1 Introduction to .NET

Juha Vihavainen
Department of Computer Science
University of Helsinki

A "shortcut" to Visual Studio

Follow the following instructions (slightly depending on VS version)

- Copy your C# file to a suitable folder (or create), say "\hello"
- **Start** the Visual Studio (from Start menu)
- Start the Create Project Wizard, by selecting the menu item:
  - **File -> New -> Project From Existing Code**..
- Select the **type of project** item "**Visual C#**". - Click "**Next** >"
- Navigate to your "\hello" folder; and fill required "**Project name**"
- Specify the project "**Output type**": "**Console application**"
- To exit the wizard and to get your project created, click "**Finish** >"
- To run, select the menu item: **Debug -> Start Debugging**
  - To enable line-by-line execution, you can select the menu item **Debug -> Step Into** (or **-> Step Over**)
- Continue from here..

Microsoft .NET

- .NET is the name for an architecture from Microsoft that runs programs
- It includes standards for the following
  - bytecode and program file format
  - data types, and system libraries
- The Microsoft's Intermediate Language (IL) is a half way between a high-level language and machine code
  - code for a hypothetical abstract stack machine
  - designed to be easy to translate into actual machine code
- The platforms (PC, Xbox, Phone) run IL from any .NET compiler
  - C#, Visual Basic, F#, IronPython, IronRuby, C++/CLI
  - Only the Windows platform offers the full set of .NET libraries

C# and .NET technology

- .NET concept grew from Win32/COM (C API/components)
  - replaces old Microsoft Foundation Class library (MFC) C++ GUI framework (built-on Win32 API)
- .NET and C# borrow ideas from Java - and C++
  - since then, Java has borrowed features from C#
  - that 's how languages keep developing (C <-> C++)
- .NET and C# are distinct but related technologies
- better to view C# as its own language, and not just a Microsoft-version of Java
- there is a learning curve, even for experienced Java programmers
- hopefully, the curve is less steep than learning C++

Original design goals of .NET framework

- simplify development and deployment
  - as compared to Windows ".dll hell" version handling
    - version conflicts, difficulty in obtaining required DLLs, and having unnecessary DLL copies
  - replace COM object model and its language-neutral IDL
- unify programming models (C, C++, Basic, Java-style)
  - under one comprehensive class-based system
- provide robust and secure execution environment
  - type safety, and virtual machine with garbage collection
- support multiple programming languages
  - similar to Java bytecode as a target code for multiple PLs
- support cross-language development (one program - many PLs)

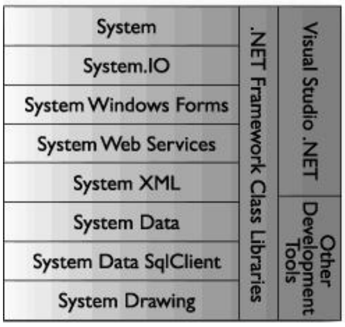
.NET basics

- ECMA/ISO: CLI (Common Language Infrastructure)
  - triggered some third-party impl. (*Mono* and *Portable* .NET)
  - .NET/CLR is MS's own commercial implementation of CLI
- the current version (?): .NET Framework 4.0
- XNA Game Studio projects that target Windows PCs have access to the full .NET Framework
- other versions include the ".NET *Compact Framework for* Xbox 360"
  - a specific subset of the .NET *Compact Framework*
  - designed and optimized specifically for the Xbox 360
- projects that target *Windows Phone* 7 use the ".NET *Compact Framework for Windows Phone* 7"

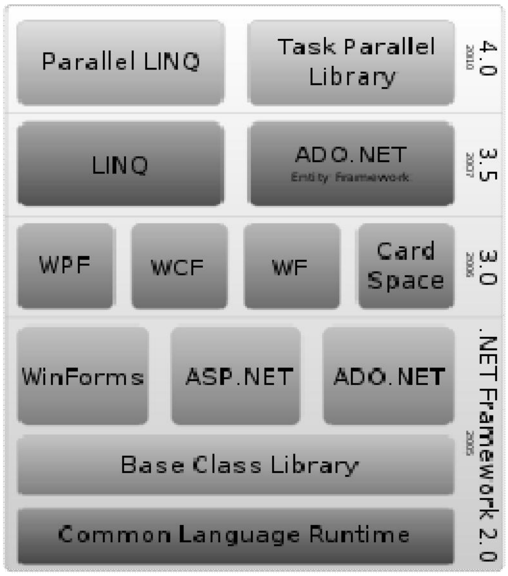
Base Class Library (BCL)

