

A Compact Guide to C#

This document assumes you have done object-oriented programming in C++ or Java and are following along in class as we present motivation, details and applications. This is only a sketch of some of the highlights of C#, emphasizing its distinctive features. C# is new, and there is no brief, high-quality book available yet. [Note added in 2003: Archer and Whitechapel, *Inside C#, 2nd Ed.* is highly recommended.]

Part I

Procedural Programming Aspects

I.1. Program / file organization

```
// Scratch.cs
using System ;
class Program
{
    static void Main()
    {
        Console.WriteLine( "nku" ) ;
        sub() ;
    }

    static void sub()
    {
        // more stuff
    }
}
```

There is no relationship between file name and class name. Methods do not need to be declared before they are called. At the command line

```
> csc Scratch.cs      -- compiles the program (into Scratch.exe)
> Scratch            -- runs the program
```

I.2. Basic console I/O

```
Console.Write( "Enter a number: " ) ;
double x= double.Parse( Console.ReadLine() ) ;

Console.WriteLine( "Cosine of " + x + " is " + Math.Cos(x) + "!" ) ; // Java style still ok
Console.WriteLine( "Cosine of {0} is {1}!", x, Math.Cos(x) ) ; // new style
Console.WriteLine( "Cosine of {0} is {1,8:###}!", x, Math.Cos(x) ) ; // kinda like %8.3f
```

I.3. Parameter passing

```
static void Main()
{
    int i, j ;

    set2( out i, out j ) ; // ok for i,j to be uninitialized
    swap( ref i, ref j ) ; // error if i,j were uninitialized
    print( i,j ) ;       // error if i,j were uninitialized
}

static void swap( ref int i, ref int j ) // inout reference parameters
{
    int temp= i ;
    i= j ;
    j= temp ;
}

static void set2( out int i, out int j ) // out reference parameters
{
    i= 100 ;
    j= 200 ;
}

static void print( int i, int j ) // value parameters
{
    System.Console.WriteLine( "i= {0}, j={1}", i, j ) ;
}
```

I.4. Assertions

Put `#define DEBUG` and `using diag = System.Diagnostics.Debug ;` near the top of your file. [In Visual Studio.NET you must *also* do this: Project | Add Reference... , select **System.dll**.] Then it's easy:

```
diag.Assert( n > 0 ) ;
```

I.5. Built-in value types

The built-in value types come from the .NET Common Type System (shared by Visual Basic .NET, etc). They can be referred to by their .NET type names or their C#-specific aliases. Here are a few:

int	<i>System.Int32</i>
double	<i>System.Double</i>
char	<i>System.Char</i>
bool	<i>System.Boolean</i>

By definition, the name of a value type names the value directly. So, for example, if **a** and **b** are **ints**, then **a == b** tests whether these two **ints** are equal (as you'd expect), *not* –of course– whether **a** and **b** refer to the very same integer object in memory. Accordingly, variables and constants of value types can never be null, and are stored on the stack. (Each of these types inherits from *System.ValueType*, a itself a subtype of *System.Object*, but discussion of this is deferred to Part III of this document.)

The **char** type is 16-bit Unicode, not 8-bit ANSI. See www.unicode.org.

I.6. Strings

The *System.String* type is aliased to **string** in C#. Strings are immutable reference types, but with **==** overloaded to compare string *contents* rather than references. [“Immutable” means that a string object cannot be changed once constructed. “Reference type” means that names of strings actually name references to string objects on the heap; string variables can be **null**. All objects on the heap are garbage-collected.]

```
string s= "WOXY" ;
string sa= null ;

sa= s ; // sa and s now point to the same "WOXY" object
s= "WDET" ; // s now points to new string "WDET"

char ch= s[0] ; // [] gets you the chars
// s[0]= 'R' ; // illegal- strings are immutable

string sb= sa.Insert( 1, "RO" ) ;
Console.WriteLine( sb ) ; // "WROOXY"

int i= sb.IndexOf( "O" ) ; // the first occurrence of "O" in sb is at i==2
string sc= sb.Substring( i, 2 ) ; // "OO" - substring of length 2 starting at i

diag.Assert( sc == "OO" ) ; // compares string contents (not references)
// diag.Assert( sc < "ZZ" ) ; // < not overloaded for strings (too US-centric!)
diag.Assert( sc.CompareTo( "ZZ" ) < 0 ) ; // ok; will allow Culture objects!

string sd= "AA" + 72 + "BB" ; // "AA72BB"; + is concatenation; note int's string rep.
```

I.7. Arrays

```
// 1D ARRAYS
string[] sar= new string[n] ;

for ( int i=0 ; i < sar.Length ; ++i )
    sar[i]= "slot(" + i + ")" ;

// 2D ARRAYS
double[,] mat= new double[n,n] ;

for ( int i=0 ; i < mat.GetLength(0) ; i++ )
    for ( int j=0 ; j < mat.GetLength(1) ; j++ )
        mat[i,j]= i + 0.1* j ;
```

There are also so-called “jagged arrays” subscripted **ar[i][j]** rather than **ar[i,j]**. Arrays are reference types (so they can be set to **null** and they exist on the heap, where they can be hauled away as garbage when necessary).

