



# *Introduction to C#*

The New Language for <sup>Microsoft</sup>.net

H.Mössenböck  
University of Linz, Austria  
moessenboeck@ssw.uni-linz.ac.at



## *Contents*

### Introduction to C#

1. Overview
2. Types
3. Expressions
4. Declarations
5. Statements
6. Classes and Structs

### Advanced C#

7. Inheritance
8. Interfaces
9. Delegates
10. Exceptions
11. Namespaces and Assemblies
12. Attributes
13. Threads
14. XML Comments

### 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

3

## *New Features in C#*



### Really new (compared to Java)

- Reference and output parameters
- Objects on the stack (structs)
- Rectangular arrays
- Enumerations
- Unified type system
- goto
- Versioning

### "Syntactic Sugar"

- Component-based programming
  - Properties
  - Events
- Delegates
- Indexers
- Operator overloading
- foreach statements
- Boxing/unboxing
- Attributes
- ...

4

## 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)

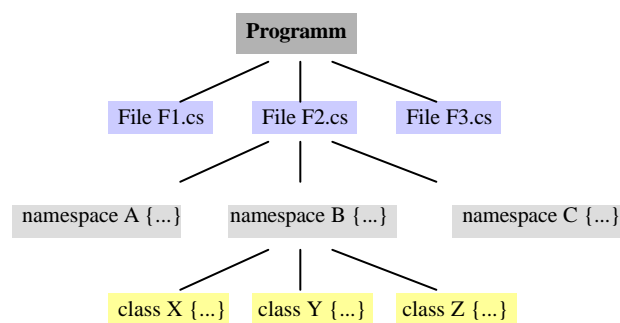
```
csc Hello.cs
```

### Execution

```
Hello
```

5

## 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

6

## A Program Consisting of 2 Files



### Counter.cs

```
class Counter {  
    int val = 0;  
    public void Add (int x) { val = val + x; }  
    public int Val () { return val; }  
}
```

### Prog.cs

```
using System;  
  
class Prog {  
  
    static void Main() {  
        Counter c = new Counter();  
        c.Add(3); c.Add(5);  
        Console.WriteLine("val = " + c.Val());  
    }  
}
```

### Compilation

```
csc Counter.cs Prog.cs  
=> generates Prog.exe
```

### Execution

```
Prog
```

### Working with DLLs

```
csc /target:library Counter.cs  
=> generates Counter.dll
```

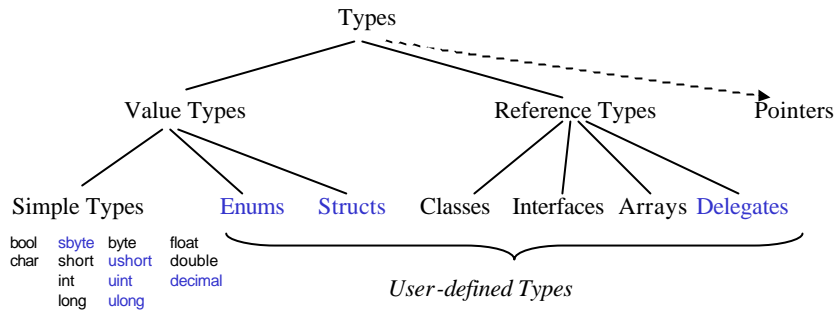
```
csc /reference:Counter.dll Prog.cs  
=> generates Prog.exe
```

7



## 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

9

## 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	<pre>int i = 17; int j = i;</pre>	<pre>string s = "Hello"; string s1 = s;</pre>

10

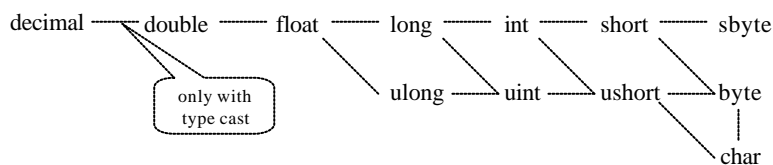
## Simple Types



	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	$\pm 1.5E-45 .. \pm 3.4E38$ (32 Bit)
double	System.Double	double	$\pm 5E-324 .. \pm 1.7E308$ (64 Bit)
decimal	System.Decimal	---	$\pm 1E-28 .. \pm 7.9E28$ (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	<u>Unicode</u> character

11

## Compatibility Between Simple Types



12



## 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");
```

13



## Operations on Enumerations

Compare	if (c == Color.red) ... if (c > Color.red && c <= Color.green) ...
+, -	c = c + 2;
++, --	c++;
&	if ((c & Color.red) == 0) ...
	c = c   Color.blue;
~	c = ~ Color.red;

The compiler does not check if the result is a valid enumeration value.

### Note

- Enumerations cannot be assigned to *int* (except after a type cast).
- Enumeration types inherit from *object* (*Equals*, *ToString*, ...).
- Class *System.Enum* provides operations on enumerations (*GetName*, *Format*, *GetValues*, ...).

14

# 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
```

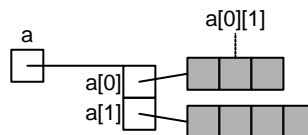
# Multidimensional Arrays



## 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[j]
- 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, ...*

17

## 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
```

18

# 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));
```

# Differences Between Classes and Structs



## Classes

- Reference Types  
(objects stored on the heap)
- support inheritance  
(all classes are derived from *object*)
- can implement interfaces
- may have a destructor

## Structs

- Value Types  
(objects stored on the stack)
- no inheritance  
(but compatible with *object*)
- can implement interfaces
- no destructors allowed

## Boxing and Unboxing



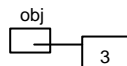
Value types (int, struct, enum) are also compatible with *object*!

