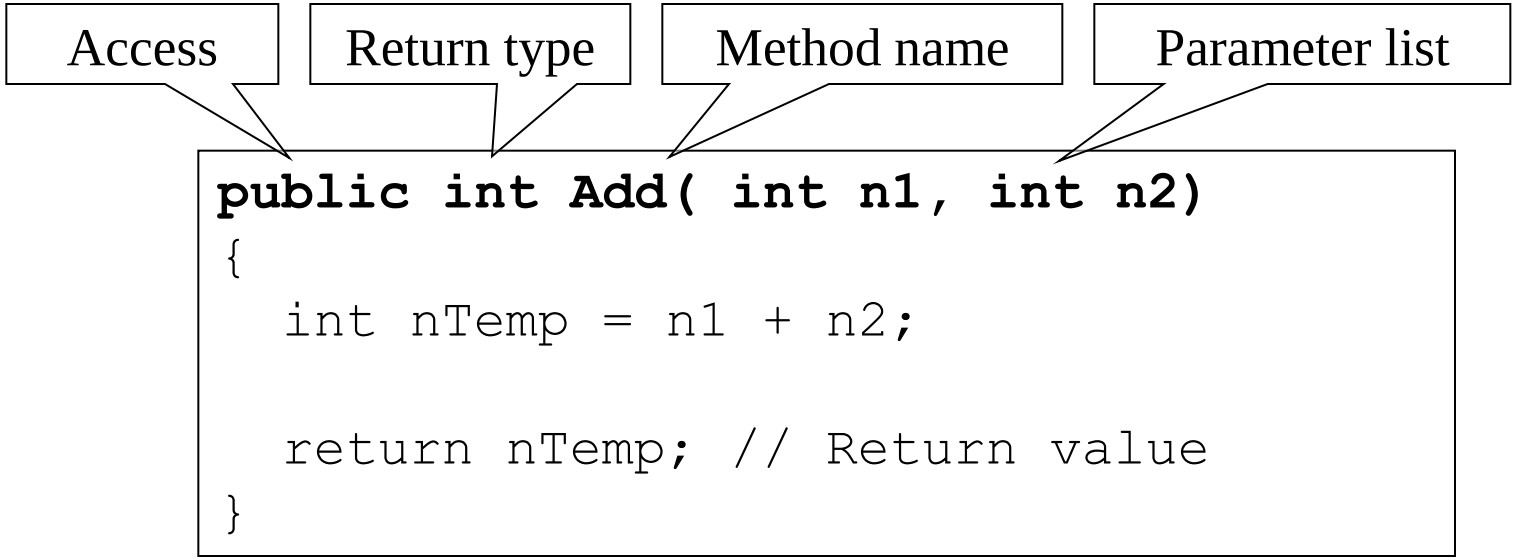
- Methods take the general form:

```
class AClass
{
  <access> <modifiers> <return type>
                    <method name> (<parameter list>)
  {
    // Implementation here
    return <return value>;
  }
}
```

Methods must be defined within a class or structure.
Methods can also be static (methods called on type).

# Methods – Example

- The following illustrates a method:



# Passing by Reference

```
int Calc( int val)
{
   return val*100;
}


void DoWork()
{
   int ans;

   ans = Calc( 5);
}
```

Value returned through return value

```
void Calc( int val,
      ref int passOut)
{
   passOut = val*100;
}

void DoWork()
{
   int ans = 0;

   Calc( 5, ref ans);
}
```

Variable requires initialising

Value returned through reference

# Uninitialised References

```
int Calc( int val)
{
   return val*100;
}


void DoWork()
{
   int ans;

   ans = Calc( 5);
}
```

Value returned through return value

```
void Calc( int val,
      out int passOut)
{
   passOut = val*100;
}

void DoWork()
{
   int ans;

   Calc( 5, out ans);
}
```

Variable does not require initialising

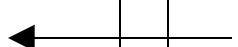
out guarantees initialisation of variable

# Method Overloading

- Many methods having the same name may be defined
- Must differ in number or type of parameters; not return type

```
double Max(double x,
   double y)
{
   return x < y ? y : x;
}
int Max(int x, int y)
{
   return x < y ? y : x;
}
```

```
void DoSomeWork()
{
   int i = 34, j = 54, k;

   k = Max( i, j);

   ...
}
```



# Default Arguments

```
double Power( double d, int p = 2)
{
   double result = 1.0;

   for( int i = 0; i < p; ++i)
       result *= d;

   return result;
}
void main()
{
   double val = 3.5, result;

   result = Power( val, 4);
   result = Power( val);
   result = Power( val, p:3)
   return 0;
}
```

Default argument

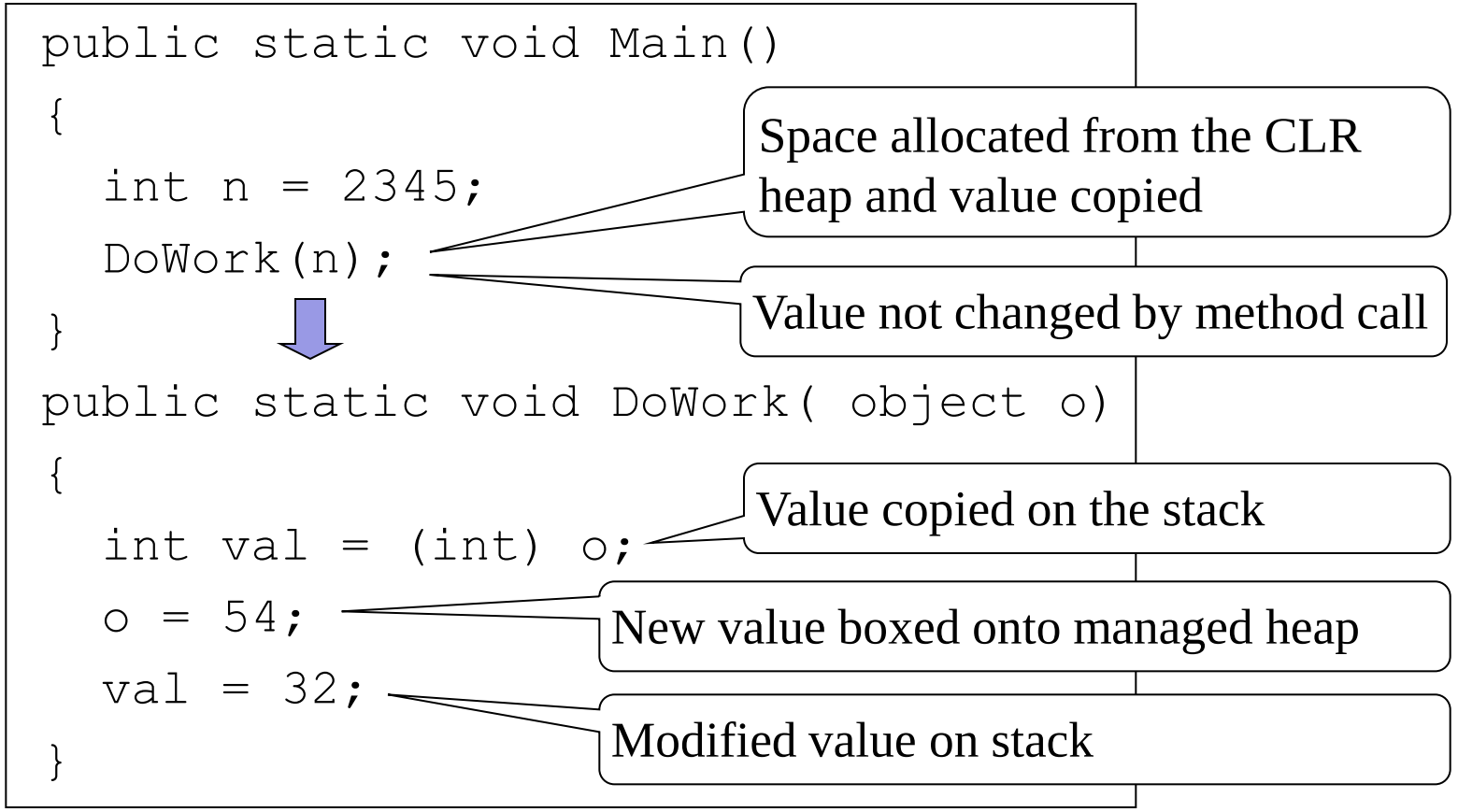
Use of default argument
second argument
defaults to 2

Named Argument

# Boxing!

- Value types are derived from System.Object
  - therefore can be assigned to object reference
- When a value is assigned to an object:
  - memory is allocated from the CLR heap
  - the value is copied
- When the object reference is assigned to a value variable of the appropriate type the value is again copied (Unboxed)!

# Boxing Illustrated



# Methods Summary

- This section has give an overview of Methods and their usage, as follows:
  - Methods General Form
  - Passing by Reference
  - Uninitialised Reference
  - Method Overloading
  - Default Arguments
  - Boxing

# Iteration and Flow of Control

- This section covers:
  - Relational Operators
  - Iteration
    - for and while
    - do while
    - for each
  - Operators
  - Flow of Control
  - Conditional expression
  - Nullable Types
  - Selection

# Relational Operators

- Relational operators may be used on primitive types:

```
{
   int j = 34, k = 54;
   bool result = false;

   result = j < k;
   …
}
```

```
Operators are:
>  greater than
<  less than
>= greater than or equal
<= less than or equal
== equal
!= not equal
```

# Iteration – for and while

- Iteration can be achieved in many ways:

```
int sum = 0;

for( int j = 0; j < 100; ++j)
{
  sum += j;
}
```

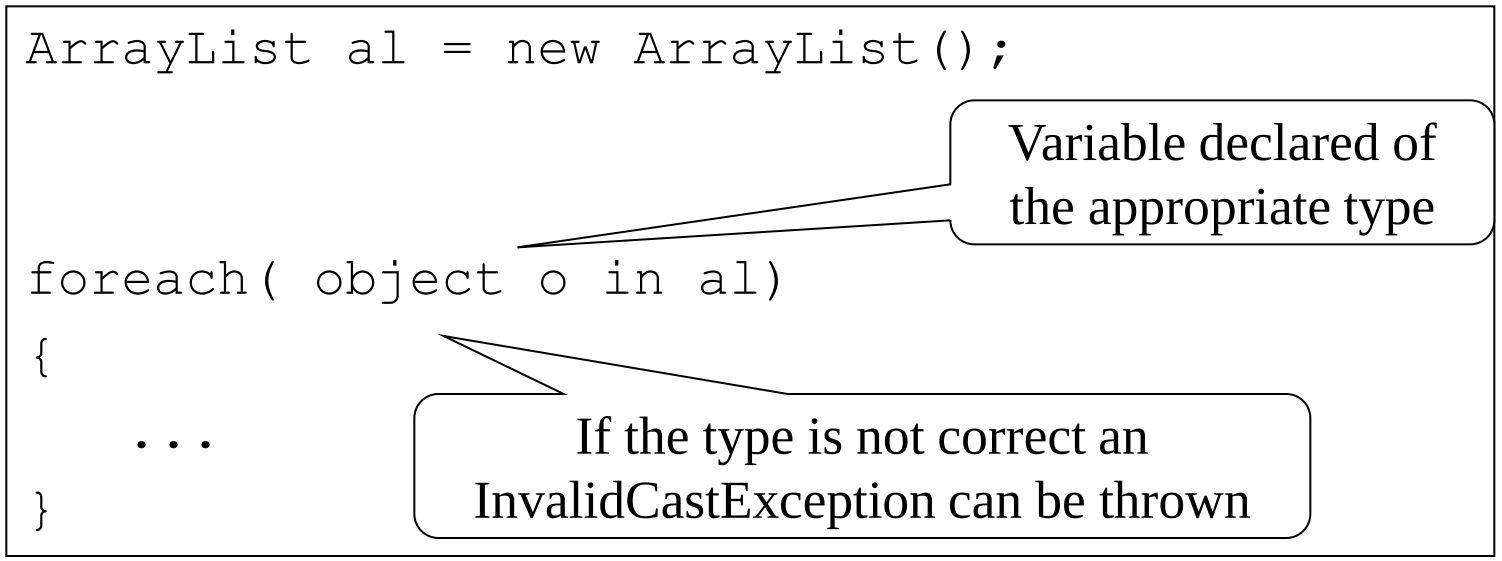
```
int sum = 0, j = 0;

while( j < 100)
{
  sum += j;
  ++j;
}
```

The above two approaches are equivalent and consist of initialisation, condition and increment.

```
bool found = false;
do
{
  found = find( …);

}while( !found);
```

# Iteration – For Each

- The **foreach** construct allows the iteration over collections implementing the IEnumerable interface:



# Logical Operators

- There are many other operators available:

| | Boolean | Bitwise |
|---|---|---|
| `and` | `&&` | `&` |
| `or` | `\|\|` | `\|` |
| `not` | `!` | `~` |
| `xor` | `^` | `^` |

- Sub-expressions may not be evaluated:

```
if( a < b || DoWork( c))
{
    ...
}
```

Not evaluated if ‘a < b’ is true!

