MT UNIT-1

By

K. Bhaskara Rao

Asst. Prof, IT Dept.

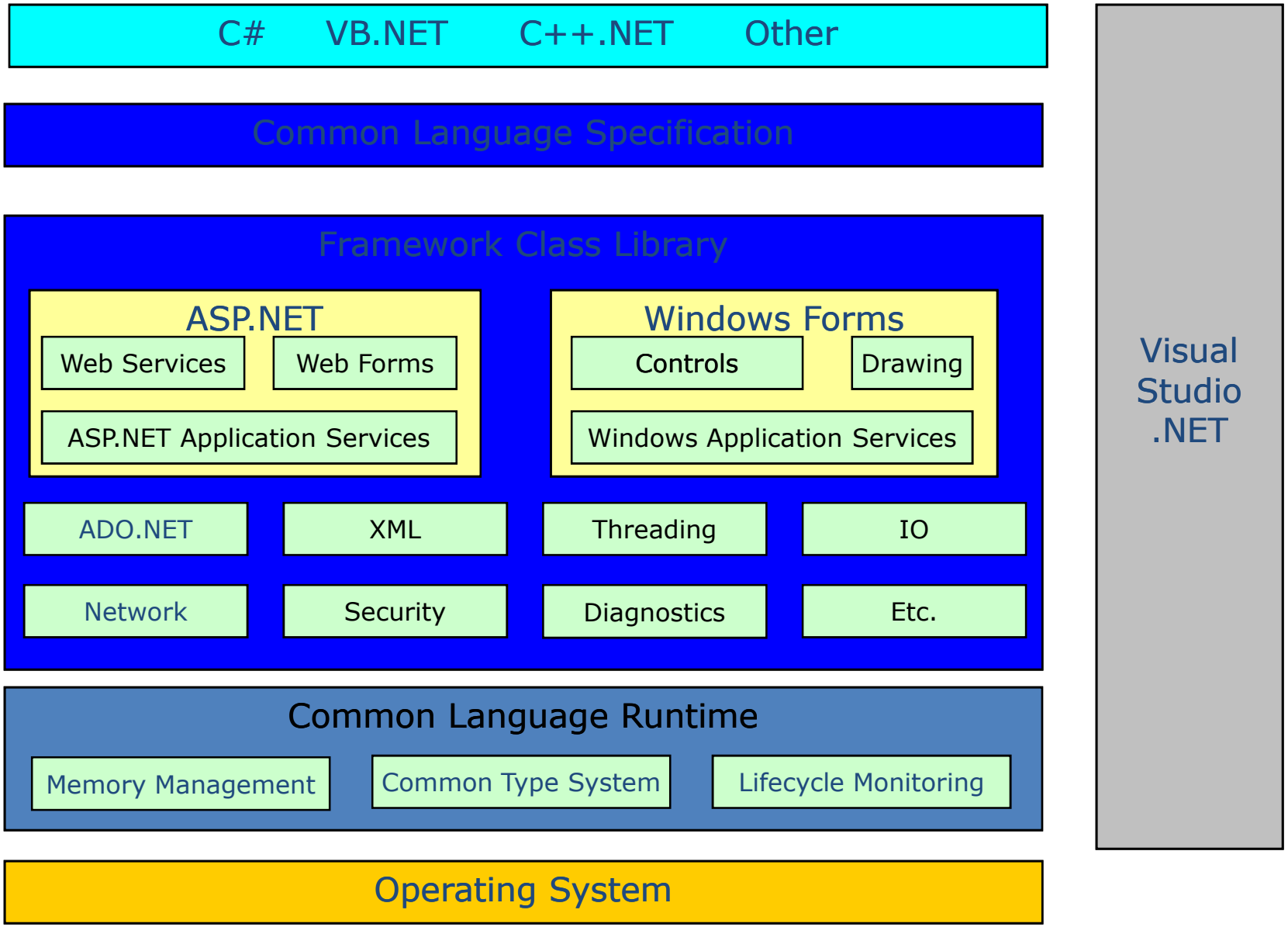
Before .NET

- Windows GUI development: Win32 API, MFC, Visual Basic
- Web development: ASP
- Java – “Write once, run anywhere.”

.net Framework

- First developed by Microsoft in 2000
- Targets primarily Windows OS, but Mono Project (headed by Novell) supports Linux, Unix, FreeBSD, Mac OS X, Solaris
- Primary languages: C#, Visual Basic .NET, C++/CLI, and J#
- Third-party languages: Ada, COBOL, LISP, Perl, Ruby, and many more.

.NET Framework



.NET framework versions

.NET framework version	Visual Studio	Released Year
1.0	Visual Studio	2002
1.1	Visual Studio 2003	2003
2.0	Visual Studio 2005	2005
3.5	Visual Studio 2008	2007
4.0	Visual Studio 2010	2010
4.5	Visual Studio 2012	2012
4.5.1	Visual Studio 2013	2013
4.6	Visual Studio 2015	2015

.NET framework versions

.NET framework version	Visual Studio	Released Year
4.7.2	**Visual Studio 2017**	2017
4.8	**Visual Studio 2019**	2019
4.8	**Visual Studio 2022**	2022

.NET design principles

- Interoperability
- Language Independence
- Portability
- Security
- Memory Management
- Easy Deployment
- Performance

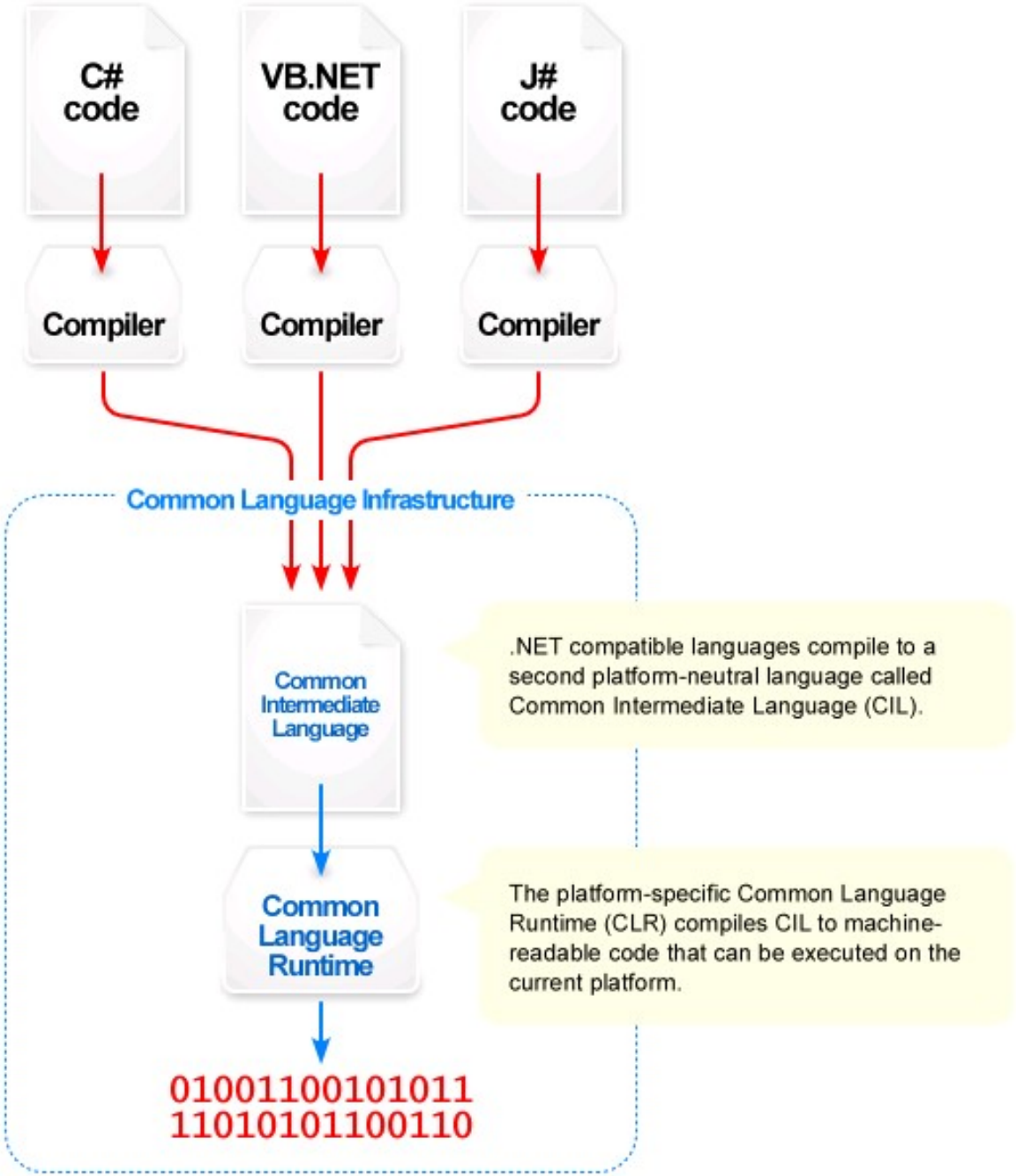
Some .NET Languages

- **C#**
- **COBOL**
- **Eiffel**
- **Fortran**
- **Mercury**
- **Pascal**
- **Python**
- **SML**
- **Perl**
- **Smalltalk**
- **VB.NET**
- **VC++.NET**
- **J#.NET**
- **Scheme**

