



C# Coding Part I

Basic Language Features



Writing Files Containing C# Code

- Your C# code goes in files with the '.cs' suffix
 - A single file can contain the declaration of many classes
 - The file name does not need to match the class name(s)
- Each file contains the following:
 - Using declarations
 - A namespace declaration
 - One or more class declarations
- Your code will be easiest to browse if:
 - Each file contains one class only
 - The file is named after the class it contains



Using Partial Classes

- Partial classes are a new feature in VS 2005
 - A class declaration can be split across multiple source files
 - Simply declare the class as 'partial' in each file
- This feature is aimed at code generation only
 - In VS 2003 generated code was inserted into the same file as type declarations and event handlers
 - This adds complexity and makes update errors likely
 - Since VS 2005 generated code is kept separate
 - This is much better for beginners but can become annoying
 - Don't use partial classes in your own code without good reason



Keywords Within the C# Language

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			



Basic Data Types in C# Programs

- All .NET languages share the same data types
 - Otherwise they could not be interoperable
 - This is the Common Type System (CTS)
- Each language has its own names for the types
 - An 'int' in C# and an 'Integer' in VB .NET are identical types
 - They both map to the 'System.Int32' type recognized by the CLR
 - Using the CTS names directly is possible but not advised
 - This is why the compiler lets you use 'string' or 'String'
 - You will see the CTS names appearing frequently when debugging or using tools like the IL Disassembler (ILDASM)



Core Data Types And Classes

C# Name	CTS Name
byte	System.Byte
sbyte	System.SByte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
decimal	System.Decimal
bool	System.Boolean

C# Name	CTS Name
object	System.Object
string	System.String



Literal Values

- String literals are characters in double quotes
 - A regular string literal may contain the escape sequences familiar to Java and C++ developers
 - `'\n'` for new line, `'\t'` for tab, `'\"'` to escape a quote etc...
 - Verbatim string literals begin with the `'@'` character
 - Escape sequences in the string are not processed
 - This is useful for file paths e.g. `'@"c:\dev\bin\app1.exe"'`
- Whole number literals are integers by default
 - If bigger than an int the value is a uint, long or ulong
- Real number literals are double by default
 - If they are suffixed by `'f'` or `'m'` they are a `'float'` or `'decimal'`



The Convert Class

- This class is a ‘junction box’
 - It contains static methods for type conversions
 - You can convert each basic type into any other
- Each conversion method is overloaded to take instances of all the other basic types
 - An exception will be thrown if information is lost

```
string str1 = "16";  
int ival = System.Convert.ToInt32(str1);  
double dval = System.Convert.ToDouble(str1);  
string str2 = "True";  
bool bval = System.Convert.ToBoolean(str2);
```



Making Choices

- If statements work the same as in C++ and Java
 - The condition must be a boolean expression
- Switch statements are also very similar
 - Each case value must be unique
 - The default case statement is optional
 - Unlike C++ you do not ‘fall through’ to the next case
 - Each case must end with an explicit transfer of control
 - This will typically be a ‘break’, ‘goto’, ‘return’ or ‘throw’
- However it is also possible to switch on strings
 - This is a very useful feature e.g. for command line arguments



Making Choices

```
if (password.Equals("wn1hgb")) {  
    //do stuff  
} else if (password.Equals("admin")) {  
    //do other stuff  
} else {  
    //do default  
}
```

```
switch ( obj.func() ) {  
    case 1:  
        //do stuff  
        break;  
    case 8:  
        return true;  
    case 12:  
        //do stuff  
        break;  
    default:  
        //do stuff  
        break;  
}
```



Iteration

- C# supports the same loops as C++ and Java
 - The 'for' loop for bounded iteration
 - The 'while' loop for unbounded iteration
 - The 'do...while' loop which executes at least once
- The condition must be a boolean expression
 - You can 'continue' past the current iteration
 - You can 'break' out of one or more loops
- The 'goto' statement transfers control to a label
 - You can only transfer control out of a nested scope
 - This ensures that you do not skip over any initializations



Iteration

```
while (i < 7) {  
    if(someCondition) {  
        i++  
    }  
    //do stuff  
}
```

```
do {  
    if(someCondition) {  
        i++  
    }  
    //do stuff  
} while (i < 7);
```

```
for (int i=0; i< LIMIT; i++) {  
    //do stuff  
}
```

Iteration

```
while (i < 7) {  
    if(conditionA) {  
        //exit the loop  
        break;  
    } else if (conditionB) {  
        //skip the rest of the body  
        continue;  
    }  
    //do stuff  
}
```

```
while (i < 7) {  
    while (x > 10) {  
        if(conditionA) {  
            //exit the inner loop  
            break;  
        } else if (conditionB) {  
            //exit both loops  
            goto FINISHED;  
        }  
    }  
}  
FINISHED:  
//do stuff
```