# Comparisons

- Test for equality of references using == and inequality !=

```
TestType t1, t2;
   //...
if( t1 == t2){
   //...
}
```

- Test for the type of an object use **is**

```
object obj = new TestType();
   //...
if( obj is TestType){
   //...
}
```

# Conversions

- Before performing a casts it is important to determine the types are correct
- Using **is**:

```
object obj = new TestType();
   ...
if( obj is TestType)
{
   TestType tt = (TestType) obj;
}
```

Valid for reference and value types

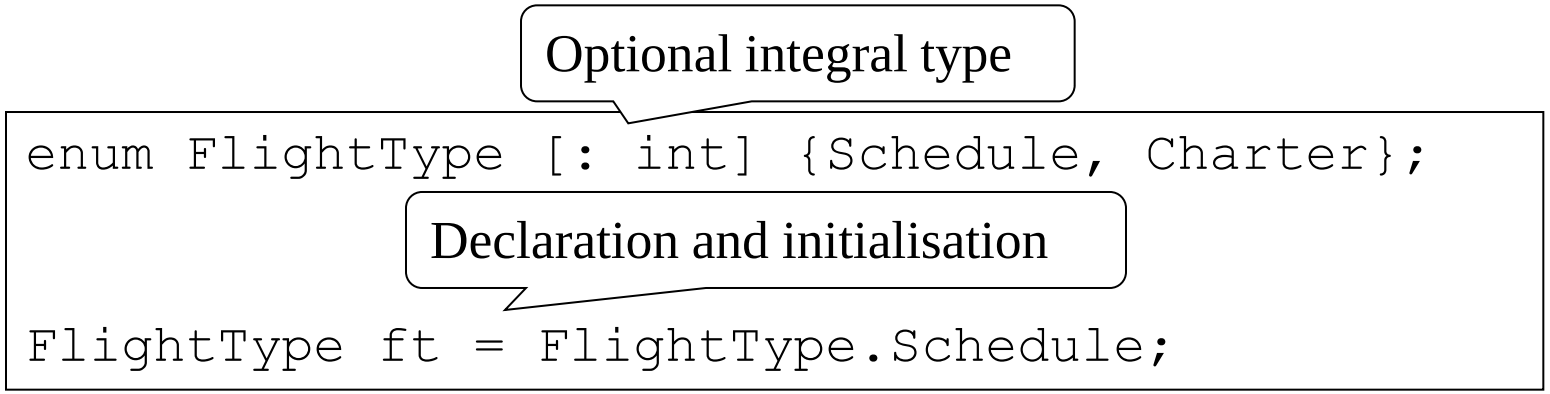
- Using **as**:

```
object obj = new TestType();
TestType tt = obj as TestType;
if( tt != null){
    ...
}
```

Valid for reference types

# Enumerated Data Types

- **enum**s are very useful user defined types where a limited set of values is required:




- These provide named constants

# Flow of control

- One means of controlling flow is with the use of the “if” statement:

Statement or block statement

else not required

```
FlightType ft = FlightType.Charter;
    ...
    // "==" not "="
if( ft == FlightType.Charter)
{
    Console.WriteLine( "Charter flight");
}
else
{
    Console.WriteLine( "Scheduled flight");
}
```

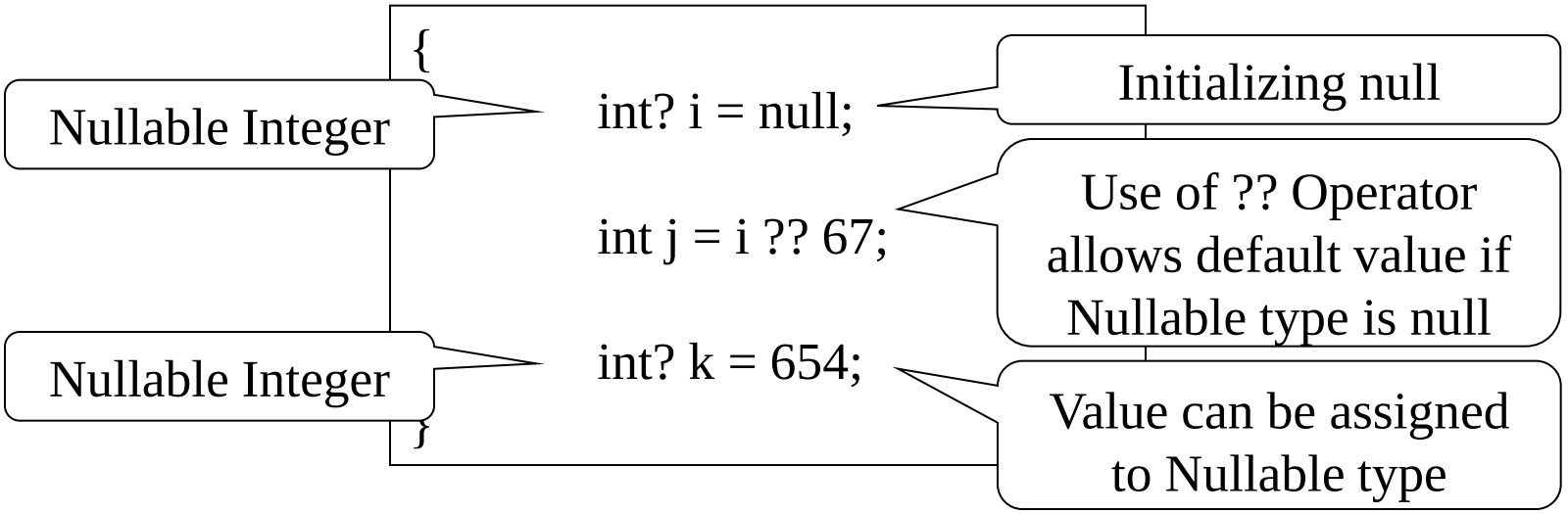
# Conditional Expression

- The only ternary operator ? :
- Useful where execution and return value is determined by some condition:

```
{
  int max, a = 43, b = 54;

  max = a < b ? b : a;
  // max == 54
}
```

# Nullable Types

- .NET Framework 2.0 supports Nullable types
- Nullable types allow for the ‘null’ value in addition to a value
- Value types can be used as Nullable types
- Declaration of Nullable types is achieved by adding ‘?’ to the value type:



# Nullable Reference Type

- Traditionally reference types can be null
- Why have nullable reference types?
  - Reference types can be null and therefore before use (call methods or access properties) should check for null
    - Otherwise possible NullReferenceException
- Enabling nullable context allows checking for use of ‘null’ reference variables
  - Thus requiring initialisation with non-null value

# Enabling Nullable References

- The ability enable ‘nullable context’ checking for possible null was available from .NET Core 3.1 projects
  - Options: Disable, Enable, Warnings and Annotations
- .NET 6.0 and later default to Enabled
  - Thus, requiring initialisation of reference or use of ‘nullable reference’

# Nullable Reference

- If nullable reference is enabled then the following usage of string will result in warnings:

```
ṣʧsîŋĝ ṇṣĝ   ŋụľľ

îŋʧ ľêŋ   ṇṣĝ Ľêŋĝʧḥ
```

- To prevent warnings (use with caution):

```
ṣʧsîŋĝ  ṇṣĝ   ŋụľľ

îŋʧ ľêŋ   ṇṣĝ  Ľêŋĝʧḥ
```

Null reference type

Null forgiving (suppressing) operator

Intended for use where ‘no’ possibility of null!

# switch statement

```
FlightType ft = FlightType.Charter;
...
switch( ft)   // Value for selection
{
  case FlightType.Charter: // Must be constant value
  {
    Console.WriteLine( "Charter flight");
    break;    // Prevents fall through
  }
  case FlightType.Schedule:
  {
    Console.WriteLine( "Schedule flight");
    break;
  }
  default:    // Optional - should we get here?
    break;
}
```

# Iteration and Flow of Control Summary

- This section has taken a look at:
  - Relational Operators
  - Iteration
    - for and while
    - do while
    - for each
  - Operators
  - Flow of Control
  - Conditional expression
  - Nullable Types
  - Selection

# Useful Types

- The following section introduced a few standard types:
  - System.String
  - System.Text.StringBuilder
  - Decimal
  - DateTime and TimeSpan

# string – System.String

- **string** is a standard reference type and may be used as follows (the value is immutable):

```
public static void Main()
{
  string firstName = "Fred", surName = "Jones";
  string fullName;
  fullName = firstName + " " + surName;

  string upper = fullName.ToUpper();
}
```

Operators cause creation of new temporary object

Upper case string returned, but original string unmodified

# String Methods

- String has a static/Shared method used to format a string
  - The following returns a string:

```
string.Format( "Point ({0},{1})", 32, 5)
```

- Various methods allow searching within a string for a string or character:
  - IndexOf( …) – various overloads
  - LastIndexOf( …) – various overloads
- A string can be divided into an array of strings:

```
string s = "Hello, there";
string[] parts = s.Split( ',');
```

# String Compare Method

- String has a number of overloads of the static/Shared method used to Compare strings
  - Returns a System.Int32 indicating relationship of strings:

| Comparison in relation to Strings | Return Value |
|---|---|
| Less than | < Zero |
| Equal | Zero |
| Greater than | > Zero |

```
String.Compare( s1,
      s2,
      false,
      CultureInfo.CurrentCulture)
```

Strings to be Compared

Ignore Case

Culture – System.Globalization

# System.Text.StringBuilder

- **StringBuilder** may be used to compose a string in a far more efficient way than using strings:

```
using System.Text;

...

public static void Main( string[] args)
{
  string firstName = "Fred", surName = "Jones";
  StringBuilder fullName = new StringBuilder();

  fullName.Append( firstName);
  fullName.Append( " ");
  fullName.Append( surName);
}
```

```
Useful Methods:
Insert()
Remove()
Replace()
```

# Decimal (Value Type)

- Large numbers and currency is often represented using Decimal
  - 96 digit with exponent 0 to 28
    - Range $((-2^{96})-(2^{96}))/10^{(0-28)}$
    - Many Static Method and fields

```
Decimal d1 = 342.2M;
Decimal d2 = new Decimal(1000000000.00);
Decimal d = d2 * d2;

Decimal one = Decimal.One + Decimal.Zero;
```

Use m/M to indicate literal Decimal

Constructor

Decimal Constants

# DateTime and TimeSpan

- Both date and time can be represented using DateTime

```
DateTime dt = DateTime.Now;

string date = dt.ToLongDateString();
string time = dt.ToLongTimeString();

Console.WriteLine(date);
Console.WriteLine(time);
string custom = dt.ToString("dd MMM yyyy");


TimeSpan ts = new TimeSpan(5, 10, 30);

dt += ts;
```

Current Date and Time

To Standard Date and Time formatted strings

Custom formatted strings

Hours, Minutes, Seconds

Difference between DateTimes

# Useful Types - Summary

- This section introduced a number of standard types:
  - System.String
  - System.Text.StringBuilder
  - Decimal
  - DateTime and TimeSpan

# Arrays

- This section covers:
  - Declaration of Arrays
  - Array Indexing
  - Iterating over an Array
  - Multi-dimensional Arrays
  - Copying Arrays
  - Range

