C#

for C++ Programmers

Krzysztof Mossakowski
Faculty of Mathematics and Information Science
Warsaw University of Technology, Poland
http://www.mini.pw.edu.pl/~mossakow

C# vs C++ - Differences

- Compile target
- Memory management
- Pointers
- Operator overloads
- Target environments
- Preprocessor directives
- Enumerators
- Destructors
- Classes versus structs

C# vs C++ - Similarities

- Syntax
- Execution flow
- Exceptions
- Inharitance model
- Constructors

C# vs C++ - New Features

- Delegates
- Events
- Properties
- Interfaces
- Attributes
- Threading
- Reflection

C# vs C++ - Unsupported

- Multiple inheritance
- Templates
  - supported as **`Generics`** in version 2.0

General

- No header files, no **`#include`** directive
- No requirement for items to be defined before they are referenced in any individual file
- No linking after compilation to an assembly
- Program entry point:
  `static int Main(string[] args)`
  (metod of a class)

Language Syntax

- Does not requires semicolon after a class definition
- Permits expressions to be used as statements even if they have no effect (e.g. **`i+1`**)
- Case-sensitive
  - (Visual Basic .NET is case-insensitive – potential problems with interoperatibility)
- No forward declarations
  - an order of definitions is not significant
- No separation of definition and declaration

Program Flow

- All condition expressions must evaluate to a **`bool`** type

```
int x = 10;
if ( x != 0 )                // OK
if ( x == 0 )                // OK
if ( x )                     // compilation error
if ( x = 0 )                 // compilation error
```

- **`switch`**
  - string can be used as test variable
  - no falling throught to the next **`case`** clause
  - **`goto case ___`**

Foreach Statement

- Foreach loop iterates through all items in an array or collection
  - collection is a class which implements the interface **`IEnumerable`** or declares **`GetEnumerator`** method
- Read-only access to iterated elements

```
foreach (double someElement in MyArray)
{
    Console.WriteLine(someElement);
    someElement *= 2; // Error! read-only access
}
```

Enumerations

```
enum Color : short { Red, Green = 5, Blue }
      // ending semicolon is optional
      // Red=0, Green=5, Blue=6
      // (default type is int, here should is used)

Color myColor = Color.Red;
Console.WriteLine("{0}", myColor);
      // Displays Red

Color secondColor = (Color)Enum.Parse(
      typeof(Color), "GREEN", true );
      // true – ignore case
```

Basic Data Types

- Integer:
  - sbyte, byte – 8-bit
  - short, ushort – 16-bit
  - int, uint – 32-bit
  - long, ulong – 64-bit
- Floating point:
  - float – 32-bit
  - double – 64-bit
  - decimal – 28 significant digits
- Characters and string:
  - char – 16-bit Unicode
  - string – set of Unicode characters
- **`object`**
- No **`unsigned`**, **`signed`** keywords

Basic Data Types as Classes

- **`string` ≡ `System.String`**
  `int` ≡ `System.Int32` etc.
- Very useful methods and fields, e.g.:
  - char: **`IsLetter, ToUpper`**
  - string: **`Length, Format, Substring`**
  - numeric types: **`MinValue, MaxValue, Parse`**
  - floating point types: **`Epsilon, PositiveInfinity, NegativeInfinity`**

Casting

- Implicit and explicit

```
float f1 = 40.0F;
long l1 = (long)f1;     // explicit: possible rounding
short s1 = (short) l1; // explicit: possible overflow
int i1 = s1;            // implicit: no problems
uint i2 = (uint)i1;     // explicit: possible sign error
uint i2 = uint(i1);     // wrong syntax
```

- Checked casting (default: not checked)
  - **`OverflowException`** at runtime

```
checked {
    int i1 = -3;
    uint i2 = (uint)i1;
}
```

Initialization of Variables

- Member fields of classes and structs are initialized by **`0`**, **`false`** or **`null`**
- Variables local in methods are not initialized
  - compilation error: Use of unassigned local variable