The 'foreach' Loop

- C# additionally supports a 'foreach' loop
 - This replaces boilerplate for loops
 - E.g. `for(int i=0; i<myArray.Length;i++) { ... }`
 - The syntax is `foreach(<type><name> in <expression>)`
- The expression must be a type which:
 - Implements the interface 'IEnumerable' OR
 - Has a 'GetEnumerator' method that returns a type with a 'MoveNext' method and a 'Current' property
- This means the type counts as a collection type
 - All arrays inherit from 'System.Array' which is a collection type



The 'foreach' Loop

```
int [] iarray = {1,2,3,4,5};
foreach(int val in iarray) {
    Console.WriteLine("{0}",val);
}
```

```
string[] tstData = {"abc","def","ghi","jkl","mno"};
foreach(string s in tstData) {
    Console.WriteLine("{0}",s);
}
```

```
void iterateListManually(IEnumerator list) {
    while(list.MoveNext()) {
        Console.WriteLine("{0} ",list.Current);
    }
}
void iterateListViaForEach(IEnumerable list) {
    foreach(object obj in list) {
        Console.WriteLine("{0} ",obj);
    }
}
```



Boxing and Unboxing

- Boxing allows any type to be considered as an object
 - The compiler will implicitly convert value types into references of type 'object', 'Enum' or 'ValueType'
 - A copy of the value type is 'boxed' into an object
 - This means you can write 'int i=10; string s = i.ToString()'
- Unboxing converts a boxed type back to a value type
 - For example 'object boxed = 23; int i= (int)boxed;'
- Boxing makes the language more object oriented
 - You can pass integers as parameters of type 'object'
 - Boxing was introduced to Java in version five



Nullable Types in C# 2.0

- Nullable types are a feature introduced in C# 2.0
 - They allow a primitive type to be assigned to null
 - This is especially useful in data access objects (DAO's)
- Nullable types are based on 'System.Nullable<T>'
 - 'int? i = func()' is shorthand for 'System.Nullable<int> i = func()'
 - 'Nullable' defines 'HasValue' and 'Value' properties
 - You can use these or directly compare the variable to 'null'
- An operator is introduced for use with nullable types
 - '??' is used when a nullable type appears on the right hand side of an assignment (and the left hand side is non nullable)
 - E.g. 'int x = i ?? 123' assigns 123 to 'x' if 'i' is null



Checked Statements

- C# does not check for numeric overflows at runtime
 - If the result of an expression is too large or small its value is truncated to fit into the receiving variable
 - Compile time expressions are checked by the compiler
- But you can place expressions in a checked context
 - By placing the 'checked' keyword before the expression
 - For example 'checked(a * b)'
 - By surrounding multiple expressions in a checked statement
 - For example 'checked { int x = y * z; x++; x *= w; }'
 - If an overflow occurs an 'OverflowException' is thrown



Boxing and Checking

```
byte b1 = 255;
byte b2 = 255;

// b1 takes value 0
b1++;

//generates OverflowException
checked {
    b2++;
};
```

```
private static void test() {
    // show boxing and unboxing
    int ivar = 20;
    object obj = showBoxing(ivar);
    int myInt = (int)obj;
    Console.WriteLine("Integer value is: {0}\n",myInt);
}

private static object showBoxing(object obj) {
    return obj;
}
```



Enumerations

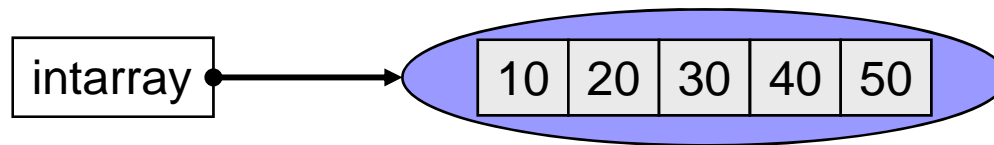
- An enumeration represents a set of constants
 - Each of which has a name and value
- Enums have a default type and values
 - E.g. `'enum Color { Red, Green, Blue} '` declares an enum called Color containing three integer constants of values 0,1 and 2
- You can define different types and values
 - E.g. `'enum State : long { On = 20, Standby = 30, Off = 60 }'`
 - If the first constant has no value it is set to zero
 - Other constants take the value of the preceding one plus one
- Enums can be cast to and from numbers
 - Which is why they were not included in Java



Arrays

- C style arrays are a contiguous series of types
 - Such as ten integers laid out end to end
 - These cause safety, security and usability issues
- In .NET all arrays are objects
 - An array is an object which manages a sequence of values
 - New array classes are produced by the CLR as required
 - They all inherit from the base class 'System.Array'
 - You can find the length of any array via the 'Length' property
- Arrays may contain basic types or references
 - An array of 'MyClass' contains references to 'MyClass' objects
 - You never have objects actually contained in the array itself
 - This is possible with local objects in unmanaged C++

Arrays in C#



```
int [] intarray = new int[5];  
intarray[0] = 10;  
intarray[1] = 20;  
intarray[2] = 30;  
intarray[3] = 40;  
intarray[4] = 50;
```

```
int [] intarray = new int[] {10,20,30,40,50};
```

```
int [] intarray = {10,20,30,40,50};
```