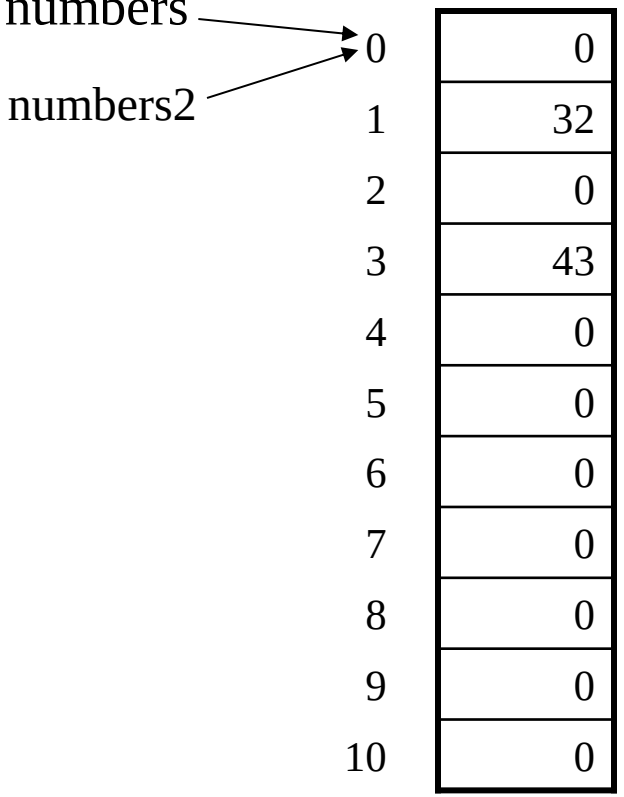
# Declaration of Arrays

- Arrays of Value Types are declared in a similar way, however arrays are reference type:

```
int[] numbers = new int[11];
int[] numbers2;
numbers[1] = 32;
numbers[3] = 43;
numbers2 = numbers;
```

**null**

Reference assigned




- Arrays are reference types
- Index starts at 0

# Initialising an Array

- An array can be initialised when declared (various forms are possible!):

```
{
    int[] numbers1 = { 23, 54, 65, 76};

    int[] numbers2 = new int[4]{ 23, 54, 65, 76};

    numbers1[3] = 34;
    int val = numbers1[3];
    int length = numbers1.Length;
}
```

Must be empty or match length of initialisation list

Length property

# Iterating over an Array

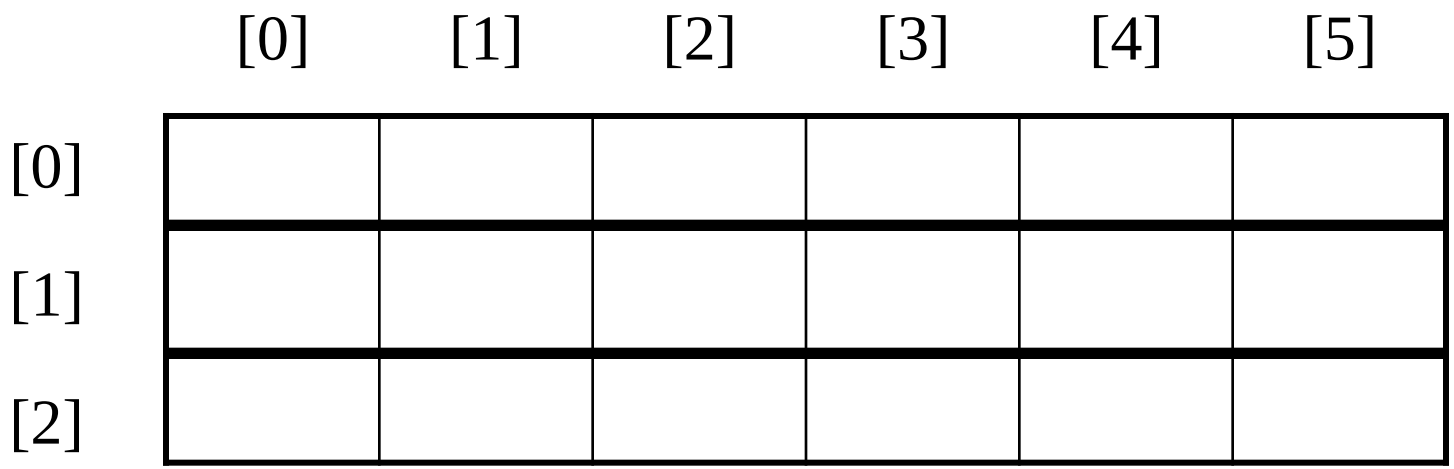
- ‘for each’ loops allow iteration across an array:

```
const int MAXNUMBERS = 7;
int[] numbers = new int[MAXNUMBERS];
int sum = 0;

foreach( int val in numbers)
{
     sum += val;
}
```

# Multi-dimensional Arrays

- A multidimensional array can be thought of as an array of arrays. One syntax allows defining regular arrays:

```
int[,] twoDim = new int[3,6];
```



# Copying Arrays

- Assigning Arrays only causes assigning of array references
- Copying of arrays can be performed using either Clone() or Array.Copy()

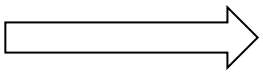
```
int[] arrSource = new int[] { 23, 3, 465, 46 };

int[] arrDest = (int[])arrSource.Clone();
```

```
int[] arrSource = new int[] { 23, 3, 465, 46 };
int[] arrDest = new int[10];

Array.Copy(arrSource, arrDest, arrSource.Length);
```

# Range (C#8)

- C# 8 introduces ranges, which allows referring to multiple items within an array:

```
{ // C#8
  var data = new int[]{1,2,3,4,5,6,7,8,9,10};
  var rng = 2..5;
  Console.WriteLine($"Range: {rng}");
  var data2 = data[rng];
  Console.Write($"Data: -");
  foreach (var item in data2)
  {
    Console.Write($" {item}");
  }
  Console.WriteLine();
}
```

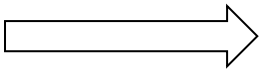


Data: - 3 4 5

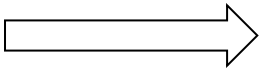
# Range (options)

- Ranges can be open ended:
  - ..5 – from the start to index 5
  - 2.. – from index 2 to the end

```
var data = new int[]{1,2,3,4,5,6,7,8,9,10};

int[] data2 = null;

data2 = data[..5];

data2 = data[2..];
```



Data: - 1 2 3 4 5



Data: - 3 4 5 6 7 8 9 10

# Arrays - Summary

- This section covered:
  - Declaration of Arrays
  - Array Indexing
  - Iterating over an Array
  - Multi-dimensional Arrays
  - Copying Arrays
  - Range

# Data Structure (Value Types)

- This section starts looking at Defining and using Structures, as follows:
  - Structures
  - Structure Implementation (Traditional)
  - Structure Implementation (New)

# Structures

- Whilst both Structures and Classes can be used as the templates for objects structures are intended to be ‘immutable’
  - Structures are not inherently immutable
  - Need to implement appropriately
- Ideal structure
  - Read only backing field
  - Define constructor to initialise fields
  - Property with getter for backing field

# Struct Implementation (Traditional)

```
enum FlightType { Ordinary, Silver}

struct Flight
{
        private readonly FlightType _flightType;
        private readonly string _flightNo;

        public FlightType FlightType { get { return _flightType; } }
        public string FlightNo { get { return _flightNo; } }
        public Flight(FlightType ft, string fn)
        {
            _flightType = ft;
            _flightNo = fn;
        }
}
```

Read only Fields

Read only Properties explicitly get backing fields

Must initialise all Fields

# Struct Implementation (C# 6)

```
enum FlightType { Ordinary, Silver}

struct Flight
{
        public FlightType FlightType { get; }
        public string FlightNo { get; }

        public Flight(FlightType ft, string fn)
        {
            FlightType = ft;
            FlightNo = fn;
        }
}
```

Auto Read Only Properties with backing Fields

Must initialise all Properties

# Data Structure - Summary

- This section covered:
  - Structures
  - Structure Implementation (Traditional)
  - Structure Implementation (New)

# Classes

- This section covers:
  - Defining classes
  - Class Diagram
  - Class Details
  - Refactoring
  - Properties
  - Indexers
  - Constructors
  - Constructor calls
  - Destructor
  - Self reference
  - Static/Shared members
  - Partial Class

# Defining Classes

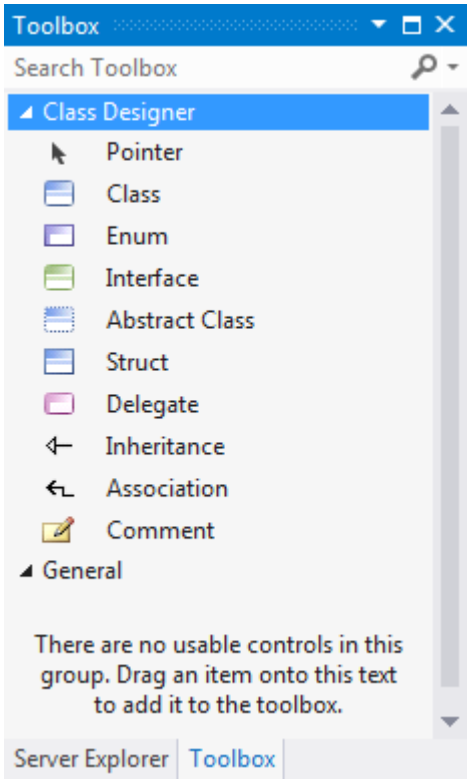
```
public class AClass
{
   int _a;

}

public static void Main()
{
    AClass ac = new AClass();


   \\ ac._a = 34;
   \\ int val = ac._a;
}
```