Value and Reference Types

Value Types	Reference Types
The variable contains the value directly	The variable contains a reference to the data (data is stored in separate memory area)
Allocated on stack	Allocated on heap using the new keyword
Assigned as copies	Assigned as references
Default behavior is pass by value	Passed by reference
== and != compare values	== and != compare the references, not the values
simple types, structs, enums	classes

Boxing and Unboxing

- Boxing
  - to treat a value type as it were a reference type

```
int p = 123;
object box;
box = p;
```

- Unboxing

```
p = (int)box
```

- Ways to Reduce Boxing and Unboxing Operations of Value Types
  - Encapsulate value types in a class
  - Manipulate value types through interfaces

Strings

- Escaping special characters (**\' \" \\ \0 \a \b \f \n \r \t \v**)
  - \0 is not a string terminator
  - @ prevents any character from being escaped
    @"C:\Program Files\"
- Unicode
  - **\u####** - can be used in variable names

`System.String` Class

- Many methods and properties, e.g.:
  - **`[], Length, Copy, Insert, Concat, Trim, Pad, ToUpper, ToLower, Join, Split`**
  - **`Compare -`** dictionary ordering, case-insensitive option, locale-specific options
  - **`Equals`**, **`==`**, **`!=`** - value comparison
- **`string`** is immutable
  - **`StringBuilder`** class - allows to modify a string without creating a new object

```
int MyInt = int.Parse("123,456",
   System.Globalization.NumberStyles.AllowThousands);
string MyString = 100.ToString("C");     // "$100.00"
```

Regular Expressions

- Regular expressions – powerful text processing
- Pattern-matching notation allows to:
  - find specific character patterns
  - extract, edit, replace, or delete text substrings
  - add the extracted strings to a collection to generate a report
- Designed to be compatible with Perl 5
- **`System.Text.RegularExpressions.RegEx`**

Arrays

```
double [] array;
array = new double[3];
```

```
double [] array = new double[3];
```

```
double [] array = new double[3] { 1.0, 2.0, 3.0 };
```

```
double [] array = { 1.0, 2.0, 3.0 };
```

```
class Example3 {
    static void Main(string[ ] args) {
        foreach (string arg in args) {
            System.Console.WriteLine(arg);
        }
    }
}
```

System.Array

- All arrays are instances of the base class System.Array
  - many useful properties and methods
    - **`Rank, Length, Sort, Clear, Clone, GetLength, IndexOf`**
  - the overhead is greater than that for C++ arrays
  - **`IndexOutOfRangeException`**
  - copying an array variable copies the reference only
  - implement **`ICloneable, IList, ICollection, IEnumerable`**

Multidimensional Arrays

- Rectangular

```
int [,] myArray2d;
myArray2d = new int[2,3] { {1, 0}, {3, 6}, {9, 12} };
int [,,] myArray3d = new int[2,3,2];
      // default constructor is called on each element

int x = myArray3d[1,2,0] + myArray2d[0,1];
```

- Jagged

```
int [][] myJaggedArray = new int[3][];
for (int i=0; i<3; i++)
    myJaggedArray[i] = new int[2*i + 2];

int x = myJaggedArray[1][3];
```

Sorting an Array

- Sort Method Using Element's IComparable.CompareTo

```
Array.Sort( anArray );
```

- IComparable.CompareTo Design Pattern

```
public int CompareTo(Object anObject) {
      if ( anObject == null) return 1;
      if ( !(anObject is <classname>) ) {
            throw new ArgumentException(); }
      // Do comparison and return a
      //   negative integer if instance < anObject
      //   0 if instance == anObject
      //   positive integer if instance > anObject
}
```

System.Collections

- Examples:
  - ArrayList
    - implements **IList** by using a dynamically-sized array
  - DictionaryBase
    - provides abstract base class for strongly-typed collection of associated keys and values
  - Hashtable
    - collection of keys and values using key's hash code
  - SortedList
    - Represents the collection of keys and values, sorted by keys and accessible by key and index
  - BitArray, Queue, Stack

