C# Basics & Overview

Edited by Faik Kerem Örs
CS408 – Computer Networks

Extracted from (with very minor modifications)
H.Mössenböck
University of Linz, Austria
moessenboeck@ssw.uni-linz.ac.at

Contents

Introduction to C#

Advanced C#

References:

- B.Albahari, P.Drayton, B.Merrill: **C# Essentials**. O'Reilly, 2001
- S.Robinson et al: **Professional C#**, Wrox Press, 2001
- Online documentation on the .NET SDK CD

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
- Dynamic loading of code
- ...

As in C++

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

Hello World

File Hello.cs

```
using System;

class Hello {

    static void Main() {
        Console.WriteLine("Hello World");
    }

}
```

- uses the namespace *System*
- entry point must be called *Main*
- output goes to the console
- file name and class name need *not* be identical

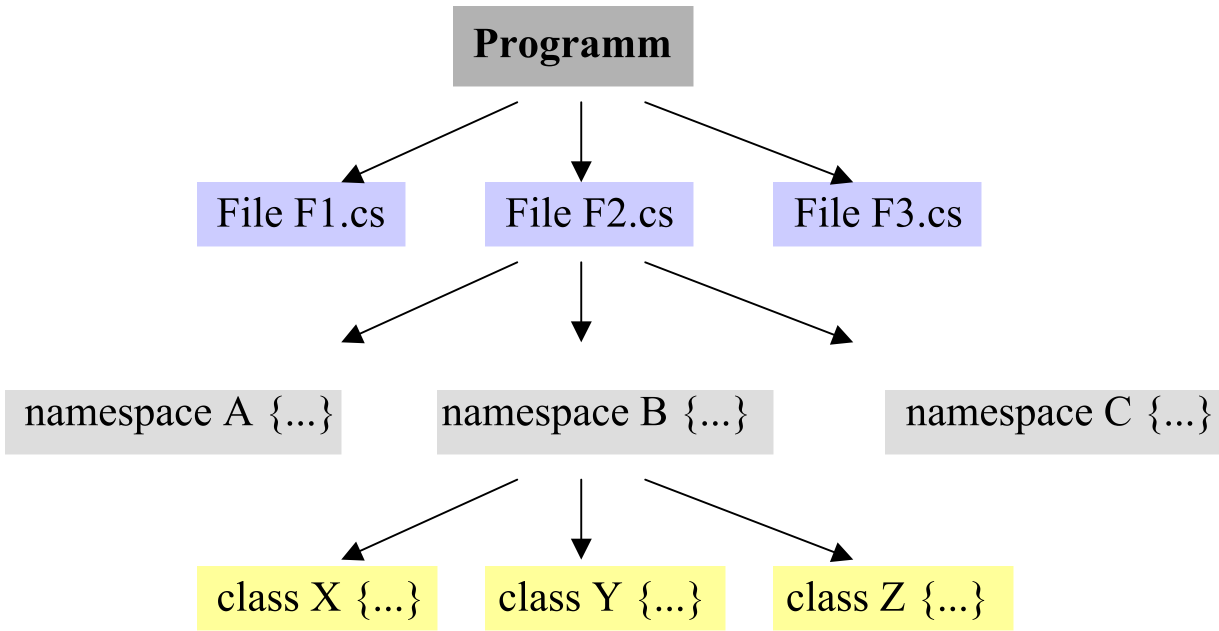
Compilation (in the Console window)

```
csc Hello.cs
```