Multidimensional Arrays

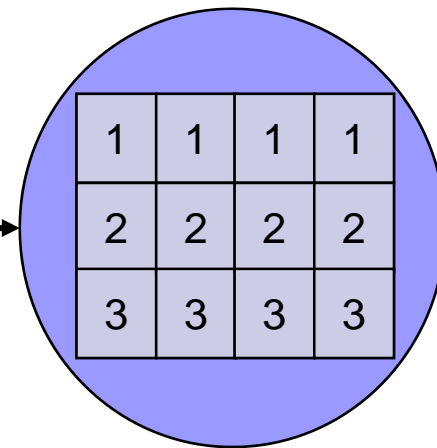
- All .NET arrays have a rank
 - The number of dimensions in the array
 - Multidimensional arrays have rank >1
- There are two kinds of multidimensional array
 - Rectangular arrays are implemented as in C
 - All of the elements are stored in a grid within a single object
 - Jagged arrays are implemented as arrays of arrays
 - A 2d array contains elements which are references to 1d arrays
 - The size of each row in each dimension can vary
 - The 'GetLength' method finds the length of each dimension
 - 'myarray.GetLength(0)' is equivalent to 'myarray.Length'

Rectangular Arrays in C#

```
int [ , ] myarray = new int [3,4];  
for(int i=0; i<myarray.GetLength(0);i++) {  
    for(int j=0; j<myarray.GetLength(1);j++) {  
        myarray[i,j] = i + 1;  
    }  
}
```

```
int [ , ] myarray = {  
    {1,1,1,1},  
    {2,2,2,2},  
    {3,3,3,3}  
};
```

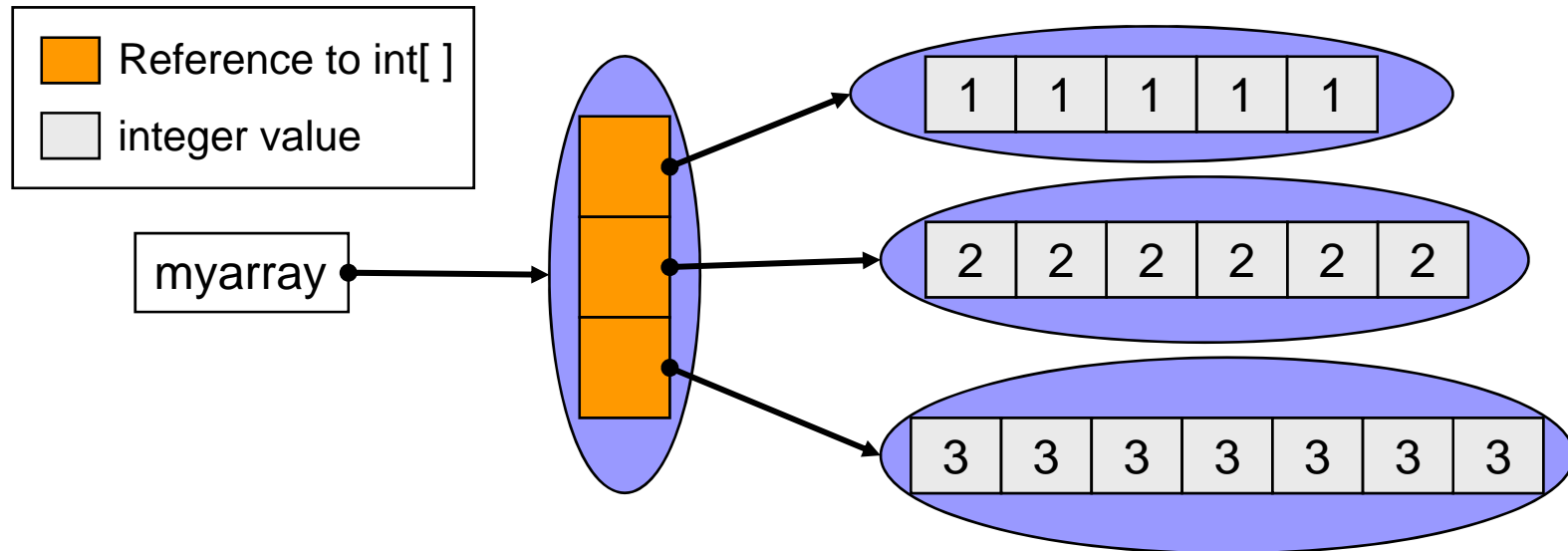
myarray



Jagged Arrays in C#

```
int[][] myarray = new int[3][];  
myarray[0] = new int[5];  
myarray[1] = new int[6];  
myarray[2] = new int[7];
```

```
for(int i=0; i<myarray.Length;i++) {  
    for(int j=0; j<myarray[i].Length;j++) {  
        myarray[i][j] = i + 1;  
    }  
}
```