- a library of classes available to all languages using the .NET Framework
- common functions such as
  - basic data structures: lists, queues, stacks, and dictionaries, with C++-style generics (but instantiated at run time)
  - file reading and writing
  - graphic rendering
  - database interaction
  - XML document manipulation
  - regular expressions
  - and so forth

.NET class library

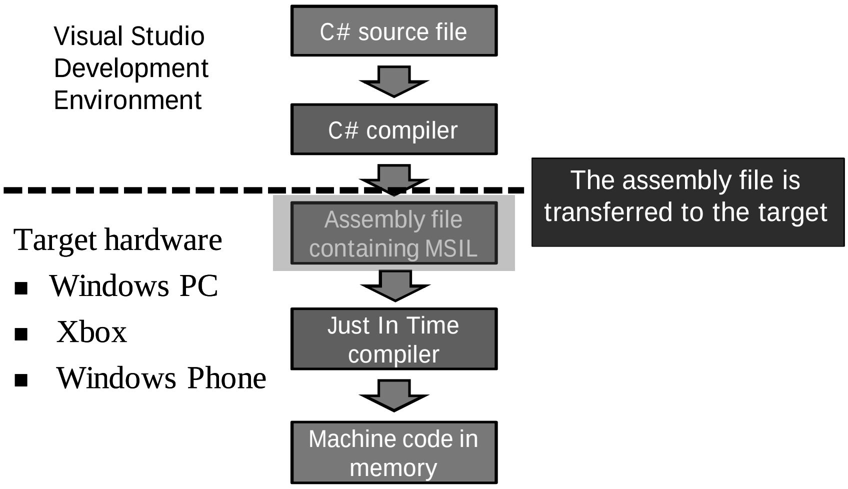


also:
System.Threading
System.Text . .



The .NET Framework Stack

Just-in-Time Compilation



Visual Studio Development Environment

The assembly file is transferred to the target

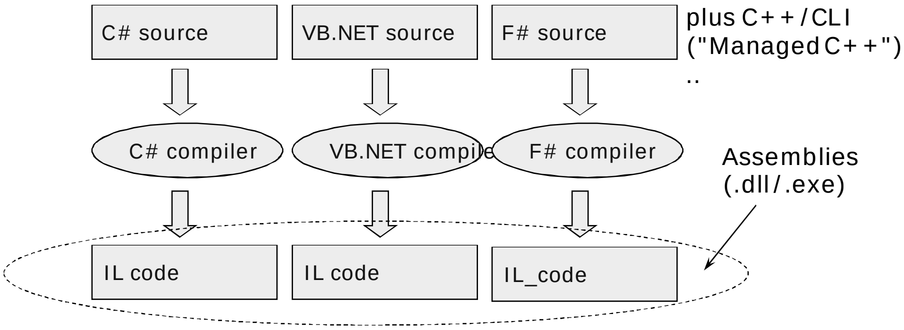
Target hardware

- Windows PC
- Xbox
- Windows Phone

Compilation to assemblies

IL-based assemblies enable *cross-language* development

- assemblies are the binary unit of deployment for classes
- referenced during compilation for checks and bindings
- loaded at run time for execution



plus C++/CLI ("Managed C++") ..

