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/
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

```csharp
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”
using System;

namespace ConsoleHello
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        /// <param name="args" /></param>
        static void Main(string[] args)
        {
            // Use the system console object
            Console.WriteLine( "Hello World!" );
        }
    }
}
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**

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

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

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

**Prog.cs**

```csharp
using System;

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

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

- **getter property**
- **use of counter class**
- **automatic conversion to string**
Multi-file Projects

One Program:
```csharp
csc Counter.cs Prog.cs
→ Prog.exe
```
```bash
c:\> Prog
```

Working with DLL's:
```csharp
csc /target:library Counter.cs
→ Counter.dll
```
```bash
requires “public” class Counter
```
```csharp
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

<table>
<thead>
<tr>
<th></th>
<th>Value Types</th>
<th>Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable contains</td>
<td>value</td>
<td>reference</td>
</tr>
<tr>
<td>stored in</td>
<td>stack</td>
<td>heap</td>
</tr>
<tr>
<td>initialization</td>
<td>0, false, ‘\0’</td>
<td>null</td>
</tr>
<tr>
<td>assignment</td>
<td>copies of values</td>
<td>copies of references</td>
</tr>
<tr>
<td>example</td>
<td>int i = 17;</td>
<td>string s = “Hello”;</td>
</tr>
<tr>
<td></td>
<td>int j = i;</td>
<td>string t = s;</td>
</tr>
</tbody>
</table>

```
i = 17
j = i;
s = "Hello"
t = s;
```

Com S 430
# System.Object

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

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

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

**Com S 430**
Type Compatibility

- decimal
- double
- float
- long
- int
- short
- sbyte
- ulong
- uint
- ushort
- byte
- char

only with type cast
Enumerations

List of named constants

Declaration (directly in a namespace)

```csharp
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

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

if ( (Access.User & lAccess) != 0)
    System.Console.WriteLine( "access granted" );
```
# Operations on Enumerations

<table>
<thead>
<tr>
<th>Tests</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td><code>c = c + 2;</code></td>
</tr>
<tr>
<td>++, --</td>
<td><code>c++</code></td>
</tr>
<tr>
<td>&amp;</td>
<td><code>if ((c &amp; Color.Red) == 0) … // logical and</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td><code>a = ~Access.Group;</code> // bitwise complement</td>
</tr>
</tbody>
</table>

**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.

Com S 430
Arrays

One-dimensional arrays:

```csharp
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:
```csharp
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:
```csharp
int[,] a = new int[2, 3];
int x = a[0,1];
int len = a.Length; // 6
len = a.GetLength(0); // 2
```
C# treats strings as first-class types that are flexible, powerful, and easy to use.

```csharp
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.
## Structures

### Declaration:

```c
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:

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

Declaration:

```java
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:

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

<table>
<thead>
<tr>
<th>Classes</th>
<th>Structs</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference types</td>
<td>value types</td>
</tr>
<tr>
<td>(objects stored on the heap)</td>
<td>(objects stored on the stack)</td>
</tr>
<tr>
<td>support inheritance</td>
<td>No inheritance</td>
</tr>
<tr>
<td>(all classes are derived from</td>
<td>(but compatible with Object)</td>
</tr>
<tr>
<td>Object)</td>
<td></td>
</tr>
<tr>
<td>Can implement interfaces</td>
<td>Can implement interfaces</td>
</tr>
<tr>
<td>May have a destructor</td>
<td>No destructor allowed</td>
</tr>
</tbody>
</table>

Com S 430
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.

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

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

```java
Queue q = new Queue();
q.Enqueue( new String( "Hello World" ) );
q.Enqueue( 3 );
String s = (String)q.Dequeue();
int x = (int)q.Dequeue();
```
## Operators

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked</td>
</tr>
<tr>
<td>Unary</td>
<td>+ - ! ~ ++x --x (T)x</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &gt; &lt;= &gt;= is as</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>Logical XOR</td>
<td>^</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
</tr>
<tr>
<td>Conditional AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Conditional OR</td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
</tr>
<tr>
<td>Assignment</td>
<td>= *= /= %= += -= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
</tr>
</tbody>
</table>

- **Left-associative**: `left-associative`
- **Right-associative**: `right-associative`
Overflow Check

- Overflow is not checked by default.

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

- Overflow check can be turned on.

```csharp
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.