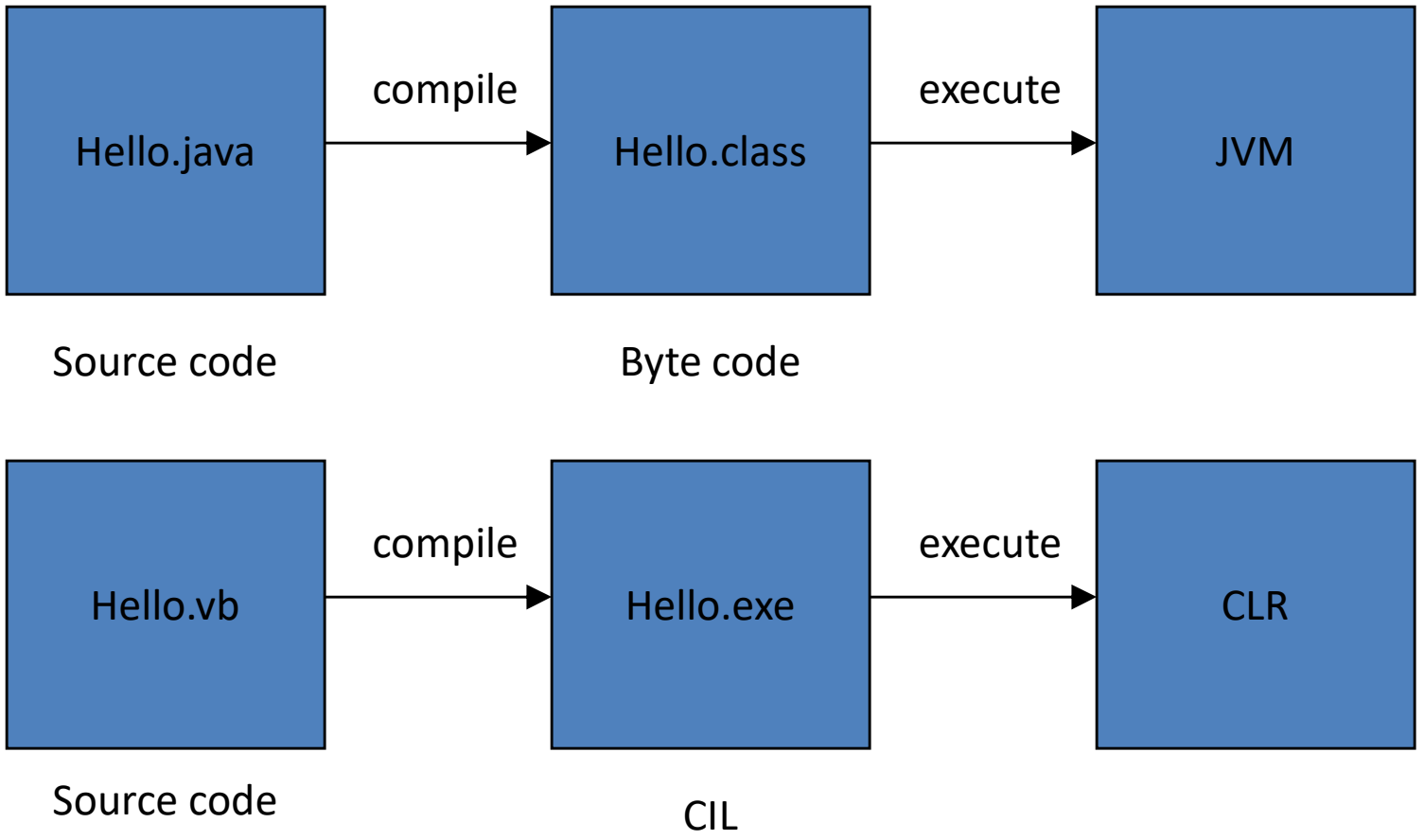
….

Language Compiler List

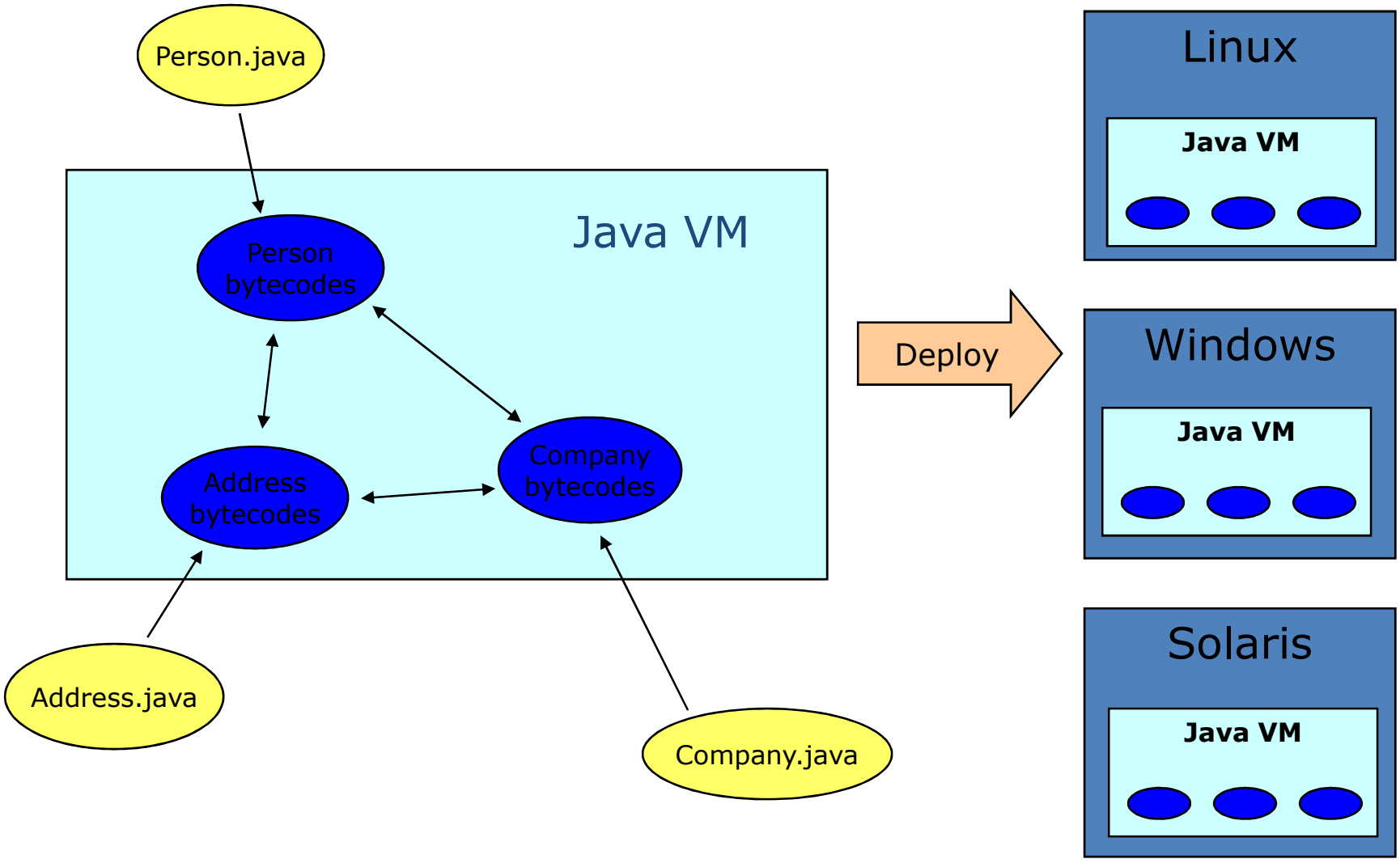
- Ada
- APL
- Basic (Visual Basic)
- C#
- C
- C++
- Java
- COBOL
- Component Pascal (Queensland U Tech)
- ECMAScript (JScript)
- Eiffel (Monash U.)
- Haskell (Utrecht U.)
- lcc (MS Research Redmond)
- Mondrian (Utrecht)
- ML (MS Research Cambridge)
- Mercury (Melbourne U.)
- Oberon (Zurich University)
- Oz (Univ of Saarlandes)
- Perl
- Python
- Scheme (Northwestern U.)
- SmallTalk


