Introduction to C#

Overview

- C# language fundamentals
- Classes and objects
- Exceptions, events, and delegates
- Attributes, reflection, threads, XML comments

References

- Hanspeter Mössenböck, C# Tutorial, http://www.ssw.uni-linz.ac.at/Teaching/Lectures/CSharp/Tutorial/
- Jesse Liberty, “Programming C#”, O’Reilly, 2002
- Tom Archer, “Inside C#”, Microsoft Press, 2001
- Andrew Troelsen, “C# and the .NET Platform”, Apress, 2001

Features of C#

- Very similar to Java

  70% Java, 10% C++, 5% Visual Basic, 15% new

As in Java

- Object-orientation (single inheritance)
- Interfaces
- Exceptions
- Threads
- Namespaces (like Packages)
- Strong typing
- Garbage Collection
- Reflection

As in C++

- (Operator) Overloading
- Pointer arithmetic in unsafe code
- Some syntactic details

Language Concepts

- Syntax based on C/C++
  - Case-sensitive
  - White space means nothing
  - Semicolons (`;`) to terminate statements
  - Code blocks use curly braces (`{}`)
- Some features
  - Can create methods with a variable number of arguments
  - Parameters are passed by value (by default)
    - Can create methods that take parameters by reference
    - Can create methods with out-only parameters
  - Operator overloading and type converters
  - Type-safety and code verification
- Object oriented, code is structured using the *class* keyword

New Features in C#

- Reference and output parameters
- Stack-based objects
- Rectangular arrays
- Enumerations
- Attributes
- Unified type system

Syntactic Sugar

- Component-based programming
  - Properties
  - Indexers
  - Events
- Delegates
- Operator overloading
- foreach statement
- Boxing / unboxing

Concepts, first used in Delphi

Hello World

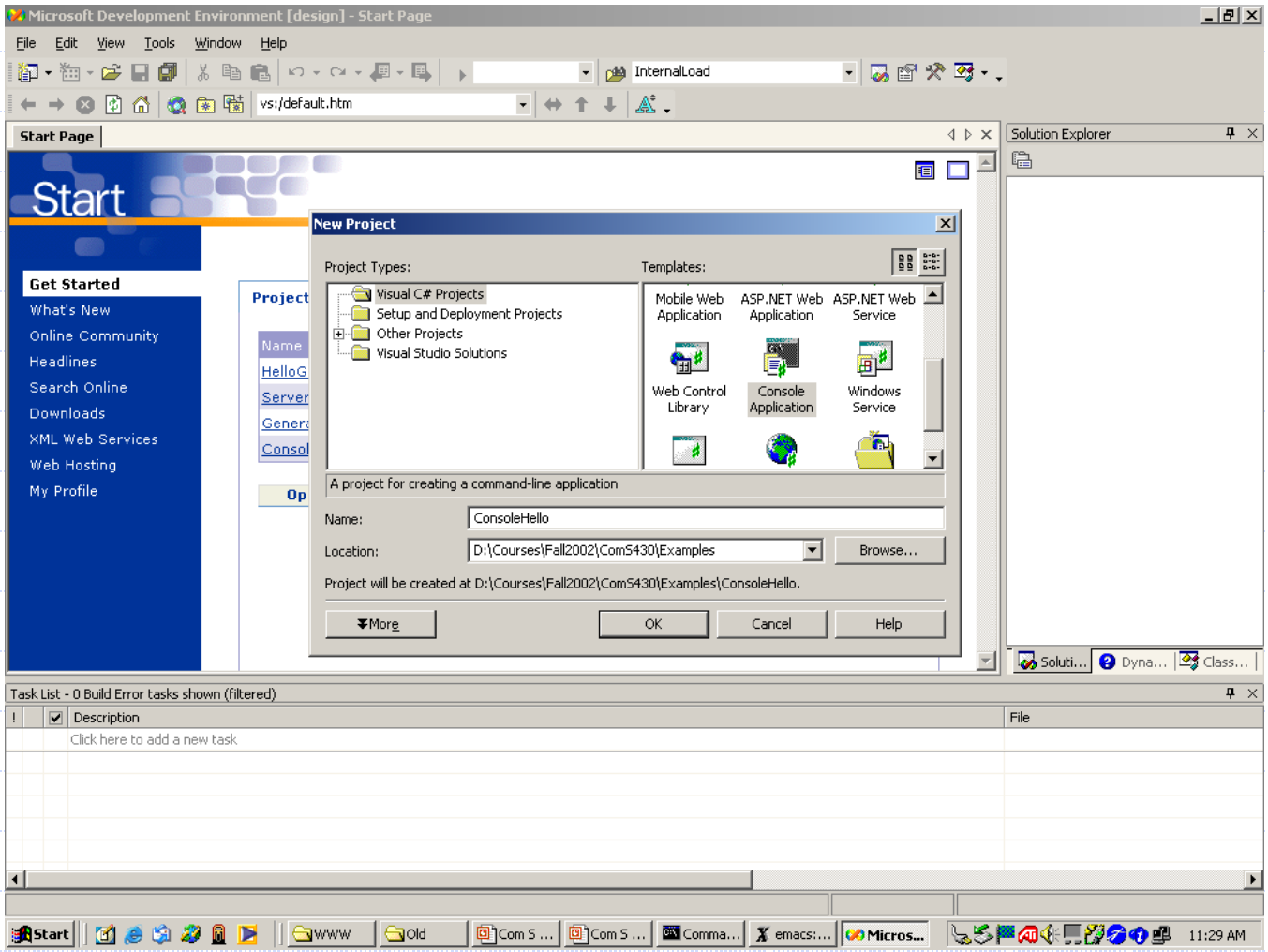
ConsoleHello.cs

```
using System;

class HelloWorld
{
    static void Main()
   {
      // Use the system console object
      Console.WriteLine( “Hello World!” );
   }
}
```

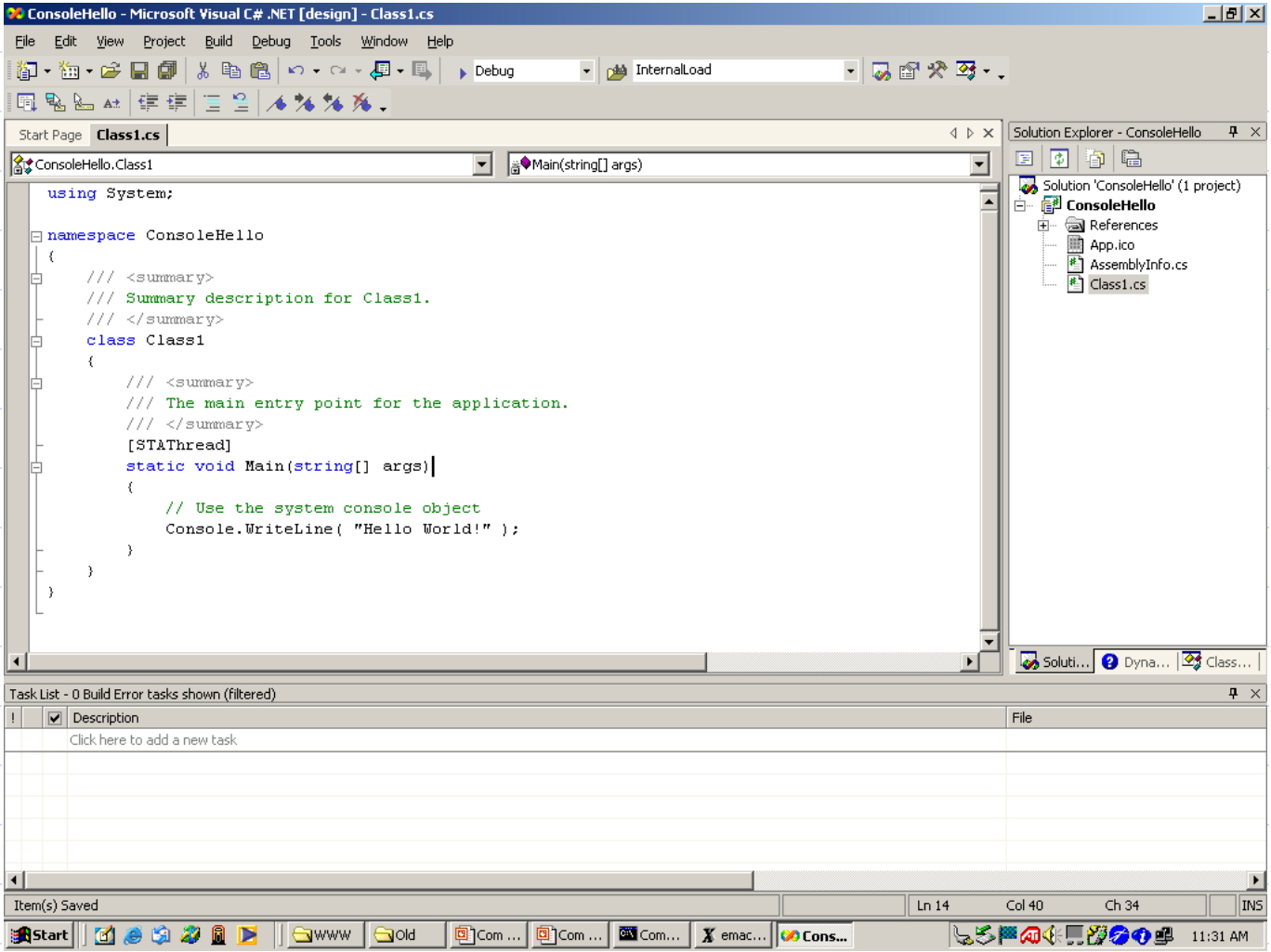
- Uses namespace *System*
- Entry point must be called Main
- Output goes to the console
- File name and class name need *not* be the same

```
C: \> csc ConsoleHello.cs
C: \> ConsoleHello
Hello World!
```