I.8. Control

There is no automatic conversion from **int** to **bool**. For example, if **n** is an integer, **if (n)** will not compile; use **if (n!=0)** instead.

Each **case** of a **switch** statement *must* end with an explicit control statement; this is usually either a **break**, **continue** or **return**.

The **foreach** statement can be used to give read-only access to array entries:

```
void f( string[] sar )
{
    foreach ( string s in sar )
        Console.WriteLine( s );
}
```

I.9. Exceptions

Exceptions follow the **try / catch** model of C++ and Java. Exceptions are reference types descended from *System.Exception*. For your own exceptions use should use (or subclass) *System.ApplicationException* :

```
if ( n < 0 )
    throw new ApplicationException( "Bad news" ) ;
```

Once nice thing to do when you catch an exception during development is print a stack trace:

```
try
{
    // . . .
}
catch ( Exception e )
{
    Console.WriteLine( e.StackTrace() ) ;
}
```

I.10. Text file I/O

In the following example, you must be **using System.IO**;

```
const string FILENAME= "myOutFile.txt" ;

// Write a text file.
StreamWriter sw = File.CreateText( FILENAME );
sw.WriteLine ( "Welcome to my file." );
sw.WriteLine ( "Certainly easier than Java!" ) ;
sw.Close();

// Read a text file, echoing it to console.
StreamReader sr = File.OpenText( FILENAME );
string lineIn= sr.ReadLine() ;
while ( lineIn != null )
{
    Console.WriteLine( lineIn );
    lineIn= sr.ReadLine() ;
}
```

Part II

ADT Programming Aspects

II.1. Structs

Structs are user-defined *value types* suited for small bundles of data. They cannot be subtypes or supertypes of other programmer-defined classes. Otherwise, they can have most of the features of classes (see next section), but in practice they tend to be much simpler. They are alternatives to classes when you want a type that is allocated on the stack; the entire object can be passed by value.¹

```
struct Point
{
    public Point ( int x, int y )           // constructor (optional, but convenient)
    {
        this.x= x ; this.y= y ;           // no initializer list for fields
    }

    public int x,y ;
} ← no semicolon

class Program
{
    static void Main()
    {
        Point ptA= new Point() ;           // (0,0), set by default ctor (NOT redefinable)
        Point ptB= new Point( 10, 20 ) ;   // calls our ctor
        Point ptC= new Point( 10, 20 ) ;   // calls our ctor

        co.WriteLine( ptB.x ) ;           // field access uses the . (dot)

        if ( ptB.Equals( ptC ) )           // == not available (unless we overload it)
            co.WriteLine( "Same" ) ;       // Equals() is value comparison (field-wise)
    }
}
```

Don't let the use of the **new** operator fool you: these Points are allocated on the stack, not the heap! They are value types, so they can never be **null**. If we have `void f(Point pt) ;` and call `f(ptA)`, the point is passed by value: the entire object is copied. Note that we cannot define our own default (no-parameter) constructor for structs.

II.2. Classes: The Basics

Classes are reference types, meaning the name of an object is bound to a pointer to the actual object, which lies on the heap. Otherwise, classes follow the model we are used to in C++ with the following points worth noting:

- **const** fields are set at compile time then frozen (no initializer lists); **readonly** fields are set at runtime in the constructor then frozen. There is no notion of **const** method.

¹ From here on, we assume the programmer is `using co = System.Console ;`

- There are no destructors. Objects that can no longer be referenced are automatically deleted.
- C# introduces the notion of *properties*—these are “field lookalikes” that the client can get and (possibly) set in a uniform way. They are roughly analogous to the following C++ idiom:

```
public:
    int& top() ;
    const int& top() const ;
```

Here’s a stripped-down array-based stack of integers designed to illustrate these language features:

```
class StackInt
{
    public StackInt( int nMaxDepth )
    {
        nMAXDEPTH= nMaxDepth ;
        ar= new int[ nMAXDEPTH ] ;
        iTop= -1 ;
    }

    public void Push( int num )
    {
        ar[ ++iTop ]= num ;
    }

    public void Pop()
    {
        iTop-- ;
    }

    public int Depth()
    {
        return iTop+1 ;
    }

    public int Top // a property
    {
        get { return ar[ iTop ] ; }
        set { ar[ iTop ]= value ; }
    }

    public readonly int nMAXDEPTH ;
    private int[] ar ;
    private int iTop ;
}
```

```
class Program
{
    static void Main()
    {
        StackInt s= new StackInt( 10 ) ;

        s.Push( 55 ) ;
        s.Push( 66 ) ;
        s.Push( 77 ) ;

        s.Top *= 100 ;

        while ( s.Depth() > 0 )
        {
            co.WriteLine( s.Top ) ;
            s.Pop() ;
        }
    }
}

/*
Output:
7700
66
55
*/
```