C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

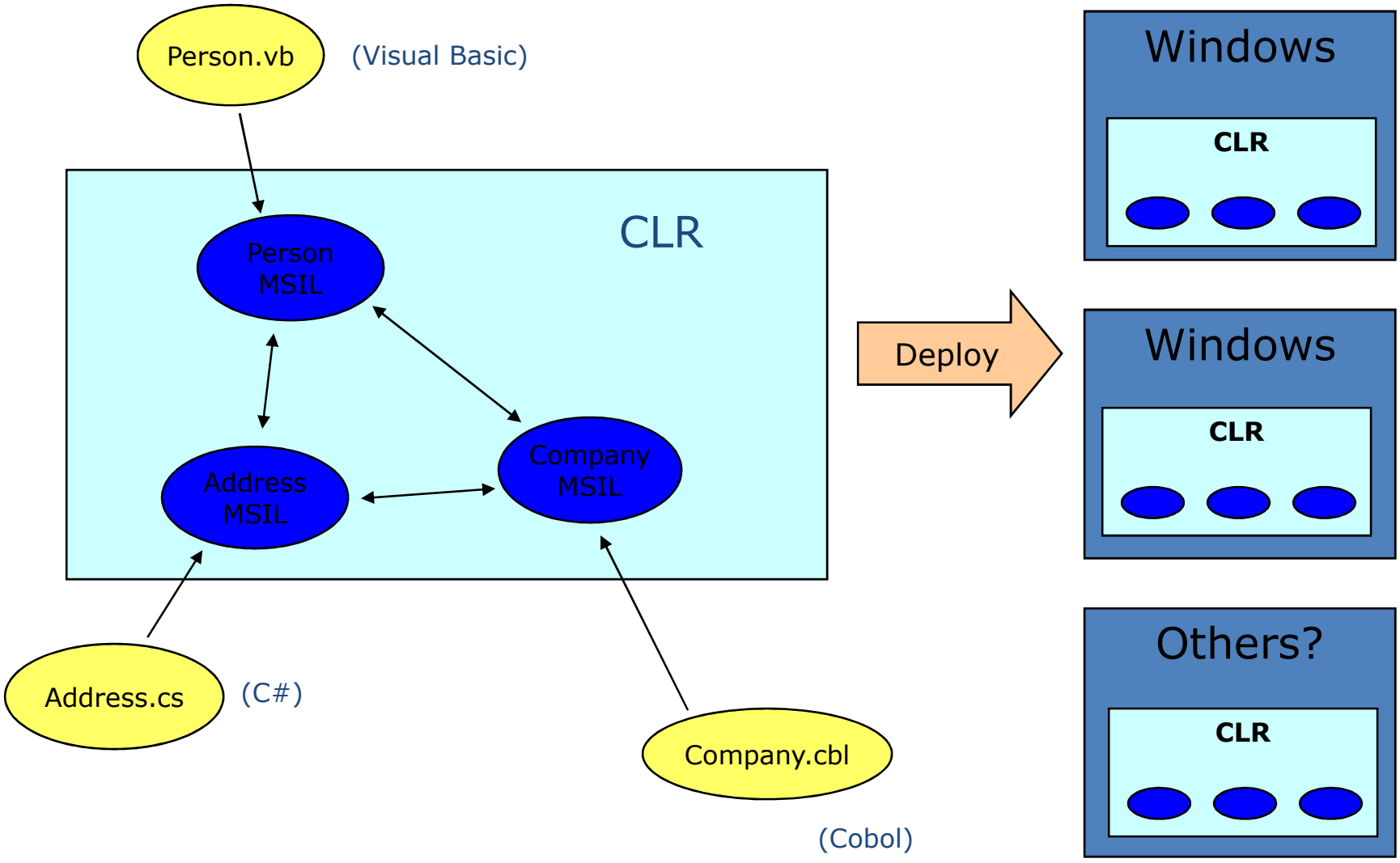
Comparison to Java



J2EE: Language-Specific, Platform- Independent



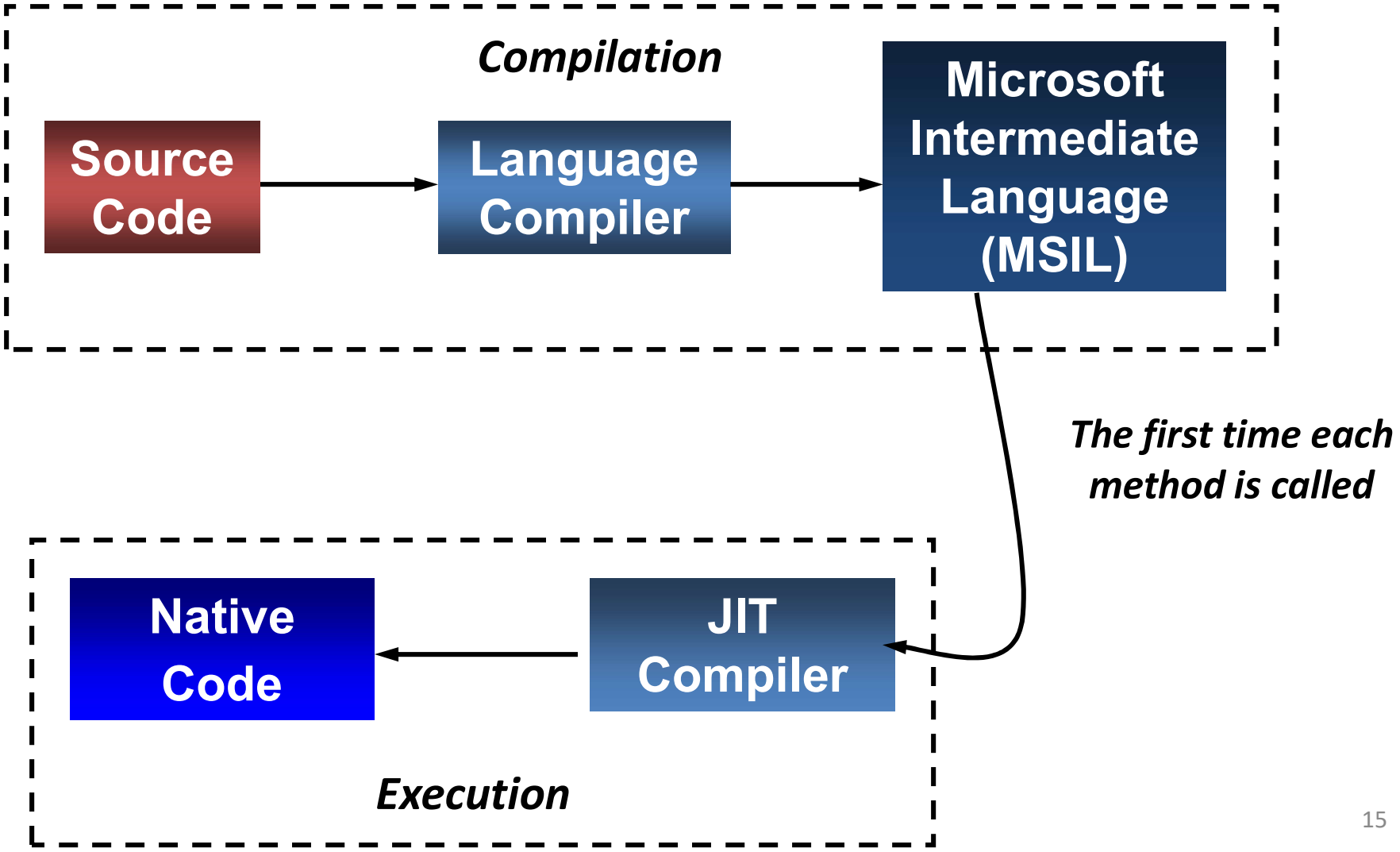
.NET: Language-Independent, (Mostly) Platform- Specific