Assemblies (.dll/.exe)

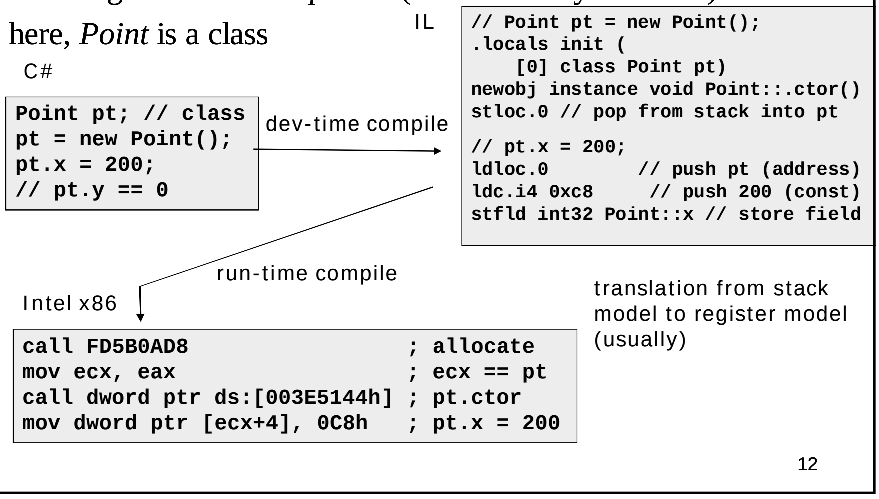
CLR = Virtual Machine

- a class-oriented virtual machine
- classes defined using MS languages C#, VB.NET, F#, and C++/CLI - and some third-party (experimental) languages
- compilers emit assemblies
  - still named .exe and .dll (dynamically linked library)
  - the names still fit their logical purpose (physically differ)
- assemblies contain intermediate language (IL) ~ Java bytecode
- IL is usually just-in-time (JIT) compiled at runtime
  - never *interpreted* (as opposed to Java that partly interprets)
  - the *NGEN* tool (Visual Studio) provides AOT (Ahead-of-Time) compilation - but may sacrifice portability and some disk space
- many built-in runtime services are provided for program execution and automated based on type info (see later)

Just-in-Time (JIT) compilation

IL is compiled into processor-specific code at runtime

- "optimized" at target, not at development machine
- *not* designed to be *interpreted* (= executed by software)
- here, *Point* is a class



C#

```
Point pt; // class
pt = new Point();
pt.x = 200;
// pt.y == 0
```

IL

```
// Point pt = new Point();
.locals init (
    [0] class Point pt)
newobj instance void Point::.ctor()
stloc.0 // pop from stack into pt

// pt.x = 200;
ldloc.0        // push pt (address)
ldc.i4 0xc8     // push 200 (const)
stfld int32 Point::x // store field
```

Intel x86

```
call FD5B0AD8                 ; allocate
mov ecx, eax                  ; ecx == pt
call dword ptr ds:[003E5144h] ; pt.ctor
mov dword ptr [ecx+4], 0C8h   ; pt.x = 200
```

translation from stack model to register model (usually)

Intermediate languages (~ bytecode)

- Benefits
  - can run on a range of platforms
  - can use lots of different programming languages (as long as they compile down to the IL)
  - programs are usually smaller than machine code
  - programs can be digitally signed and verified
    - one representation of code
- Limitations/drawbacks
  - the need to just-in-time (JIT) compile them slows down program execution