Creating Classes in C#

- The syntax of C# class declarations is closest to C++
 - However the semantics is much closer to Java
 - Its important not to be misled by familiar constructs
 - Both C++ and Java developers will find that C# uses familiar concepts in unfamiliar ways
- As with Java there are no separate header files
 - The whole class is declared within 'class MyClass { ... }'
 - Unlike Java multiple classes can be declared in the same file
- Classes are declared with public or internal accessibility
 - Members of the class can additionally use the protected, private and protected internal accessibilities



A Simple Class Declaration

```
public class Employee {
    public Employee(string NINO, string name, string department, int age) {
        this.NINO = NINO;
        this.name = name;
        this.department = department;
        this.age = age;
    }
    public override string ToString() {
        return name + " of age " + age + " working in " + department;
    }
    public string InsuranceNumber {
        get {
            return NINO;
        }
    }
    private int age;
    private string NINO;
    private string name;
    private string department;
}
```



Class Inheritance

- The syntax for inheritance is taken from C++
 - We write 'public class Derived : Base { ... }'
 - The derived class cannot be more accessible than the base
- Every object contains two built in references
 - The 'this' reference refers to the current object
 - The 'base' reference refers to the inherited part of the object
- Base classes can be marked as 'abstract'
 - This means they cannot be instantiated directly
 - Abstract classes may contain abstract methods
- Leaf classes can be marked as 'sealed'
 - This ensures they will never be used as base classes



Inheritance Example

```
public class Employee {
    //Base class constructor
    public Employee(string id, string name) : base() {
        this.id = id;
        this.name = name;
    }
    protected string id;
    protected string name;
}

public class SalesPerson : Employee {
    //Derived class constructor
    public SalesPerson(string id, string name, double target) : base(id,name) {
        this.target = target;
    }
    private double target;
}
```



Inheritance and Downcasting

- Casting from a base to a derived type is dangerous
 - An exception will be thrown if the actual type of the object is incompatible with the target type of the cast
- The 'as' operator performs a safe downcast
 - If the cast is impossible null is returned
- The 'is' operator determines if a cast is possible
 - It is similar to the 'instanceof' operator in Java
- It is possible to use 'is' to test for boxing
 - The expression 'obj is int' returns true if obj is an object reference that points to a wrapped up integer



Downcasting Safely

```
//assume Derived inherits from Base  
Base b = new Derived();
```

```
//perform safe downcast using as  
Derived d = b as Derived;  
if (d != null) {  
    d.func();  
}
```

```
//perform safe downcast using is  
if(b is Derived) {  
    Derived tmp = (Derived)b;  
    tmp.func();  
}
```



The Object Class

- Object is the ultimate base class in .NET
 - As mentioned previously all classes in every .NET language inherit from it, either directly or indirectly
 - The 'object' type is just an alias for 'System.Object'
- A reference of type object can refer to anything
 - This is useful in collections and generic functions
- Methods of Object should be overridden as required
 - 'ToString' should always be overridden in entity classes
 - 'Equals' should be overridden where comparison is required
- The 'Finalize' method cannot be overridden in C#
 - The concept of a destructor is used instead (see later)



The Contents of a C# Class

Class Member	Purpose	Notes
Constructor	Initialises data within the class	Can be static or instance
Destructor	Allows release of acquired resources	Replaces the 'finalize' method
Method	A service provided by this class	Should be query or command
Property	For safe and convenient access of fields	Similar to accessor methods
Indexer	Provides array-like syntax for objects	Same as 'operator[]' in C++
Operator	Redefine C# operators for your class	Use with caution
Event	Automate the registration of callbacks	Uses the '+=' and '-=' operators
Type	Nested classes can be declared	NOT Java Inner Classes
Field	Hold state within the object or class	Can be static or instance
Constant	Constant values required by the class	Implicitly static



Using Fields

- Fields represent data within a class or object
 - Fields declared as 'static' have a single copy
 - This is held at class level and accessible to all class instances
 - Other fields are 'instance fields' with a copy in each object
- All fields are automatically initialized to default values
 - If you do not give them an initial value in your code
 - This differs from local variables which must be explicitly assigned an initial value before first use
- A fields value may never change
 - In which case it should be declared as 'const' or 'readonly'



Using Fields

- Constants are values known at compile time
 - They are declared with the 'const' keyword
 - For example 'public const double PI = 22 / 7;'
- Each use of the constant is replaced with its value
 - Hence they can be used everywhere a literal is used
 - E.g. in case statements within a switch block
 - Constants can be used to define other constants
- Fields declared 'readonly' are runtime constants
 - The value of the constant is discovered at runtime
 - The field can only be set in the constructor or initializer and thereafter cannot be modified



Using Constructors

- A constructor is a method named after the class
 - A constructor never has a return type
- Constructors are used to initialize class instances
 - Any number of overloaded constructors may be declared
- A constructor has an initializer before its body
 - Calls to other constructors can be inserted here
 - Use 'this' like a method to call another constructor of the same class
 - Use 'base' like a method to call a constructor of the superclass
 - If no other constructor is called the compiler adds a call
 - 'public MyClass() { ...}' is equal to 'public MyClass() : base() { ...}'
 - This ensures initialization proceeds from the top down