Execution

```
Hello
```

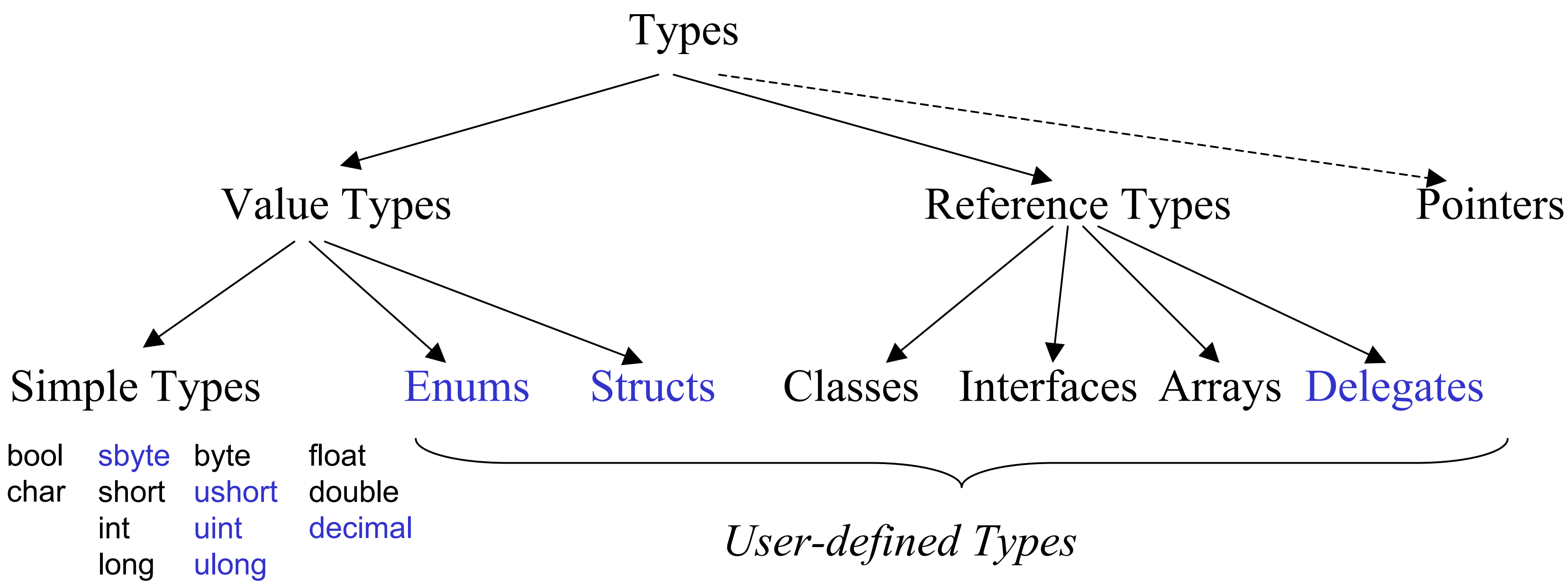
Structure of C# Programs



- 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

Types

Unified Type System

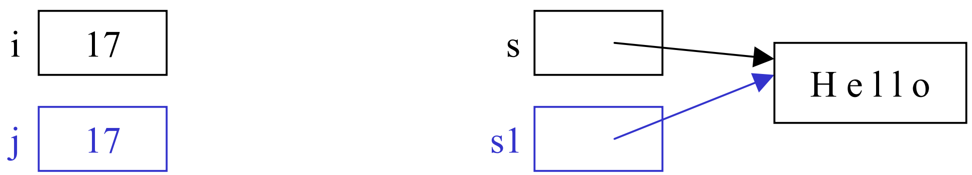


All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

Value Types versus Reference Types

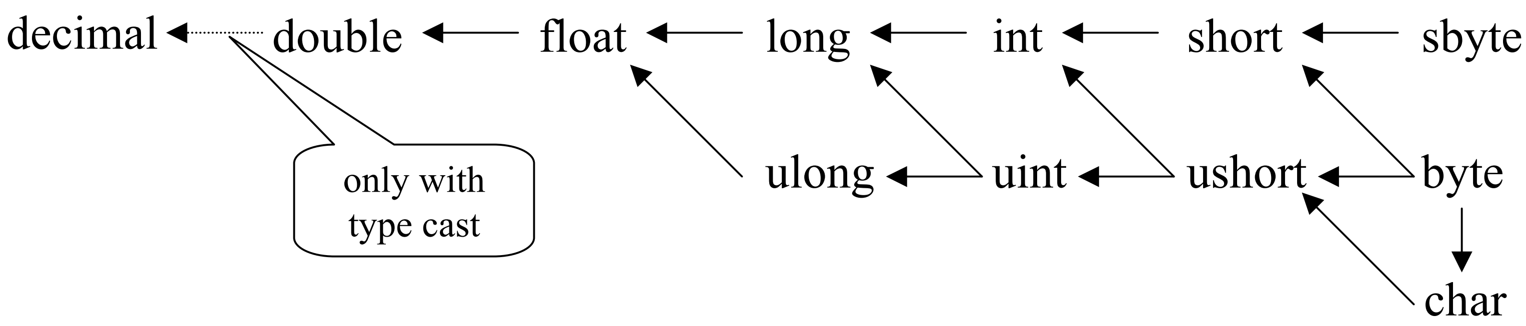
	Value Types	Reference Types
variable contains	value	reference
stored on	stack	heap
initialisation	0, false, '\0'	null
assignment	copies the value	copies the reference
example	int i = 17; int j = i;	string s = "Hello"; string s1 = s;