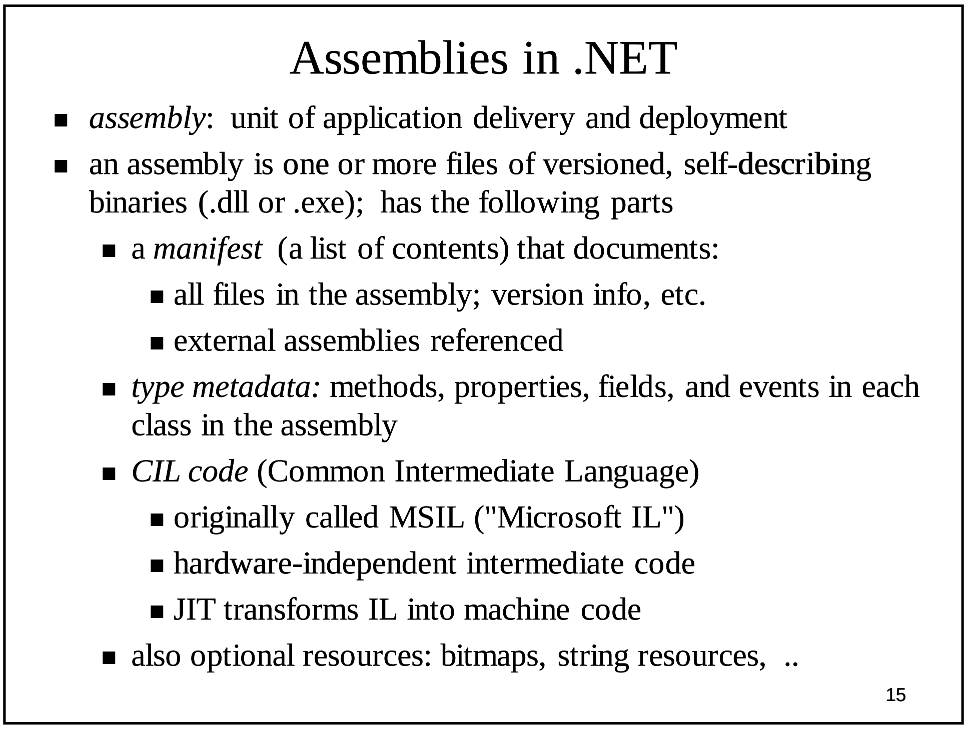
CLR virtual machine

Virtual machine provides services, such as

- run-time assembly loading and processing
- JIT compilation
- garbage collection
- serialization . .
- many services depend on run-time type information

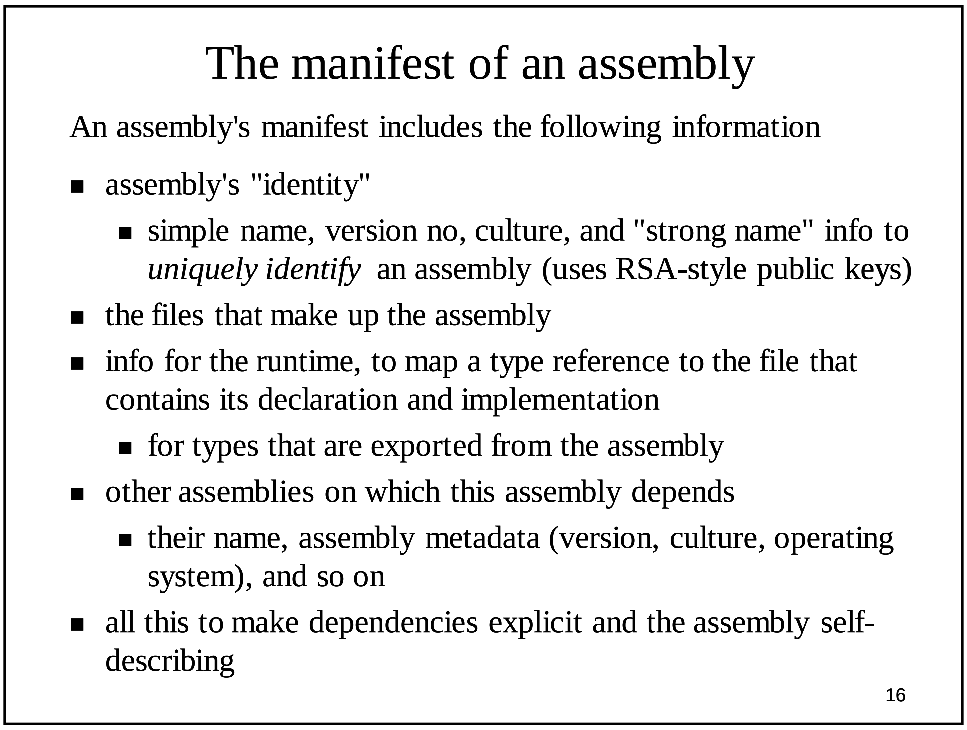
Note. Visual Studio also compiles a "managed" version of C++