Using Constructors

```
class Constructors : Base {  
  
    //Calls Base();  
    public Constructors() : base() {  
    }  
  
    //Calls Base(string)  
    public Constructors(string p1) : base(p1) {  
    }  
  
    //Reuse existing constructor above  
    public Constructors(string p1, string p2) : this(p1) {  
        derivedFieldOne = p2;  
    }  
  
    //Reuse existing constructor above  
    public Constructors(string p1, string p2, string p3) :this(p1,p2) {  
        derivedFieldTwo = p3;  
    }  
  
    //Fields  
    string derivedFieldOne;  
    string derivedFieldTwo;  
}
```



Using Constructors

- Every class must have at least one constructor
 - A default constructor is added by the compiler if required
 - For example `public MyClass() : base() {}`
 - If the class is abstract then the compiler makes it protected
- Fields can also be initialized in their declaration
 - For example `protected int myField = 27`
 - These initializations are run before any constructor
- A class may have a static constructor
 - This is declared with the `static` keyword and has no parameters
 - It is called once, when the class is loaded into the AppDomain



Destructors and Finalization

- Like the JVM the CLR has a Garbage Collector to sweep up objects which are no longer reachable
 - There are important differences in the way GC works
- As in Java you inherit a 'Finalize' method from the universal base class 'Object'
 - In VB .NET you can override the method yourself
 - In C# this must be done indirectly
 - In either case you should use with caution (as in Java...)
- There is a built in pattern for signalling to an object that its will no longer be used
 - This is based on the interface 'IDisposable'

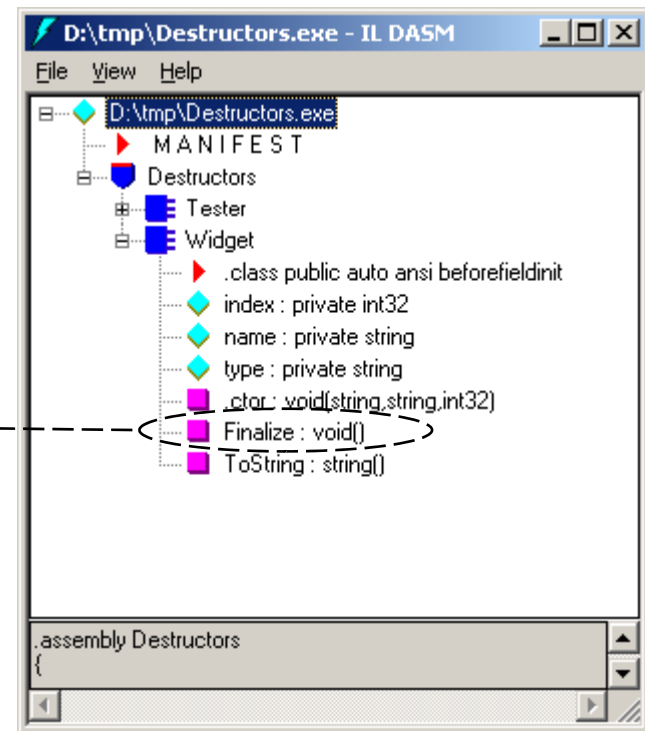


Destructors and Finalization

- C# borrows the concept of a 'destructor' from C++
 - It looks like a constructor but has a '~' before the class name
 - A class will have at most one destructor
 - Destructors cannot be called explicitly (unlike C++)
- Destructors need to be clearly understood in C#
 - They have the C++ syntax but Java semantics
- The compiler links the destructor code into 'Finalize'
 - The destructor is invoked when an object is GC'd
 - This can occur any time after the object becomes unreachable
- For derived classes all inherited destructors all called
 - Starting with the derived class and moving up the hierarchy

Destructors and Finalization

```
public class Widget {
    public Widget(String type, String name, int index) {
        this.type = type;
        this.name = name;
        this.index = index;
    }
    ~Widget() {
        Console.WriteLine("Destructor called for
                           widget with index {0}
                           in generation {1}",
                           index,
                           GC.GetGeneration(this));
    }
    public override string ToString() {
        //Implementation omitted
    }
    private string type, name;
    private int index;
}
```





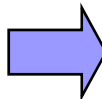
The IDisposable Interface

- Classes which need resources released in a timely manner should not rely on destructors
 - Instead they should implement the 'IDisposable' interface
- Disposable objects should have their 'Dispose' methods called by clients when the objects lifecycle is ending
 - The object can then release resources and close connections
- It may be necessary to prevent 'Finalize' being called if the 'Dispose' method has previously been triggered
 - This can be done inside the 'Dispose' method by calling 'GC.SuppressFinalize(this)'