### 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

21

## 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 and value types

```
Queue q = new Queue();  
  
q.Enqueue(new Rectangle());  
q.Enqueue(3);  
  
Rectangle r = (Rectangle) q.Dequeue();  
int x = (int) q.Dequeue();
```

22



## *Expressions*



## *Operators and their Priority*

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

Operators on the same level are evaluated from left to right



## Overflow Check

Overflow is not checked by default

```
int x = 1000000;  
x = x * x; // -727379968, no error
```

Overflow check can be turned on

```
x = checked(x * x); // → System.OverflowException  
  
checked {  
    ...  
    x = x * x; // → System.OverflowException  
    ...  
}
```

Overflow check can also be turned on with a compiler switch

```
csc /checkedTest.cs
```

25



## typeof and sizeof

typeof

- Returns the *Type* descriptor for a given type  
(the *Type* descriptor of an object *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 value types.
- Can only be used in an unsafe 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));  
}
```

26



## *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, ...)



## 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.

29



## 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;

30

## 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.

31

## 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
}
```

32



## Statements



## Simple Statements

### Empty statement

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

### Assignment

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

35



## *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).

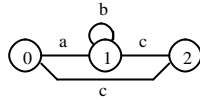
**No fall-through!**  
Every statement sequence in a case must be terminated with break (or return, goto, throw).

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

36

## switch with Gotos

E.g. for the implementation of automata



```

int state = 0;
int ch = Console.Read();
switch (state) {
  case 0: if (ch == 'a') { ch = Console.Read(); goto case 1; }
         else if (ch == 'c') goto case 2;
         else goto default;
  case 1: if (ch == 'b') { ch = Console.Read(); goto case 1; }
         else if (ch == 'c') goto case 2;
         else goto default;
  case 2: Console.WriteLine("input valid");
         break;
  default: Console.WriteLine("illegal character {0}", ch);
          break;
}
  
```

## Loops

### while

```

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

39

## *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

40



## *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  
}
```

41



## *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) ...  
}
```

43

## 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)

44

## 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 constructors cannot be declared
- Can neither inherit nor be inherited, but can implement interfaces

45

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

46

## 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 ...
```

47

## 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 ...
```

48

## 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();
```

49

## 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();
```

50



## 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);  
}
```

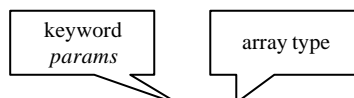
- similar to ref parameters but no value is passed by the caller.
- must not be used in the method before it got a value.

51



## Variable Number of Parameters

Last n parameters may be a sequence of values of a certain 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
```

52

## 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)

### 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) {...}
```

### Calls

```
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); => compilation error
F(i, i); // cannot distinguish F(int x, long y) and F(long x, int y); => compilation error
```

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

53

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

54



## 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, no 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
```

55



## 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).

56

## 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.

57

## 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.

58

## Properties



### Syntactic sugar for get/set methods

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

property type

property 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

59

## 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.

60

## Indexers



### Programmable operator for indexing a collection

```
class File {  
    FileStream s;  
  
    public int this[int index] {  
        get { s.Seek(index, SeekOrigin.Begin);  
            return s.ReadByte();  
        }  
        set { s.Seek(index, SeekOrigin.Begin);  
            s.WriteByte((byte)value);  
        }  
    }  
}
```

Annotations:

- type of the indexed expression (points to `int`)
- name (always `this`) (points to `this`)
- type and name of the index value (points to `[int index]`)

### Use

```
File f = ...;  
int x = f[10]; // calls f.get(10)  
f[10] = 'A'; // calls f.set(10, 'A')
```

- get or set method can be omitted (write-only / read-only)
- Indexers can be overloaded with different index types

61

## Indexers (other example)



```
class MonthlySales {  
    int[] product1 = new int[12];  
    int[] product2 = new int[12];  
    ...  
    public int this[int i] { // set method omitted => read-only  
        get { return product1[i-1] + product2[i-1]; }  
    }  
  
    public int this[string month] { // overloaded read-only indexer  
        get {  
            switch (month) {  
                case "Jan": return product1[0] + product2[0];  
                case "Feb": return product1[1] + product2[1];  
                ...  
            }  
        }  
    }  
}
```

```
MonthlySales sales = new MonthlySales();  
...  
Console.WriteLine(sales[1] + sales["Feb"]);
```

62



## 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
- Must return a value

63



## 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
```

64

## 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.