2 Introduction to C#

Juha Vihavainen
Department of Computer Science
University of Helsinki

Goals of the C# language

- a general-purpose object-oriented language
  - strong type checking (static plus dynamic)
  - array bounds checking
  - prevention of use of uninitialized variables
  - automatic garbage collection (no dangling references..)
  - object-oriented features, generics, exceptions, . .
- moderately easy to learn by programmers familiar with Java and/or C++

On history of C#

- "Microsoft 's response" to Java
  - after the retirement of some transitory (J++ /J#) languages
  - initial public release about 2000
- language name inspired by musical note C#
  - a *sharp* (#) sign means "the note that is one half step higher than the natural note"
- lead designers:
  - Anders Hejlsberg with design team members Scott Wiltamuth and Peter Golde
  - Hejlsberg experience: Turbo Pascal, Borland Delphi, J++
- C# is standardized via ECMA and ISO
  - but "Microsoft retains architectural control" (?)

3

Some language features

- unified object system
  - everything is an **object**, also (primitive) values
- inheritance and subclasses: *object polymorphism*
- *operation polymorphism* via **virtual** methods (i.e., dyn. binding)
- **interface**s define services but no implementation
- **struct**
  - restricted, lightweight, and efficiently managed value types
  - can implement interfaces but are themselves *sealed* (= final)
- **delegate**
  - an "object-oriented" version of function pointers
  - useful for *Strategy* and *Observer* design patterns
- also: namespaces, exceptions, threads, locks, deterministic *Dispose*, unsafe code (for interfacing to C/C++), preprocessor

4

"Hello, World!" example (again)

```
using System;

static class Hello {
   static void Main () {          // main entry point
        Console.WriteLine ("Hello, World!");
   }
}
```

- uses the predefined **static** *System.Console* class
- *WriteLine* is a public **static** method of *Console*
- the customary meaning for **static**: no instances needed for class-level data/methods

5

Lexical issues

- C# is case-sensitive (as C, C++, and Java)
- as usual, *whitespace* is used as token separators
  - sequences of space, tab, linefeed, carriage return
- semicolons terminate statements (";")
- curly braces "{ . . . }" enclose code blocks
- different kinds of *comments*
  - **/*** *possible multi-line comment* ***/**
  - // *comment until end-of-line*
  - XML commenting facility (supported by Visual Studio)

    /// <*summary*> . . . </*summary*>

    /// <*param name*="strFilePath"> . . . </*param*>

    public void LoadXMLFromFile (string strFilePath)

6

C# reserved keywords

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

7

Some C# keywords

can test before downcast: "as T" (or *null*) - "is T"
"base" (= "super") for calling base methods/ctor
"checked (expr)" / "checked { . . }": checks for integer overflows
"explicit/implicit": for defining custom type conversions
"extern": calling externally defined unmanaged code
"internal": visible only in this or friend assemblies (specified by it)
"namespace": logical scopes to enclose names
"operator": overloading (only of existing operator symbols)
"params": defining variable length parameters
"readonly": cannot be assigned (after initialization)
"sealed" (= "final" in Java): cannot have subclasses
"stackalloc": memory to be allocated on the call stack (unsafe code)
"struct": user-defined value type
"virtual" / "override": explicit dynamically bound methods

8

C# program structure

Organization

- no header files, code written “in-line” within classes
- no declaration order dependence (but for locals in blocks)

Namespaces (very similar to C++ namespaces)

- contain types/classes, and other namespaces
- can be devided into multiple files
- can be "repeated" in multiple assemblies

Type declarations

- **class**, **struct**, **interface**, **enum**, and **delegate**

Class members

- constants, fields, methods, properties, indexers, events, operators, constructors, finalizers, etc.