The Using Statement

- C# provides the 'using' statement to make the 'IDisposable' pattern more reliable
 - One or more local variables are created inside the brackets
 - These variables are disposed of at the end of the braces
- The effect is identical to a 'try .. finally' block with calls to 'Dispose' in the finally

```
using (MyClass mc = func()) {  
    mc.op1();  
}
```



```
try {  
    MyClass mc = func();  
    mc.op1();  
} finally {  
    if(mc != null)  
        ((IDisposable)mc).Dispose();  
}
```



Using Methods

- Methods represent services offered by an object
 - Protected methods allow special access for subclasses
 - Private methods are helper methods hidden from clients
- In OO systems methods have two roles
 - Query methods discover the state the object is in
 - Other methods request that the object change its state
 - For example compare 'isConnected' with 'makeConnection'
- Methods can be overloaded based on signature
 - A method is identified by its name and its parameters
 - As long as the type, number or order of parameters is different then two or more methods can share the same name
 - As always you cannot overload based in return type

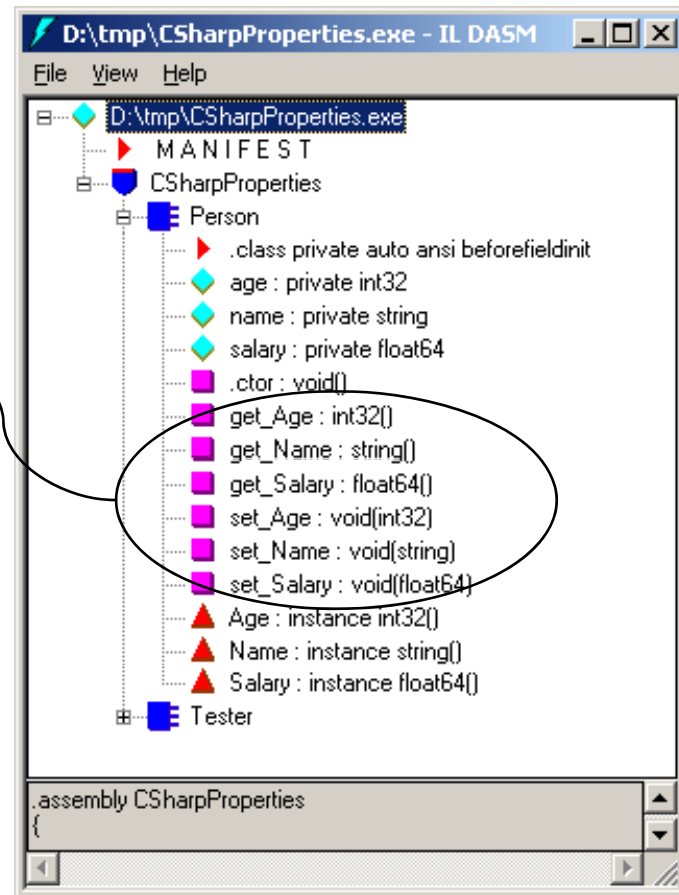


Using Properties

- Java code frequently uses accessor methods
 - A field called 'customerName' is accessed via:
 - 'public String getCustomerName()'
 - 'public void setCustomerName(String name)'
- C# code simplifies this with properties
 - A single code block is used to define the get and set methods
 - In the set method 'value' represents the property value
- Properties are accessor methods in disguise
 - The compiler transforms them into 'set_' and 'get_' methods
 - Any use of a property is turned into a call to a method
 - The compiler must be able to work out which method to call

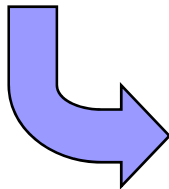
Understanding Properties

```
class Person {  
    public double Salary {  
        get { return salary; }  
        set { salary = value; }  
    }  
    public int Age {  
        get { return age; }  
        set { age = value; }  
    }  
    public string Name {  
        get { return name; }  
        set { name = value; }  
    }  
    private double salary;  
    private int age;  
    private string name;  
}
```



Understanding Properties

```
static void Main(string[] args) {  
    Person person = new Person();  
    person.Name = "Joe Bloggs";  
    person.Age = 27;  
    person.Salary = 30000.00;  
  
    Console.WriteLine("Person {0} aged {1}, earns {2}", person.Name, person.Age, person.Salary);  
}
```



```
ldstr    "Joe Bloggs"  
callvirt instance void CSharpProperties.Person::set_Name(string)  
ldc.i4.s 27  
callvirt instance void CSharpProperties.Person::set_Age(int32)  
ldc.r8   30000.  
callvirt instance void CSharpProperties.Person::set_Salary(float64)  
ldstr    "Person {0} aged {1}, earns {2}"  
callvirt instance string CSharpProperties.Person::get_Name()  
callvirt instance int32 CSharpProperties.Person::get_Age()  
callvirt instance float64 CSharpProperties.Person::get_Salary()  
call     void [mscorlib]System.Console::WriteLine(string,object,object,object)
```