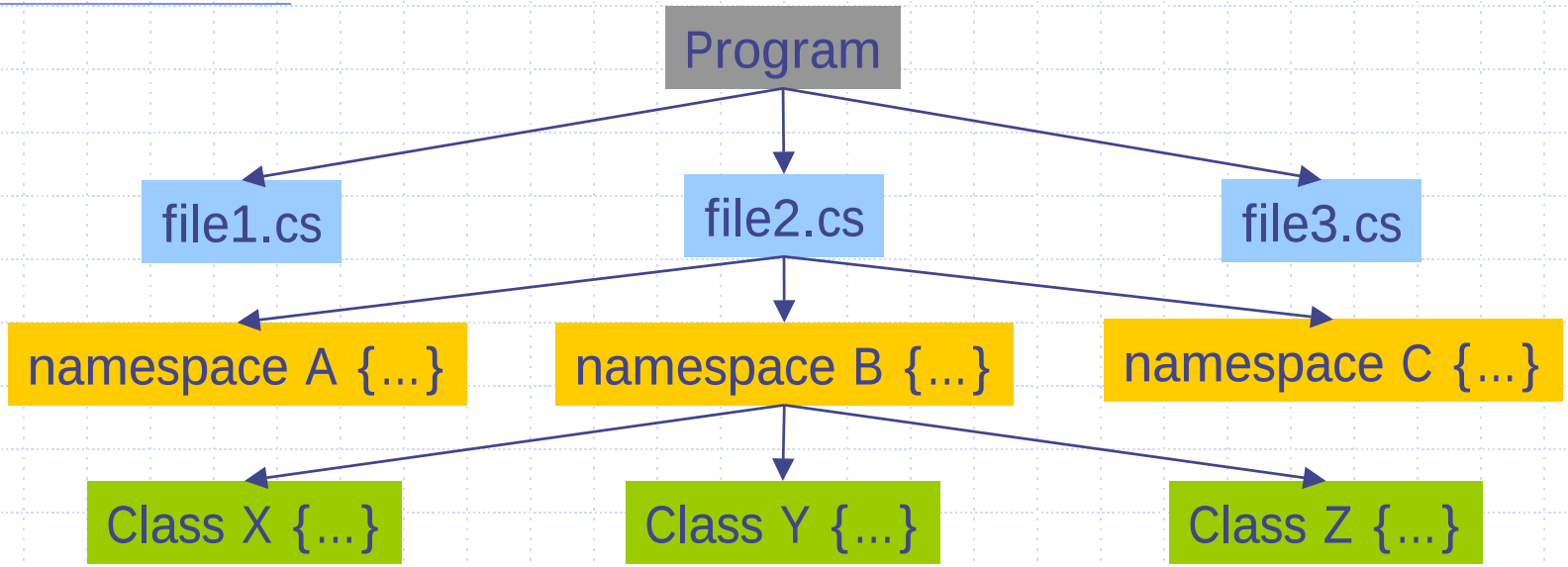
Developing “Hello World”



Namespace ConsoleHello



Structure of C# Programs



- If no namespace is specified ➔ anonymous default namespace.
- Namespaces may also contain structs, interfaces, delegates, and enums.
- Namespace may be “reopened” in other files.
- Simplest case: single class, single file, default namespace.

A Program Consisting of 2 Files

Counter.cs

```
public class Counter
{
    private int fValue = 0;

    public void Add( int aValue )
    {
       fValue = fValue + aValue;
    }

    public int Val
    {
       get { return fValue; }
    }
}
```

getter property

Prog.cs

```
using System;

class Prog
{
   static void Main()
   {
     Counter lCounter = new Counter();

     lCounter.Add( 3 );
     lCounter.Add( 5 );
     Console.WriteLine( "Val = " + lCounter.Val );
   }
}
```

use of counter class

automatic conversion to string

Multi-file Projects

- One Program:

  csc Counter.cs Prog.cs

  ➔ Prog.exe

  c:\> Prog

- Working with DLL's:

  csc /target:library Counter.cs

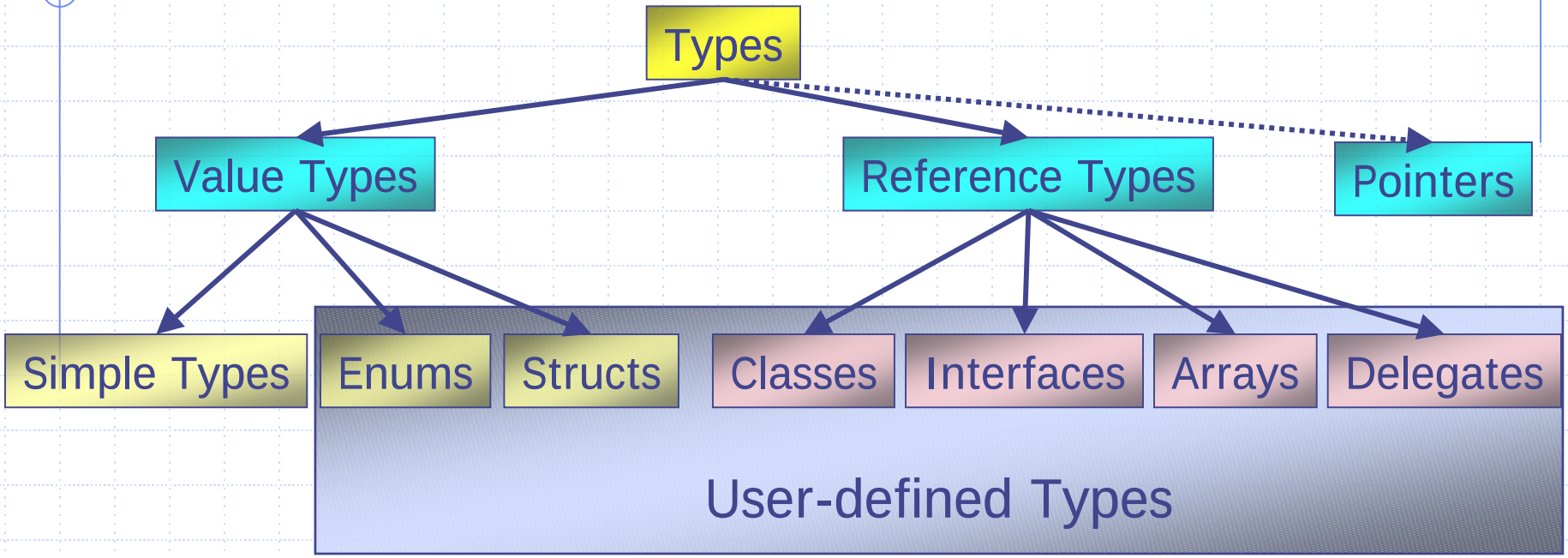
  ➔ Counter.dll

  requires "public" class Counter

  csc /reference:Counter.dll Prog.cs

  ➔ Prog.exe

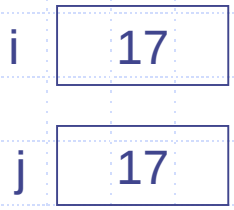
Unified Type System

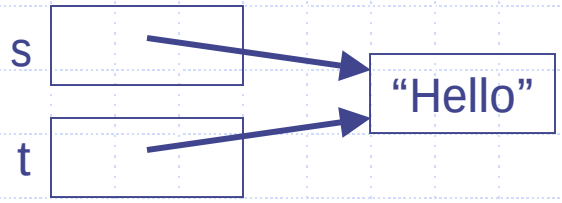


- All types are compatible with *Object*
  - Can be assigned to variables of type *Object*
  - All operations of type *Object* are applicable to them