Simple Types

	Long Form	in Java	Range
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2147483648 .. 2147483647
uint	System.UInt32	---	0 .. 4294967295
long	System.Int64	long	-2^{63} .. $2^{63}-1$
ulong	System.UInt64	---	0 .. $2^{64}-1$
float	System.Single	float	±1.5E-45 .. ±3.4E38 (32 Bit)
double	System.Double	double	±5E-324 .. ±1.7E308 (64 Bit)
decimal	System.Decimal	---	±1E-28 .. ±7.9E28 (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode character

Compatibility Between Simple Types



Arrays

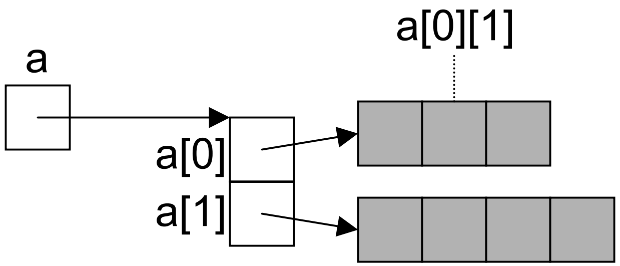
One-dimensional Arrays

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
SomeClass[] d = new SomeClass[10];    // Array of references
SomeStruct[] e = new SomeStruct[10];  // Array of values (directly in the array)

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

Multidimensional Arrays

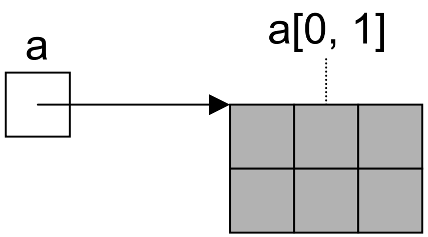
Jagged (like in Java)

```
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 (more compact, more efficient access)

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

int x = a[0, 1];
int len = a.Length;        // 6
len = a.GetLength(0); // 2
len = a.GetLength(1); // 3
```



Class System.String

Can be used as standard type *string*

```
string s = "Alfonso";
```

Note

- Strings are immutable (use *StringBuilder* if you want to modify strings)
- Can be concatenated with +: `"Don " + s`
- Can be indexed: `s[i]`
- String length: `s.Length`
- Strings are reference types => reference semantics in assignments
- but their values can be compared with == and != : `if (s == "Alfonso") ...`
- Class *String* defines many useful operations:
  CompareTo, *IndexOf*, *StartsWith*, *Substring*, ...

Expressions

Operators and their Priority

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	c?x:y
Assignment	= += -= *= /= %= <<= >>= &= ^= |=

Operators on the same level are evaluated from left to right

typeof and sizeof

typeof

- Returns the *Type* descriptor for a given <u>type</u>
  (the *Type* descriptor of an <u>object</u> *o* can be retrieved with *o.GetType()*).

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

sizeof

- Returns the size of a type in bytes.
- Can only be applied to <u>value</u> types.
- Can only be used in an <u>unsafe</u> block (the size of structs may be system dependent).
  Must be compiled with csc /unsafe xxx.cs

```
unsafe {
    Console.WriteLine(sizeof(int));
    Console.WriteLine(sizeof(MyEnumType));
    Console.WriteLine(sizeof(MyStructType));
}
```

Declarations

Declaration Space

The program area to which a declaration belongs

Entities can be declared in a ...

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

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 are 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 (private, protected, ...)

Blocks

Various kinds of blocks

```
void foo (int x) {                  // method block
        ... local variables ...

        {                           // nested block
            ... local variables ...
        }

        for (int i = 0; ...) {      // structured statement block
            ... local variables ...
        }
}
```

Note

- The declaration space of a block includes the declaration spaces of nested blocks.
- Formal parameters belong to the declaration space of the method block.
- The loop variable in a for statement belongs to the block of the for statement.
- The declaration of a local variable must precede its use.

Declaration of Local Variables

```
void foo(int a) {
	int b;
	if (...) {
		int b;		// error: b already declared in outer block
		int c;		// ok so far, but wait ...
		int d;
		...
	} else {
		int a;		// error: a already declared in outer block
		int d;		// ok: no conflict with d from previous block
	}
	for (int i = 0; ...) {...}
	for (int i = 0; ...) {...}	// ok: no conflict with i from previous loop
	int c;			// error: c already declared in this declaration space
}
```

Statements

Simple Statements

Empty statement

```
;                                   // ; is a terminator, not a separator
```

Assigment

```
x = 3 * y + 1;
```

Method call

```
string s = "a,b,c";
string[] parts = s.Split(',');     // invocation of an object method (non-static)

s = String.Join(" + ", parts);  // invocation of a class method (static)
```

if Statement

```
if ('0' <= ch && ch <= '9')
	val = ch - '0';
else if ('A' <= ch && ch <= 'Z')
	val = 10 + ch - 'A';
else {
	val = 0;
	Console.WriteLine("invalid character {0}", ch);
}
```

switch Statement

```
switch (country) {
    case "Germany": case "Austria": case "Switzerland":
        language = "German";
        break;
    case "England": case "USA":
        language = "English";
        break;
    case null:
        Console.WriteLine("no country specified");
        break;
    default:
        Console.WriteLine("don't know language of {0}", country);
        break;
}
```

Type of switch expression
numeric, char, enum or string (null ok as a case label).