Using Indexers

- Indexers allow an object to behave like an array
 - Clients can write 'ref[3]' rather than 'ref.get(3)'
 - Useful for classes which represent collections of entities
- Properties and indexers are similar
 - Your code is placed in get and set methods
 - In the set method 'value' represents the input
 - You decide what data is actually being accessed
- The compiler matches an indexer by its signature
 - An indexer is declared as '<type> this[<type><name>]'
 - You can have more than one indexer if the signatures differ



Using Indexers

```
public class Indexed {
    public int this[int index] {
        get {
            switch(index) {
                case 0: return valOne;
                case 1: return valTwo;
                case 2: return valThree;
                default: throw new IndexOutOfRangeException();
            }
        }
        set {
            switch(index) {
                case 0: valOne = value; break;
                case 1: valTwo = value; break;
                case 2: valThree = value; break;
            }
        }
    }
    private int valOne, valTwo, valThree;
}
```



Passing Parameters

- C# supports three types of parameter passing
 - Pass by value, by reference and for initialization
- Pass by value is the default
 - If the parameter is a basic type its value is copied
 - If the parameter is a reference then the address in memory of the object it points to is copied (the object itself is never copied)
- The 'ref' keyword is used to pass by reference
 - The address of a type is passed rather than its value
 - This allows the value of the original to be changed
 - The actual memory location is still hidden



Passing Parameters By Reference

```
private void test() {  
    int i = 14;  
  
    //Pass the address of the integer rather than its value  
    incrementByRef(ref i);  
  
    //Prints 'Value is now: 15'  
    Console.WriteLine("Value is now: {0}",i);  
}  
//Take an integer parameter using pass by reference  
private void incrementByRef(ref int pByRef) {  
    pByRef++;  
}
```



Passing Parameters

- The 'out' keyword allows unset values to be passed
 - An output parameter is considered to be unassigned
 - Output parameters must be assigned within the method
- The 'params' keyword enables arbitrary parameter lists
 - It performs the same role as the C++ ellipses (...)
 - A parameter declared with 'params' must be an array
 - Multiple arguments are inserted into an array to call the method
 - This lets you create methods such as 'Console.WriteLine'



Output Parameters

```
private void test() {
    float f1,f2,f3;

    //Initialise the first three parameters using the fourth
    initialiseAscending(out f1,out f2, out f3, 31.2f);

    //Prints 'Values are: 32.2 33.2 34.2'
    Console.WriteLine("Values are: {0} {1} {2}",f1,f2,f3);
}
//Pass the first three parameters by reference and without a value for initialization
private void initialiseAscending(out float p1, out float p2, out float p3, float startVal) {
    p1 = ++startVal;
    p2 = ++startVal;
    p3 = ++startVal;
}
```



Parameter Arrays

```
private void test() {  
    //Show parameter lists  
    int result = addIntegers(20,30,40,50,60,12);  
  
    //Prints 'Total of integers is: 212'  
    Console.WriteLine("Total of integers is: {0}", result);  
}  
  
//Accept any number of integers as parameters  
private static int addIntegers(params int[] args) {  
    int total = 0;  
  
    foreach(int i in args) {  
        total += i;  
    }  
    return total;  
}
```



Polymorphism

- Overriding in C# is more verbose than C++/Java
 - The base class method must be declared with 'virtual'
 - Overriding methods must be declared with 'overrides'
- The 'new' modifier lets a method opt out of overriding
 - A method which would otherwise override a base method is not considered for use when a polymorphic call is made
 - Calls via base references trigger 'most derived' implementation
 - Calls via a reference of the actual type will find the 'new' method
- The 'sealed' modifier is used to prevent further overriding
 - If a class declares 'public sealed override void func() { ... }' then inheriting classes will not be able to implement 'func' themselves

Polymorphism in C#

```
class Base {  
    //switch on polymorphism  
    public virtual void func() {  
        Console.WriteLine("Base.func");  
    }  
}  
class Middle : Base {  
    //override base class implementation  
    public override void func() {  
        Console.WriteLine("Middle.func");  
    }  
}  
class Derived : Middle {  
    //hide base class implementation  
    public new void func() {  
        Console.WriteLine("Derived.func");  
    }  
}
```

```
Base b1 = new Base();  
Base b2 = new Middle();  
Base b3 = new Derived();  
  
b1.func(); //calls Base.func  
b2.func(); //calls Middle.func  
b3.func(); //calls Middle.func  
  
//perform safe downcast  
Derived d = b3 as Derived;  
if (d != null) {  
    d.func(); //calls Derived.func  
}
```



Nested Classes

- Every C# type is either nested or non-nested
 - A non-nested type is declared outside of any scope
 - Except the current namespace if one is present
 - A nested type is declared within a class or struct
- Non nested types are public or internal
 - Classes nested in classes can have any accessibility
 - This determines if clients are allowed to reference them
- Nested type names are qualified by the enclosing type
 - So 'Inner' declared in 'Outer' has the full name of 'Outer.Inner'
 - This is how client code must declare references to 'Inner'