Value Types vs. Reference Types

	Value Types	Reference Types
variable contains	value	reference
stored in	stack	heap
initialization	0, false, '\0'	null
assignment	copies of values	copies of references
example	int i = 17; int j = i;	string s = "Hello"; string t = s;





System.Object

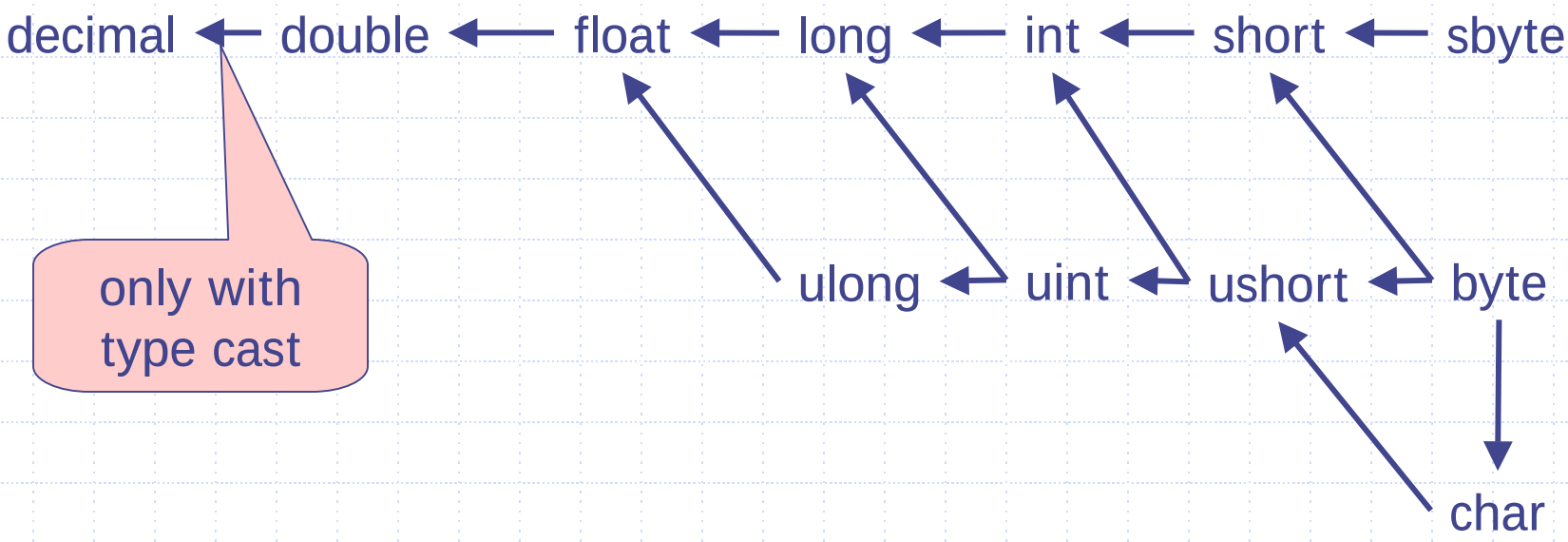
- The type *System.Object* is the root of all types.

Method Name	Description
bool Equals()	This method compares two object references to determine whether they are the exact same object.
int GetHashCode()	Retrieves the hash code specified for an object.
Type GetType()	Used with reflection methods to retrieve type information.
string ToString()	By default, this method is used to retrieve the name of the object. Should be overridden by derived class.
void Finalize()	This method is called by the runtime to allow for cleanup prior garbage collection. DO NOT OVERRIDE!
Object MemberwiseClone()	This member represents a shallow copy of the object. To support a deep copy, the *ICloneable* interface must be implemented to manually do cloning.

Built-in Types

Type	Size (in bytes)	.NET type	Description
byte	1	System.Byte	0 .. 255
char	2	System.Char	Unicode characters
bool	1	System.Boolean	**true** or **false**
sbyte	1	System.Sbyte	-128 .. 127
short	2	System.Int16	- 32,768 .. 32,767
ushort	2	System.UInt16	0 .. 65,535
int	4	System.Int32	-2,147,483,648 .. 2,147,483,647
uint	4	System.UInt32	0 .. 4,294,967,295
float	4	System.Single	1.5E-45 .. 3.4E38
double	8	System.Double	5E-324 .. 1.7E308
decimal	12	System.Decimal	1E-28 .. 7.9E28 (28 digits)
long	8	System.Int64	-2^{63} .. 2^{63} -1
ulong	8	System.UInt64	0 .. 2^{64} -1

Type Compatibility



Enumerations

List of named constants

Declaration (directly in a namespace)

```
enum Color { Red, Blue , Green }; // values 0, 1, 2
enum Access { User = 1, Group = 2, All = 4 };
enum ServingSizes : uint { Small = 1, Regular = 2, Large = 3 };
```

Use

```
Color lColor = Color.Blue; // enumeration constants must be qualified

Access lAccess = Access.User | Access.Group;
if ( (Access.User & lAccess) != 0)
   System.Console.WriteLine( “access granted” );
```

Operations on Enumerations

tests	if (c > Color.Red && c <= Color.Green) ...
+, -	c = c + 2;
++, --	c++
&	if ((c & Color.Red) == 0) ... // logical and
|	a = a | Access.Group; // logical or
~	a = ~Access.Group; // bitwise complement

Note:

- Enumerations cannot be assigned to *int* without a type cast.
- Enumeration types inherit from *Object*.
- Class *System.Enum* provides operations on enumerations.
- The compiler does not check if the result of an operation on enumerations yields a value enumeration value.

Arrays

One-dimensional arrays:

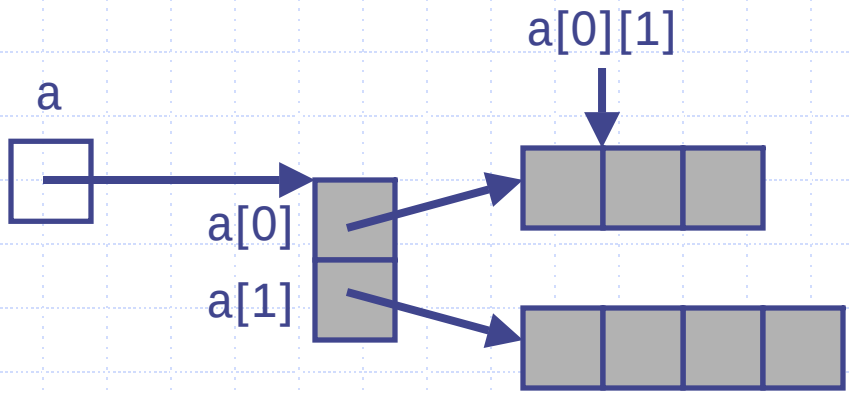
[] between type and variable name

```
int[] a = new int[3];
int[] b = new int[] {1, 2, 3};
int[] c = {4, 5, 6, 7};

SomeClass[] d = new SomeClass[10]; // array of references
SomeStruct[] e = new SomeStruct[10]; // array of values


int len = a.Length; // number of elements in a
```

Multidimensional Arrays

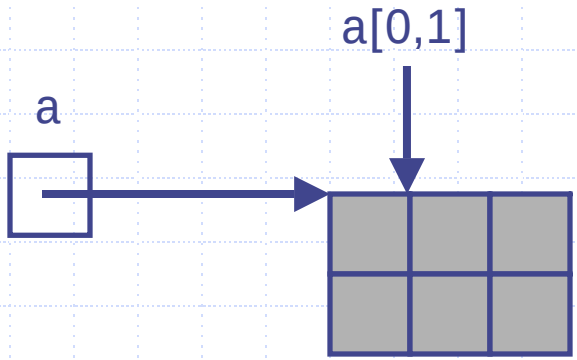
- Jagged arrays:

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

Int x = a[0][1];
Int len = a.Length; //2
Len = a[0].Length; //3
```



- Rectangular arrays:

```
Int[,] a = new int[2, 3]

Int x = a[0,1];
Int len = a.Length; // 6
Len = a.GetLength(0); // 2
```



Class *System.String*

- C# treats strings as first-class types that are flexible, powerful, and easy to use.

