Classes and Structs

Contents of Classes and Structs

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

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

   ... indexers ...                                   // for convenience
   ... 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)

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

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

Access to private Members

```
class B {
    private int x;
    ...
}


class C {
    private int x;

    public void F (C c) {
        x = ...;            // method may access private members of this.

        c.x = ...;          // method of class C may access private members
                            // of some other C object.

        B b = ...;
        b.x = ...;          // error! method of class C must not access private members
                            // of some other class.
    }
}
```

Fields and Constants

```
class C {
```

`int value = 0;`

Field

- initialization is optional
- initialization value must be computable at compile time
- fields of a struct must not be initialized

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

Constant

- must be initialized
- 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
- value must not be changed later
- occupies 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;      // -- " --
    int x, y, width,height;         // once per object
    ...
}
```

Access within the class	**Access from other classes**
... defaultColor ... scale ...	... Rectangle.defaultColor ... Rectangle.scale ...

Constants must not be declared static

Methods

Example

```
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 from class C

```
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 from Rectangle

ResetColor();

Access from other classes

Rectangle.ResetColor();

Parameters

value parameters (input parameters)

```
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 can be an expression

ref parameters (input/output parameters)

```
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 parameters)

```
void Read (out int a, out int b) {
    a = Console.Read(); b = 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

Usage

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

Another example

```
void Console.WriteLine (string format, params object[] arg) {...}
```

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);    // ambiguous between F(int x, long y) and F(long x, int y); => compilation error
F(i, i);    // ambiguous between 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*!

Method Overloading

Overloaded methods must not differ only in their function types

```
int F() {...}
string F() {...}
```

```
F();            // if the return value is ignored, the name F cannot be resolved
```

The following overloading is illegal as well

```
void P(int[] a) {...}
void P(params int[] a) {...}
```

```
int[] a = {1, 2, 3};
P(a);           // should call P(int[] a)
P(1, 2, 3);     // should call P(params int[] a)
```

Reason for this restriction lies in the implementation of the CLR:
The CIL does not contain the address of the called method but a description of it, which is identical for both cases.

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 = w; 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 constructor 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
```

default constructor initializes all fields as follows:

numeric	0
enumeration	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
```

Constructor 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 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).
- A constructor of a struct must initialize all fields.

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.
- Used for initialization of static fields.

Destructors

```
class Test {

    ~Test() {
        ... cleanup actions ...
    }

}
```

- Correspond to finalizers in Java.
- Called for an object before it is removed by the garbage collector.
- Can be used, for example, to close open files.
- Base class destructor is called automatically at the end.
- No *public* or *private*.
- Is dangerous (object resurrection) and should be avoided
- Structs must not have a destructor (reason unknown).

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

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

Properties (continued)

Properties can also be static

Properties work also with assignment operators

```
class C {
    private static int size;

    public static int Size {
        get { return size; }
        set { size = value; }
    }
}

C.Size = 3;
C.Size += 2;  // Size = Size + 2;
```

Properties (continued)

get or set can be omitted

```
class Account {
    long balance;

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

x = account.Balance;          // ok
account.Balance = ...;        // compiler reports an error
```

Why are properties a good idea?

- Allow read-only and write-only fields.
- Can validate a field when it is accessed.
- Interface and implementation of the data can differ.
- Substitute for fields in interfaces.

Indexers

Custom-defined operator for indexing a collection

type of the indexed expression

name (always *this*)

type and name of the index value

```
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);
            }
    }
}
```

Usage

```
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
- .NET library has indexers for *string* (s[i]), *ArrayList* (a[i]), etc.

Indexers (another example)

```
class MonthlySales {
    int[] apples = new int[12];
    int[] bananas = new int[12];
    ...
    public int this[int i] {                // set method omitted => read-only
        get { return apples[i-1] + bananas[i-1]; }
    }

    public int this[string month] {         // overloaded read-only indexer
        get {
            switch (month) {
                case "Jan": return apples[0] + bananas[0];
                case "Feb": return apples[1] + bananas[1];
                ...
            }
        }
    }
}
```

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

Operator Overloading

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);
    }
}
```

Usage

```
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 always return a function result
- If == (<, <=, true) is overloaded,!= (>=, >, false) must be overloaded as well.

Overloading of && and ||

In order to overload && and ||, one must overload &, |, true and false

```
class TriState {
    int state;  // -1 == false, +1 == true, 0 == undecided
    public TriState(int s) { state = s; }
    public static bool operator true (TriState x) { return x.state > 0; }
    public static bool operator false (TriState x) { return x.state < 0; }
    public static TriState operator & (TriState x, TriState y) {
        if (x.state > 0 && y.state > 0) return new TriState(1);
        else if (x.state < 0 || y.state < 0) return new TriState(-1);
        else return new TriState(0);
    }
    public static TriState operator | (TriState x, TriState y) {
        if (x.state > 0 || y.state > 0) return new TriState(1);
        else if (x.state < 0 && y.state < 0) return new TriState(-1);
        else return new TriState(0);
    }
}
```

true and false are called implicitly

```
TriState x, y;
if (x) ...          => if (TriState.true(x)) ...
x = x && y;         => x = TriState.false(x) ?  x : TriState.&(x, y);
x = x || y;         => x = TriState.true(x) ? x : TriState.|(x, y)
```

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 user-defined 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; }
}
```

Usage

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

Nested Types

```
class A {
    private int x;
    B b;
    public void Foo() { b.Bar(); }
    ...

    public class B {
        A a;
        public void Bar() { a.x = ...; ... a.Foo(); }
        ...
    }
}

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

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.

Differences to Java

- **No anonymous types like in Java**
- **Different default visibility for members**
  C#: private
  Java: package
- **Different default visibility for types**
  C#: internal
  Java: package