9

Defining a class

Simple example

```
class A {
    int num1 = 1;          // can be initialized here
    int num2;              // = zero, by default

    A (int no =  0) {      // can use default arguments
       num2 = no;          // set initial value of number
    }
    . . .
}
```

10

Defining a class

- class may have an access modifier that is either **public** or **internal** (the default)
  - **internal** means visible in this assembly - or by its "friends"
- *one base class* can be indicated (the default is **object**)
- may implement *multiple* **interface**s
- *class body*
  - defines the member fields and methods
  - may have *inner types* (**class**, **struct**, **enum**, **delegate**)
    - inner classes can't access outer non-**static** members
    - but we can pass the enclosing object as a ctor argument
- *namespace items* cannot be declared as **private**, **protected**, or **protected internal** (make no sense for namespaces)
- C# does not have *typedefs* (used extensively in C/C++)

11

On access modifiers

public	fully accessible
internal	accessible in this and friend assemblies
private	accessible only within the containing type
protected	accessible in this type and its subclasses
protected internal	union of **protected** and **internal**

- **protected internal** is more accessible than **protected** or **internal** alone
- non-nested types are **internal** by default (or can be defined as **public**)
- members of **class** and **struct** are **private** by default (vs. C++)
- members of **enum** and **interface** are implicitly **public**

12

Sample class

```
public class Stack <T> {          // a simplified version
    private Entry <T> top;
    public void Push (T data) {
        top = new Entry <T> (top, data);
    }
    public T Pop () {
        if (top == null)
            throw new InvalidOperationException ();
        T result = top.data;  top = top.next;
        return result;
    }
    class Entry <V> { . . . }  // C++-like (static) class..
}
```

13

Constructor

- has the same name as the class (as is customary)
- can take arguments, and may be overloaded
  - can also use default arguments (C# version 4.0, 2010)
- "**new**" is used to create a new instance and to call (one of) its constructors (and is used even for "**struct**" values)
- if we don't write *any* constructor for a class
  - C# provides an implicit default constructor (a 0-arg ctor)
  - so, a similar convention as in C++ and Java
- typically classes have explicitly provided constructors
- constructors are usually **public**, but not always
  - e.g., *Singleton* design pattern makes constructors **private** to ensure that only one instance could be created

14

Inheritance

- use "**:**" to indicate inheritance (vs. "**extends**" in Java)
  - C++-like notation (but without **public**/**private** clauses)
- constructors can invoke base-class constructor by special base-class constructor call:

  public Child (int x, int y) : base (x, y).. // *note keyword*

- casting up and down as in C/Java
- must use "**virtual**" keyword to indicate *virtual functions*
- must use "**override**" when *redefining* a virtual method
  - can call base-class implementation via "**base**"
- also: *abstract class* concept ("**abstract**" keyword)

15

Type system (revisited)

Value types

primitives	**int i;**
enums	**enum State { Off, On };**
structs	**struct Point { public int x, y; }**

Reference types

classes	**class Foo: Bar, IFoo { ... }**
interfaces	**interface IFoo: IBar, IBaz { ... }**
arrays	**string [ ] a = new string [10];**
delegates	**delegate void Op ();** // *typedef* !
	Op op = this.Fun; // *Foo.StaticFun*
string	**string s = "abcd";**

System.Text.StringBuilder ... a mutable string of chars

16

"*Everything is an object*"

Some "traditional" views

- C++: all values are "*objects*" within memory, stack, or heap
- Java: primitive values are handled separately "inline"
- Smalltalk, Lisp: primitive values act like objects, but at some performance cost (may need extra run-time type checking)

C# unifies all types into one tree hierarchy (with single root)

- simplifies the system throughout
- should have no significant performance cost (?)

Improved extensibility and reusability

- new value types: **decimal**, *Point*, etc.
- generic collections, etc., work uniformly for all types

17

Unified type system and boxing

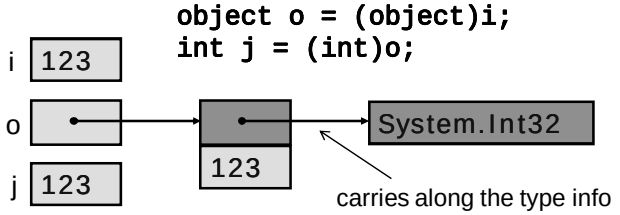
Boxing

- can wrap a value inside an object (of *System.Object* or some **interface** type), and store it on heap
- think as allocating a "box", then copying the value into it
  - note generics and overridden operations for primitives don't require any boxing -- e.g., "**123.ToString ()**" is OK

Unboxing

- checks type and extracts the value from the box object

```
int i = 123;
object o = (object)i;
int j = (int)o;
```



18

C# predefined types

Reference	**object, string,** arrays, etc.
Signed	**sbyte, short, int, long**
Unsigned	**byte, ushort, uint, ulong**
Character	**char** (2-byte, Unicode)
Floating-point	**float, double**
Fixed-precision	**decimal**
Logical	**bool**