```
string s = “World”;
```

- Each string object is an immutable sequence of Unicode characters.
  - Strings can be concatenated with +: “Hello ” + s
  - Strings can be indexed: s[i]
  - Strings have a length: s.Length
  - Strings are reference types!
  - String values can be compared with == and !=.
  - The class *System.String* provides a huge set of string operations.

Structs

- Declaration:

```
struct Point
{
   public int x, y;                                          // fields
   public Point( int x, int y ) { this.x = x; this.y = y; }  // constructor
   public void MoveTo( int a, int b ) { x = a; y = b; }      // method
}
```

- Use:

```
Point p = new Point( 5, 6 );    // constructor initializes object on the
                                // stack
p.MoveTo( 30, 45 );             // method call
```

Classes

- Declaration:

```
class Point
{
   int x, y;                                                    // fields
   public Point( int x, int y ) { this.x = x; this.y = y; } // constructor
   public void MoveTo( int a, int b ) { x = a; y = b; }  // method
}
```

- Use:

```
Point p = new Point( 5, 6 );    // constructor initializes object on the
                                // heap
p.MoveTo( 30, 45 );             // method call
```

Classes versus Structs

Classes	Structs
reference types (objects stored on the heap)	value types (objects stored on the stack)
support inheritance (all classes are derived from Object)	No inheritance (but compatible with Object)
Can implement interfaces	Can implement interfaces
May have a destructor	No destructor allowed

Boxing and Unboxing

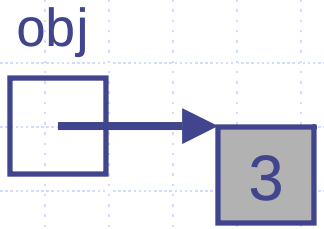
- Value types (e.g. int, struct, enum) are also compatible with *Object*!

Boxing

The assignment

```
Object obj =3;
```

wraps up the value 3 into a heap object.



Unboxing

The assignment

```
int x = (int)obj;
```

unwraps the value again.

Generic Container Types

- The boxing/unboxing mechanism allows the easy implementation of generic container types.

```
class Queue
{ ...
   public void Enqueue( Object obj ) {...}
   public Object Dequeue() {...}
   ... }
```

- This Queue can then be used for both reference types and value types.

```
Queue q = new Queue();
q.Enqueue( new String( “Hello World” ) );
q.Enqueue( 3 );
String s = (String)q.Dequeue();
int x = (int)q.Dequeue();
```

Operators

Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

left-associative

right-associative

Overflow Check

- Overflow is not checked by default.

```
int x = 1000000;
x = x * x; // -72737968, no error
```

- Overflow check can be turned on.

```
x = checked( x * x );  // Throws System.OverflowException

checked { ...
          x = x * x; // Throws System.OverflowException
          ...}
```

- Overflow check can also be turned on with a compiler switch.

```
csc /checked Test.cs
```

typeof and sizeof

- The *typeof* operator is used to obtain the System.Type object for a type.

```
Type t = typeof(int);
System.Console.WriteLine(t.Name); // ➔ Int32
```

- The *sizeof* operator is used to obtain the size, in byte, of a given type.
  - This operator can only be applied to value types.
  - This operator can only be used in an unsafe block.

```
unsafe { System.Console.WriteLine( sizeof(int) ); // ➔ 4 }

csc /unsafe Test.cs
```

Declarations

Program entities can be declared in a

- namespace
  - classes, interfaces, structs, enums, delegates
- class, interface, struct
  - fields, methods, properties, events, indexers, ...
- enum
  - enumeration constants
- block
  - local variables

Scoping and Visibility

Scoping rules:

- A name must not be declared twice in the same declaration space.
- Declarations may occur in arbitrary order.

  Exception: Local variables must be declared before they can be used.

Visibility rules:

- A name is only visible within its declaration space.

  Local variables are only visible after their point of declaration.
- The visibility can be restricted by modifiers (e.g. private, protected).

Namespaces

X.cs

```
namespace A{
        ... classes ...
      ... interfaces ...
        ... structs ...
        ... enums ...
      ... delegates ...
  namespace B {
    ... // full name: A.B
  }
              ...
}
```

Y.cs

```
namespace A{
                ...
  namespace B { ... }
                ...
}
namespace C { ... }
```

- Equally named namespaces in different files constitute a single declaration space.
- Nested namespaces constitute a declaration space of their own.

Using Namespaces

- Foreign namespaces
  - must be either imported (e.g. using System;)
  - or specified in a qualified name (e.g. System.Console)
- Most programs need the namespace System.
  - Therefore, you need to specify using System.

```
using System;
using System.Configuration;
using System.IO;
using System.Data;
using System.Data.OleDb;
```

Statement Lists and Blocks

Statement	Example

Statement lists and block statements	`static void Main()`
	`{`
	`  F();`
	`  G();`
	`  {`
	`  H();`
	`  I();`
	`  }`
	`}`

Labeled Statement and goto

Statement	Example

Labeled statements and *goto* statements	`static void Main(string[] args)`
	`{`
	`  if (args.Length == 0)`
	`  goto done;`
	
	`  Console.WriteLine(args.Length);`
	
	`done:`
	`  Console.WriteLine("Done");`
	`}`

Local Constant Declaration

Statement	Example

Local constant declarations	`static void Main()`
	`{`
	`  const float pi = 3.14f;`
	`  const int r = 123;`
	`  Console.WriteLine(pi * r * r);`
	`}`

Local Variable Declaration

Statement	Example

Local variable declarations	`static void Main()`
	`{`
	`  int a;`
	`  int b = 2, c = 3;`
	`  a = 1;`
	`  Console.WriteLine(a + b + c);`
	`}`

Expression Statement

Statement	Example

Expression statements	`static int F(int a, int b)`
	`{`
	`    return a + b;`
	`}`
	
	`static void Main()`
	`{`
	`    F(1, 2);  // Expression statement`
	`}`

if Statement

Statement	Example
if statements	

```
static void Main(string[] args)
{
    if (args.Length == 0)
        Console.WriteLine("No args");
    else
        Console.WriteLine("Args");
}
```

Test expression must be a Boolean value!

switch Statements

Statement	Example
switch statements	

```
static void Main(string[] args)
{
    switch (args.Length)
    {
        case 0:
            Console.WriteLine("No args");
            break;
        case 1:
            Console.WriteLine("One arg ");
            break;
        default:
            int n = args.Length;
            Console.WriteLine("{0} args", n);
            break;
    }
}
```

No Fall-Through!

while Statements

Statement	Example
while statements	

```
static void Main(string[] args)
{
    int i = 0;
    while (i < args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

do Statements

Statement	Example

do statements	`static void Main()`
	`{`
	`  string s;`
	`  do { s = Console.ReadLine(); }`
	`  while (s != "Exit");`
	`}`

for Statements

Statement	Example

for statements	`static void Main(string[] args)`
	`{`
	`  for (int i = 0; i < args.length; i++)`
	`  Console.WriteLine(args[i]);`
	`}`

foreach Statements

Statement	Example

foreach statements	`static void Main(string[] args)`
	`{`
	`    foreach (string s in args)`
	`        Console.WriteLine(s);`
	`}`

break Statements

Statement	Example

break statements	`static void Main(string[] args)`
	`{`
	`  int i = 0;`
	`  while (true)`
	`  {`
	`  if (i == args.Length)`
	`  break;`
	`  Console.WriteLine(args[i++]);`
	`  }`
	`}`

continue Statements

Statement	Example
continue statements	

```
static void Main(string[] args)
{
   int i = 0;
   while (true)
   {
      Console.WriteLine(args[i++]);
      if (i < args.Length)
         continue;
      break;
   }
}
```

return Statements

Statement	Example
return statements	

```
static int F(int a, int b)
{
    return a + b;
}

static void Main()
{
    Console.WriteLine( F(1, 2) );
    return;
}
```

throw and try Statements

Statement	Example

throw statements
and *try* statements
	`static int F(int a, int b)`
	`{`
	`  if (b == 0)`
	`  throw new Exception("Divide by zero");`
	`  return a / b;`
	`}`
	
	`static void Main()`
	`{`
	`  try {`
	`  Console.WriteLine(F(5, 0));`
	`  }`
	`  catch(Exception e) {`
	`  Console.WriteLine("Error");`
	`  }`
	`}`

lock Statements

Statement	Example

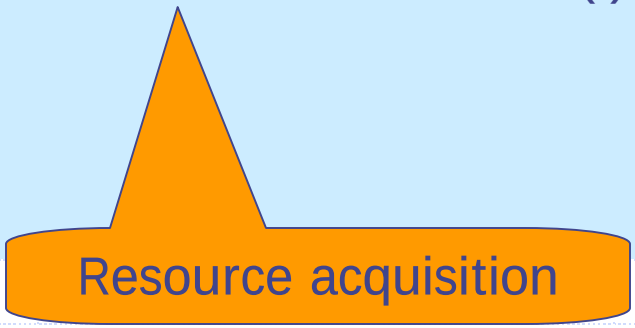
lock statements	`static void Main()`
	`{`
	`  A a = ...;`
	`  lock(a)`
	`  {`
	`  a.P = a.P + 1;`
	`  }`
	`}`

Lock does not support the full array of features found in the Monitor class!

using Statements

Statement	Example

using statements	`static void Main()`
	`{`
	`  using (Resource r = new Resource())`
	`  {`
	`  r.F();`
	`  }`
	`}`



The using statement obtains one or more resources, executes a statement, and then disposes of the resource.