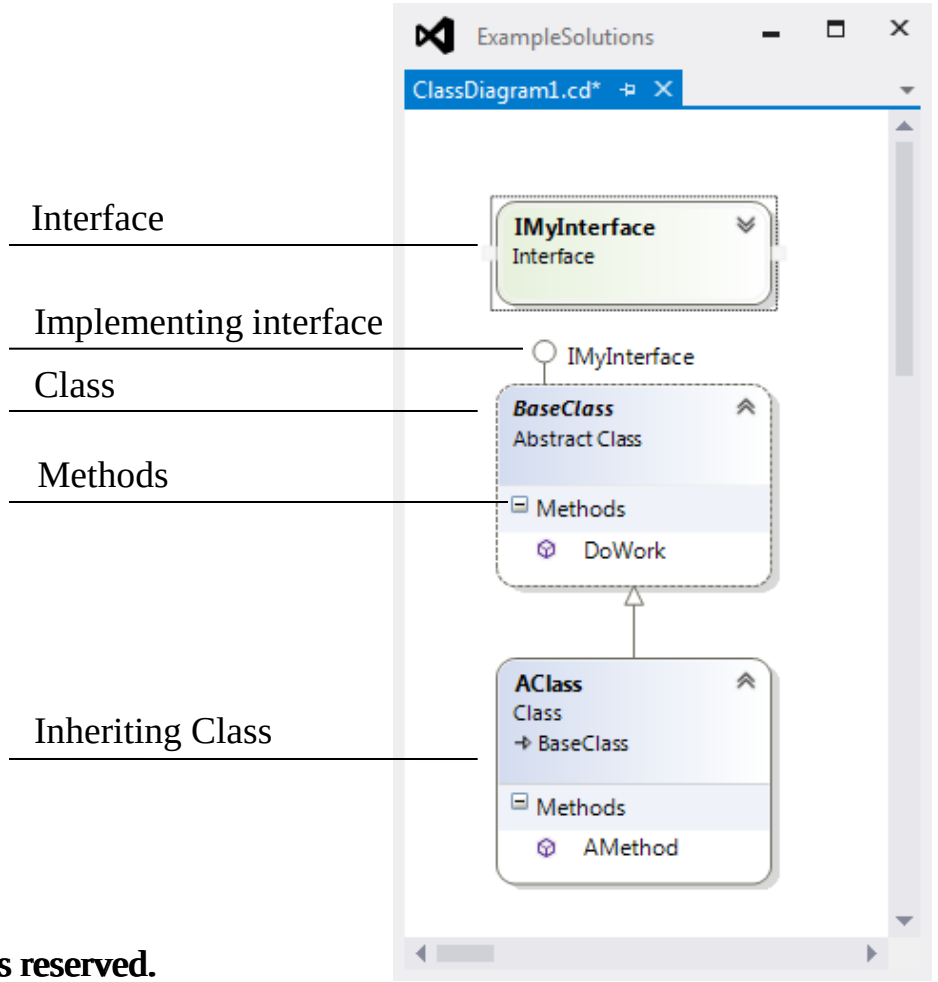
Default private in class

Allocated from the CLR heap

Not accessible

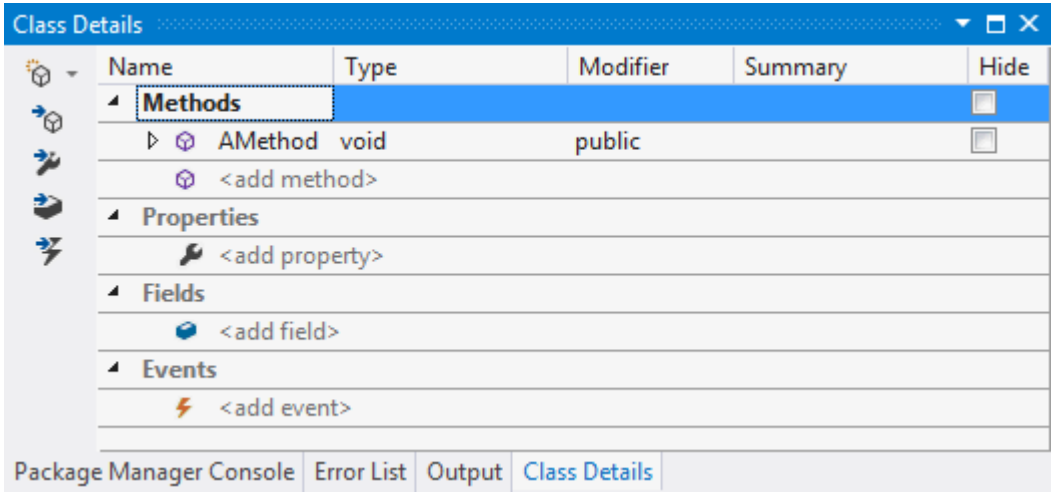
# Class Diagrams within VS.NET

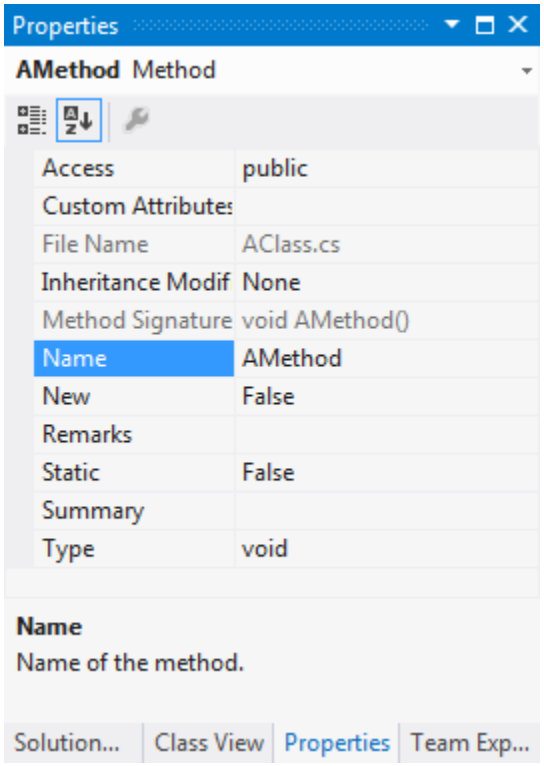
- Visual Studio .NET supports creation of Class Diagrams:
- 

# Class Diagrams – Class Details

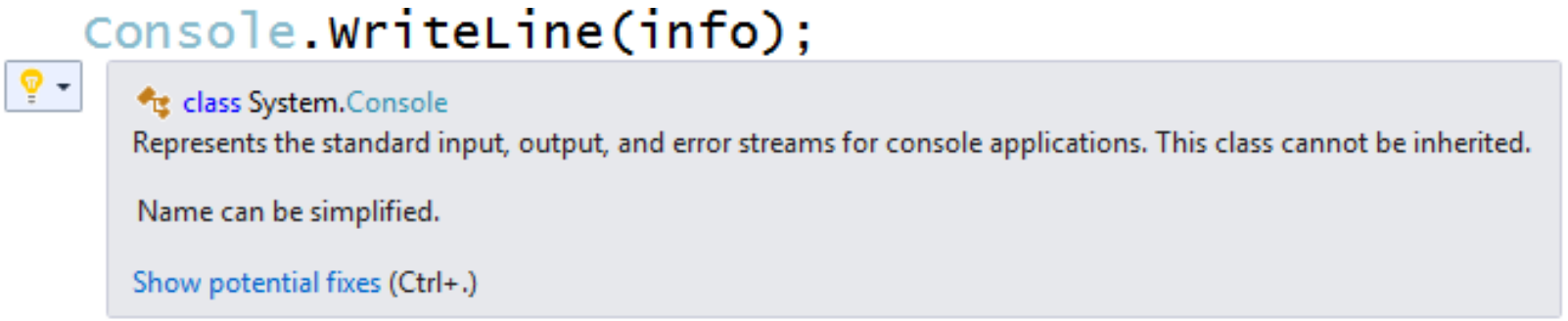
- Members of class can be modified using the Class Details window
  - Select type within Class Diagram:



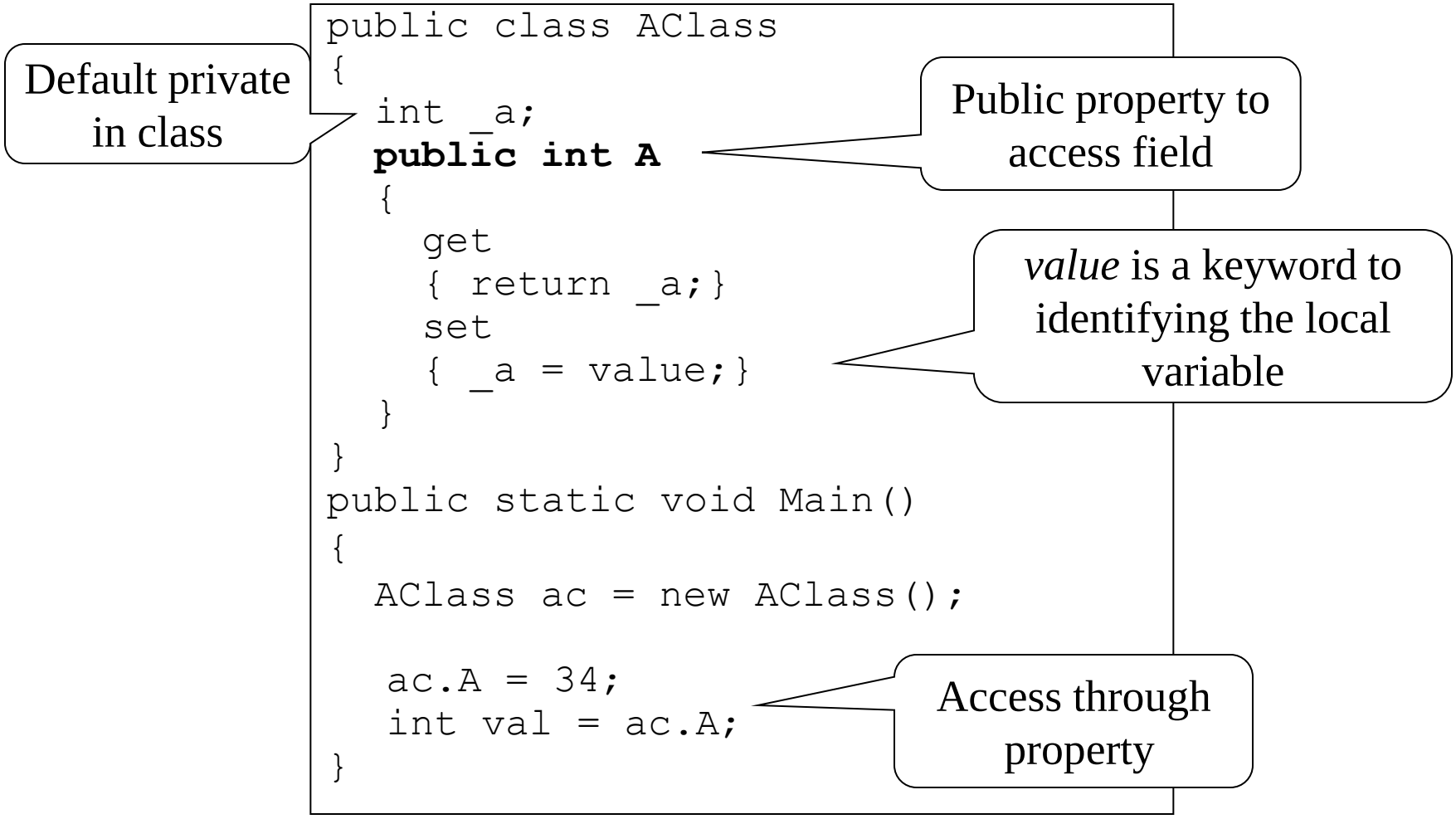



# Quick Actions (and Refactorings)

- Quick Actions (and Refactorings) provides tools for common actions:



# Properties



# Automatic and Immutable Properties

```
public class AClass
{
  public int A { get; set; }

}
```

Automatic Property

Backing Field provided with unknown name

```
public class AClass
{
  public int A { get;}

  public AClass(int a)
  {
    A = a;
  }
}
```

Immutable Property

Property can be set within Constructor

# Indexers

```
public class TestColl
{
  int[] _arr = new int[11];

  public int this[int ind]
  {
    get
    {
      return _arr[ind];
    }
    set
    {
      _arr[ind] = value;
    }
  }
}
```

*value* is a keyword to identifying the local variable

```
public static void Main()
{
  TestColl tc = null;

  tc = new TestColl();

  tc[2] = 543;

  int i = tc[2];
}
```

Access through indexer

# Constructors

```
public class AClass
{
  private int _a;
  private int _b;

  public AClass( int i)
  {
    _a = i;
    _b = 0;
  }

  public AClass( int i, int j)
  {
    _a = i;
    _b = j;
  }
}
```

No default constructor now constructor defined

```
public static void Main()
{
  AClass ac1;

  ac1 = new AClass( 43);

  AClass ac2;

  ac2 = new AClass(43,65);

}
```

# Initialisers (.NET 3.5)

```
public class AClass
{
  public int A { get; set;}
  public string B { get; set;}
}

...
public static void Main()
{
  AClass ac;

  ac = new AClass() { A = 23, B = "Hello"};

}
```

# Destructor

```
public class AClass
{
  private int _a;
  private int _b;

  ~AClass()
  {
  }
  public AClass( int i, int j)
  {
    _a = i;
    _b = j;
  }
  public AClass( int i): this( i, 0)
  {
  }
}
```

Only one destructor taking no parameters.

In C# the destructor is used to generate the Finalize() method. Therefore in C# if unmanaged resources need to be released a destructor should be defined but not a Finalize() method. However, see IDisposable!

# Null Condition Operator

- A classic problem is needing to check for null references. This can result in nested if statements:

```
var msg = default(string);
msg = "Hello";

if( msg != null)
{
  int ind = msg.IndexOf('e');
  char ch = msg[1];
}
```

Test for null

```
var msg = default(string);
msg = "Hello";

{
  int? ind = msg?.IndexOf('e');
  char? ch = msg?[1];
}
```

Null condition operator

Nullable type as result could be null!