- Standard C++ features available (STL, raw memory, **delete**)
- at the same run, "managed objects" can be created (**gcnew**) at the CLR heap and be served by CLR and .NET libraries
- so, C++/CLI can interface to programs in .NET languages
  - native C++ and .NET code in the same C++ program



Assemblies in .NET

- *assembly*: unit of application delivery and deployment
- an assembly is one or more files of versioned, self-describing binaries (.dll or .exe); has the following parts
  - a *manifest* (a list of contents) that documents:
    - all files in the assembly; version info, etc.
    - external assemblies referenced
  - *type metadata:* methods, properties, fields, and events in each class in the assembly
  - *CIL code* (Common Intermediate Language)
    - originally called MSIL ("Microsoft IL")
    - hardware-independent intermediate code
    - JIT transforms IL into machine code
  - also optional resources: bitmaps, string resources, ..



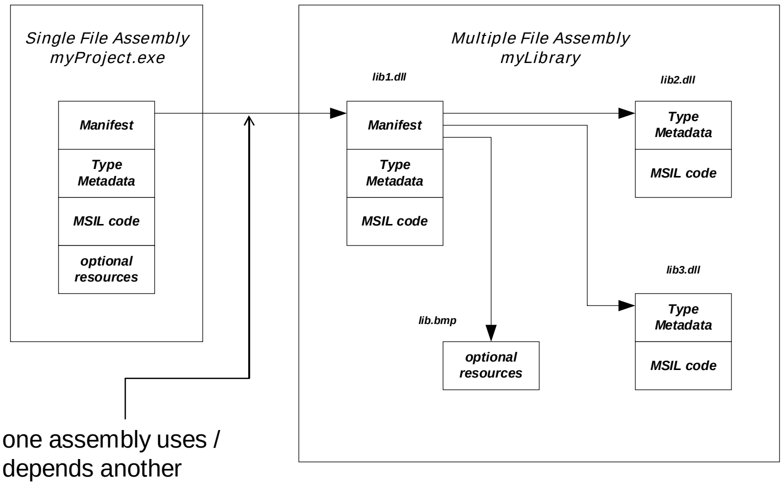
The manifest of an assembly

An assembly's manifest includes the following information

- assembly's "identity"
  - simple name, version no, culture, and "strong name" info to *uniquely identify* an assembly (uses RSA-style public keys)
- the files that make up the assembly
- info for the runtime, to map a type reference to the file that contains its declaration and implementation
  - for types that are exported from the assembly
- other assemblies on which this assembly depends
  - their name, assembly metadata (version, culture, operating system), and so on
- all this to make dependencies explicit and the assembly self-describing

Assembly "modules" = files

- the contents of an assembly can be packaged within one or more intermediate containers, called *modules*
  - a module corresponds to a file that contains parts of an assembly
- so, an assembly can be contained in multiple files
  - the one "main" module contains the manifest, and (optionally) IL + type metadata and various resources
    - the manifest describes the other modules
  - the other modules contain IL + metadata and/or resources
- the need for multi-file assemblies is rare, and they may require use of special tools (command line compiler)



one assembly uses / depends another

- But don't worry: Visual Studio does most of the work in configuring an assembly for us

.NET object model

- "**classes**" correspond to Java object model
  - instances of *reference types* live on the heap
  - they are garbage collected: non-deterministic life-time model
- the programmer can define new *value types* (as in C++)
  - "**struct**" keyword identifies a value type (vs. "**class**")
  - some restrictions on use (e.g., no implementation inheritance)
  - value types are stack-based, i.e., (usually) inlined within a call frame and deterministically "managed by scope"
- a special **using** construct can provide similar deterministic management for dynamic resources (OS "handles"/whatever)

Type system

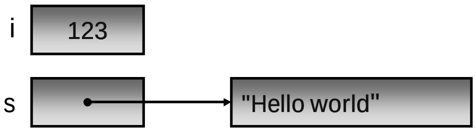
Value types

- directly contain data (but of course may include references as member fields)
- cannot itself be ***null*** (of course fields can be ***null***)
- can be allocated on the stack and efficiently reclaimed