Exceptions

- Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions.
  - All exceptions must be represented by an instance of a class type derived from *`System.Exception`*.
  - A *finally* block can be used to write termination code that executes in both normal execution and exceptional conditions.
  - System-level exceptions such as overflow, divide-by-zero, and null dereferences have well defined exception classes and can be used in the same way as application-level error conditions.

Causes of Exceptions

Exception can be thrown in two different ways:

- A *`throw`* statement throws an exception immediately and unconditionally. Control never reaches the statement immediately following the `throw`.
- Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation throws a *`System.DivideByZeroException`* if the denominator is zero.

try Statement

```
FileStream s = null;
try {
        s = new FileStream(curName, FileMode.Open);
        ...
    }
catch (FileNotFoundException e) {
        Console.WriteLine("file {0} not found", e.FileName); }
catch (IOException) {
        Console.WriteLine("some IO exception occurred"); }
catch {
        Console.WriteLine("some unknown error occurred"); }
finally {
         if (s != null) s.Close();
}
```

Exception Handling

- *catch* clauses are checked in sequential order.
- *finally* clause is always executed (if present).
- Exception parameter name can be omitted in a *catch* clause.
- Exception type must be derived from *System.Exception*.
- If exception parameter is missing, *System.Exception* is assumed.

System.Exception

- The type *System.Exception* is the root of all exceptions.

Method Name	Description
string Message	Gets a message that describes the current exception.
string StackTrace	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
string Source	Gets or sets the name of the application or the object that causes the error.
MethodBase TargetSite	Gets the method that throws the current exception.
string ToString()	Creates and returns a string representation of the current exception.

Common Exception Classes

Exception Name (System.)	Description
`ArithmeticException`	Base class for exceptions that occur during arithmetic operations.
`DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`IndexOutOfRangeException`	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
`InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived types fails at run time.
`NullReferenceException`	Thrown when a *`null`* reference is used in a way that causes the referenced object to be required.
`OverflowException`	Thrown when an arithmetic operation in a `checked` context overflows
`TypeInitializationException`	Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

No Throws Clause in Signature

Java:

```
void myMethod() throws IOException {
                    ... throw new IOException(); ... }
```

Callers of *myMethod* must either

- catch *IOException* or
- specify *IOExceptions* in their own signature

C#:

```
void myMethod() {
                ... throw new IOException(); ... }
```

Callers of *myMethod* may handle *IOException* or not.

+ convenient
- less robust

Contents of Classes or Structs

```
class C {
    ... fields, constants ...              // for object-oriented programming
    ... methods ...
    ... constructors, destructors ...

    ... properties ...                     // for component-oriented programming
    ... events ...

    ... indexers ...                       // for amenity
    ... overloaded operators ...

    ... nested types (classes, interfaces, delegates, etc.) ...
}
```

Visibility Modifiers

Access Modifier	Restrictions
public	No restriction. Members marked *public* are visible to any element in the declaration domain.
private	The member is only accessible in the program text of the defined class.
protected	Members marked *protected* are visible in the program text of the defining class and all its subclasses.
internal	Members marked *internal* are accessible to all methods of any class in the current assembly.
protected internal	Either *protected* or *internal* applies.

Fields and Constants

```
class C {
    int fValue = 0;
            Field:      - Initialization is optional.
                        - Initialization must not access other fields or
                          methods of the same type.
                        - Fields of a struct must not be initalized.

    const long cSize = ((long)int.MaxValue + 1)/4;
            Constant: - Value must be computable at compile time.

    readonly DateTime fDate;
            Read OnlyField:   - Must be initialized in their declaration or in
                                in a constructor.
                              - Value does not need to be computable at
                                compile time .
                              - Consumes a memory location (like a field).
}
```

Static Fields and Constants

- A static member belongs to a class, not to a particular object.
- All instances share the same static member.

```
class Rectangle
{
    static Color fDefaultColor;      // once per class
    static readonly int fScale;      // once per class
    int x, y, height, width;         // once per object
    ...
}
```

Unlike in Java, static constants are not allowed.

Methods

```
class C
{
    int fSum = 0;
    int n = 0;

    public void Add( int x )          // procedure
    {
        fSum = fSum + x;
        n++;
    }

    public float Mean()               // function
    {
        return (float)fSum / n;
    }

}
```

Class-based methods are annotated with *static*.

Call-by-value Parameters

- Formal parameter is a copy of the actual parameter.
- The actual parameter is an expression.

```
void Inc( int x )
{
    x = x + 1;
}

void f()
{
    int val = 3;
    Inc( val );
}
```

local change of “x”

Value of “val” is still 3.

Call-by-reference Parameters

- The formal parameter is an alias for the actual parameter (address of actual parameter is passed).
- The actual parameter must be a variable.

```
void Inc( ref int x )
{
    x = x + 1;
}

void f()
{
    int val = 3;
    Inc( ref val );
}
```

global change of “x”

Value of “val” is 4.

Out Parameters

- Out parameters are similar to reference parameters, but no value is passed by the caller.
- Out parameters must not be used in the method before they have been assigned a valid value.

```
void Read( out int first, out int next )
{
    first = System.Console.Read();
    next = System.Console.Read();
}

void f()
{
    int first, next;
    Read( out first, out next );
}
```

Parameters carry no value.

Parameter Arrays

- The last n parameters may be a sequence of values of a certain type.
- They are represented be a parameter array – a mechanism to model a variable number of parameters.

```
void Add( out int sum, params int[] val )
{
    sum = 0;
    foreach ( int i in val )
        sum += i;
}
```

```
Add( out sum, 1, 2, 3, 4 ); // sum == 10
```

Note: The keyword params cannot be used in for ref and out parameters.

Method Overloading

Methods of a class may have the same name

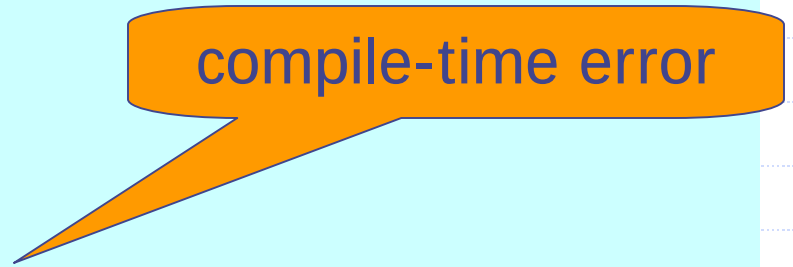
- If they have different numbers of parameters, or
- If they have different parameter types, or
- If they have different parameter kinds (value, ref/out)

- Overloaded methods must not differ only in their function types, in the presence of *params* or in *ref* versus *out*!

Examples

```
void F (int x) {...}
void F (char x) {...}
void F (int x, long y) {...}
void F (long x, int y) {...}
void F (ref int x) {...}
```

```
int i; long n; short s;

F(i); // F(int x)
F('a'); // F(char x)
F(i, n); // F(int x, long y)
F(n, s); // F(long x, int y);
F(i, s); // cannot distinguish F(int x, long y) and F(long x, int y)
F(i, i); // cannot distinguish F(int x, long y) and F(long x, int y)
```



Constructors for Classes

- Constructors can be overloaded.
- A constructor may call another constructor with this specified in the constructor head, not in the body as in Java.
- Before a constructor is called, fields are possibly initialized.

```
class Rectangle
{
    int x, y, width, height;

    public Rectangle (int x, int y, int w, int h)
       { this.x = x; this.y = y; width = w; height = h; }
    public Rectangle (int w, int h) : this(0, 0, w, h) {}
    public Rectangle () : this(0, 0, 0, 0) {}
    ...
}
```

Default Constructor

- If no constructor has been specified for a given class, the compiler generates a parameter-less default constructor:

```
class C{ int x; }

C c = new C(); // ok
```

- If a constructor has been specified for a given class, no default constructor is generated:

```
class C{ int x;
         public C( int y ) { x = y; } }

C c1 = new C(); // error
C c2 = new C( 3 ); // ok
```

Constructors for Structs

- For **every** struct the compiler generates a parameter-less default constructor, even if there are other constructors.