# *this* - reference to object

```
class AClass
{
  private int _a;

  public AClass(int i)
  {
    _a = i;
  }

   . . .
  public void MethodChange( int i)
  {
    this._a = i;
  }
}
```

# Nesting of Types

- Types my be nested within other types:

```
namespace MyProject
{
  public class Outer
  {
    public class Nested
    {  ...
    }
  }
}
```

Classes and Structures can be nested

Accessed similar to using Namespaces

```
public static void Main()
{
  MyProject.Outer.Nested nc = new MyProject.Outer.Nested()
}
```

Fully qualified name

# Static Members

- Static fields:
  - only one instance of this data irrespective of the number of objects of the class created
- Static methods:
  - do not have a reference to an object
    - *this*
  - called on the type (except when used within defining type)
  - can be used for accessing static/Shared fields

# Static Members – example

```
public class ACounter
{
  private static int _count = 0;

  public ACounter(){ ++_count;}

  public static int GetCount() { return _count;}
}
...
static void Main()
{
  int count = ACounter.GetCount();    // count set to 0
  ACounter ac = new ACounter();   // Create one ACounter
  count = ACounter.GetCount();    // count set to 1
}
```

static field

# Static Constructors

```
public class AClass
{
  private const int MAX_SQUARES = 1000;
  private static int[] squares = new int[MAX_SQUARES];
  private int _a, _b;
  static AClass()
  {
    for( int i = 0; i < MAX_SQUARES; ++i)
    {
      squares[i] = i*i;
    }
  }
  public AClass( int i, int j)
  {
    _a = i;
    _b = j;
  }
}
```

Static Constructor – No Parameters

Ordinary Constructor

# Partial Class Definition

- A class can have its definition split across multiple file (.NET Framework 2.0):

```
// Part of class in file aclassp1.cs
public partial class AClass
{
  public void AMethod1(){ }
}
```

```
// Part of class in file aclassp2.cs
public partial class AClass
{
  public void AMethod2(){ }
}
```

Files within same project produce one class

# Classes - Summary

- This section covered:
  - Defining classes
  - Class Diagram
  - Class Details
  - Properties
  - Indexers
  - Constructors
  - Constructor calls
  - Destructor
  - Self reference
  - Static/Shared members
  - Partial Class

# Collections

- This section covers:
  - Collections
  - Generic Collections
  - List
  - Dictionary

# System.Collections

- The System.Collections namespace contains a number of standard collections
- Many collections can hold any type in the form System.Object
  - Only mechanism with .NET Framework 1.0 and 1.1
- Collections Available:
  - ArrayList - Dynamically sized array
  - BitArray - Dynamically sized array of Boolean
  - Hashtable - Dictionary collection of key/values
  - Queue - First In/First Out (FIFO) collection
  - SortedList - Sorted list!
  - Stack - Last In/First Out (LIFO) collection

# System.Collections.Generic

- The .NET Framework 2.0 introduces a number of Generic classes
  - These include Collections:
    - List - list collection
    - LinkedList - doubly linked list collection
    - Dictionary - stores key/value pairs
    - Queue - first in/first out collection
    - Stack - last in/first out collection
- Many of these collections are used in a similar way to the none generic collections
- Generic collections are type safe (avoids need for casting)

# List

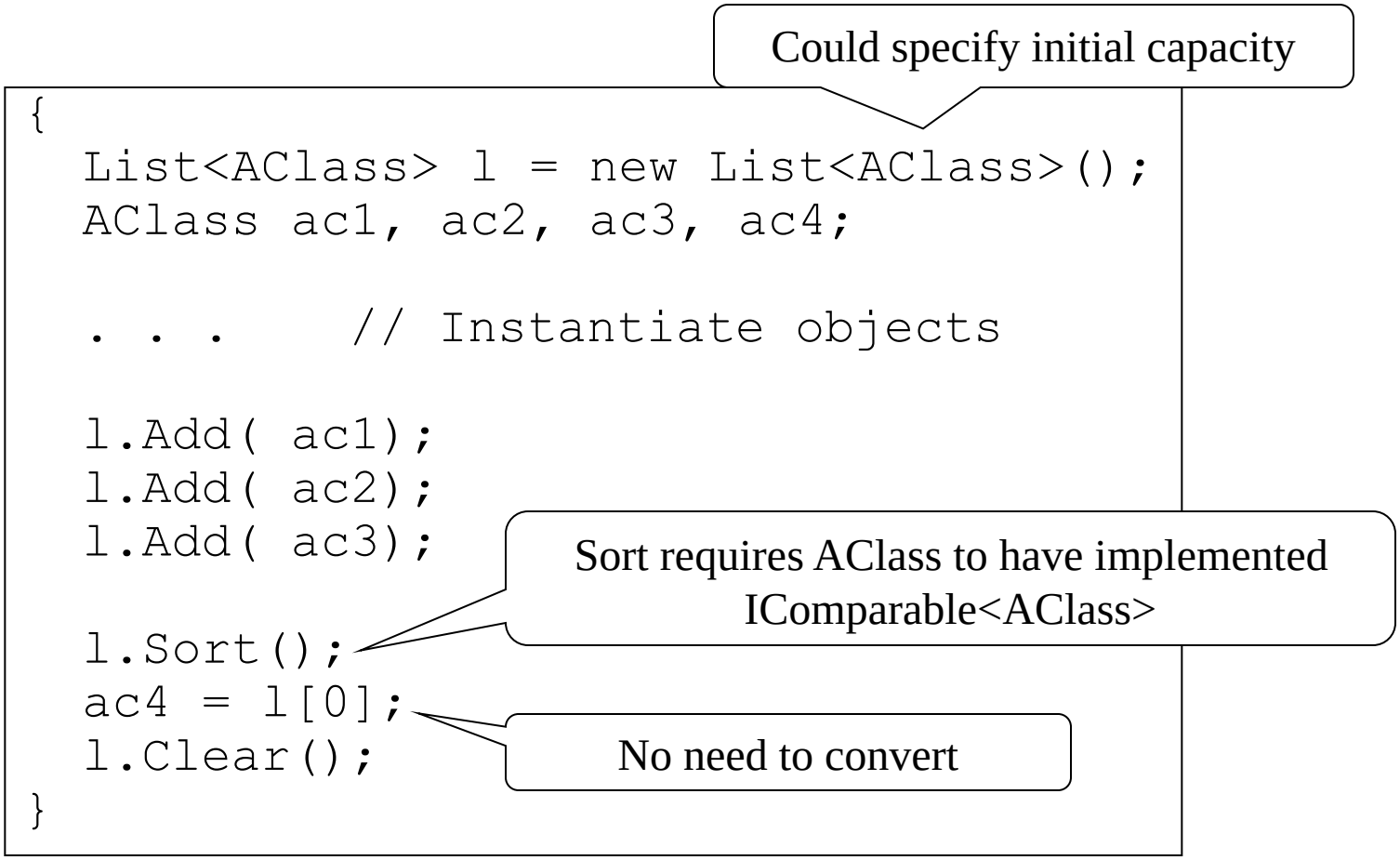
- Similar to a dynamically sized array
- Properties

| | | |
|---|---|---|
| Capacity | - | Size of container |
| Count | - | Number of elements |

- Methods:

| | | |
|---|---|---|
| Add() | - | Add element |
| AddRange() | - | Adds contents of another collection |
| BinarySearch() | - | Search for element using CompareTo() |
| Clear() | - | Removes all elements |
| Contains() | - | Returns Boolean indicating if element present |
| IndexOf() | - | Returns the index of an item |
| Insert() | - | Insert element at index |
| Remove() | - | Removes element |
| RemoveAt() | - | Remove element at index |
| Sort() | - | Sorts elements using CompareTo() or a Comparer |

# List Example



# Dictionary Collection

- The Dictionary stores values keyed by the hash value associated with a key
  - GetHashCode() Method is used to evaluate the key

```
AClass ac1 = new AClass();
AClass ac2 = new AClass();
Dictionary<AClass, string> d;

d = new Dictionary<AClass, string>();

d.Add(ac1, "Hello");
d.Add(ac2, "there");
if( d.ContainsKey(ac2))
{
  string s = d[ac2];
}
```

Optional: Initial capacity

Key

Value

Only one entry can be added for a key

Check for key before indexing

# C# 6 Dictionary Initialisation

- C# 6 has introduced the syntax allowing initialisation of Dictionary collections:

```
Dictionary<int, int> dict =
          new Dictionary<int, int>()
   {
   [3] = 54,
   [2] = 65,
   [5] = 7
   };

foreach (var item in dict)
{
  Console.WriteLine(
          $"Key {item.Key}, Value {item.Value}");
}
```

Initialise with key values pairs

# Collections - Summary

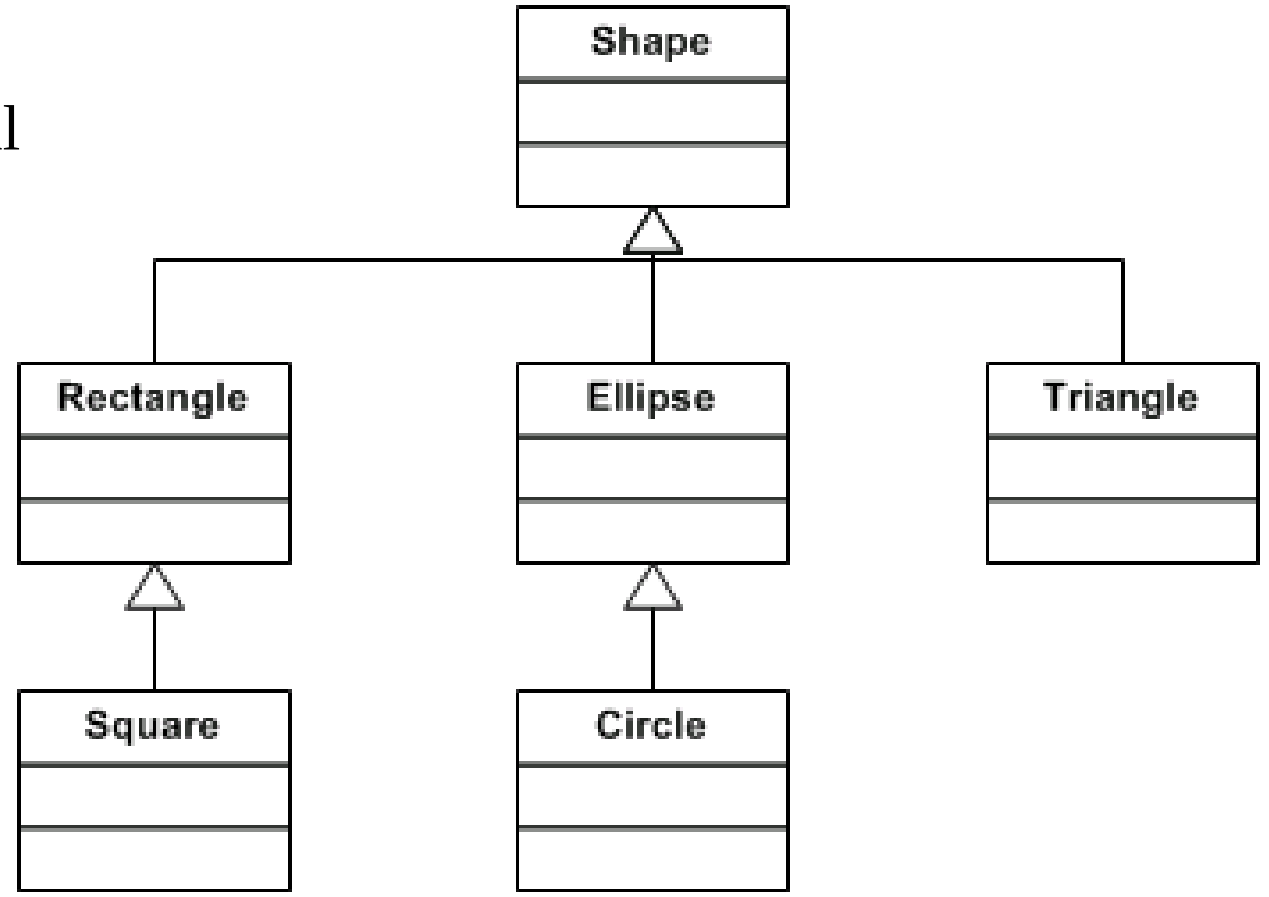
- This section covered:
  - Collections
  - Generic Collections
  - List
  - Dictionary

# Inheritance and Polymorphism

- This section covers:
  - Inheritance
  - Shape classes
  - Polymorphism
  - Constructors
  - Protected
  - Modifiers

# Inheritance

More General




More Specialised

# Shape class

```
public struct Point
{
   public double X {get;}
   public double Y {get;}
   public Point( double xx, double yy)
   {
     X = xx;
     Y = yy;
   }
}
public class Shape
{
   private Point _pos;
   public Shape( Point p) { _pos = p;}
   public virtual void Draw() { ...}
   ...
}
```

Point structure used to represent position

Virtual method can be overridden within inheriting classes

# Rectangle class – Polymorphism

```
public class Rectangle: Shape
{
  private double _w, _h; // Width and Height
  public Rectangle( Point p, double w, double h)
     :base( p)
  { _w = w; _h = h; }
  public override void Draw() { base.Draw(); ...}
}
public class Class1
{
  public static void Main()
  {
    Shape sh;
    Point p = new Point( 10.0, 10.0);
    sh = new Rectangle( p, 2.3, 43.3);
    sh.Draw();
  }
}
```

Initialisation of base class required

Base class method can be called

Which Draw method? Shape?