II.3. Enums

Enumerated types in C# are value types. This means they work just like you’d expect from C++:

```
enum Mood { HAPPY, SAD, ANGRY }
```

```
void f( Mood m )
{
    if ( m == HAPPY ) // . . .
}
```

II.4. Indexers

If you are writing a sequentially ordered container class in C++, overloading the [] operator is a nice way to grant random access to its contents. In C# a so-called *indexer* achieves the same effect. For example, if you wanted to grant read access to all the entries in a **StackInt** object (see #2 above), you would add an indexer:

```
class StackInt
{
    // . . .

    public int this [ int i ← square brackets !
    {
        get { return ar[i] ; }
        // set omitted, since we do not want write-access to the interior of a stack
    }
}
```

This is very similar to a C# *property*. Now if we have **StackInt s** then we can print the 3rd element from the bottom of the stack with `co.WriteLine(s[3])`. Since we decided not to define **set**, we cannot assign to it: `s[3]=7` will not compile. (If we allowed this it would hardly be a stack.)

II.5. Main

Any C# class can have a static **Main()** method. It's sometimes nice to bundle an ADT class with a little **Main()** that demos its usage. At the command line, you decide which main to run. If you want **StackInt's** **Main()**, do:

```
> csc MyCode.cs /main:StackInt
> MyCode
```

II.6. Inheritance (ADT style)

C# supports single inheritance of classes. The superclass is known as the *base* class; its constructor can be called from the subclass constructor using the `: base(...)` syntax. Methods can be redefined at the subclass level, though unlike in C++ this *requires* the use of a keyword (**new**) to indicate that the method is indeed a redefinition.

```
class Gadget
{
    public Gadget ( int n ) { ... }

    public void twist() { ... }
    // . . .
}

class FastGadget : Gadget
{
    public FastGadget( int n, double x ) : base( n ) { ... }

    public new void twist() { ... }
    // . . .
}

void play( Gadget gA, FastGadget fgB )
{
    gA.twist() ;           // Gadget's twist(), (even if gA actually refers to a FastGadget)
    fgB.twist() ;         // FastGadget's twist()
}

void test()
{
    FastGadget fgX, fgY ;
    // . . .
    play( fgX, fgY ) ;
}
```

static binding

Part III

OO Programming Aspects

III.1. Inheritance (OO style)

To enable dynamic binding there are two steps: mark the methods to be overridden as **virtual** (like C++), and mark the overriding method as **override** (peculiar to C#).

```
class Gadget
{
    public Gadget ( int n )    { ... }

    public virtual void twist() { ... }
    // . . .
}

class FastGadget : Gadget
{
    public FastGadget( int n, double x ) : base( n ) { ... }

    public override void twist() { ... }
    // . . .
}

void play( Gadget gA, FastGadget fgB )
{
    gA.twist() ;           // FastGadget's twist() - a virtual function call
    fgB.twist() ;         // FastGadget's twist()
}

void test()
{
    FastGadget fgX, fgY ;
    // . . .
    play( fgX, fgY ) ;
}
```

dynamic binding

III.2. Sealed classes

A **sealed** class is one that cannot serve as a base for further derivation. This is often done to provide a “floor” on dynamic binding so that the compiler can optimize it into static binding. For example, *System.String* is sealed.

III.3. Downcasting

There are two ways to downcast in C#, analogous to `dynamic_cast<T&>` and `dynamic_cast<T*>` in C++:

```
Gadget g1= new SlowGadget() ;
Gadget g2= new FastGadget() ;
SlowGadget sg ;

// Method 1: Explicit cast.

sg= (SlowGadget) g1 ; // succeeds
sg= (SlowGadget) g2 ; // fails: throws System.InvalidCastException

// Method 2: The as operator.

sg= g1 as SlowGadget ; // succeeds
sg= g2 as SlowGadget ; // fails: sets sg to null
```

III.4. System.Object

Every type in C# is a subtype of *System.Object*, even humble little `int` (which is really *System.Int32*). The C# alias for *System.Object* is `object`.

The *Equals* method of `object` is virtual, and tests for reference equality by default. *System.ValueType* (the mother of all value types) overrides it to test value equality.