  The default constructor initializes all fields with their default value.

- The programmer must not declare a parameter-less constructor for structs, due to implementation reasons of the CLR.

Static Constructors

- Static constructors can be used for both classes and structs.
- Static constructors must be parameter-less and have no public or private modifier.
- Only one static constructor is allowed per class or struct.
- The static constructor is invoked once before this type is used for the first time.

Destructors

You cannot override Finalize()!

- A destructor corresponds to finalizers in Java.
- The destructor is called for an object before it is garbage collected.
- You must not specify the public of private modifier for a destructor.
- It is in general not recommended (since dangerous) to use destructors. Rather use the method *Dispose*.

Dispose Design Pattern

```
class C
{
   ...
   ~C() { ... cleanup object, call super destructors ... }
   ...
}
```

Properties

- Properties are syntactic sugar for get/set methods.
- Properties are used as *virtual* or *smart fields*, since the programmer can specify additional code that is associated with the accessor methods.

```
class Data
{
    FileStream fStream;

    public String FileName
    {
        set { fStream = new FileStream( value, FileMode.Create ); }

        get { return fStream.Name;}
    }
}
```

default input parameter

get or set can be omitted

Why Properties?

- Properties allow the specification of read-only and write-only fields.
- Properties are used to validate a field when it is assigned a value (setter method) and when its value is accessed (getter method).
- Properties are especially useful in component-oriented programming.
- Properties help to build reliable and robust software.

Indexer

- An indexer is a C# construct that can be used to access collections contained by a class using the [] syntax for arrays.
- Like properties an indexer uses a get() and set() method to specify its behavior.

```
class File
{
    FileStream fStream;

    public int this [int index]
    {
        get { fStream.Seek( index, SeekOrigin.Begin );
              return fStream.Read(); }

        set { fStream.Seek( index, SeekOrigin.Begin );
              fStream.WriteByte((byte)value); }
    }
}
```

Overloaded Indexers

- Indexers can be overload with different index types.

```
class ListBoxText
{
    String[] fStrings;

    String FindString( String aString ) { ... }

    public String this [int index] {
          get { return fString[index]; }
        }

    public String this [String index] {
          get { FindString( index ); }
        }

}
```

Nested Types

- Nested types are used for auxiliary classes that should be hidden.
  - Members of an inner class can access all members of the outer class (even private members).
  - Members of the outer class can access only public members of the inner class.
  - Members of other classes can access members of an inner class only if it is public.
- Nested types can also be structs, enums, interfaces and delegates.

A Nested Class Example

```
class A
{
    int x;
    B b = new B(this);
    public void f() { b.f(); }

    public class B
    {
        A a;

        public B(A a) { this.a = a; }
        public void f() { a.x = ...; ... a.f(); }
    }
}
```

```
class C
{
    A a = new A();
    A.B b = new A.B(a);
}
```

Inheritance

- C# supports single inheritance:
  - A class can only inherit from one base class.
  - A class can implement multiple interfaces.
- A class can only inherit from a class, not from a structs.
- Structs cannot inherit from another type, but they can implement multiple interfaces.
- A class without explicit base class inherits from *Object*.

Inheritance Example

```
class A // base class
{
    int a;
    public A() {...}
    public void F() {...}
}

class B : A  // subclass (inherits from A, extends A)
{
    int b;
    public B() {...}
    public void G() {...}
}
```

Class B inherits *a* and *F*(), it adds *b* and *G*()

- constructors are not inherited
- inherited methods can be overridden

Object Assignment

```
class A {...}
class B : A {...}
class C: B {...}
```

```
A a = new A(); // static type of a: declaration type A
               // dynamic type of a: the type of the
               // object in a (also A)

a = new B(); // dynamic type of a is B
a = new C(); // dynamic type of a is C

B b = a; // forbidden; compilation error
```

Runtime Type Checks

```
class A {...}
class B : A {...}
class C: B {...}
```

```
a = new C();
if (a is C) ... // true, if dynamic type of a is C or a subclass;
                // otherwise false

if (a is B) ... // true
if (a is A) ... // true, but warning because it makes no sense

a = null;
if (a is C) ... // false: if a == null, (a is T) always returns false
```

Checked Type Casts

```
class A {...}
class B : A {...}
class C: B {...}
```

```
A a = new C();
B b = (B) a; // if (a is B ) static type of a is B in this expression;
             // else exception
C c = (C) a;

a = null;
c = (C) a; // ok, null can be casted to any reference type
```

cast

```
A a = new C();
B b = a as B;  // if (a is B) b = (B)a; else b = null;
C c = a as C;

a = null;
c = a as C;     // c == null
```

as

Method Overriding

- Methods need to be declared as virtual in order to be overridden in subclasses.
- Overriding methods must be declared as override.
- Method signatures must be identical
  - Same number and types of parameters (including function type),
  - Same visibility (public, protected, ...).
- Properties and indexers can also be overridden (virtual, override).
- Static methods cannot be overridden.

Overriding Example

```
class A
{
    public void F() {...}          // cannot be overridden
    public virtual void G() {...} // can be overridden in a subclass
}
```

enables dynamic method lookup

```
class B : A
{
    public void F() {...}     // warning: hides inherited F() → use new
    public void G() {...}     // warning: hides inherited G() → use new
    public override void G() // ok: overrides inherited G
        {  ... base.G();  ... }  // calls inherited G()
}
```

Hiding

- Members of a class can be declared as new in a subclass.
- They *hide* inherited members with the same name.

```
class A
{
    public int x;
    public void F() {...}
    public virtual void G() {...}
}
```

```
class B : A
{
    public new int x;
    public new void F() {...}
    public new void G() {...}
}
```

Dynamic Binding & Hiding

```
class A { public virtual void M() { Console.WriteLine("A"); } }

class B : A { public override void M() { Console.WriteLine("B"); } }

class C : B { public new virtual void M() { Console.WriteLine("C"); } }

class D : C { public override void M() { Console.WriteLine("D"); } }
```

```
C c = new D();
c.M(); // “D”

A a = new D();
a.M(); // “B”
```

Static type A is used to find M().

Constructors & Inheritance

Implicit call of the base class constructor			Explicit call

class A {
...
}
class B : A {
public B(int x)
{...}
}
	class A {
	public A() {...}
	}
	class B : A {
	public B(int x)
	{...}
	}
		class A {
		public A(int x) {...}
		}
		class B : A {
		public B(int x)
		{...}
		}
			class A {
			public A(int x) {...}
			}
			class B : A {
			public B(int x)
			: base(x)
			{...}
			}

B b = new B(3);	B b = new B(3);	B b = new B(3);	B b = new B(3);
OK - default const. A() - B(int x)	**OK** - A() - B(int x)	**Error!** - no explicit call of the A() constructor - default constr. A() does not exist	**OK** - A(int x) - B(int x)

Abstract Classes

- The *abstract* modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class.
  - An abstract class cannot be instantiated directly.
  - An abstract class is permitted (but not required) to contain abstract members.
  - An abstract class cannot be sealed.

Abstract Class - Example

```
abstract class Stream
{
    public abstract void Write( char aChar );

    public void WriteString( String aString )
    {
        foreach ( char ch is aString)
            Write( ch );
    }
}
```

In C++: =0;

concrete class

```
class File : Stream
{
    public override void Write( char aChar )
    {  ... write aChar to disk ... }
}
```

Abstract Properties & Indexers

```
abstract class Sequence
{
    public abstract void Add( Object aObj );          // method
    public abstract String Name { get; }              // property
    public abstract Object this [int i] { get; set; } // indexer
}
```

```
class List : Sequence
{
    public override void Add( Object aObj ) { ... }
    public override String Name { get { ... }; }
    public override Object this [int i] { get { ... }; set { ... }; }
}
```

Note: Overridden properties and indexers must have the same get and set methods as in the base class.