- predefined types are aliases for system-provided (.NET) types, for example: **int** == *System.Int32*
- **bool**: "**true**" or "**false**"; 0 does not equal "**false**" and no implicit conversion from **int** to **bool**
- **decimal**: a fixed precision number up to 28-29 digits, used for money calculations ("**300.5m**") - "**m**" like "money"

19

User-defined value types: **struct**

- value type; no subclasses allowed (sealed) => "non-polymorphic"
- can implement interfaces (but cannot inherit **class**/**struct** - why?)
- no field initializers are allowed (instead, use constructors)
- the *default ctor* sets fields to zero/null and *cannot be overridden*
- when overloading a ctor, must assign every field

```
public struct  Point  {
    int x = 1;              // Error:  initializers are not allowed
    int y;                  // ok
    public Point () { }     // Error:  overriding default ctor
    public Point (int x) { this.x = x;  }   // Error: "y" missing
}
```

- changing "**struct**" to "**class**" makes this example legal!

20

Enumerations

Example

```
enum Grades  {              //  default "base type" is int
  GradeA = 94,              //  replaces the default start value 0
  GradeAminus = 90,
  GradeBplus = 87,
  GradeB = 84
}
```

- unassigned values keep incrementing from the last one
- optionally specify another "base type" for values, e.g., " **: byte**"
- part of class hierarchy: *Grades* : *Enum* : *ValueType* : **object**
- can cast to and from an integer value

21

Variables

```
type variable-id [ = init-expression ]       // or
var variable-id = init-expression
```

- the latter automatically deduces the type from an initializer

Examples

```
int number_of_slugs = 0;
string name;
var myfloat = 0.5f;                          // so it's float
var hotOrNot = true;
```

- must be initialized/assigned to before its use (as in Java)

Also constants

```
const int freezingPoint = 32;   // must be compile-time value
```

- cannot change, so value can be *inlined* into code

22

if statement

- familiar C-like syntax for **if** statement but the condition must must evaluate to a **bool** value (**false**, **true**)

if (*expression*) *statement*1 [else *statement*2]

Example:

```
if (i < 5) {
  System.Console.WriteLine ("i is smaller than 5");
} else {
  System.Console.WriteLine ("i is greater or equal to 5");
}
```

23

switch statement

- better alternative to a long sequence of **if** statements
- implicit "fall through" to the next **case** is not permitted
  - but can "**goto case**" to continue to another branch
- **case** label is an integral literal, a **string**, or an **enum** value

```
switch (weather)  {
case Snowing:                  // must be a statically evaluated value
     Console.WriteLine ("It is snowing!");
     goto case Raining;        // also: goto default
case Raining:
     Console.WriteLine ("I am wet!");   break;
default:
     Console.WriteLine ("Weather OK");   break;
}
```

24

Usual C-like looping constructs

for statement

for (j = 0; j < 5; ++j) { … }

- the classic loop syntax in C-like languages
  - beware of off-by-one errors in array indexing

while statement

while (j < 5) { … ; ++j; }

- loops while the condition is **true**

do-while statement

do { … ; ++j; } while (j < 5)

- first perform action, then do condition check

foreach statement to iterate through a collection of items

foreach (int i in intArray) { … } *// or:* **"var i in intArray"**

- locally scoped index variable

25

foreach statement

- iterates through all elements in an array, or a collection
- declare an identifier to hold the current element within the given collection

```
string [] sArray = { "Cherry", "Apple", "Banana"} ;
Array.Sort (sArray);                    // sort elements
foreach (string s in sArray)
   System.Console.Write ("{0}  : ", s);
     // output: "Apple : Banana : Cherry : "
```

- also, a generic version of *Sort* available . .

Array.Sort <T>(T [ ] arr, Comparison <T> comp)

26

goto statement (!)