The new operator

```
MyClass Mine;        // Just declares a reference.
                     // Similar to declaring
                     // an uninitialized pointer in C++.
Mine = new MyClass(); // Creates an instance of MyClass.
                     // Calls no-parameter constructor.
                     // In the process, allocates
                     // memory on the heap.
Mine = null;         // Releasing the object.

MyStruct Struct;     // Creates a MyStruct instance
                     // but does not call any constructor.
                     // Fields in MyStruct will be
                     // uninitialized.
Struct = new MyStruct(); // Calls constructor,
                     // so initializing fields.
                     // But doesn't allocate any memory
                     // because Struct already exists
                     // on stack.
```

Operator =

- Simple data types: copies the value
- Structs: does a shallow copy of the struct
- Classes: copies the reference
  - to shallow copy the instance:
    - **`MemberwiseCopy()`** – protected method of Object class
    - **`ICloneable`** interface with **`Clone()`** method

- Shallow copy: members which are objects are not duplicated (their references are duplicated)

Syntax of Class Definition

- Each member is explicitly declared with an access modifier
- Member variables can be initialized in the class definition (also static variables)
  - the only items that can be placed in the constructor initializer is another constructor
- No ending semicolon

Classes

- Inheritance is always public
- A class can only be derived from one base class
  - but it can implement any number of interfaces
- No **`inline`** methods
- Additional access modifiers:
  - **`internal`** – access is limited to the current assembly
  - **`protected internal`** ≡ protected OR internal

Constructors

- The constructor at the top of the hierarchy (**`System.Object`**) is executed first, followed in order by constructors down the tree
- Static constructors
  - executed once only
  - allow static fields to be initialized
  - no access modifier, no parameters
- Default constructors
  - the compiler will generate this default constructor only when there are no constructors

Constructor Initialization List

```
class MyClass : MyBaseClass {
    MyClass(int X)
        : base(X)       // executes the MyBaseClass
                        // 1-parameter constructor
        { /* ... */ }
    MyClass()
      : this (10)       // executes the 1-parameter
                        // MyClass constructor
                        // passing in the value of 10
        { /* ... */ }
```

- Only one other constructor can be placed in the list

Destructors

- **`Finalize`** method
  - in C# syntax is the same as in C++
- The order and timing of destruction is undefined
  - not necessarily the reverse of construction
- Destructors are guaranteed to be called
  - cannot rely on timing and thread ID
- Avoid destructors if possible
  - performance costs
  - complexity
  - delay of memory resource release

IDisposable Interface

- To reclaim a resource:
  - inherit from **`IDisposable`** Interface and implement **`Dispose`** method that releases resources
  - call **`GC.SuppressFinalize`** method
  - calling **`Dispose`** more than once must be benign
  - do not try to use a reclaimed resource
- **`Dispose`** is automatically called at the end of the using block

```
using (Resource r1 = new Resource()) {
    r1.Method();
}
```

Inheritance

- No multiple implementation inheritance
  - but there is a possibility to implement many interfaces
- A reference to a class can refer to instances of that class or to instances of any derived class
- All classes are inherited from **`object`** (**`System.Object`**)
  - the way of using a reference to anything (e.g. in collection classes)

Virtual Methods

```
class MyBaseClass {
    public virtual void VirtualMethod() { ... }
    public void NormalMethod() { ... }
}

class MyClass {
    public override void VirtualMethod() { ... }
    public new void NormalMethod() { ... }
       // hides MyBaseClass.NormalMethod
}
```

Abstract Classes and Methods

- **`abstract`** keyword
- Abstract class
  - cannot be instantiated
  - cannot be sealed
  - can contain implementation
  - can declare non-public members
  - can extend a class and interfaces
- Abstract method
  - cannot contain a method body
  - is virtual

Sealed Classes

- It is not possible to derive from a sealed class
- Sealed classes can be used for optimizing operations at run time
- Many .NET Framework classes are sealed: String, StringBuilder, and so on
- **`sealed`** keyword

Operators `is` `as`

- The is operator
  - returns true if a conversion can be made
- The as operator
  - converts between reference types, like cast
  - on error returns null, does not raise an exception

```
Bird b = (Bird)a;        // Unsafe – possible
                         // InvalidCastException
if (a is Bird)
    b = (Bird) a;        // Safe
b = a as Bird;           // Convert, no exception
if (b == null)
      // not a bird
```