# Constructors and Base Classes

```
class A
{
  private int _a;

  public A()
  {
    _a = 0;
  }

  public A( int i)
  {
    _a = i;
  }
}
```

```
class B: A
{
  private int _b;

  public B()
  {
    _b = 0;
  }
  public B( int i):base()
  {
    _b = i;
  }
  public B( int i, int j):base(j)
  {
    _b = i;
  }
}
```

Implicit call to A()

No access to _a

Explicit call to A()

Explicit call to A(int)

# Shapes?

```
public static void Main( string[] args)
{
  shape sh;

  sh = new Shape( new Point( 10.0, 10.0));

  sh.Draw();
}
```

Should it be possible to create a Shape object?

What does this code mean?

# Abstract Class

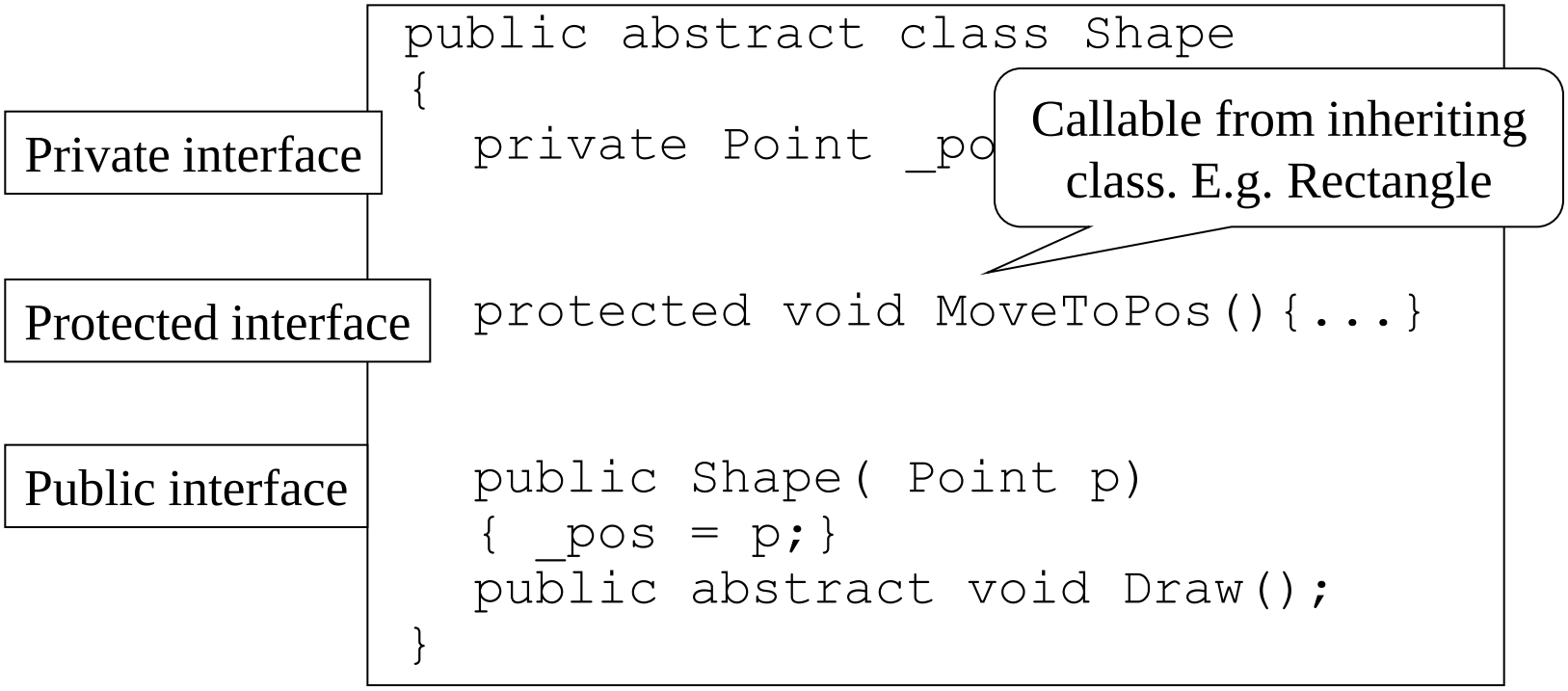
```
public abstract class Shape
{
  private Point _pos;
  public Shape( Point p) {_pos = p;}
  ...
  public abstract void Draw();
}
public class Class1
{
  public static void Main( string[] args)
  {
    Shape sh;
    Point p = new Point( 10.0, 10.0);
    sh = new Shape( p);
    sh.Draw();
    ...
  }
}
```

Method must be defined in derived (inheriting) classes

Cannot create object of abstract class

# Protected

- Protected members are only visible in the current class and its methods and derived classes and their methods.



# Method Modifiers

| C# Keyword | VB.NET Keyword | Description |
|---|---|---|
| virtual | Overridable | Method can be overridden in derived class. |
| sealed | NotOverridable | Override method cannot be overridden in derived class |
| new | Shadows | New / Shadowed method within base class is not available within the derived class |
| override | Overrides | Indicates method overrides method declared as virtual / Overridable in base class which has same parameter list and return type. That is, correct method will be called dependent upon type referenced. |
| abstract | MustOverride | Method must be implemented in the derived class. Abstract / MustOverride method cannot itself have an implementation. |

# Class Modifiers

| C# Keyword | VB.NET Keyword | Description |
|---|---|---|
| sealed | NotInheritable | Class cannot be used as a base class |
| abstract | MustInherit | Class must be inherited to create concrete class |
| static | Not applicable | All methods must be static |

- E.g.

```
abstract class AbBase
{
        …
}
```

```
static class Utilities
{
        …
}
```

# Inheritance and Polymorphism - Summary

- This section covered:
  - Inheritance
  - Shape classes
  - Polymorphism
  - Constructors
  - Protected
  - Modifiers

# Exception Handling

- This section looks at Exception handling and its use, as follows:
  - Exception Handling
  - Try- Finally
  - Errors as Exceptions
  - Exception members

# Exception Handling

- .NET uses the termination model of exception handling.
- When a problem occurs and an exception is thrown, the flow of execution is terminated
- The stack is unwound back to the nearest/latest handler for that exception type

```
public void DoWork()
{
  try
  {
    int.Parse( "Fred");
  }
  catch( FormatException ex)
  {
    Console.WriteLine( "Exception: {0}", ex.Message);
  }
}
```

Exception fo ‘not an integer’, will be thrown

# Try Finally

- When resources are being allocated it is necessary to tidy up when they are no longer require (disposed)
- This tidying needs to occur whether or not an exception occurred
- The finally block can be used to perform this process

```
public void DoWork()
{
  try
  {
    // Allocate resources and use
  }
  catch( FormatException ex)
  {
    // Handle any exceptions
  }
  finally
  {
    // Tidy resources
  }
}
```

Multiple catch blocks can appear

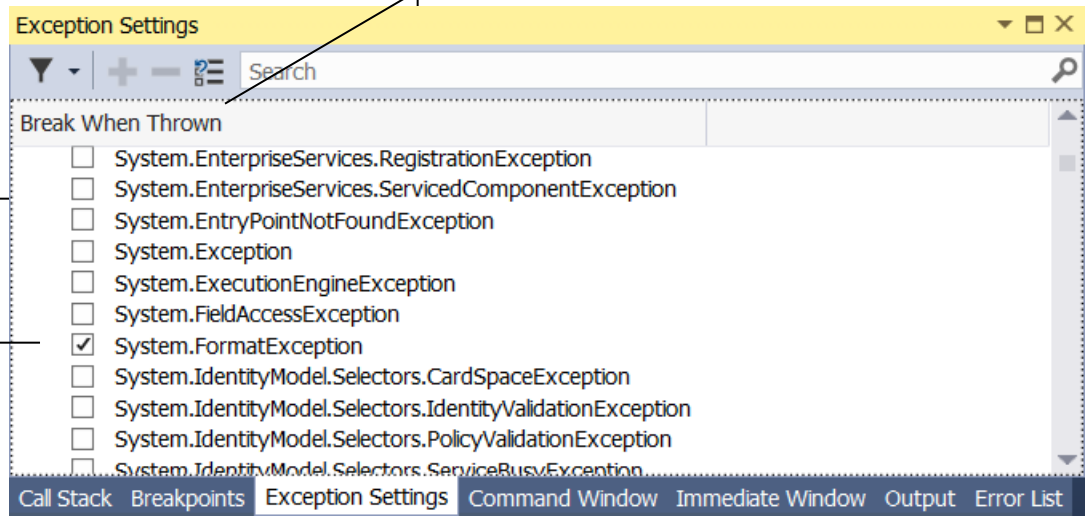
# Errors are handled as Exceptions

- The base class of all exceptions is **System.Exception**

```
try
{
  throw new MyException( "Problem");
}
catch( MyException ex)
{
  // Handle exception
}
```

Break When Thrown

Check Those to Break On



# Exception Filter

- Exception Filters can be used in a number of ways:

```
try
{
    using (var sw = new StreamWriter("C:/Temp2/data.txt"))
    {
        sw.WriteLine("Greetings");
    }
}
catch (Exception ex) when (LogIt(ex)){ }
catch (IOException ex)
{
    Console.WriteLine($"Problem: {ex?.Message}");
}
```

Function or express returning bool

Catch block entered if correct exception type and expression true!

# Operation Errors

- In C# the default for numeric operations is for values to wrap around:

```
int val = int.MaxValue;
int ans = val + val;
```

-2 !

- This behaviour can be changed in two ways
  - For the whole project, using project properties
    - Build tab – Advanced – ‘Check for arithmetic overflow/underflow’
  - Alternatively, use checked blocks:

```
int ans = 0
int val = int.MaxValue;

checked
{
  ans = val + val;
}
```

# Exception Members

- Useful properties:

```
Message     -     description of error
Source      -     Assembly name
StackTrace  -     location at which error occurred
TargetSite  -     originating method
InnerException -  reference to inner exception
```

# Exception Handling - Summary

- This section looked at Exception handling and its use, as follows:
  - Exception Handling
  - Try- Finally
  - Errors as Exceptions
  - Exception members

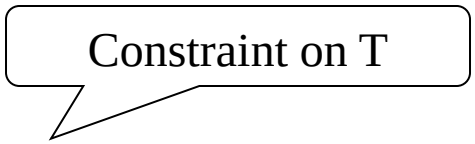
# Generics

- This section covers:
  - Writing Generic Code
  - Constraints on Generics
  - Writing Generic Classes

# Writing Generic Code

- Generic Methods and Types are instantiated using other Types
- Below is an example of a Generic max method:

```
namespace Utilities
{
  public class Utils
  {
    public static T Max<T> (T t1, T t2) where T:IComparable<T>
    {
      return t1.CompareTo( t2) > 0 ? t1 : t2;
    }
  }
}
```