No fall-through!
Every statement sequence in a case must be terminated with break (or return, goto, throw).

If no case label matches → default
If no default specified → continuation after the switch statement

Loops

while

```
while (i < n) {
      sum += i;
      i++;
}
```

do while

```
do {
      sum += a[i];
      i--;
} while (i > 0);
```

for

```
for (int i = 0; i < n; i++)
      sum += i;
```

Short form for

```
int i = 0;
while (i < n) {
    sum += i;
    i++;
}
```

foreach Statement

For iterating over collections and arrays

```
int[] a = {3, 17, 4, 8, 2, 29};
foreach (int x in a) sum += x;
```

```
string s = "Hello";
foreach (char ch in s) Console.WriteLine(ch);
```

```
Queue q = new Queue();
q.Enqueue("John"); q.Enqueue("Alice"); ...
foreach (string s in q) Console.WriteLine(s);
```

Jumps

break; For exiting a loop or a switch statement.
There is no break with a label like in Java (use *goto* instead).

continue; Continues with the next loop iteration.

goto case 3: Can be used in a switch statement to jump to a case label.

```
myLab:
...
goto myLab;
```

Jumps to the label *myLab*.
Restrictions:
- no jumps into a block
- no jumps out of a finally block of a try statement

return Statement

Returning from a void method

```
void f(int x) {
    if (x == 0) return;
    ...
}
```

Returning a value from a function method

```
int max(int a, int b) {
    if (a > b) return a; else return b;
}

class C {
    static int Main() {
        ...
        return errorCode;        // The Main method can be declared as a function;
    }                            // the returned error code can be checked with the
                                 // DOS variable errorlevel
}
```

Classes and Structs

Contents of Classes or Structs

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

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

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

   ... nested types (classes, interfaces, structs, enums, delegates) ...
}
```

Classes

```
class Stack {
    int[] values;
    int top = 0;

    public Stack(int size) { ... }

    public void Push(int x) {...}
    public int Pop() {...}
}
```

- Objects are allocated on the heap (classes are reference types)
- Objects must be created with *new*

```
Stack s = new Stack(100);
```

- Classes can inherit from *one* other class (single code inheritance)
- Classes can implement multiple interfaces (multiple type inheritance)

Classes

Declaration

```
class Rectangle {
	Point origin;
	public int width, height;
	public Rectangle() { origin = new Point(0,0); width = height = 0; }
	public Rectangle (Point p, int w, int h) { origin = p; width = w; height = h; }
	public void MoveTo (Point p) { origin = p; }
}
```

Use

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
r.MoveTo(new Point(3, 3));
```

Structs

```
struct Point {
    int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public MoveTo(int x, int y) {...}
}
```

- Objects are allocated on the stack not on the heap (structs are value types)
  - \+ efficient, low memory consumption, no burden for the garbage collector.
  - \- live only as long as their container (not suitable for dynamic data structures)
- Can be allocated with new

```
Point p;                // fields of p are not yet initialized
Point q = new Point();
```

- Fields must not be initialized at their declaration

```
struct Point {
    int x = 0;          // compilation error
}
```

- Parameterless construcors cannot be declared
- Can neither inherit nor be inherited, but can implement interfaces

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; }         // methods
}
```

Use

```
Point p = new Point(3, 4);    // constructor initializes object on the stack
p.MoveTo(10, 20);             // method call
```

Differences Between Classes and 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 destructors allowed

Visibility Modifiers (excerpt)

public visible where the declaring namespace is known

- Members of interfaces and enumerations are public by default.
- Types in a namespace (classes, structs, interfaces, enums, delegates) have default visibility *internal* (visible in the declaring assembly)

private only visible in declaring class or struct

- Members of classes and structs are private by default (fields, methods, properties, ..., nested types)

Example

```
public class Stack {
    private int[] val;          // private is also default
    private int top;            // private is also default
    public Stack() {...}
    public void Push(int x) {...}
    public int Pop() {...}
}
```

Parameter Passing

Parameters

Value Parameters (input values)