Structs

- All structs are value types
- No inheritance
  - it is not possible to inherit from a struct, nor can a struct inherit from another struct or class
  - no virtual and abstract methods
- The default (no-parameter) constructor of a struct is always supplied by the compiler and cannot be replaced

`const` and `readonly`

- **`const`**: value is set at complite time
  - const variables are static (access: **`ClassName.Variable`**), but **`static`** keyword cannot be used in declaration
- **`readonly`**: value is set at runtime, in a constructor
  - instance constant
- Cannot be applied to methods or parameters

Methods

- Always members of classes
  - no global functions
- Definition and declaration in one place
- No **`const`** methods
- Can be overloaded (the same name, different parameters)
  - no default values for parameters

Method parameters

- Default: passed by value
  - structs are duplicated, classes passed by reference
- Reference parameters: **`ref`** keyword
  - must be initialized before passing to a method
  - no **`const`** modifier
- Output parameters: **`out`** keyword

```
public void MultiplyByTwo(ref double d, out double square) {
    d *= 2;
    square = d * d;
}
//...
double value, square;
value = 4.0;
MultiplyByTwo(ref value, out square);
```

Variable-Length Parameters

- Use **`params`** keyword
- Declare as an array at the end of the parameter list
- Always pass by value

```
static long AddList(params long[ ] v) {
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( ) {
    long x = AddList(63,21,84);
}
```

Properties

```
class MyClass {
    private int length;
    public int Length {
        get { return length; }
        set { length = value; }
    }
}
//...
MyClass MyObject = new MyClass;
MyObject.Length = 10;
int Length = MyObject.Length;
```

- It is possible to omit the **`get`** or **`set`** accesor
  - the way of implementing write-only or read-only access

Operator Overloading

- Overloadable operators:
  + - * / %
  ++ -- (prefix version only)
  == != < > <= >=
  & | ~ ^ !
  true false

```
public static Time operator+(Time t1, Time t2)
{
      int newHours = t1.hours + t2.hours;
      int newMinutes = t1.minutes + t2.minutes;
      return new Time(newHours, newMinutes);
}
```

Operators Restrictions

- = cannot be overloaded
- && and || cannot be overloaded directly
  - are evaluated using &, |, **`true`**, **`false`**
- *=, /=, +=, -=, %= cannot be overloaded
  - are evaluated using *, /, +, -, % operators
- &=, |=, ^=, >>=, <<= cannot be overloaded
- Relational operators must be paired (< and >, <= and >=, == and !=)
- Override the **`Equals`** method if overloading == !=
- Override the **`GetHashCode`** method if overriding **`Equals`** method

Indexers

- Provide array-like access to an object

```
class String {
    public char this[int index] {
        get {
          if (index < 0 || index >= Length)
              throw new IndexOutOfRangeException( );
          ...
        }
    }
    ...
}


string s = "Hello world!";
char ch = s[3];
```

Conversion Operators

```
public static explicit operator Time (float hours)
{ ... }
public static explicit operator float (Time t1)
{ ... }
public static implicit operator string (Time t1)
{ ... }


Time t;
string s = t;
float f = (float)t;
```

- If a class defines a string conversion operator, it should override ToString

Exceptions

- **`finally`** block
  - executed as soon as control leaves a catch or try block, and typically contains clean-up code for resources allocated in the try block
  - optional
- **`catch`** blocks – optional
- The exception must be a class derived from **`System.Exception`**

Excepions Handling

- Throwing
  - avoid exceptions for normal or expected cases
  - never create and throw objects of class **`Exception`**
  - include a description string in an **`Exception`** object
  - throw objects of the most specific class possible
- Catching
  - arrange **`catch`** blocks from specific to general
  - do not let exceptions drop off **`Main`**

Unsafe Code

- Possibilities:
  - declaring and operating on pointers
  - conversions between pointers and integral types
  - taking the address of variables
  - fixing data on the heap (**`fixed`** keyword)
  - declaring arrays on the stack (**`stackalloc`** keyword)
- **`unsafe`** keyword used for:
  - class or struct
  - member field
  - block statement
- **`/unsafe`** flag of compilation is required