# Constraints on Generics

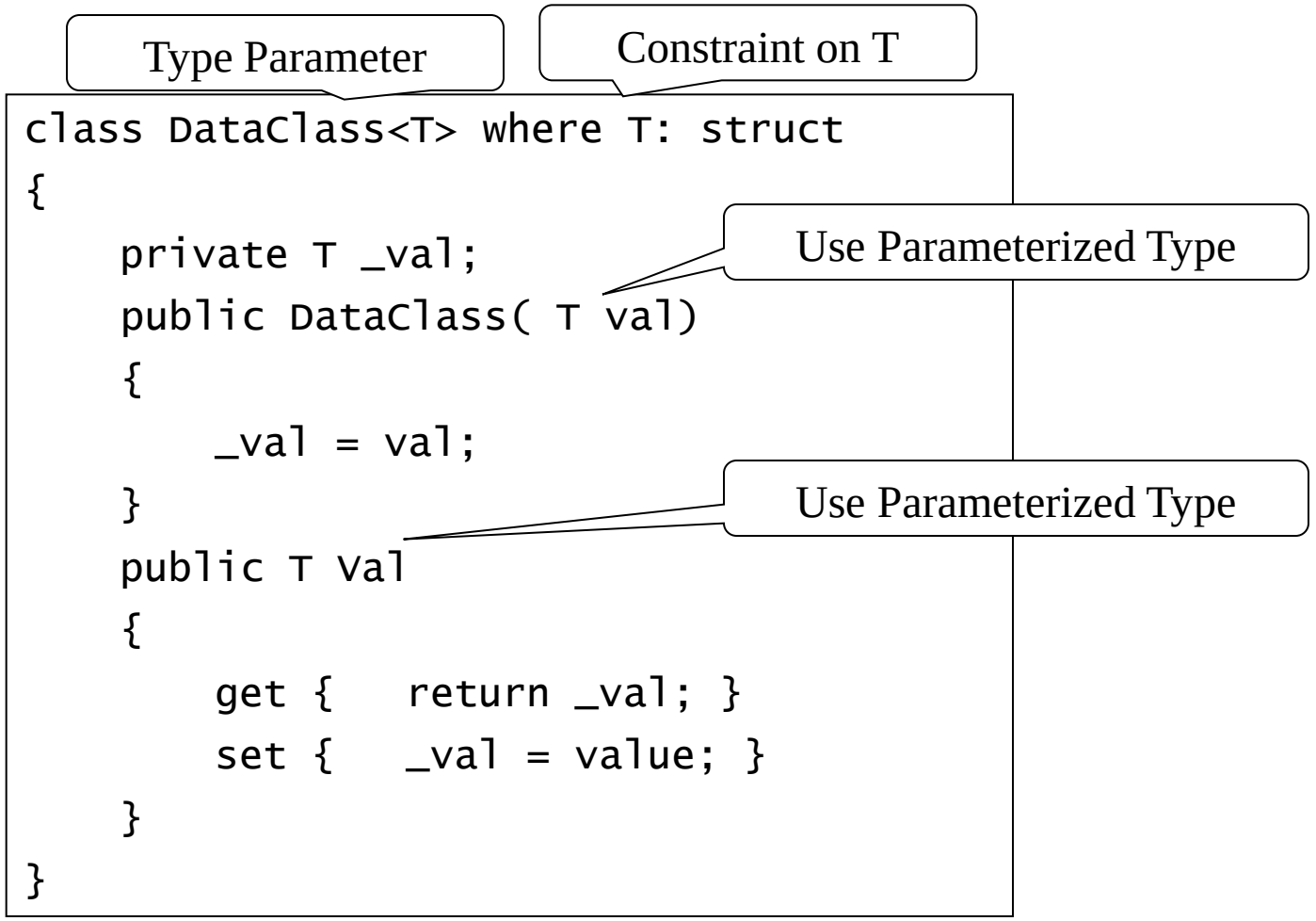
- To allow methods to be called on arguments of the Type Parameter the argument use must satisfies some constraints
  - The max method illustrate showed the need for IComparable<> to be implemented

| **Constraint** | **Description** |
|---|---|
| class | Reference type argument |
| struct(ure) | Value type argument |
| new | Argument must supply parameterless constructor |
| X | T must be of the type X or inherit from X |
| IX | T must be implement interface IX |

- Constraint clause can contain comma separated list of constraints

# Writing Generic Classes

- Generic Generic classes need to be defined with care!




```
class DataClass<T> where T: struct
{
    private T _val;
    public DataClass( T val)
    {
        _val = val;
    }
    public T Val
    {
        get {   return _val; }
        set {   _val = value; }
    }
}
```

# Generics - Summary

- This section covered:
  - Writing Generic Code
  - Constraints on Generics
  - Writing Generic Classes

# Interfaces and Generic Interfaces

- This section covers:
  - Inheritance
  - Interfaces
  - Generic Interfaces
  - Defining Interfaces
  - Defining Generic Interfaces
  - ICloneable
  - IComparable (Generic)
  - IComparer (Generic)
  - IDisposable
  - **using** Statement
  - Garbage Collection

# Inheritance and Interfaces

- Multiple Inheritance is not supported
- A Class may inherit from
  - one base class
  - Implement zero or more Interfaces
- Interfaces are Reference types
- .NET Framework 2.0 introduced Generic Interfaces
  - Generally more useful than non-generic types
  - Needed for use with Generic Collections

# Defining Interfaces

- Defining an interface:

```
public interface IMyInterface
{
      void DoWork( int i);
}
```

No access modifier specified

- Implementing the interface:

```
public class AClass: IMyInterface
{
      public void DoWork( int i){ ...}
}
```

# Defining Generic Interfaces

- Defining an interface:

```
public interface IMyInterface<T>
{
  void DoWork( T t);
}
```

No access modifier specified

- Implementing the interface:

```
public class AClass: IMyInterface<AClass>
{
  public void DoWork( AClass ac){ ...}
}
```

# ICloneable

- Allows the implementation of cloning functionality (Not recommended for public API)!
- Classes implementing this interface must implement the instance method
  - Clone()
- The Clone method is used to create a new object which is a copy of the cloned object
  - Can implement as **deep or shallow** copy
    - For a shallow copy simply call MemberwiseClone within the implementation of the Clone() method
- Many predefined classes support ICloneable

# IComparable

- Allows the implementation of comparison functionality
- Classes implementing this interface must implement the one instance method
  - **CompareTo()** - could throw ArgumentException
    - Return type **System.Int32**

| **Comparison in relation to object of same type** | **Return Value** |
|---|---|
| Less than | < Zero |
| Equal | Zero |
| Greater than | > Zero |

- Must be implemented for objects within List in order to call Sort()

# Implementing IComparable (Generic)

- The following illustrates implementation of a CompareTo method:

```
public class AClass: IComparable<AClass>
{
  private int _i;
  public AClass( int i) { _i = i; }

  public int CompareTo( AClass other)
  {
    return this._i - other._i;
  }
}
```

# Implementing IComparer (Generic)

- The following illustrates implementation of a Compare method:

```
public class MyComparer: IComparer<AClass>
{
  public int Compare(AClass x, AClass y)
  {
    return  x.I – y.I;
  }
}
```

Property gives access to numeric fields being compared

# Implementing IEquatable (Generic)

- Default implementations:

```
public class FlightInfo : IEquatable<FlightInfo>
{
  public override bool Equals(object obj)
  {
    return base.Equals(obj);
  }
  public override int GetHashCode()
  {
    return base.GetHashCode();
  }
  public bool Equals([AllowNull] FlightInfo other)
  {
    throw new NotImplementedException();
  }
}
```

Comparison by overriding methods on 'object' (should be treated as a pair)

Required to implement interface

# IDisposable

- Implemented where resources need to have there lifetime managed
- Garbage Collection occurs at indeterminate times
- If resources (unmanaged) require to be explicitly remove
  - These classes should implement IDisposable
    - This forces the class to implement the **Dispose()** method
- When developer has finished using the object call **Dispose()** method
  - Where the more natural method call would be **Close()** then implement **Close()** method to call **Dispose()**

# **using** Statement

- To help control the lifetime of objects:
  - Scope of the **using** statement defines the lifetime of the object (implementing IDisposable)
  - Leaving the scope causes call to Dispose() method

```
using( AClass ac = new AClass())
{
  // Use ac of type AClass

}
```

Dispose() method called when scope is exited

# Garbage Collection

- It is possible to force Garbage Collection using:

```
System.GC.Collect()
System.GC.Collect( n)
```

Garbage collect all generations

Garbage collect generation zero through to n

- ‘Generational’ Garbage Collector
  - Most recently allocated checked earliest!

| **Method/Property on System.GC** | **Description** |
|---|---|
| MaxGeneration (property) | Maximum number of generations |
| GetGeneration | Get generation of an object |
| AddMemoryPressure | Call in constructor (customise garbage collection process) |
| RemoveMemoryPressure | Call in destructor |

# Interfaces - Summary

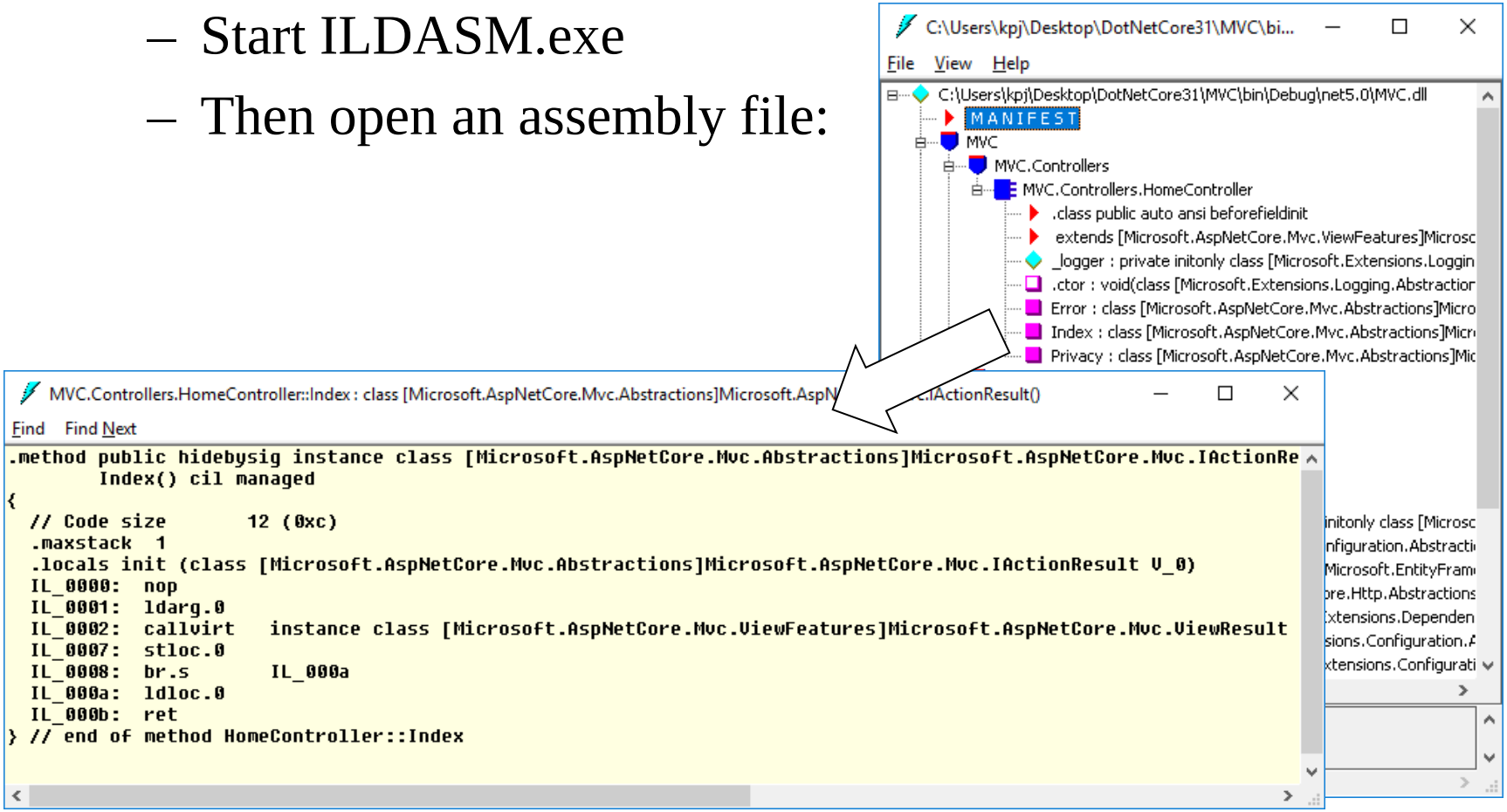
- This section covered:
  - Inheritance
  - Interfaces
  - Generic Interfaces
  - Defining Interfaces
  - Defining Generic Interfaces
  - ICloneable
  - IComparable (Generic)
  - IComparer (Generic)
  - IDisposable
  - **using** Statement
  - Garbage Collection