```
void Inc(int x) {x = x + 1;}
void f() {
    int val = 3;
    Inc(val); // val == 3
}
```

- "call by value"
- formal parameter is a copy of the actual parameter
- actual parameter is an expression

ref Parameters (transition values)

```
void Inc(ref int x) { x = x + 1; }
void f() {
    int val = 3;
    Inc(ref val); // val == 4
}
```

- "call by reference"
- formal parameter is an alias for the actual parameter
  (address of actual parameter is passed)
- actual parameter must be a variable

out Parameters (output values)

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

- similar to ref parameters
  but no value is passed by the caller.
- must not be used in the method before
  it got a value.

Variable Number of Parameters

Last n parameters may be a sequence of values of a certain type.

keyword *params*

array type

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

params cannot be used for *ref* and *out* parameters

Use

```
Add(out sum, 3, 5, 2, 9);  // sum == 19
```

Exceptions

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();
}
```

- *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

Properties

e.Message	the error message as a string; set in *new Exception(msg);*
e.StackTrace	trace of the method call stack as a string
e.Source	the application or object that threw the exception
e.TargetSite	the method object that threw the exception
...	

Methods

e.ToString()	returns the name of the exception
...	

Throwing an Exception

By an invalid operation (implicit exception)

Division by 0
Index overflow
Acess via a null reference
...

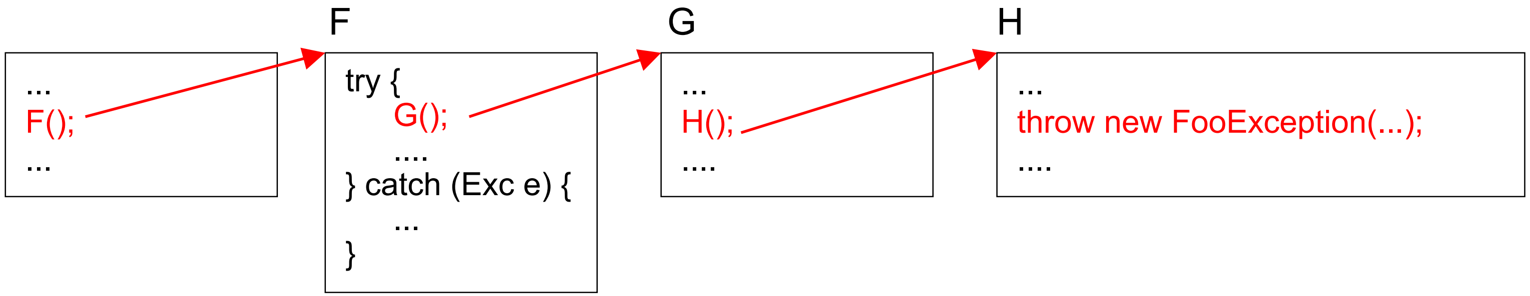
By a throw statement (explicit exception)

```
throw new FunnyException(10);

class FunnyException : ApplicationException {
    public int errorCode;
    public FunnyException(int x) { errorCode = x; }
}
```

Exception Hierarchy (excerpt)

Exception
- **SystemException**
  - ArithmeticException
    - DivideByZeroException
    - OverflowException
    - ...
  - NullReferenceException
  - IndexOutOfRangeException
  - InvalidCastException
  - ...
- **ApplicationException**
  - ... custom exceptions
  - ...
- **IOException**
  - FileNotFoundException
  - DirectoryNotFoundException
  - ...
- **WebException**
- ...

Searching for a catch Clause



Caller chain is traversed backwards until a method with a matching catch clause is found.
If none is found => Program is aborted with a stack trace

Exceptions don't have to be caught in C# (in contrast to Java)

No distinction between
- *checked exceptions* that have to be caught, and
- *unchecked exceptions* that don't have to be caught

Advantage: convenient
Disadvantage: less robust software

No Throws Clause in Method 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

More…

(Not covered)

Enumerations

List of named constants

Declaration (directly in a namespace)

```
enum Color {red, blue, green}    // values: 0, 1, 2
enum Access {personal=1, group=2, all=4}
enum Access1 : byte {personal=1, group=2, all=4}
```

Use

```
Color c = Color.blue;     // enumeration constants must be qualified

Access a = Access.personal | Access.group;
if ((Access.personal & a) != 0) Console.WriteLine("access granted");
```

Boxing and Unboxing

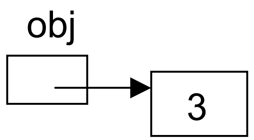
Value types (int, struct, enum) are also compatible with *object*!