Interfaces

- Set of definitions for methods and properties
- Restrictions for methods:
  - no access modifiers
  - no implementation in the interface
  - cannot be declared as virtual or abstract
- Implementing interfaces
  - a class can implement zero or more interfaces
  - a class must implement all inherited interface methods
  - the implementing method can be virtual or non-virtual

Explicit Implementation

```
interface IDimensions {
    float Length();
    float Width();
}
class Box : IDimensions {
    float IDimensions.Length() {}
    float Width() {}
}

Box myBox = new Box();
IDimensions myDimensions = (IDimensions)myBox;
float a = myBox.Length();                  // error
float b = (myBox as IDimensions).Width(); // OK
float c = myBox.Width();                   // OK
float d = myDimensions.Length();           // OK
float e = myDimensions.Width();            // OK
```

Delegates

- Idea: the method pointer is wrapped in a specialized class, along with a reference to the object against which the method is to be called (for an instance method, or the null reference for a static method)
- A delegate is a class that is derived from the class System.Delegate
- Delegate contains a reference to a method
- All methods invoked by the same delegate must have the same parameters and return value

Using Delegates

```
delegate void MyOp(int X);
class MyClass {
    void MyMethod(int X) { ... }
}

MyClass Mine = new MyClass();
MyOp DoIt = new MyOp(Mine.MyMethod);

// Invoking the method via the delegate
DoIt();
```

Events

```
// System namespace:
    public delegate void EventHandler(
                          object sender, EventArgs e );
// System.Windows.Forms.Control class:
    public event EventHandler Click;

public class MyForm : Form {
  private Button button;      // derived from Control
  public MyForm() : base() {
    button.Click += new EventHandler(Button_Clicked);
  }

  private void Button_Clicked(
                          object sender, EventArgs e) {
    MessageBox.Show( "Button was clicked" );
  }
}
```

Delegates, Events, Interfaces

- Use a delegate if:
  - you basically want a C-style function pointer
  - you want single callback invocation
  - the callback should be registered in the call or at construction time, not through an add method call
- Use events if:
  - client signs up for the callback function through an add method call
  - more than one object will care
  - you want end users to be able to easily add a listener to the notification in the visual designer
- Use an interface if:
  - the callback function entails complex behavior, such as multiple methods

Attributes

- Derived from System.Attribute
- Declarative tags that convey information to the runtime
- Stored with the metadata of the element
- Predefined attributes in .NET:
  - general attributes
  - COM interoperability attributes
  - transaction handling attributes
  - visual designer component building attributes

`Conditional` Attribute

- Serves as a debugging tool
  - causes conditional compilation of method calls, depending on the value of a programmer-defined symbol
  - does not cause conditional compilation of the method itself

```
using System.Diagnostics;
class MyClass {
  [Conditional ("DEBUGGING")]
  public static void MyMethod( )  { ... }
}
```

  - compilation parameter /d:DEBUGGING

DllImport Attribute

```
using System.Runtime.InteropServices;

public class Win32 {
     [DllImport("user32.dll", CharSet=CharSet.Auto)]
     public static extern int MessageBox(int hWnd,
            String text, String caption, uint type);
}

public class HelloWorld {
    public static void Main() {
       Win32.MessageBox(0, "Hello World",
                        "Platform Invoke Sample", 0);
    }
}
```

Custom Attributes

```
using System;
[AttributeUsage( AttributeTargets.Class |
                 AttributeTargets.Struct,
                 AllowMultiple = true )]
public class AuthorAttribute : Attribute {
    public Author(string name) {
        this.name = name;
        version = 1.0;
    }
    public double version;
    string name;
}
```

```
[Author("H. Ackerman", version=1.1)]
class SomeClass { ...
}
```

Documentation Comments

- Special comment syntax that contains Extensible Markup Language (XML)
- Documentation comments must immediately precede a user-defined type (class, interface, delegate) or member (field, property, method, event)
- Documentation generator produces file, which can be used as an input for documentation viewer
  - NDoc – http://ndoc.sourceforge.net