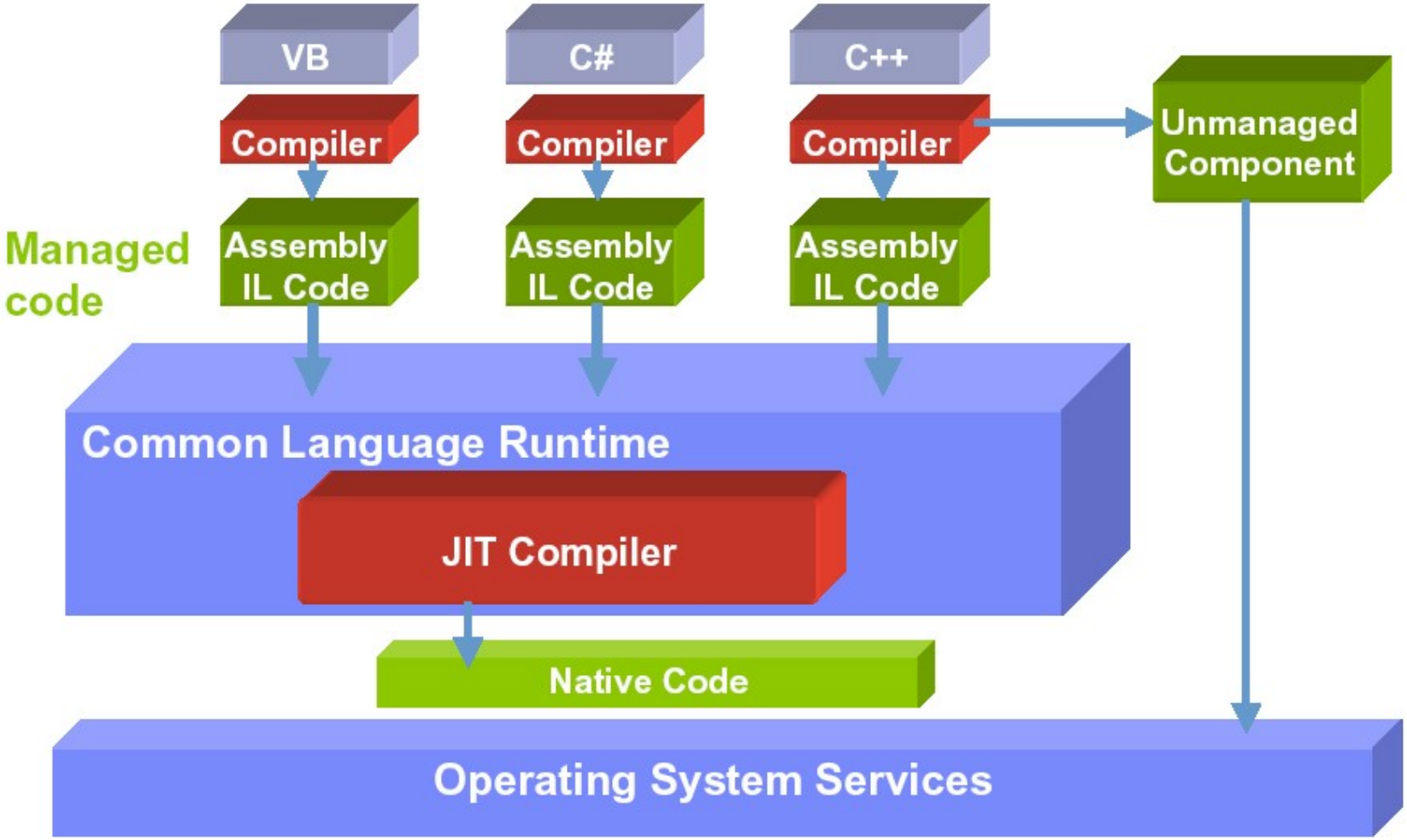
Java and .NET: Runtime environments

- Java
  - Intermediate language is *bytecode*
  - Original design targeted interpretation
  - Java VMs with JIT compilation are now also used
- .NET Framework
  - Intermediate language is *MSIL*
  - Provides JIT compilation
  - What is JIT?
  - Just-In-Time compilation: translates a bytecode method into a native method on the fly, so as to remove the overhead of interpretation

Compiling and executing managed code



Common Language Runtime



Basic Truths

- .NET
  - Windows-centric and language-neutral
  - .NET is a Microsoft product strategy that includes a range of products from development tools and servers to end-user applications.
- Platform-neutral version of .NET is available
  - Mono –Novell-sponsored, open source implementation of the .NET development environment
  - (http://www.go-mono.com)

.NET Class Library

- IO
- GUI Programming
- System Information
- Collections
- Components
- Application Configuration
- Connecting to Databases (ADO.NET)
- Tracing and Logging
- Manipulating Images/Graphics

Class Library

- Interoperability with COM
- Globalization and Internationalization
- Network Programming with Sockets
- Remoting
- Serialization
- XML
- Security and Cryptography
- Threading
- Web Services

Namespace

Namespace	*Description*
System	Provides base data types and almost 100 classes that deal with situations like exception handling, mathematical functions, and garbage collection.
System.CodeDom	Provides the classes needed to produce source files in all the .NET languages.
System.Collections	Provides access to collection classes such as lists, queues, bit arrays, hash tables, and dictionaries.
System.ComponentModel	Provides classes that are used to implement runtime and design-time behaviors of components and controls.
System.Configuration	Provides classes and interfaces that allow you to programmatically access the various configuration files that are on your system, such as the web.config and the machine.config files.
System.Data	Provides classes that allow data access and manipulation to SQL Server and OleDb data sources. These classes make up the ADO.NET architecture.
System.Diagnostics	Provides classes that allow you to debug and trace your application. There are classes to interact with event logs, performance counters, and system processes.

Namespace

Namespace	*Description*
System.DirectoryServices	Provides classes that allow you to access Active Directory.
System.Drawing	Provides classes that allow you to access the basic and advanced features of the new GDI+ graphics functionality.
System.EnterpriseServices	Provides classes that allow you to access COM+ services.
System.Globalization	Provides classes that access the global system variables, such as calendar display, date and time settings, and currency display settings.
System.IO	Provides classes that allow access to file and stream control and manipulation.
System.Management	Provides access to a collection of management information and events about the system, devices, and applications designed for the Windows Management Instrumentation (WMI) infrastructure.
System.Messaging	Provides classes that allow you to access message queue controls and manipulators.
System.Net	Provides access to classes that control network services. These classes also allow control over the systems sockets.

Namespaces

Namespace	*Description*
System.Reflection	Provides classes that allow control to create and invoke loaded types, methods, and fields.
System.Resources	Provides classes that allow you to create and manage culture-specific resources.
System.Runtime.Remoting	Provides classes that allow the management of remote objects in a distributed environment.
System.Security	Provides classes that allow access to authentication, authorization, cryptography, permissions, and policies.
System.ServiceProcess	Provides classes that give control over Windows services.
System.Text	Provides classes for working with and manipulating text strings.
System.Threading	Provides classes for threading issues and allows you to create multithreaded applications.
System.Timers	Provides the capability to raise events on specified intervals.

Namespaces

Namespace	*Description*
System.Web	Provides numerous classes that are used in ASP.NET Web application development.
System.Web.Services	Provides classes that are used throughout this book to build, deploy, and consume Web services.
System.Windows.Forms	Provides classes to build and deploy Windows Forms applications.
System.Xml	Provides classes to work with and manipulate XML data.