Sealed Classes

```
sealed class Account : Asset
{
    long fValue;
    public void Deposit( long aAmount ) { ... }
    public void Withdraw( long aAmount ) { ... }
}
```

- A sealed class cannot be extended (same as final classes in Java).
- Methods can be marked sealed individually.

Why do we want to use sealed classes?

- Security (no modifications in subclasses)
- Efficiency (methods may be called using static binding)

Interfaces

- Interface = purely abstract class; only signatures, no implementation
- May contain methods, properties, indexers and events (no fields, constants, constructors, destructors, operators, and nested types)
- Interface members are implicitly *public abstract virtual*.
- Interface members must not be static.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

Interfaces - Example

```
public interface IList : ICollection, IEnumerable
{
    int Add( Object aObj );                    // method
    bool IsReadOnly { get; }                   // property
    Object this [int i] { get; set; }          // indexer
    ...
}
```

```
class MyClass : MyBaseClass, IList, ICollection, IEnumerable
{
    public int Add( Object aObj );
    public bool IsReadOnly { get { ... }; }
    public Object this [int i] { get { ... }; set { ... }; }
    ...
}
```

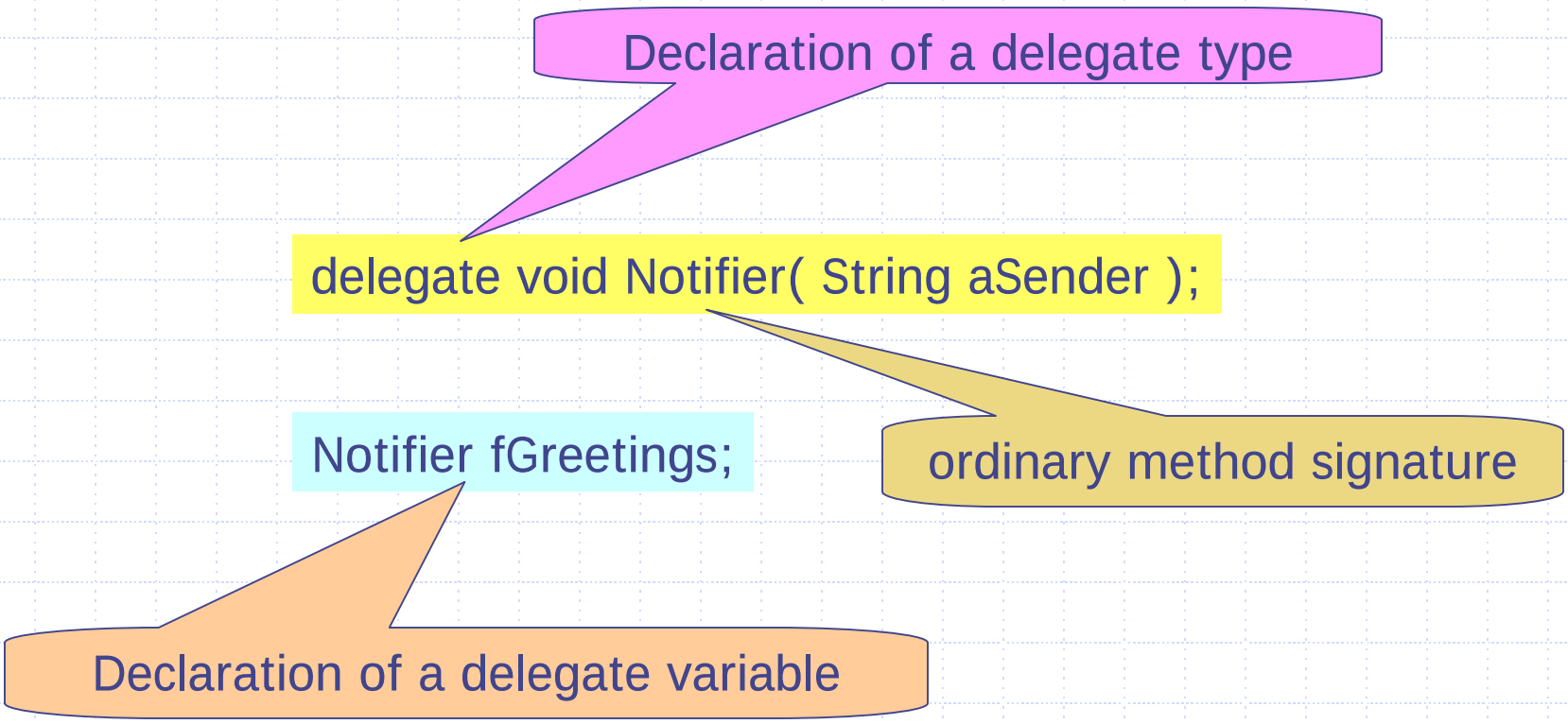
Interface Implementation

- A class can inherit a single base class, but can implement multiple interfaces.
- A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, and indexer) must be implemented or inherited from a base class.
- Implemented interface methods must not be declared as override.
- Implemented interface methods can be declared virtual or abstract (i.e., an interface can be implemented by an abstract class).

Delegates

- Delegates enable scenarios that other languages (e.g. C++, Pascal, and Modula) have addressed with function pointers.
- Delegates are fully object oriented.
- Delegates encapsulate both an object instance and a method.

Delegate Declaration



Delegate Use

```
class X
{
    ...
    public void SayHello( String aSender )
    {
        System.Console.WriteLine( “Hello from ” + aSender );
    }
    ...
}

X lObj = new X();
fGreetings = new Notifier( lObj.SayHello );
```

Declaration of a delegate handler

Assigning a handler to a delegate variable

```
fGreetings( “Daisy”);
```

Calling a delegate variable

Multicast Delegates

- A delegate variable can hold multiple values at the same time.

Note:

- If a multicast delegate is a function, the value of the last call is returned.
- If a multicast delegate has an out parameter, the parameter of the last call is returned.

A Multicast Delegate Example

```
delegate void D(int x);

class C
{
    public static void M1(int i)
    { Console.WriteLine("C.M1: " + i); }

    public static void M2(int i)
    { Console.WriteLine("C.M2: " + i); }

    public void M3(int i)
    { Console.WriteLine("C.M3: " + i); }
}
```

delegate D

handler

Delegate Test

```
...
    static void Main()
    {
        D cd1 = new D( C.M1 );
        D cd2 = new D( C.M2 );
        D cd3 = cd1 + cd2;
        cd3(10);                    // call M1 then M2
        cd3 += cd1;
        cd3(20);                    // call M1, M2, then M1
        cd3 -= cd1;                 // remove last M1
        cd3(40);                    // call M1 then M2
        cd3 -= cd2;
        cd3(60);                    // call M1
        cd3 -= cd2;                 // impossible removal is benign
        cd3(60);                    // call M1
    cd3 -= cd1;                     // invocation list is empty
//  cd3(70);                        // System.NullReferenceException thrown
    }
...
```

Test Output

```
C.M1:    10
C.M2:    10
C.M1:    20
C.M2:    20
C.M1:    20
C.M1:    40
C.M2:    40
C.M1:    60
C.M1:    60
```

Events

- Events are special delegate variables.
- Only the class that declares the event can fire it.

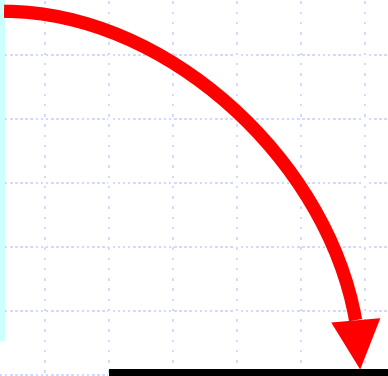
```
class Model
{
    public event Notifier notifyViews;
    public void Change() { ... notifyViews("Model"); }
}
```

Events Example

```
class View1
{
    public View1( Model aModel )
    { aModel.notifyViews += new Notifier( this.Update1 ); }
    void Update1( String aSender )
    { Console.WriteLine( aSender + " was changed" ); }
}

class View2
{
    public View2( Model aModel )
   { aModel.notifyViews += new Notifier( this.Update2 ); }
    void Update2( String aSender )
    { Console.WriteLine( aSender + " was changed" ); }
}
```

Event – Example cont.

```
class Test
{
    static void Main()
    {
        Model m = new Model();
        new View1(m);
        new View2(m);
        m.Change();
    }
}
```



```
Model was changed
Model was changed
```

Attributes

- Attributes provide a convenient way to specify user-defined meta information about program elements:
  - Attributes can be attached to types, members, assemblies, etc.
  - Attributes extend predefined attributes such as *public*, *sealed* or *abstract*.
  - Attributes are implemented as classes that are derived from *System.Attribute*.
  - Attributes are stored in the metadata of an assembly.
  - Attributes are often used by CLR services (serialization, remoting, COM interoperability).
  - Attributes can be queried at run time using reflection.