Recommended Tags

- **<c>** - text in a code-like font
- **<code>** - one or more lines of source code or program output
- **<example>** - an example
- **<exception>** - the exceptions a method can throw
- **<include>** - includes XML from an external file
- **<list>** - list or table
- **<para>** - permits structure to be added to text
- **<param>** - parameter for a method or constructor
- **<paramref>** - identifies that a word is a parameter name
- **<permission>** - the security accessibility of a member
- **<remarks>** - a type
- **<returns>** - the return value of a method
- **<see>** - a link
- **<seealso>** - See Also entry
- **<summary>** - a memberof a type
- **<value>** - a property

Naming Conventions

- Suggestions:
  - use **`camelCase`** for:
    - local variables
    - parameters of methods
  - use **`_camelCase`** or **`camelCase`** for :
    - private contants
    - private fields
    - private static fields
  - use **`PascalCase`** in other cases
  - do not use Hungarian notation

C# 2.0

- Generics
- Anonymous methods
- Iterators
- Partial types
- Static classes
- Nullable types
- **`global`** keyword
- Access modifiers for **`get`** and **`set`** accesors
- Covariance and contravariance

Generics

```
using System.Collections.Generic;
public class Stack<T> {
      T[] items;
      int count;
      public void Push(T item) {...}
      public T Pop() {...}
}
// ...
static void PushMultiple<T>(Stack<T> stack,
                            params T[] values) {
      foreach (T value in values)
            stack.Push(value);
}
// ...
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

Anonymous Methods

```
// 1.x
public MyForm() {
    addButton.Click += new EventHandler(AddClick);
}
void AddClick(object sender, EventArgs e) {
    MessageBox.Show(textBox.Text);
}
```

```
// 2.0
public MyForm() {
    addButton.Click += delegate {
        MessageBox.Show(textBox.Text);
    }
}
// when parameter names are needed
    addButton.Click += delegate
            (object sender, EventArgs e) {
        MessageBox.Show(((Button)sender).Text);
    }
```

Iterators

```
using System.Collections.Generic;
public class Stack<T>: IEnumerable<T> {
    T[] items;
    int count;
    public IEnumerator<T> GetEnumerator() {
        for (int i = count – 1; i >= 0; --i)
            yield return items[i];
    }
    public IEnumerable<T> TopToBottom {
        get { return this; }
    }
    public IEnumerable<T> BottomToTop {
        get {
            for (int i = 0; i < count; i++)
                yield return items[i];
        }
    }
}
```

Partial Types

- **`partial`** keyword used before **`class`**
- Possibility to write definition of classes, structs and interfaces in many source files

- Good programming practice:
  - maintain all source code for a type in a single file
- May be used to seperate machine-generated and user-written parts of types

Static Classes

- **`static`** keyword used before **`class`**
- Static classes
  - contain only static members
  - cannot be instantiated
  - are sealed
  - cannot contain instance constructor (only static constructor)

Nullable Types

- Possibility to assign value types a null value
  - especially useful for database solutions

```
System.Nullable<T> variable
// or
T? variable

System.Nullable<int> myNullableInt;
int? myOtherNullableInt;

if (myNullableInt.HasValue)
// or
if (myNullableInt != null)
```

global Keyword

```
using System;
namespace GlobalNameSpace {
    class Program {
        public class System {
        }

        static void Main(string[] args) {
            bool Console = true;
            int x = 5;

            Console.WriteLine(x);        // compilation error
            System.Console.WriteLine(x); // compilation error
            global::System.Console.WriteLine(x);       // OK
            global::System.Console.WriteLine(Console); // OK
        }
    }
}
```

Access Modifiers in Properties

- Accessibility level of the **`get`** and **`set`** accessors within a property can be restricted
- Access cannot be broadened

```
public string Something
{
    get { return something; }
    protected set { something = value; }
}
```

Covariance & Contravariance

- Covariance
  - delegate method with a return type that is derived (directly or indirectly) from the delegate's defined return type
- Contravariance
  - delegate method signature in which one or more of the parameters is derived from the type defined by the delegate