The virtual method in `object` most commonly overridden by programmers is probably `ToString()`; this returns a concise string representation for the object.

```
class Gadget
{
    // . . .
    public override void string ToString() { return "Gadget " + n ; }
}
```

Some methods that take `objects`, such as *WriteLine*, invoke the object's *ToString()* method:

```
try
{
    // . . .
}
catch ( Exception e )
{
    co.WriteLine( e ) ; // writes e.ToString()
}
```

`Object` also has a *GetType()* method that returns information about the object's actual type; its own *ToString()* method is useful. For example, if `g` refers to an object constructed as a *FastGadget*, then `co.WriteLine(g.GetType())` prints "FastGadget" even if `g` was declared as a `Gadget`.

The `int GetHashCode()` method in `object` is handy for use in hash tables. It is often overridden to take advantage of the programmer's knowledge of the specific structure of the type.

III.5. Abstract classes

Abstract classes must be tagged with the **abstract** keyword. What we call “pure virtual functions” (**virtual** . . . =0) in C++ are called “abstract methods” and must also be tagged as **abstract**. (The keyword **abstract** implies they are virtual; you cannot also add the redundant keyword **virtual**.)

```
abstract class Pile
{
    public abstract void flatten() ;
    public void sayHey() { co.WriteLine( "Hey" ) ; }
}

class ConcretePile : Pile
{
    public override void flatten() { /* . . . */ }
}
```

```
static void Main()
{
    Pile p= new ConcretePile() ;
    p.sayHey() ;
    p.flatten() ;
}
```

III.6. Interfaces

All methods in an interface are public and abstract. The key difference from abstract classes is that a class can be a subclass of only *one* class (abstract or no) but can implement multiple interfaces.

```
interface IFlippable
{
    void flip() ;
} ;

interface ISwattable
{
    void swat() ;
    void swatComplete() ;
}

class Waffle
{
    // . . .
}

class BelgianWaffle : Waffle, IFlippable, ISwattable
{
    void flip() { /* . . . */ }
    void swat() { /* . . . */ }
    void swatComplete() { /* . . . */ }
    // .. .
}
```

base class name listed first

```
static void Main()
{
    IFlippable pf= new BelgianWaffle() ;
    ISwattable ps= pf as ISwattable ;
    // . . .
}
```

Note that the keyword **override** is not used when providing an implementation for an interface method.

III.7. Operator overloading

Unlike C++, where operator overloading is vital to polymorphism in the generic (template) programming model, C# uses operator overloading only as syntactic sugar. It is best to avoid it unless you are writing operator-intensive number-like types (like matrices, for example, – and even then it can be inefficient at runtime). (or the same reasons, it should be avoided in pure object-oriented C++.)

So instead of sketching how to do operator overloading (it's actually easier than in C++), we'll just mention some warnings. Some classes have overloaded the `==` operator to check for *value* equality even though classes are *reference* types; `string` does this. So if you really want to test reference equality you must be careful. It is best to use the static method `ReferenceEquals(o1,o2)` rather than `o1==o2`.

You might think that comparison operators (like `<`) could be usefully overloaded for ordered keys. But even `string` does not overload `<`! Instead, if you want your type to be used as an ordered key, then implement the `System.IComparable` interface; this just means you provide an `int CompareTo(object)` method, and ask whether `x.CompareTo(y) < 0` instead of `x<y`. This makes interoperability with other .NET languages that do not have operator overloading a lot easier.

One great relief: the assignment operator `=` cannot be overloaded in C#. (But if you've overloaded `+`, then `+=` is automatically overloaded, etc.)

III.8. Boxing

OO polymorphism allows us to have general containers. The most general would be a container of objects. See how easy we can use an array of objects:

```
object[] oar= new object[ 4 ] ;

oar[0]= 26.4 ;
oar[1]= true ;
oar[2]= "dog" ;
oar[3]= new Gadget( 21 ) ;

foreach ( object o in oar )
    co.WriteLine( o ) ;
```

Output (thanks to the ToString() method):

```
26.4
True
dog
Gadget 21
```

Behind this simple façade some important things are happening. Clearly an array of **objects** is an array of references to objects on the heap—essentially an array of pointers. (An array of different *value* objects would have slots of variable size, which would be quite unworkable.) Putting **strings** and **Gadgets** into the array is straightforward, as they are reference types. But it's different for value types like **double** and **bool** here. They are value types, so behind the scenes the system is “boxing” them for us into reference types.

Getting them out requires dynamic casting, of course:

```
double x= (double) oar[0] ;
bool b= (bool) oar[1] ;
string s= (string) oar[2] ;
Gadget g= (Gadget) oar[3] ;
```

The .NET collections framework (stacks, queues, hash tables, etc) is built around **objects**, and automatic boxing means we can easily use them with value types as well as reference types.