Primitive Types

C# Type	.NET Framework type
bool	System.Boolean
byte	System.Byte
sbyte	System.Sbyte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single

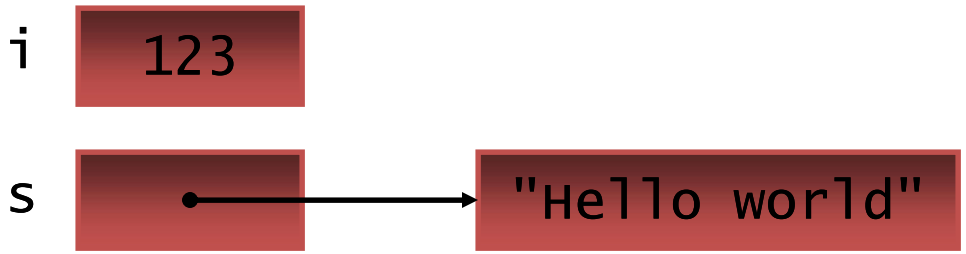
Primitive Types (contd.)

int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Type System

- Value types
  - Directly contain data
  - Cannot be null
- Reference types
  - Contain references to objects
  - May be null

```
int i = 123;
string s = "Hello world";
```



Type System

- Value types
  - Primitives `int i;`
  - Enums `enum State { Off, On }`
  - Structs `struct Point { int x, y; }`
- Reference types
  - Classes `class Foo: Bar, IFoo {...}`
  - Interfaces `interface IFoo: IBar {...}`
  - Arrays `string[] a = new string[10];`
  - Delegates `delegate void Empty();`
  - events
  -

Predefined Types

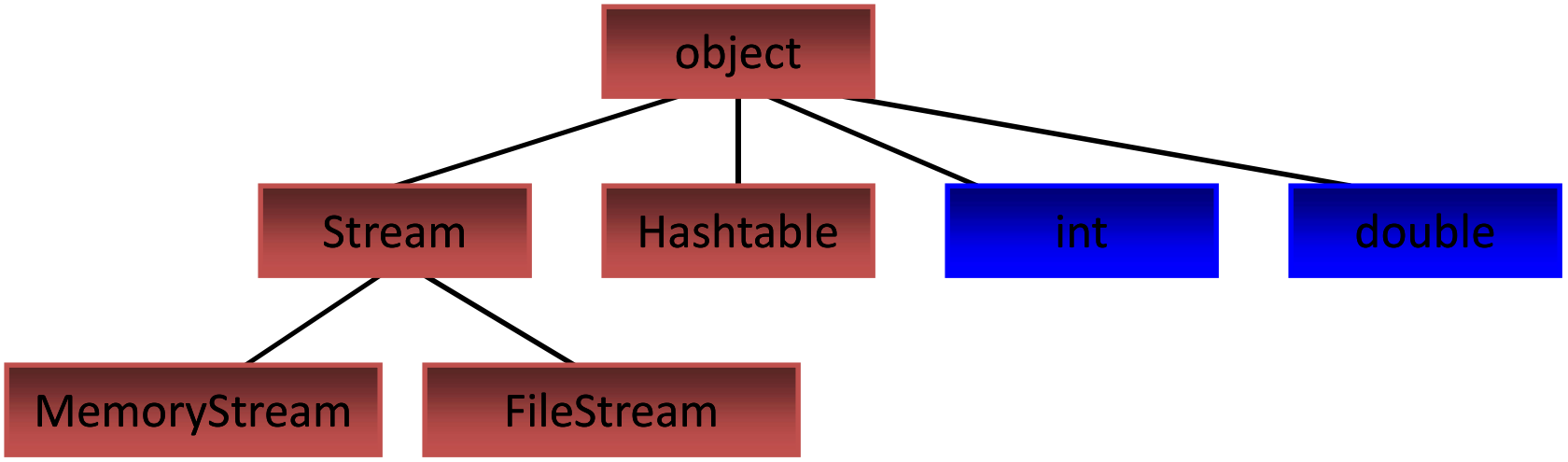
- C# predefined types
  - Reference object, string
  - Signed sbyte, short, int, long
  - Unsigned byte, ushort, uint, ulong
  - Character char
  - Floating-point float, double, decimal
  - Logical bool
- Predefined types are simply aliases for system-provided types
  - For example, int == System.Int32

Predefined Types

- C# predefined types
  - Reference object, string
  - Signed sbyte, short, int, long
  - Unsigned byte, ushort, uint, ulong
  - Character char
  - Floating-point float, double, decimal
  - Logical bool
- Predefined types are simply aliases for system-provided types
  - For example, int == System.Int32

Unified Type System

- Everything is an object
  - All types ultimately inherit from object
  - Any piece of data can be stored, transported, and manipulated with no extra work



Statements And Expressions

- High C++ fidelity
- If, while, do require bool condition
- goto can't jump into blocks
- Switch statement
  - No fall-through, "goto case" or "goto default"
- foreach statement
- Checked and unchecked statements
- Expression statements must do work

```
void Foo() {
   i == 1;    // error
}
```

Comments in C#

- Both /* … */ and // can be used for comments.
- VS provides comment/uncomment selections.
  - Use the menu bar, or Ctrl-K Ctrl-C for comment and Ctrl-K Ctrl-U for uncomment

Control Statements in C#

- ✓ if statement
- ✓ if else statement
- ✓ nested if statement
- ✓ switch statement
- ✓ goto
- ✓ while
- ✓ do while
- ✓ for
- ✓ foreach

Control Statements in C#

- ✓ if statement
- ✓ if else statement
- ✓ nested if statement

syntax of if else:

if (*expression*) *statement1* [**else** *statement2*]

Syntax of nested if else:

```
if ( expression )
        code_block
  else if ( expression_1 ) code_block
  else if ( expression_2 ) code_block
  else code_block
```

Control Statements in C#

switch statement:

syntax:

switch (*expression***)**
{ case *constant-expression***:** *statements*
break;
case *constant-expression***:** *statements*
break;
[**default:** *jump-statement*]
}

while statement:

Syntax: while (expression) { ... }

Control Statements in C#

do while statement:

Syntax:

```
do { ... }
while ( expression );
```

for statement:

Syntax:

```
for ( initializer; exit condition; iterative code ) { ...
}
```

foreach statement:

Syntax:

foreach (*type identifier* **in** *expression***)** *statement*

Predefined Data types

Integer types :

Name	CTS Type	Description	Range (min:max)
sbyte	System.sByte	signed 8bit int	-128 : 127
short	System.Int16	signed 16bit int	-32768 : 32767
int	System.Int32	signed 32bit int	-2^{31} : $2^{31} - 1$
long	System.Int64	signed 64bit int	-2^{63} : $2^{63} - 1$
byte	System.Byte	unsigned 8bit	0 : 255
ushort	System.UInt16	unsigned 16bit	0 : 65535
uint	System.UInt32	unsigned 32bit	0 : $2^{32} - 1$
ulong	System.UInt64	unsigned 64bit	0 : $2^{64} - 1$

Example :

```
uint ui = 1234U;
long l=123L;
ulong ul=122UL;
```

Floating point type :

float	System.Single single precision 32bit
double	System.Double double precision 32bit
for float,	range : +/-1.5 * 10^{-45} to +/-3.4 * 10^{38}
for double,	range: +/-5.0 * 10^{-324} to +/- 1.7 * 10^{308}

Example :

```
float f = 12.4F;
```

Predefined Data types

Name	CTS Type	Description	Range (min:max)
decimal	System.Decimal	128bit precision	range : +/-1.0 x 10^{-28} to +/- 7.9 x 10^{28}
boolean	System.Boolean		
char	System.Char		
object	System.Object	the root type	
string	System.String	unicode char string	

Parameter passing

- Small data are defined as value types
- Large objects are defined as reference types

Parameter passing mechanisms :

1 . call by Value
2. call by Reference

- By default, all parameters are passed using call by value (even refence types also)

ref keyword :

- Allows passing of both value types and reference types to methods using call by reference mechanism.
- class A{ public void method1(ref int i) { } }

out keyword :

- Used to pass values out of a method. Class B{ public void m2(out int a, out double b){ int c; --- return c;}
- Calling : B bob=new B(); bob.m2(out x, out y); }

Passing Parameters

- Passing a value variable by default refers to the Pass by Value behavior as in Java

```
public static void foo(int a)
{
        a=1;
}

static void Main(string[] args)
{
        int x=3;
        foo(x);
        Console.WriteLine(x);
}
```

This outputs the value of 3 because x is passed by value to method foo, which gets a copy of x's value under the variable name of a.

Passing by Reference

- C# allows a ref keyword to pass value types by reference:

```
public static void foo(int ref a)
{
        a=1;
}

static void Main(string[] args)
{
        int x=3;
        foo(ref x);
        Console.WriteLine(x);
}
```

The ref keyword must be used in both the parameter declaration of the method and also when invoked, so it is clear what parameters are passed by reference and may be changed.
Outputs the value of 1 since variable a in foo is really a reference to where x is stored in Main.

Strings

- ✓ Strings are immutable (to make operations efficient)
- ✓ Strings are differently represented in different languages.