Reference types

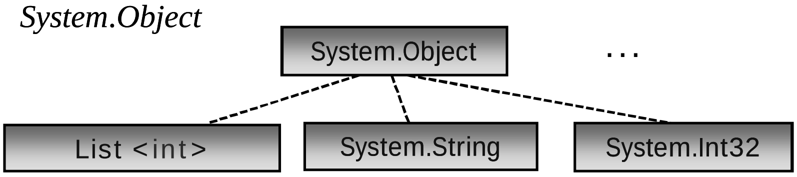
- are references to objects located in the heap
- may be ***null***
- allocated on the heap and garbage collected

```
System.Int32 i  = 123;
System.String s  =
               "Hello world";
```



Uniform type hierarchy

- *all* types ultimately inherit from *System.Object*
  - classes, arrays, structs, enums, delegates, ..
  - an implicit conversion exists from any type to type *System.Object*



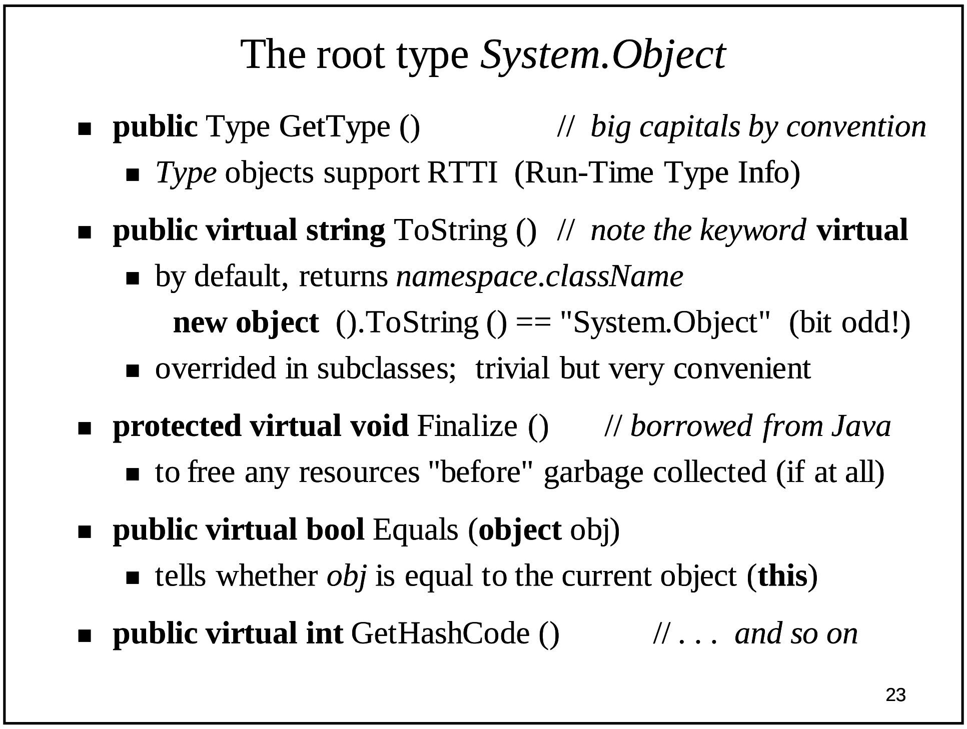
- logically uniform to the programmer
  - but actually optimized by the compiler
- e.g., *System.Int32* is a value type and not a reference
- in C#, keyword "**int**" is reserved symbol to mean *System.Int32*
- in C#, generic instantiations (*List*<**int**>)are new run-time types

the name used in assemblies

.NET/C# primitive types

.NET Base Types (BCL)	C# reserved type names
System.Byte	byte
System.SByte	sbyte
System.Int16	short
System.Int32	int
System.Int64	long
System.UInt16	ushort
System.UInt32	uint
System.UInt64	ulong
System.Single	float
System.Double	double
System.Object	object
System.Char	char
System.String	string
System.Decimal	decimal
System.Boolean	bool