```csharp
csc /checked Test.cs
```
typeof and sizeof

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

```csharp
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.

```csharp
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

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.

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

Com S 430
Statement Lists and Blocks

**Statement**

Statement lists and block statements

**Example**

```csharp
static void Main()
{
    F();
    G();
    {
        H();
        I();
    }
}
```
Labeled Statement and goto

Labeled statements and goto statements

```csharp
static void Main(string[] args)
{
    if (args.Length == 0)
        goto done;

    Console.WriteLine(args.Length);

done:
    Console.WriteLine("Done");
}
```

Com S 430
Local Constant Declaration

**Statement**  
Local constant declarations

<table>
<thead>
<tr>
<th><strong>Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>```csharp</td>
</tr>
</tbody>
</table>
| static void Main()
| {         |
|   const float pi = 3.14f;
|   const int r = 123;
|   Console.WriteLine(pi * r * r); |
| }         |
|```
Local Variable Declaration

<table>
<thead>
<tr>
<th>Statement</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variable declarations</td>
<td>static void Main()</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int a;</td>
</tr>
<tr>
<td></td>
<td>int b = 2, c = 3;</td>
</tr>
<tr>
<td></td>
<td>a = 1;</td>
</tr>
<tr>
<td></td>
<td>Console.WriteLine(a + b + c);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
## Expression Statement

<table>
<thead>
<tr>
<th><strong>Statement</strong></th>
<th><strong>Example</strong></th>
</tr>
</thead>
</table>
| Expression statements | ```
static int F(int a, int b)
{
    return a + b;
}
``` |
|                     | ```
static void Main()
{
    F(1, 2);  // Expression statement
}
``` |
if Statement

**Statement**

if statements

**Example**

```csharp
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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>switch</code> statements</td>
<td><code>static void Main(string[] args)</code></td>
</tr>
<tr>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td></td>
<td><code>switch (args.Length)</code></td>
</tr>
<tr>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td></td>
<td><code>case 0:</code></td>
</tr>
<tr>
<td></td>
<td><code>Console.WriteLine(&quot;No args&quot;);</code></td>
</tr>
<tr>
<td></td>
<td><code>break;</code></td>
</tr>
<tr>
<td></td>
<td><code>case 1:</code></td>
</tr>
<tr>
<td></td>
<td><code>Console.WriteLine(&quot;One arg&quot;);</code></td>
</tr>
<tr>
<td></td>
<td><code>break;</code></td>
</tr>
<tr>
<td></td>
<td><code>default:</code></td>
</tr>
<tr>
<td></td>
<td><code>int n = args.Length;</code></td>
</tr>
<tr>
<td></td>
<td><code>Console.WriteLine(&quot;{0} args&quot;, n);</code></td>
</tr>
<tr>
<td></td>
<td><code>break;</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

No Fall-Through!
while Statements

**Statement**

while statements

**Example**

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>do statements</td>
<td><code>static void Main()</code></td>
</tr>
<tr>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td></td>
<td><code>string s;</code></td>
</tr>
<tr>
<td></td>
<td><code>do { s = Console.ReadLine(); }</code></td>
</tr>
<tr>
<td></td>
<td><code>while (s != &quot;Exit&quot;);</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
for Statements

- **Statement**: for statements
  - **Example**:
    ```csharp
    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)`

```csharp
{
    foreach (string s in args)
    {
        Console.WriteLine(s);
    }
}
```
**break Statements**

**Statement**

- `break` statements

**Example**

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

**Statement**

continue statements

**Example**

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Example</th>
</tr>
</thead>
</table>
| `return`      | `static int F(int a, int b) {
|               |         |       return a + b; }
| statements    | `static void Main() {
|               |         | Console.WriteLine( F(1, 2) );
|               |         | return; }` |

Com S 430
throw and try Statements

**Statement**

```
static int F(int a, int b)
{
    if (b == 0)
        throw new Exception("Divide by zero");
    return a / b;
}
```

**Example**

```
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**

```csharp
using { Resource r = new Resource(); }
```

**Example**

```csharp
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.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string Message</td>
<td>Gets a message that describes the current exception.</td>
</tr>
<tr>
<td>string StackTrace</td>
<td>Gets a string representation of the frames on the call stack at the time the current exception was thrown.</td>
</tr>
<tr>
<td>string Source</td>
<td>Gets or sets the name of the application or the object that causes the error.</td>
</tr>
<tr>
<td>MethodBase TargetSite</td>
<td>Gets the method that throws the current exception.</td>
</tr>
<tr>
<td>string ToString()</td>
<td>Creates and returns a string representation of the current exception.</td>
</tr>
</tbody>
</table>
# Common Exception Classes

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

Com S 430
No Throws Clause in Signature

Java:

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

Callers of `myMethod` must either

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

C#:

```csharp
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.) ...
}

Com S 430
## Visibility Modifiers

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>No restriction. Members marked <em>public</em> are visible to any element in the declaration domain.</td>
</tr>
<tr>
<td>private</td>
<td>The member is only accessible in the program text of the defined class.</td>
</tr>
<tr>
<td>protected</td>
<td>Members marked <em>protected</em> are visible in the program text of the defining class and all its subclasses.</td>
</tr>
<tr>
<td>internal</td>
<td>Members marked <em>internal</em> are accessible to all methods of any class in the current assembly.</td>
</tr>
<tr>
<td>protected internal</td>
<td>Either <em>protected</em> or <em>internal</em> applies.</td>
</tr>
</tbody>
</table>
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 initialized.

    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 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.