Example : string s1="Hello World";
string s2=s1;

String class methods :

Method	Description
CompareTo	- Compares this string instance with another string instance.
Contains	- Returns a Boolean indicating whether the current string instance contains the given substring.
CopyTo	- Copies a substring from within the string instance to a specified location within an array of characters.
EndsWith	- Returns a Boolean value indicating whether the string ends with a given substring.
Equals	- Indicates whether the string is equal to another string. You can use the '==' operator as well.
IndexOf	- Returns the index of a substring within the string instance.

Strings contd…

Method	Description
IndexOfAny	- Returns the first index occurrence of any character in the substring within the string instance.
PadLeft	- Pads the string with the specified number of spaces or another Unicode character, effectively right-justifying the string.
PadRight	- Appends a specified number of spaces or other Unicode character to the end of the string, creating a left-justification.
Remove	- Deletes a given number of characters from the string.
Replace	- Replaces all occurrences of a given character or string within the string instance with the specified replacement.
Split	- Splits the current string into an array of strings, using the specified character as the splitting point.

Strings contd…

StartsWith - Returns a Boolean value indicating whether the string instance starts with the specified string.

SubString - Returns a specified portion of the string, given a starting point and length.

ToCharArray - Converts the string into an array of characters.

ToLower - Converts the string into all lowercase characters.

ToUpper - Converts the string into all uppercase characters.

Trim - Removes all occurrences of a given set of characters from the beginning and end of the string.

TrimStart - Performs the TRim function, but only on the beginning of the string.

TrimEnd - Performs the TRim function, but only on the end of the string.

Strings contd…

Copy() vs Clone()

- The Copy() method creates a new instance of string with the same value as a specified string.
- The Clone() method returns a reference to the string which is being cloned. It is not an independent copy of the string on the Heap. It is another reference on the same string.

Strings contd…

Copy() vs Clone()

```
using System;
public class CopyClone
{
      static void Main()
      {
                                    string str = "ZetCode";
            string cloned = (string) str.Clone();
            string copied = string.Copy(str);
           Console.WriteLine(str.Equals(cloned)); // prints true
           Console.WriteLine(str.Equals(copied)); // prints true
           Console.WriteLine(ReferenceEquals(str, cloned)); // prints true
                        Console.WriteLine(ReferenceEquals(str, copied)); // prints false
      }
}
```

Strings contd…

== operator will test for value equivalence for value types
and for address (or reference) equivalence for reference types
Equals method → will see for content equivalenve

```
namespace StrEquEx{
    public class Thing
    {
            private int i;
            public Thing(int i)
            { this.i=i; }
     }
   static void Main()
  {
            Thing t1=new Thing(123);
            Thing t2=new Thing(123);
            Console.Write(t1==t2);     // false
            string a="Hello"; string b="Hello";
            Console.Write(a==b);  //true
  }
```

StringBuilder class

Constructor	Description
StringBuilder()	Initializes a new instance of the StringBuilder class.
StringBuilder(Int32)	Initializes a new instance of the StringBuilder class using the specified capacity.
StringBuilder(String)	Initializes a new instance of the StringBuilder class using the specified string.
StringBuilder(Int32, Int32)	Initializes a new instance of the StringBuilder class that starts with a specified capacity and can grow to a specified maximum.
StringBuilder(String, Int32)	Initializes a new instance of the StringBuilder class using the specified string and capacity.

StringBuilder class

Properties	Description
Capacity	Gets or sets the maximum number of characters that can be contained in the memory allocated by the current instance.
Chars	Gets or sets the character at the specified character position in this instance.
Length	Gets or sets the length of the current StringBuilder object.
MaxCapacity	Gets the maximum capacity of this instance.

Properties:

Topics

- Types of Parameters → value parameters, ref, out, parameter array
- Method Overloading and Constructor overloading
- Types of constructors → default, parameterized, private, static, copy constructor
- Default constructor → a constructor with out any parameters. This is called when class is instantiated
-

Classes & Objects

- A *class is a template that defines the form of an*
- object. It specifies both the data and the code that will operate on that data.
- C# uses a class specification to construct *objects.*
- *Objects are instances of a class.* class is a logical abstraction.
- It is not until an object of that class has been created that a physical representation of that class exists in memory.
- When you define a class, you declare the data that it contains and the code that operates on it. While very simple classes might contain only code or only data, most real-world classes contain both.
- data is contained in *data members defined by the class, and code is* contained in *function members.*
- **data members** (also called *fields) include instance variables and static variables.* ***Function members*** *include methods,* constructors, destructors, indexers, events, operators, and properties

General Form of a class

class *classname {*
// declare instance variables
access type var1;
access type var2;
// ...
access type varN;
// declare methods
access ret-type method1(parameters) {
// body of method
}
access ret-type method2(parameters) {
// body of method
}
// ...
access ret-type methodN(parameters) {
// body of method
}
}

Class example

```
class Class1
{
        static void Main(string[] args)
        {
                // Your code would go here, e.g.
                Console.WriteLine("hi");
        }
        /* We can define other methods and vars for the class */
        // Constructor
        Class1()
        {
                // Code
        }
        // Some method, use public, private, protected
        // Use static as well just like Java
        public void foo()
        {
                // Code
        }
        // Instance, Static Variables
        private int m_number;
        public static double m_stuff;
}
```

Class Example

```
class Building {
public int Floors; // number of floors
public int Area; // total square footage of building
public int Occupants; // number of occupants
}
```

Object creation:

```
Building house = new Building();     // create an object of type
                                        building
```

Accessing members :

```
house.Floors = 2;
```

Class Example

```
public class BankAccount
{
        public double m_amount;
        BankAccount(double d) {
                m_amount = d;
        }
        public virtual string GetInfo() {
                return “Basic Account”;
        }
}

public class SavingsAccount : BankAccount
{
        // Savings Account derived from Bank Account
        // usual inheritance of methods, variables
        public double m_interest_rate;
        SavingsAccount(double d) : base(100) { // $100 bonus for signup
                m_interest_rate = 0.025;
        }
        public override string GetInfo() {
                string s = base.GetInfo();
                return s + “ and Savings Account”;
        }
}
```

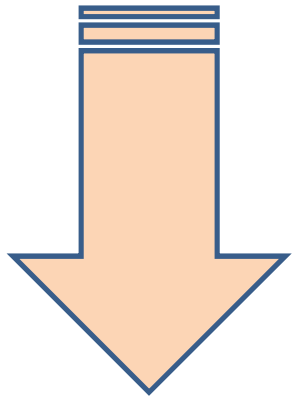
Sample Class Usage

```
SavingsAccount a = new SavingsAccount(0.05);
CoConsole.WriteLine(a.m_interest_rate);
Console.WriteLine(a.GetInfo());
```

Then the output is:

```
        100
        0.05
        Basic Account and Savings Account
nsole.WriteLine(a.m_amount);
```

Constructors



Constructor

A *constructor* *initializes an object when it is created. It has the same name as its class and* is syntactically similar to a method. However, constructors have no explicit return type.

- ✓ The general form of a constructor is shown here:

```
access class-name(param-list) {
// constructor code
}
```

- ✓ Typically, you will use a constructor to give initial values to the instance variables defined by the class.
- ✓ usually, *access is* ***public because constructors are normally called from outside*** their class. The *param-list can be empty, or it can specify one or more parameters.*
- C# automatically provides a default constructor that causes all member variables to be initialized to their default values. For most value types, the default value is zero. For **bool, the default is false.** For reference types, the default is null.

Parameterized Constructors

- ✓ Most often you will need a constructor that has one or more parameters.
- ✓ Parameters are added to a constructor in the same way they are added to a method: just
- ✓ Declare them inside the parentheses after the constructor's name.