Using Attributes

Makes this class serializable.

```
[Serializable]
class X { ... }
```

```
[Serializable][Obsolete]
class X { ... }
```

Multiple attributes

```
[Serializable, Obsolete]
class X { ... }
```

Some Attribute Targets

Target	Usage
All	Applied to any of the following elements: assembly, class, constructor, delegate, enum, event, field, interface, method, module, parameter, property, return value, or struct.
Assembly	Applied to the assembly itself
Class	Applied to instances of the class
Delegate	Applied to delegate methods
Method	Applied to a method
Parameter	Applied to a parameter of a method
ReturnValue	Applied to a return value

Attribute Example

- We define a PlatformAttribute that is used to specify that the application can only run on a particular application.
- We need to define a new attribute class that is derived from class Attribute.
- We add a method to the application that checks for the existence of the PlatformAttribute.

PlatformTypeAttribute

```
public enum PlatformTypes { Win2000 = 0x0001, WinXP = 0x0002 }

public class PlatformAttribute : Attribute
{
   private PlatformType fPlatform;

   public PlatformAttribute( PlatformType aPlatform ) { fPlatform = aPlatform; }

   public override Boolean Match( object obj ) { ... } // true if objects match

   public override Boolean Equals( object obj ) { ... } // true if objects are equal

   public override Int32 GetHashCode() { return (Int32)fPlatform; }
}
```

Test Application

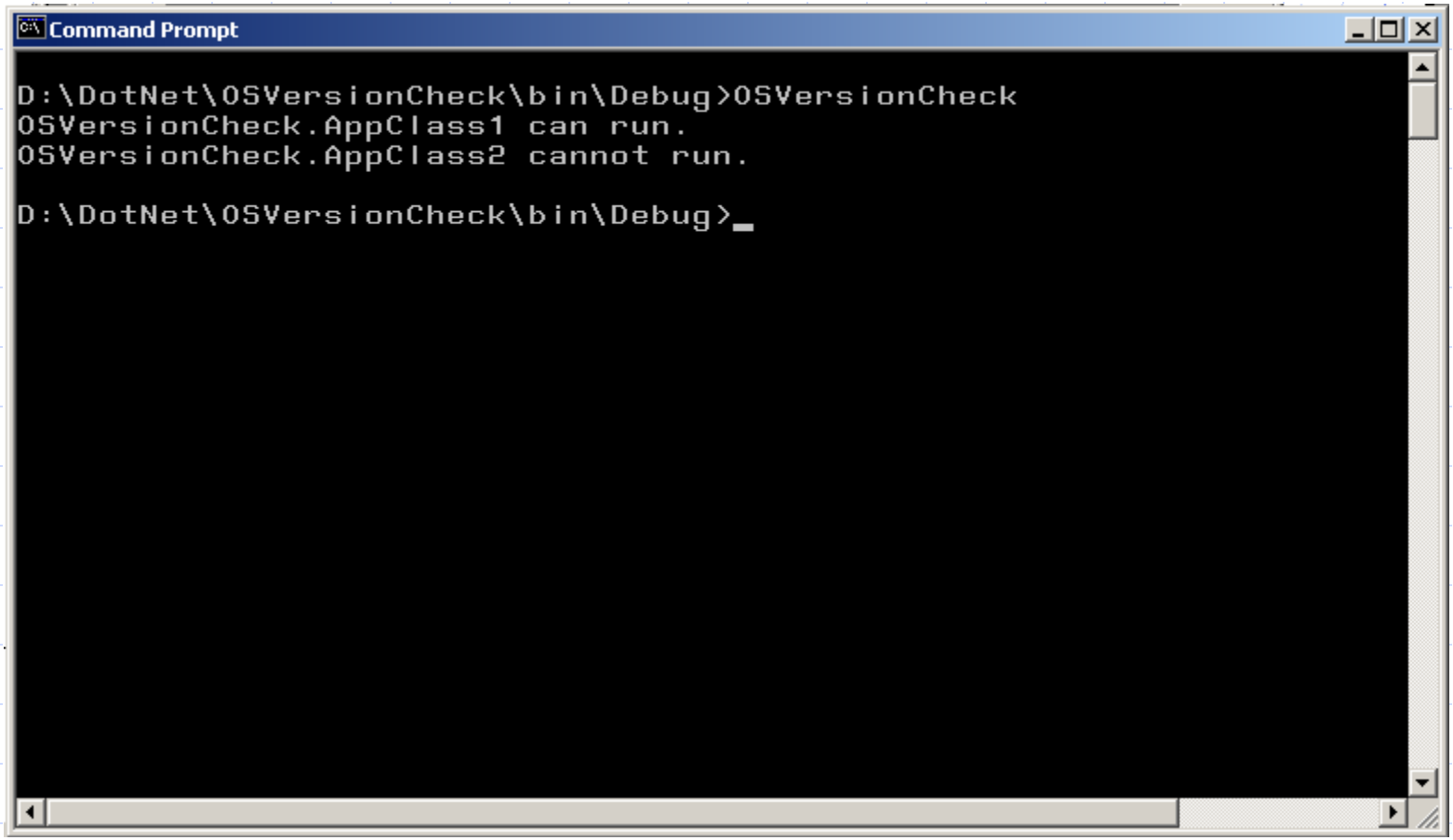

```
[Platform(PlatformTypes.Win2000)]
class AppClass1 {}

[Platform(PlatformTypes.WinXP)]
class AppClass2 {}

public class MainClass
{
        static void Main(string[] args)
        {
                CanRunApplication( new AppClass1() );
                CanRunApplication( new AppClass2() );
        }
```

MainClass.CanRunApplication

```
public static void CanRunApplication( object obj )
{
    Attribute lCheck = new PlatformAttribute( PlatformTypes.Win2000 );
    Attribute lApp = Attribute.GetCustomAttribute( obj.GetType(),
                                                   typeof(PlatformAttribute),
                                                   false );

    if ( (lApp != null) && lCheck.Match( lApp ) )
        Console.WriteLine( "{0} can run.", obj.GetType() );
    else
        Console.WriteLine( "{0} cannot run.", obj.GetType() );
}
```

Run time reflection

Output


Command Prompt
D:\DotNet\OSVersionCheck\bin\Debug>OSVersionCheck
OSVersionCheck.AppClass1 can run.
OSVersionCheck.AppClass2 cannot run.
D:\DotNet\OSVersionCheck\bin\Debug>

XML Documentation Comments

- C# supports a new Documentation Comment style, with three slash marks (///).

Example:

```
/// ... comment ...
class C {
    /// ... comment ...
    public int f;

    /// ... comment ...
    public void foo() {...}
}
```

XML Generation

- The C# compiler processes the Documentation Comments into an XML file:

  csc /doc:MyFile.xml MyFile.cs

- The compiler
  - Checks if comments are complete and consistent, e.g. if one parameter of a method is documented, all parameters must be documented; Names of program elements must be spelled correctly.
  - Generates an XML file with the commented program elements.
- XML can be formatted for the Web browser with XSLT.

Example

```
/// <summary> A counter for accumulating values and computing the mean
    value.</summary>
class Counter {
        /// <summary>The accumulated values</summary>
        private int value;
        /// <summary>The number of added values</summary>
        public int n;
        /// <summary>Adds a value to the counter</summary>
        /// <param name="x">The value to be added</param>
        public void Add(int x) { value += x; n++; }
        /// <summary>Returns the mean value of all accumulated values</summary>
        /// <returns>The mean value, i.e. <see cref="value"/>
        ///   / <see cref="n"/></returns>
        public float Mean() { return (float)value / n; }
}
```

XML Main Tags

<summary>
short description of a program element </summary>

<remarks>
extensive description of a program element </remarks>

<param name="*ParamName*">
description of a parameter </param>

<returns>
description of the return value </returns>

Nested Tags

Tags that are used within other tags:

<exception [cref="*ExceptionType*"]>
used in the documentation of a method: describes an exception </exception>

<example> *sample code* </example>

<code> *arbitrary code* </code>

<see cref="*ProgramElement*"> *name of a crossreference link* </see>

<paramref name="*ParamName*"> *name of a parameter* </paramref>