```csharp
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

```java
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.

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

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

Value of “val” is still 3.

local change of “x”
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.

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

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

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.

```csharp
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 by a parameter array – a mechanism to model a variable number of parameters.

```csharp
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)

compile-time error
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.

```java
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:

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

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

```java
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

- 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 or private modifier for a destructor.
- It is in general not recommended (since dangerous) to use destructors. Rather use the method `Dispose`.

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

You cannot override Finalize()!
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.

```java
class Data {
    private 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 overloaded with different index types.

```java
class ListBoxText {
    String[] fStrings;
    String FindString(String aString) { … }

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

Com S 430
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

```java
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

```java
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
```

```java
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
```
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
}

class B : A
{
    public void F() {...}      // warning: hides inherited F() \to use new
    public void G() {...}      // warning: hides inherited G() \to 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.

```csharp
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**

```csharp
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

<table>
<thead>
<tr>
<th>Implicit call of base class constructor</th>
<th>Explicit call</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A {</td>
<td>class A {</td>
</tr>
<tr>
<td>...</td>
<td>public A()</td>
</tr>
<tr>
<td>}</td>
<td>{...}</td>
</tr>
<tr>
<td>class B : A {</td>
<td>class B : A</td>
</tr>
<tr>
<td>public B(int x)</td>
<td>public B()</td>
</tr>
<tr>
<td>{...}</td>
<td>{...}</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

B b = new B(3);

**OK**
- default const.
- A()
- B(int x)

### Explicit call

<table>
<thead>
<tr>
<th>Explicit call</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A {</td>
</tr>
<tr>
<td>public A(int x) {...}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>class B : A {</td>
</tr>
<tr>
<td>public B(int x) {...}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

B b = new B(3);

**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

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

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

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) { ... } // method
    public override String Name { get { ... }; } // property
    public override Object this[int i] { get { ... }; set { ... }; } // indexer
}

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

```csharp
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 { ... }; } // property
    public Object this[int i] { get { ... }; set { ... }; } // indexer
...
}
```
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

Declaration of a delegate type

delegate void Notifier( String aSender );

Declaration of a delegate variable

Notifier fGreetings;

ordinary method signature
Delegate Use

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

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

fGreetings( "Daisy");
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

downarrow
handler

text
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);
    cd3 -= cd1; // invocation list is empty
    // cd3(70); // System.NullReferenceException thrown
}
...
### Test Output

<table>
<thead>
<tr>
<th>C.M1: 10</th>
<th>C.M1: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.M2: 10</td>
<td>C.M1: 60</td>
</tr>
<tr>
<td>C.M1: 60</td>
<td>C.M1: 40</td>
</tr>
<tr>
<td>C.M1: 40</td>
<td>C.M2: 40</td>
</tr>
<tr>
<td>C.M1: 20</td>
<td>C.M2: 20</td>
</tr>
<tr>
<td>C.M1: 20</td>
<td>C.M1: 20</td>
</tr>
<tr>
<td>C.M1: 20</td>
<td>C.M1: 40</td>
</tr>
<tr>
<td>C.M1: 40</td>
<td>C.M1: 60</td>
</tr>
<tr>
<td>C.M2: 40</td>
<td>C.M1: 60</td>
</tr>
</tbody>
</table>
Events

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

```csharp
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();
    }
}
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

[Serializable]
class X { … }

Makes this class serializable.

[Serializable][Obsolete]
class X { … }

Multiple attributes

[Serializable, Obsolete]
class X { … }
### Some Attribute Targets

<table>
<thead>
<tr>
<th>Target</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Applied to any of the following elements: assembly, class, constructor, delegate, enum, event, field, interface, method, module, parameter, property, return value, or struct.</td>
</tr>
<tr>
<td>Assembly</td>
<td>Applied to the assembly itself</td>
</tr>
<tr>
<td>Class</td>
<td>Applied to instances of the class</td>
</tr>
<tr>
<td>Delegate</td>
<td>Applied to delegate methods</td>
</tr>
<tr>
<td>Method</td>
<td>Applied to a method</td>
</tr>
<tr>
<td>Parameter</td>
<td>Applied to a parameter of a method</td>
</tr>
<tr>
<td>ReturnValue</td>
<td>Applied to a return value</td>
</tr>
</tbody>
</table>
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**.
public enum PlatformTypes { Win2000 = 0x0001, WinXP = 0x0002 }

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() );
    }

    // Method to run applications
    public static void CanRunApplication(object app)
    {
        // Implementation
    }
}
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() );
}
Output

```
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:

```csharp
/// ... 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"/></returns>
    public float Mean() { return (float)value / n; }
}

Com S 430
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:

```xml
<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>
```