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

Boxing/Unboxing

Allows the implementation of generic container types

```
class Queue {
	...
	public void Enqueue(object x) {...}
	public object Dequeue() {...}
	...
}
```

This *Queue* can then be used for reference types <u>and</u> value types

```
Queue q = new Queue();

q.Enqueue(new Rectangle());
q.Enqueue(3);

Rectangle r = (Rectangle) q.Dequeue();
int x = (int) q.Dequeue();
```

Namespaces

File: X.cs

```
namespace A {
    ... Classes ...
    ... Interfaces ...
    ... Structs ...
    ... Enums ...
    ... Delegates ...
    namespace B {  // full name: A.B
        ...
    }
}
```

File: 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 on their own.

Using Other Namespaces

Color.cs

```
namespace Util {
    public enum Color {...}
}
```

Figures.cs

```
namespace Util.Figures {
    public class Rect {...}
    public class Circle {...}
}
```

Triangle.cs

```
namespace Util.Figures {
    public class Triangle {...}
}
```

```
using Util.Figures;

class Test {
    Rect r;         // without qualification (because of using Util.Figures)
    Triangle t;
    Util.Color c;   // with qualification
}
```

Foreign namespaces

- must either be imported (e.g. *using Util;)*
- or specified in a qualified name (e.g. *Util.Color*)

Most programs need the namespace System => using System;

Fields and Constants

```
class C {
```

```
int value = 0;
```

Field
- Initialization is optional
- Initialization must not access other fields or methods of the same type
- Fields of a struct must not be initialized

```
const long size = ((long)int.MaxValue + 1) / 4;
```

Constant
- Value must be computable at compile time

```
readonly DateTime date;
```

Read Only Field
- Must be initialized in their declaration or in a constructor
- Value needs not be computable at compile time
- Consumes a memory location (like a field)

```
}
```

Access within C

```
... value ... size ... date ...
```

Access from other classes

```
C c = new C();
... c.value ... c.size ... c.date ...
```

Static Fields and Constants

Belong to a class, not to an object

```
class Rectangle {
	static Color defaultColor;	// once per class
	static readonly int scale;	// -- " –
	// static constants are not allowed
	int x, y, width,height;		// once per object
	...
}
```

Access within the class

... defaultColor ... scale ...

Access from other classes

... Rectangle.defaultColor ... Rectangle.scale ...

Methods

Examples

```
class C {
    int sum = 0, n = 0;

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

    public float Mean() {              // function (must return a value)
        return (float)sum / n;
    }
}
```

Access within the class

```
this.Add(3);
float x = Mean();
```

Access from other classes

```
C c = new C();
c.Add(3);
float x = c.Mean();
```

Static Methods

Operations on class data (static fields)

```
class Rectangle {
	static Color defaultColor;

	public static void ResetColor() {
		defaultColor = Color.white;
	}
}
```

Access within the class

ResetColor();

Access from other classes

Rectangle.ResetColor();

Constructors for Classes

Example

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

```
Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle(2, 5);
Rectangle r3 = new Rectangle(2, 2, 10, 5);
```

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

Default Constructor

If no constructor was declared in a class, the compiler generates a parameterless default constructor:

```
class C { int x; }
C c = new C();      // ok
```

The default constructor initializes all fields as follows:

numeric	0
enum	0
bool	false
char	'\0'
reference	null

If a constructor was declared, <u>no</u> default constructor is generated:

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

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

Constructors for Structs

Example

```
struct Complex {
	double re, im;
	public Complex(double re, double im) { this.re = re; this.im = im; }
	public Complex(double re) : this (re, 0) {}
	...
}
```

```
Complex c0;                      // c0.re and c0.im are still uninitialized
Complex c1 = new Complex();      // c1.re == 0, c1.im == 0
Complex c2 = new Complex(5);     // c2.re == 5, c2.im == 0
Complex c3 = new Complex(10, 3); // c3.re == 10, c3.im == 3
```

- For every struct the compiler generates a parameterless default constructor (even if there are other constructors).
  The default constructor zeroes all fields.

- Programmers must not declare a parameterless constructor for structs (for implementation reasons of the CLR).

Static Constructors

Both for classes and for structs

```
class Rectangle {
    ...
    static Rectangle() {
        Console.WriteLine("Rectangle initialized");
    }
}
```

```
struct Point {
    ...
    static Point() {
        Console.WriteLine("Point initialized");
    }
}
```

- Must be parameterless (also for structs) and have no *public* or *private* modifier.
- There must be just one static constructor per class/struct.
- Is invoked once before this type is used for the first time.