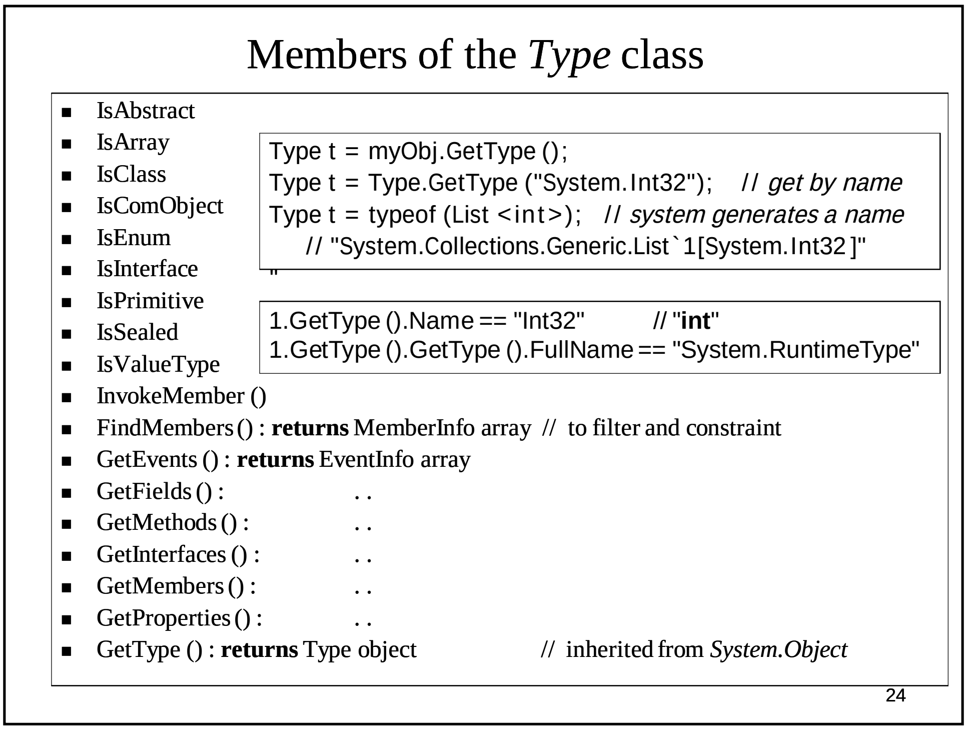
- note that "*System.Int32* " and "**int**" mean exactly the same thing (vs. Java where "*Integer*" and "**int**" are different types)



The root type *System.Object*

- **public** Type GetType () // *big capitals by convention*
  - *Type* objects support RTTI (Run-Time Type Info)
- **public virtual string** ToString () // *note the keyword* **virtual**
  - by default, returns *namespace.className*

    new object ().ToString () == "System.Object" (bit odd!)
  - overrided in subclasses; trivial but very convenient
- **protected virtual void** Finalize () // *borrowed from Java*
  - to free any resources "before" garbage collected (if at all)
- **public virtual bool** Equals (**object** obj)
  - tells whether *obj* is equal to the current object (**this**)
- **public virtual int** GetHashCode () // . . . *and so on*



Members of the *Type* class

```
Type t = myObj.GetType ();
Type t = Type.GetType ("System.Int32");    // get by name
Type t = typeof (List <int>);  // system generates a name
    // "System.Collections.Generic.List`1[System.Int32]"
```

```
1.GetType ().Name == "Int32"          // "int"
1.GetType ().GetType ().FullName == "System.RuntimeType"
```

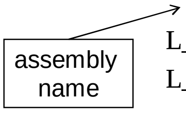
- IsAbstract
- IsArray
- IsClass
- IsComObject
- IsEnum
- IsInterface
- IsPrimitive
- IsSealed
- IsValueType
- InvokeMember ()
- FindMembers () : **returns** MemberInfo array // to filter and constraint
- GetEvents () : **returns** EventInfo array
- GetFields () : . .
- GetMethods () : . .
- GetInterfaces () : . .
- GetMembers () : . .
- GetProperties () : . .
- GetType () : **returns** Type object // inherited from *System.Object*