Constructor

```
access class-name(param-list) {
// constructor code
}
```

Ex:

```
// A parameterized constructor.
using System;
class MyClass {
public int x;
public MyClass(int i) {
x = i;
}
}
class ParmConsDemo {
static void Main() {
MyClass t1 = new MyClass(10);
MyClass t2 = new MyClass(88);
Console.WriteLine(t1.x + " " + t2.x);
}
}
```

Parameters to methods

- value types → built in types, structs, enum etc…
- reference types → arrays, class instances etc…
- ref type
- out type
- params → allows variable no of parameters to methods

const, readonly keywords

```
class A{
 public const int i=10;   //static by default
public readonly int j;
void m1()
{
   j=20;   //after assignment j becomes const
}
}
```

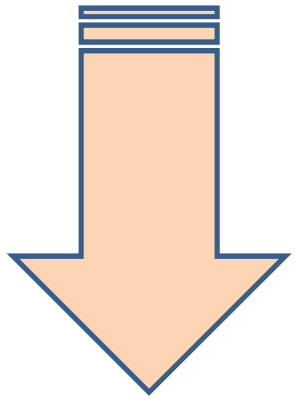
constructors

- default constructor
- parameterized constructor
- private constructor
- static constructor
- copy constructor
- overloaded constructor

Invoke an Overloaded Constructor Through this keyword:

constructor-name(parameter-list1) : this(parameter-list2) {
// ... body of constructor, which may be empty
}

base keyword



base Keyword – calling base class constructor

- ✓ When a derived class specifies a **base** clause, it is calling the constructor of its immediate base class.
- ✓ **base always refers** to the base class immediately above the calling class
- ✓ You pass arguments to the base constructor by specifying them as arguments to **base.**
- ✓ A reference variable of a base class can be assigned a reference to an object of any class derived from that base class.

Syntax:

derived-constructor(parameter-list) : base(arg-list) {

// body of constructor

}

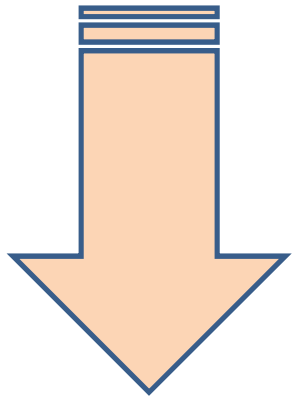
arg-list → *specifies any arguments needed by the constructor in the base class*

base Keyword – calling base class constructor

Example:

```
class A {
    public int i = 0;
}
// Create a derived class.
class B : A {
    new int i; // this i hides the i in A
    public B(int a, int b) {
    base.i = a; // this uncovers the i in A
    i = b; // i in B
}
```

Member access

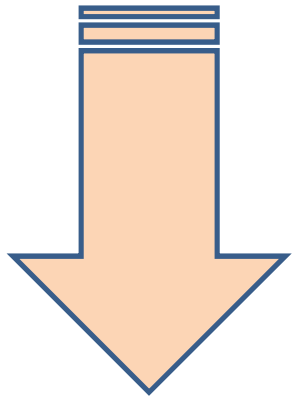


Class Member Access

	public	protected	internal	protected internal	private	private protected
Entire program	Yes	No	No	No	No	No
Containing class	Yes	Yes	Yes	Yes	Yes	Yes
Current assembly	Yes	No	Yes	Yes	No	No
Derived types	Yes	Yes	No	Yes	No	No
Derived types within current assembly	Yes	Yes	Yes	Yes	No	Yes

- ✓
- ✓
- ✓
- ✓

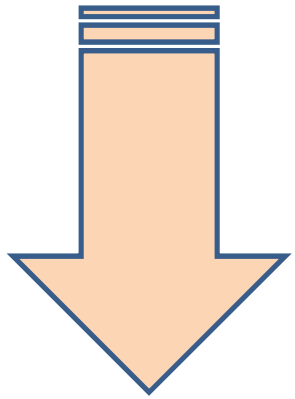
Method overloading & Constructor overloading



Method overriding using inheritance

```
using System;
class a //base class
{
    public void cname()
    {
        Console.WriteLine("This is first method");
    }
}
```

Member access



Method Overloading

- ✓ In C#, two or more methods within the same class can share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- ✓ Method overloading is one of the ways that C# implements polymorphism.
- ✓ In declaring Overloaded Methods, It is not sufficient for two methods to differ only in their return types. They must differ in the types or number of their parameters.
- ✓ When an overloaded method is called, the version of the method executed is the one whose parameters match the arguments.

```
class b : a //drived class
{
    new public  void cname()
// new keyword is used to hiding method cname of
class a
    {
        Console.WriteLine("This is second method");
    }
    public static void Main()
    {
        b obj = new b();
        b.cname();
        Console.Read();
    } }
```

Custom DateTime Format Specifiers

Specifier	Description
d	Displays the current day of the month.
dd	Displays the current day of the month, where values < 10 have a leading zero.
ddd	Displays the three-letter abbreviation of the name of the day of the week.
dddd(+)	Displays the full name of the day of the week represented by the given DateTime value.
f(+)	Displays the x most significant digits of the seconds value. The more f's in the format specifier, the more significant digits. This is total seconds, not the number of seconds passed since the last minute.
F(+)	Same as f(+), except trailing zeros are not displayed.
g	Displays the era for a given DateTime (for example, "A.D.")
h	Displays the hour, in range 1-12.
hh	Displays the hour, in range 1-12, where values < 10 have a leading zero.
H	Displays the hour in range 0-23.

Custom DateTime Format Specifiers

Specifier	Description
HH	Displays the hour in range 0-23, where values < 10 have a leading zero.
m	Displays the minute, range 0-59.
mm	Displays the minute, range 0-59, where values < 10 have a leading zero.
M	Displays the month as a value ranging from 1-12.
MM	Displays the month as a value ranging from 1-12 where values < 10 have a leading zero.
MMM	Displays the three-character abbreviated name of the month.
MMMM	Displays the full name of the month.
s	Displays the number of seconds in range 0-59.
ss(+)	Displays the number of seconds in range 0-59, where values < 10 have a leading 0.
T	Displays the first character of the AM/PM indicator for the given time.
tt(+)	Displays the full AM/PM indicator for the given time.
y/yy/yyyy	Displays the year for the given time.
z/zz/zzz(+)	Displays the timezone offset for the given time.

Numeric Custom Formatting Specifiers

Numeric Custom Format Specifiers

Specifier	Description
0	The zero placeholder.
#	The digit placeholder. If the given value has a digit in the position indicated by the # specifier, that digit is displayed in the formatted output.
.	Decimal point.
,	Thousands separator.
%	Percentage specifier. The value being formatted will be multiplied by 100 before being included in the formatted output.
E0/E+0/e/e+0/e-0/E	Scientific notation.
'XX' or "XX“	Literal strings. These are included literally in the formatted output without translation in their relative positions.
;	Section separator for conditional formatting of negative, zero, and positive values.

Arrays

- An array is a collection of variables of the same type that are referred to by a common name.
- Arrays are implemented as objects

One dimensional array:

type[ ] array-name = new type[size];

Ex : int[] sample = new int[10];

(or)

int[] sample;

sample = new int[10];

Initialization:

int[] nums = new int[] { 99, 10, 100, 18, 78, 23, 63, 9, 87, 49 };

Arrays

Initialization:

int[] nums;

nums = new int[] { 99, 10, 100, 18, 78, 23,63, 9, 87, 49 };

Multi dimensiona arrays:

int[,] table = new int[10, 20];

type[, ...,] name = new type[size1, size2, ..., sizeN];

int[,,] multidim = new int[4, 10, 3];

multidim[2, 4, 1] = 100;

Initialization:

type[,] array_name = {

{ *val, val, val, ..., val },*

{ *val, val, val, ..., val },*

...

{ *val, val, val, ..., val }*

};

Arrays

System.Array methods and properties:

Method or property	Purpose
BinarySearch( )	Overloaded public static method that searches a one-dimensional sorted array.
Clear( )	Public static method that sets a range of elements in the array either to 0 or to a null reference.
Copy( )	Overloaded public static method that copies a section of one array to another array.
CreateInstance( )	Overloaded public static method that instantiates a new instance of an array.
IndexOf( )	Overloaded public static method that returns the index (offset) of the first instance of a value in a one-dimensional array.
LastIndexOf( )	Overloaded public static method that returns the index of the last instance of a value in a one-dimensional array.

Arrays

Method or property	Purpose
Reverse( )	Overloaded public static method that reverses the order of the elements in a one-dimensional array.
Sort( )	Overloaded public static method that sorts the values in a one-dimensional array.
IsFixedSize	Required because Array implements ICollection. With arrays, this will always return true (all arrays are of a fixed size).
IsReadOnly	Public property (required because Array implements IList) that returns a Boolean value indicating whether the array is read-only.
IsSynchronized	Public property (required because Array implements ICollection) that returns a Boolean value indicating whether the array is thread-safe.
Length	Public property that returns the length of the array.
Rank	Public property that returns the number of dimensions of the array.