Destructors

```
class Test {

    ~Test() {
        ... finalization work ...
        // automatically calls the destructor of the base class
    }

}
```

- Correspond to finalizers in Java.
- Called for an object before it is removed by the garbage collector.
- No *public* or *private*.
- Is dangerous (object resurrection) and should be avoided.

Properties

Syntactic sugar for get/set methods

property type

property name

```
class Data {
    FileStream s;

    public string FileName {
        set {
            s = new FileStream(value, FileMode.Create);
        }
        get {
            return s.Name;
        }
    }
}
```

"input parameter"
of the set method

Used as "smart fields"

```
Data d = new Data();

d.FileName = "myFile.txt";      // invokes set("myFile.txt")
string s = d.FileName;          // invokes get()
```

JIT compilers often inline get/set methods ➔ no efficiency penalty

Properties (continued)

get or set can be omitted

```
class Account {
    long balance;

    public long Balance {
        get { return balance; }
    }
}

x = account.Balance;          // ok
account.Balance = ...;        // compilation error
```

Why are properties a good idea?

- Interface and implementation of data may differ.
- Allows read-only and write-only fields.
- Can validate a field when it is assigned.
- Substitute for fields in interfaces.

Overloaded Operators

Static method for implementing a certain operator

```
struct Fraction {
    int x, y;
    public Fraction (int x, int y) {this.x = x; this.y = y; }

    public static Fraction operator + (Fraction a, Fraction b) {
        return new Fraction(a.x * b.y + b.x * a.y, a.y * b.y);
    }
}
```

Use

```
Fraction a = new Fraction(1, 2);
Fraction b = new Fraction(3, 4);
Fraction c = a + b;    // c.x == 10, c.y == 8
```

- The following operators can be overloaded:
  - arithmetic: +, - (unary and binary), *, /, %, ++, --
  - relational: ==, !=, <, >, <=, >=
  - bit operators: &, |, ^
  - others: !, ~, >>, <<, true, false
- <u>Must</u> return a value

Conversion Operators

Implicit conversion

- If the conversion is always possible without loss of precision
- e.g. long = int;

Explicit conversion

- If a run time check is necessary or truncation is possible
- e.g. int = (int) long;

Conversion operators for custom types

```
class Fraction {
    int x, y;
    ...
    public static implicit operator Fraction (int x) { return new Fraction(x, 1); }
    public static explicit operator int (Fraction f) { return f.x / f.y; }
}
```

Use

```
Fraction f = 3;        // implicit conversion, f.x == 3, f.y == 1
int i = (int) f;       // explicit conversion, i == 3
```

Nested Types

```
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);
}
```

For auxiliary classes that should be hidden

- Inner class can access all members of the outer class (even private members).
- Outer class can access only public members of the inner class.
- Other classes can access an inner class only if it is public.

Nested types can also be structs, enums, interfaces and delegates.

Inheritance

Syntax

```
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() {...}
}
```

- B inherits *a* and *F*(), it adds *b* and *G*()
  - constructors are not inherited
  - inherited methods can be overridden (see later)
- Single inheritance: a class can only inherit from one base class, but it can implement multiple interfaces.
- A class can only inherit from a class, not from a struct.
- Structs cannot inherit from another type, but they can implement multiple interfaces.
- A class without explicit base class inherits from *object*.

Asignments and Type Checks

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

Assignments

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

Run time type checks

```
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

Cast

```
A a = new C();
B b = (B) a;        // if (a is B) stat.type(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
```

as

```
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
```

Overriding of Methods

Only methods that are declared as virtual can be overridden in subclasses

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

Overriding methods must be declared as override

```
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()
    }
}
```

- 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.

Constructors and 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 constr. 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)

Visibility protected *and* internal

protected	Visible in declaring class and its subclasses (more restricive than in Java)
internal	Visible in declaring assembly (see later)
protected internal	Visible in declaring class, its subclasses and the declaring assembly

Example

```
class Stack {
    protected int[] values = new int[32];
    protected int top = -1;
    public void Push(int x) {...}
    public int Pop() {...}
}
class BetterStack : Stack {
    public bool Contains(int x) {
        foreach (int y in values) if (x == y) return true;
        return false;
    }
}
class Client {
    Stack s = new Stack();
    ... s.values[0] ...     // compilation error!
}
```

Abstract Classes

Example

```
abstract class Stream {
    public abstract void Write(char ch);
    public void WriteString(string s) { foreach (char ch in s) Write(s); }
}

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

Note

- Abstract methods do not have an implementation.
- Abstract methods are implicitly *virtual*.
- If a class has abstract methods it must be declared *abstract* itself.
- One cannot create objects of an abstract class.