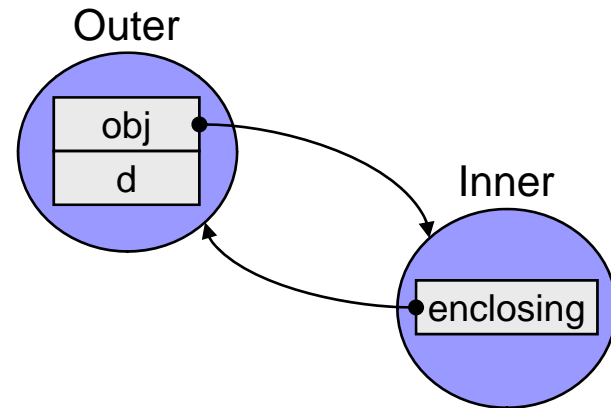


Nested Classes

- Nested classes are typically used as helpers
 - They encapsulate a job that must be performed by the enclosing class but is not its primary purpose
 - For this reason they are declared private and hidden from clients
- A nested class is considered as part of the implementation of its enclosing type
 - Hence it has permission to access all the members of the enclosing type, regardless of their accessibility
- There is no built in reference to an enclosing object
 - A nested object does not know which object it is 'inside'
 - If you require this functionality you must add it yourself
 - This differentiates nested classes from Java's inner classes

Nested Classes

```
class Outer {
  Outer() {
    obj = new Inner(this);
    obj.func(23);
  }
  private class Inner {
    Inner(Outer enclosing) {
      this.enclosing = enclosing;
    }
    void func(double p) {
      enclosing.d = p;
    }
    private Outer enclosing;
  }
  private Inner obj;
  private double d;
}
```





Using Interfaces

- Constants in C# are stated as interfaces
 - An interface is similar to a class except it only defines behaviour
 - Just like an abstract class holding only abstract methods
- Interfaces can inherit from other interfaces
 - Because one contract may depend on another
- All interface members are implicitly public and virtual
 - Interface member declarations cannot have any modifiers
 - Members may be methods, properties, events or indexers
- The UML term for implementing interfaces is 'realization'



Implementing Interfaces

- As with inheritance the colon is used for realisation
 - If you are using both the name of the base class comes first
- To realise an interface you must implement its members
 - You do not use the 'override' keyword when doing this
 - Abstract classes can declare interface members as 'abstract'
 - Derived classes don't automatically override the method
 - The base class method must be declared as 'virtual'
 - Otherwise the derived call must re-implement the interface
- You can implement an interface method explicitly
 - Using the syntax 'public void MyInterface.func() {}'
 - Useful if you inherit the same method from two interfaces



Implementing Interfaces

```
public interface IDrawable
{
    void draw(int size);

    int X { get; set; }
    int Y { get; set; }
}

public interface IColorable : IDrawable {
    void paint(int color);
}
```

```
public abstract class Shape : IColorable {
    public abstract void paint(int color);
    public abstract void draw(int size);

    public int X {
        get { return xPosition; }
        set { xPosition = value; }
    }

    public int Y {
        get { return yPosition; }
        set { yPosition = value; }
    }

    private int xPosition;
    private int yPosition;
}
```



Implementing Interfaces

```
public class Square : Shape {
    public override void paint(int color) {}
    public override void draw(int size) {
        drawTopOrBottom(size);
        drawSides(size);
        drawTopOrBottom(size);
    }
    private void drawTopOrBottom(int size) {
        for(int i=0;i<size;i++) { Console.Write("+"); }
        Console.WriteLine();
    }
    private void drawSides(int size) {
        for(int i=0;i<size;i++) {
            Console.Write("+");
            for(int x=2;x<size;x++) { Console.Write(" "); }
            Console.WriteLine("+");
        }
    }
}
```



Exception Handling

- C# and Java both adopt the C++ syntax for exceptions
 - Fortunately the .NET exception model is much cleaner than C++
- Code executes inside ‘try { ... } catch(...) { ... }’ blocks
 - When an exception is thrown the CLR searches for the closest matching ‘catch’ statement on the call stack
 - The stack will be ‘unwound’ until a matching catch block is found or we reach the top of the stack and the thread exits
 - If the unwinding reaches a static initializer or static constructor a ‘System.TypeInitializationException’ is thrown
 - An exception thrown from a destructor is not propagated
 - It is discarded and execution continues with the base destructor



Exception Handling

- As in Java there is a hierarchy of exception classes
 - 'System.Exception' is the base type of all exceptions
- C# does not support Java compile time exceptions
 - That is exception types which are stated in a method declaration and for which the compiler ensures error handlers are written
- Your code can throw its own exception types
 - Use the 'throw' keyword to throw or re-throw an exception
- Cleanup code can be placed in a 'finally' block
 - This ensures the code executes and avoids duplication
 - It is possible to have 'try { ... } finally { ... }' without any 'catch'
 - As in Java code within 'finally' can replace the current exception
 - Unlike Java the 'return' statement cannot be used in a 'finally'