Arrays

GetEnumerator( )	Public method that returns an IEnumerator.
GetLowerBound( )	Public method that returns the lower boundary of the specified dimension of the array.
GetUpperBound( )	Public method that returns the upper boundary of the specified dimension of the array.
Initialize( )	Initializes all values in a value type array by calling the default constructor for each value. With reference arrays, all elements in the array are set to null.
SetValue( )	Overloaded public method that sets the specified array elements to a value.

Jagged Arrays

Definition : A jagged array (also referred to as an array of arrays) is an array (single-dimension or multidimensional) in which the elements are themselves arrays.

Declarations:

```
string[ ][ ] jaggedStrings;
int[ ][ ] jaggedInts;
Customer[ ][ ] jaggedCustomers;

int[ ][ ] jaggedInts = new int[2][ ];
 jaggedInts[0] = new int[10];
jaggedInts[1] = new int[20];
```

Jagged Arrays

Initialization during declaration:

```
int[ ][ ] ji2 = new int[2][ ];
ji2[0] = new int[ ] { 1, 3, 5, 7, 9 };
 ji2[1] = new int[ ] { 2, 4, 6 };
                    (or)
int[ ][ ] ji3 = {
                        new int[ ] { 1, 3, 5, 7, 9 },
                        new int[ ] { 2, 4, 6 }
                    };
                        (or)
int[ ][ ] ji4 = new int[ ][ ]
                    {
                        new int[ ] { 1, 3, 5, 7, 9 },
                        new int[ ] { 2, 4, 6 }
                    };
```

Jagged Arrays

Mixed jagged and rectangular array :

```
int[ ][,] mixedJagged = new int[ ][,] {
    new int[,] { {0,1}, {2,3}, {4,5} },
    new int[,] { {9,8}, {7,6}, {5,4}, {3,2}, {1,0} }
     };
    Console.WriteLine(mixedJagged[1][1, 1]);
    Console.ReadLine();
```

Class member visibility

Class Member Visibility Levels

Visibility Keyword	Description
public	This member is accessible from both inside the assembly and outside the assembly, as well as by members inside and outside the class.
private	This member is accessible only from code within the class definition itself.
protected	This member is accessible only to derived types, both inside and outside the assembly.
internal	This member is accessible by any code within the assembly.
public protected	This member is accessible by all code within the assembly, but only by derived types outside the assembly.
private protected	This member is protected within the assembly (accessible only to derived types), but is inaccessible outside the assembly.

Class member visibility

Class Member Visibility:

Object Oriented Programming

- Object Oriented Programming (OOP) is a programming model where programs are organized around objects and data rather than action and logic.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.

Abstraction :

Representing essential features without representing the background details

✓ Abstraction lets you focus on what the object does instead of how it does it.

Object Oriented Programming

```
abstract class MobilePhone
    {
        public void Calling();
        public void SendSMS();
    }

    public class Nokia1400 : MobilePhone
    {
    }
```

Object Oriented Programming

Encapsulation: Wrapping up a data member and a method together into a single unit (in other words class) is called Encapsulation.

Example for encapsulation is class.

```
class Bag
  {
      book;
      pen;
      ReadBook();
  }
```

Encapsulation means hiding the internal details of an object, in other words how an object does something.

Object Oriented Programming

```
class Demo
  {
     private int _mark;

     public int Mark
     {
        get { return _mark; }
        set { if (_mark > 0) _mark = value; else _mark = 0; }
     }
  }
```

Ex: Mobile user learns how to use the phone but not how it works.

Object Oriented Programming

Inheritance:

✓ When a class includes a property of another class it is known as inheritance.
✓ Inheritance is a process of object reusability.

Ex:
```
public class ParentClass
    {
        public ParentClass()
        {
            Console.WriteLine("Parent Constructor.");
        }
        public void print()
        {
            Console.WriteLine("I'm a Parent Class.");
        }
    }
public class ChildClass : ParentClass
    {
        public ChildClass()
        {
            Console.WriteLine("Child Constructor.");
        }
        public static void Main()
        {
            ChildClass child = new ChildClass();
            child.print();
        }
    }
```

inheritance

Syntax :

```
class derived-class-name : base-class-name {
   // body of class
}
```

inheritance

```
class TwoDShape {
public double Width;
public double Height;
public void ShowDim() {
      Console.WriteLine("Width and height are " +
      Width + " and " + Height);
}
}
// Triangle is derived from TwoDShape.
class Triangle : TwoDShape {
      public string Style; // style of triangle
      // Return area of triangle.
      public double Area() {
      return Width * Height / 2;
      }
/     / Display a triangle's style.
      public void ShowStyle() {
      Console.WriteLine("Triangle is " + Style);
      }
}
```

Object Oriented Programming

Polymorphism: Polymorphism means one name, many forms.

•One function behaves in different forms.

•Many forms of a single object is called Polymorphism.

Ex: A person behaves the son in a house at the same time that the person behaves an employee in an office.

Ex: Your mobile phone, one name but many forms:

•As phone

•As camera

•As mp3 player

•As radio

Class: It is template or blueprint that describes the details of an object.

Object:

static keyword

- Applies for classes, methods, variables and constructor.
- Normally, a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static.**
- When a member is declared static, it can be accessed before any objects of its class are created and without reference to any object.
- You can declare **both methods and variables to be static.**
- Because **static fields are independent of any specific object, they are useful when you** need to maintain information that is applicable to an entire class.
- The most common example of a static member is **Main( ),** which is declared static because it must be called by the operating system when your program begins.
- Outside the class, to use a static member, you must specify the name of its class followed by the dot operator. No object needs to be created. In fact, a static member cannot be accessed through an object reference. It must be accessed through its class name. For example, if you want to assign the value 10 to a static variable called count that is part of a class called Timer, use this line:

```
Timer.count = 10;
```

- A **static method can be called in the same way—by use** of the dot operator on the name of the class.

static keyword

- Variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, **all instances of the class share the same static variable.**
- **A static variable is initialized before its class is used.** If no explicit initializer is specified, it is initialized to zero for numeric types, null in the case of reference types, or false for variables of type bool. Thus, a static variable always has a value.
- The difference between a static method and a normal method is that the static method can be called through its class name, without any instance of that class being created. You have seen an example of this already: the Sqrt( ) method, which is a static method within C#'s System.Math class.

static methods and static variables

- Methods declared with static keyword are called static methods..

Rules:

- • A static method does not have a this reference. This is because a static method does not execute relative to any object.
- • A static method can directly call only other static methods of its class. It cannot
- directly call an instance method of its class. The reason is that instance methods
- operate on specific objects, but a static method is not called on an object. Thus, on what object would the instance method operate?
- • A similar restriction applies to static data. A static method can directly access only other static data defined by its class. It cannot operate on an instance variable of its class because there is no object to operate on.
- Error occurs when trying to call a non-**static method from within a static** method of the same class.

static method accessing a non static method - Error

```
using System;
class AnotherStaticError {
// A non-static method.
void NonStaticMeth() {
Console.WriteLine("Inside NonStaticMeth().");
}
/* Error! Can't directly call a non-static method
from within a static method. */
static void staticMeth() {
NonStaticMeth(); // won't compile
}
}
```

static classes

- A class can be declared **static. There are two key features of a static class. First, no object of** a **static class can be created. Second, a static class must contain only static members. A static** class is created by modifying a class declaration with the keyword **static, shown here:**

  static class *class-name { // ... }*

- Within the class, all members must be explicitly specified as **static. Making a class static** does not automatically make its members **static.**

Use of static classes:

- static classes have two primary uses. First, a static class is required when creating an *extension method.*
- Second, a static class is used to contain a collection of related static methods.

static constructor

- A constructor can also be specified as **static. A static constructor is typically used to** initialize features that apply to a class rather than an instance.
- Thus, it is used to initialize aspects of a class before any objects of the class are created.

Topics

- Properties - examples
- Base reference – Derived objects
- Using abstract classes
- Exception Handling

Base reference – Derived objects

- C# is a strongly typed language. Aside from the standard conversions and automatic promotions that apply to its value types, type compatibility is strictly enforced.
- Therefore, a reference variable for one class type cannot normally refer to an object of another class type.
- In general, an object reference variable can refer only to objects of its type. There is, however, an important exception to C#'s strict type enforcement. A reference
- variable of a base class can be assigned a reference to an object of any class derived from that base class. This is legal because an instance of a derived type encapsulates an instance of the base type. Thus, a base class reference can refer to it.

THANK YOU

IO in C#