Abstract Properties and Indexers

Example

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

class List : Sequence {
    public override void Add(object x) {...}
    public override string Name { get {...} }
    public override object this [int i] { get {...} set {...} }
}
```

Note

- Overridden indexers and properties must have the same get and set methods as in the base class

Interfaces

Syntax

```
public interface IList : ICollection, IEnumerable {
    int Add (object value);                  // methods
    bool Contains (object value);
    ...
    bool IsReadOnly { get; }                 // property
    ...
    object this [int index] { get; set; }    // indexer
}
```

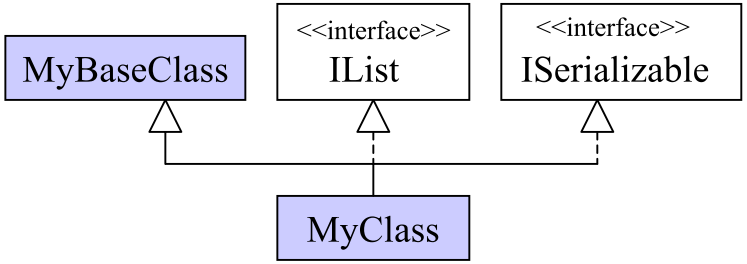
- Interface = purely abstract class; only signatures, no implementation.
- May contain methods, properties, indexers and events
  (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly *public abstract* (*virtual*).
- Interface members must not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

Implemented by Classes and Structs

```
class MyClass : MyBaseClass, IList, ISerializable {
    public int Add (object value) {...}
    public bool Contains (object value) {...}
    ...
    public bool IsReadOnly { get {...} }
    ...
    public object this [int index] { get {...} set {...} }
}
```

- A class can inherit from a *single base class*, but implement *multiple interfaces*.
  A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, 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).

Working with Interfaces

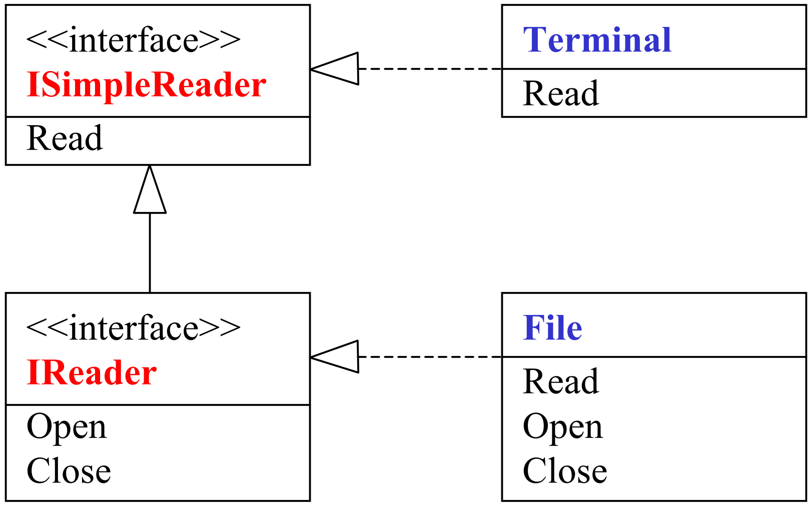


Assignments:	MyClass c = new MyClass(); IList list = c;	
Method calls:	list.Add("Tom");	// dynamic binding => MyClass.Add
Type checks:	if (list is MyClass) ...	// true

Type casts:	c = list as MyClass;
	c = (MyClass) list;
	
	ISerializable ser = (ISerializable) list;	

Example

```
interface ISimpleReader {
    int Read();
}

interface IReader : ISimpleReader {
    void Open(string name);
    void Close();
}

class Terminal : ISimpleReader {
    public int Read() { ... }
}

class File : IReader {
    public int Read() { ... }
    public void Open(string name) { ... }
    public void Close() { ... }
}
```




```
ISimpleReader sr = null;     // null can be assigned to any interface variable
sr = new Terminal();
sr = new File();

IReader r = new File();
sr = r;
```

Summary

Summary of C#

- Familiar
- Safe
  - Strong static typing
  - Run time checks
  - Garbage Collection
  - Versioning
- Expressive
  - Object-oriented (classes, interfaces, ...)
  - Component-oriented (properties, events, assemblies, ...)
  - Uniform type system (boxing / unboxing)
  - Enumerations
  - Delegates
  - Indexers
  - `ref` and `out` parameters
  - Value objects on the stack
  - Threads and synchronization
  - Exceptions
  - User attributes
  - Reflection
  - ...