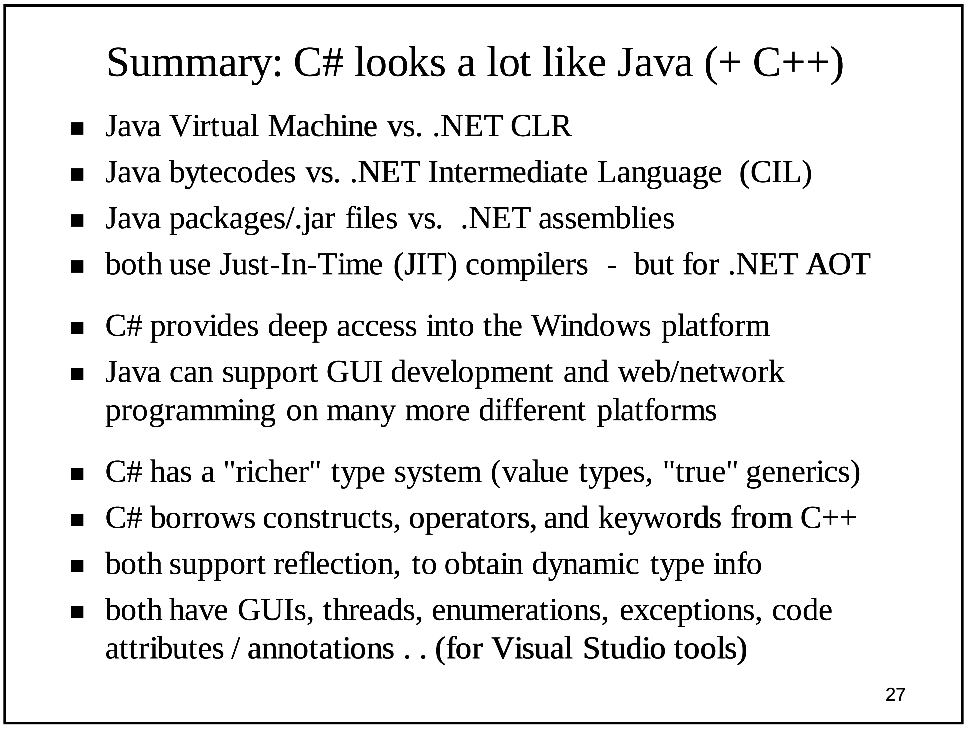
C# "Hello, world!" program

```
using System;                    //  required to use "Console"
class HelloWorld
{
    public static void Main () {
        Console.WriteLine ("Hello, world!");
        Console.ReadKey ();  //   or System.Console
    }
}
```

In Common Intermediate Language (CIL)

```
.class private auto ansi beforefieldinit HelloWorld
        extends [mscorlib]System.Object   {
    .method public hidebysig static void Main () cil managed {
        .entrypoint
        .maxstack  8                    //  max size of IL eval stack
        L_0000: nop                     //  fill space (potential patching)
        L_0001: ldstr "Hello, world!"   //  push constant string
        L_0006: call void               //  call method WriteLine
          [mscorlib]System.Console::WriteLine (string)
        L_000b: nop                     //  note that everything is fully named
        L_000c: call valuetype [mscorlib]System.ConsoleKeyInfo
          [mscorlib]System.Console::ReadKey ()
        L_0011: pop                     //  pop returned value
        L_0012: ret                     //  exit from Main
    }
}
```


assembly name



Summary: C# looks a lot like Java (+ C++)

- Java Virtual Machine vs. .NET CLR
- Java bytecodes vs. .NET Intermediate Language (CIL)
- Java packages/.jar files vs. .NET assemblies
- both use Just-In-Time (JIT) compilers - but for .NET AOT
- C# provides deep access into the Windows platform
- Java can support GUI development and web/network programming on many more different platforms
- C# has a "richer" type system (value types, "true" generics)
- C# borrows constructs, operators, and keywords from C++
- both support reflection, to obtain dynamic type info
- both have GUIs, threads, enumerations, exceptions, code attributes / annotations . . (for Visual Studio tools)

27