- transfers execution directly to a label

```
var i = 0;
StartLoop:
if (i < 3) {
    Console.WriteLine (i);  ++i;
    goto StartLoop;
}
```

  - prints out "0 1 2" (one number/line)
- the **goto** statement may be useful to get out of deeply nested loops
- of course, **goto** statements are not usually recommmended but useful for machine-generated source code (say, automata)

27

C# properties: "logical fields"

- provide access to an "abstract data" property
  - hidden custom implementations of getters and setters
- have syntax similar to direct variable access
  - can write "**foo.X**" instead of "**foo.GetX ()**"
  - can write "**foo.X = value**" instead of "**foo.SetX (value);**"
- the actual accessors can additionally
  - check state and so enforce *invariants*
  - provide *lazy evaluation* or some other kind of late binding
- in a way, a minor feature (syntactic sugar)
  - improves readability and uniformity of notation (~ Eiffel)
  - used extensively in C# libraries

28

C# property example

```
public class GameInfo  {
    private string name;
    public string Name {          // can have access modifiers
        get  { return name;  }
        set  { name = value;  }   // special keyword value
    }
}
```

- **get** returns a value of the given property type
- **set** uses an implicit parameter "**value**" to set some internal data representation (here simply a field but not necessarily)
- can omit **get** or **set** (but not both)
- by convention, property names have initial capital, so often use property "*X*" to access a **private** field "*x*"

29

Automatic properties

- often, a property just reads and writes one variable
- *such* **get** and **set** can be automatically created (by the compiler)

```
public class GameInfo  {
    public string Name { get;  set; }      // automatic property
}
```

  - here, a needed "**private string**" member is implicitly created
- here too, can omit one, or can specify *restricted* accessibility
- behaves the same way as the previous *Name* property

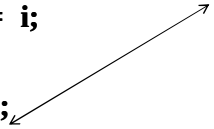
```
GameInfo g = new GameInfo ();
g.Name = "Radiant Silvergun";          // calls "set"
System.Console.Write (g.Name);         // calls "get"
```

30

Passing arguments in C#

- values are usually *passed by value* (of course)
- heap-allocated objects are *passed by reference*
- *values* can be *passed by-reference* with keyword "**ref**"

```
public static void Add (int i, ref int result) {
    result  +=  i;
}   . . .
int total = 20;
Add (10, ref total);      // note obligatory keyword "ref"
Console.WriteLine ("Value after Add () call: {0}", total);
```



  - resembles C++ *reference types* ("&")
- can also pass a object-ref. variable *by reference*: passes the address of the variable itself (of course)

31

Passing arguments in C# (cont.)

- "**out**" parameters are useful for getting multiple return values from a method
  - must be assigned within the method
  - an "**out**" parameter need not be assigned before it goes into the method

```
void Split (string s, out string first, out string last) {
    var i = s. LastIndexOf (' ');      // searches from the end
    first = s.Substring (0, i);  last = s.Substring (i + 1);
}
```

- "**out**" is just like "**ref** " except (any) initial value cannot be used, and it must be assigned before returning
- again, an obligatory keyword "**out**" neede when calling

32

Exceptions and try-catch-finally

```
try {
    throw new MyException ("Oops!");
} catch (MyException e) {
    . . . handle exception
} catch {
    . . . catch all other exceptions
} finally {
    . . . clean up - always, even if no exception occurred
}
```

- the C# language forces increasing generality when processing multiple kinds of exceptions
- no (Java-style) *checked* exceptions are provided

33

finally/using blocks to release resources

```
StreamReader reader = null;
try {  reader = File.OpenText ("file.txt");      // open file for reading
       if (reader.EndOfStream) return;           // test for empty file
       Console.WriteLine (reader.ReadToEnd ());
} catch {  . . .   // show error: non-existing or badly formed file
} finally {        // the standard convention to release resources
       if (reader != null) reader.Dispose ();    // IDisposable interface
}
```

- Above can be *shortened* by using a **using** statement (~ C++ dtor)

```
using (StreamReader reader = File.OpenText ("file.txt")) {
   if (reader.EndOfStream) return;
   Console.WriteLine (reader.ReadToEnd ());
}        // implicitly calls Dispose on reader (when not null)
```

Overloading indexers

- to provide array-like interface to a collection class
- defined as property, adding "**this**" and index brackets ("[]")

```
public class DblBuf {                          // illustrative only !
    string [] buffer  = { "Joe", "Jane" };
    public string this [int index]  {          // overloaded operation
        get  { return buffer [index]; }
        set  { buffer [index] = value; }       // the special keyword
    }
}
var seq = new DblBuf  ();  seq [0]  = "Moe";  seq [1]  = "Mary";
Console.WriteLine ("{0} & {1} ", seq [0], seq [1]);
```

- index does need not be an integer, and
- an indexer can take any number of parameters

35