# Code and Assemblies

- This section covers:
  - Disassembling Code
  - Assembly Strong Name
  - AssemblyInfo
  - Properties – Signing Assembly

# Disassembling Code

- Start the Visual Studio .NET command prompt
  - Start ILDASM.exe
  - Then open an assembly file:



# Assembly Strong Name

- The Strong Name identifies the Assembly
- The Strong Name is Globally Unique
  - Consisting of
    - Simple Name, Version Number, Cultural information (if present), Public Key
- An assembly becomes a strong named assembly by signing
- Create a key file using the sn.exe utilities
  - Use ‘Developer PowerShell’ within Visual Studio

# Attributes

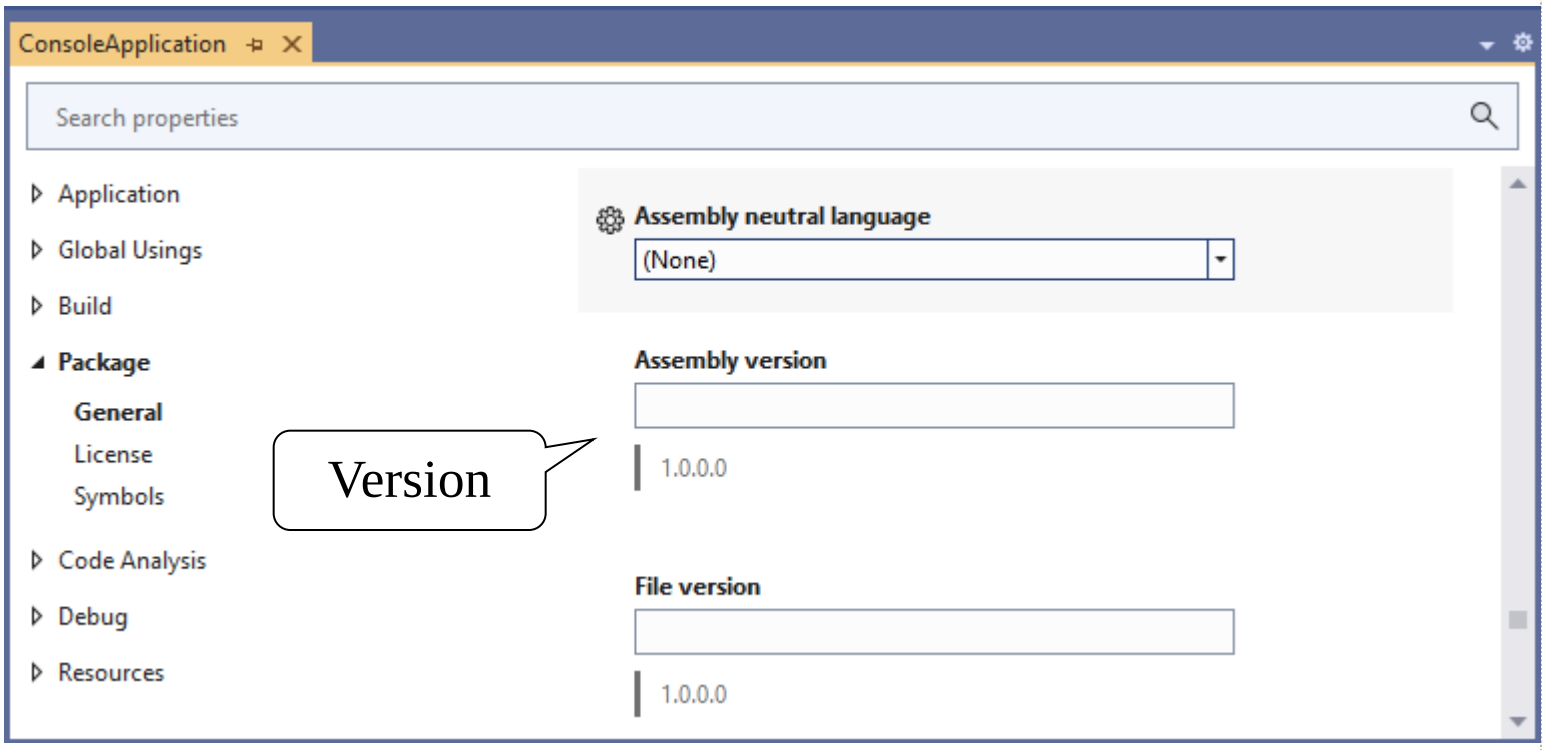
- Attributes provide additional information about code elements
  - Can apply to Assemblies, Classes, Methods, Properties, etc.
  - Attribute information is stored within the metadata of Assemblies
  - Many attributes may be applied to the same element
  - Attributes may be parameterised
  - Many predefined Attributes, but can define your own
- Attribute syntax:

```
[AttributeType(parameters)]
```

The word ‘Attribute’ can be dropped from the end of the type name when using the Attribute

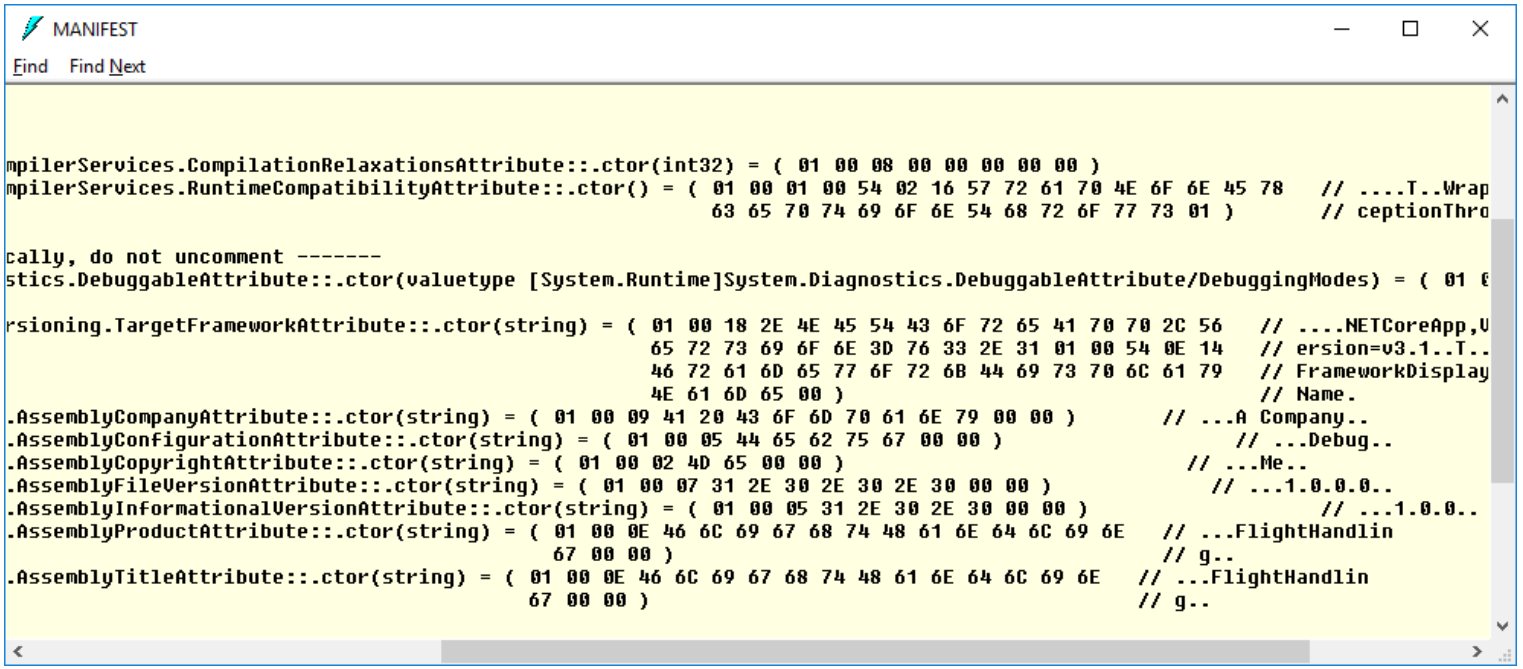
# AssemblyInfo.cs File

- Developer supplied information about the Assembly



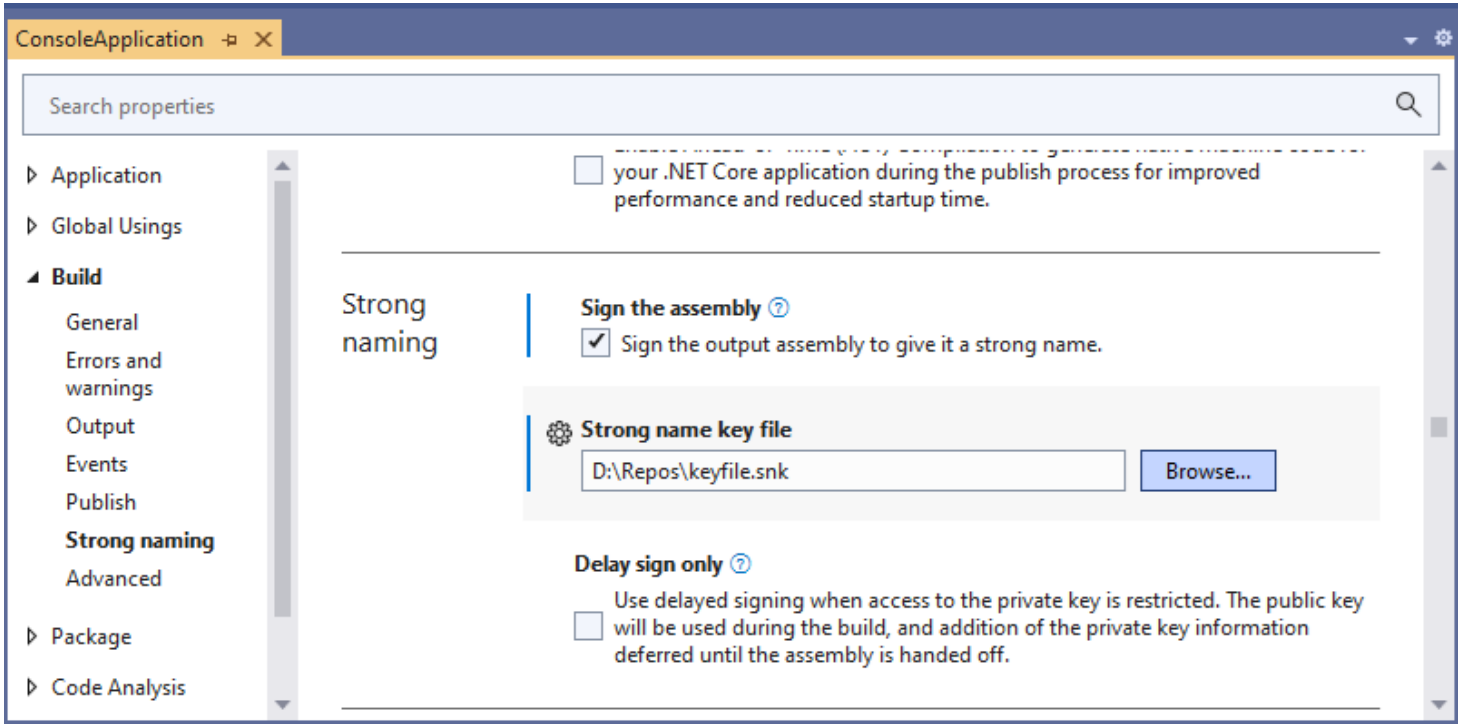
# ILDASM Manifest

- Use the Intermediate Language Disassembler to view Intermediate Language
- Can also view Manifest with ILDASM:



# Properties - Signing Assembly

- Strong Naming Properties (Build option):




- One key file can be used for many assemblies
  - This key pair should be protect, in particular
    - **Do not expose the private key!**

# Code and Assemblies - Summary

- This section covered, as follows:
  - Disassembling Code
  - Assembly Strong Name
  - AssemblyInfo
  - Properties – Signing Assembly