Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies

Windows Applications Programming

Microsoft .NET Framework, C# Language

Liviu-Adrian Cotfas, PhD.

liviu.cotfas@ase.ro

Microsoft .NET Framework, C# Language

C#

- url: https://dotnet.microsoft.com, docs.microsoft.com/en-us/dotnet/csharp/

Few words about me…

https://ro.linkedin.com/in/cotfasliviu

Evaluation

- Final exam – 60%
  - similar to the OOP exam
- Seminar – 40%
  - test – 30%
  - project – 10%

Recommended Reading / Watching

- Slides, Examples, Books:
  - http://online.ase.ro

Further Reading / Watching

- Courses on Microsoft Learn - https://docs.microsoft.com/en-us/learn
  - Free

API reference and Source code

- API reference:
  - https://docs.microsoft.com/en-us/dotnet/api/?view=netframework-4.8
- .NET Framework source code:
  - http://referencesource.microsoft.com/#mscorlib/system/string.cs,8281103e6f23cb5c
- Official samples:
  - https://code.msdn.microsoft.com/

Road map

1. .NET Framework and C# Language
2. Differences between C++ and .NET
3. System Data Types
4. Value Types and Reference Types
5. Boxing / Unboxing

.NET Framework and C# language

Why .NET? - Various types of apps

Web
ASP.NET

Mobile
Xamarin

Desktop

Cloud
Microsoft Azure

Games
Unity

Machine Learning
ML.NET

https://dotnet.microsoft.com/learn

Why .NET? - Various types of apps

- Desktop:
  - Universal Windows Platform (Windows 10);
  - Windows Presentation Foundation – WPF (Windows Vista +);
  - WinForms;
  - Console.

- Comparison:
  - https://docs.microsoft.com/en-us/windows/apps/desktop/
  - https://docs.microsoft.com/en-us/windows/apps/desktop/choose-your-platform

Why .NET?

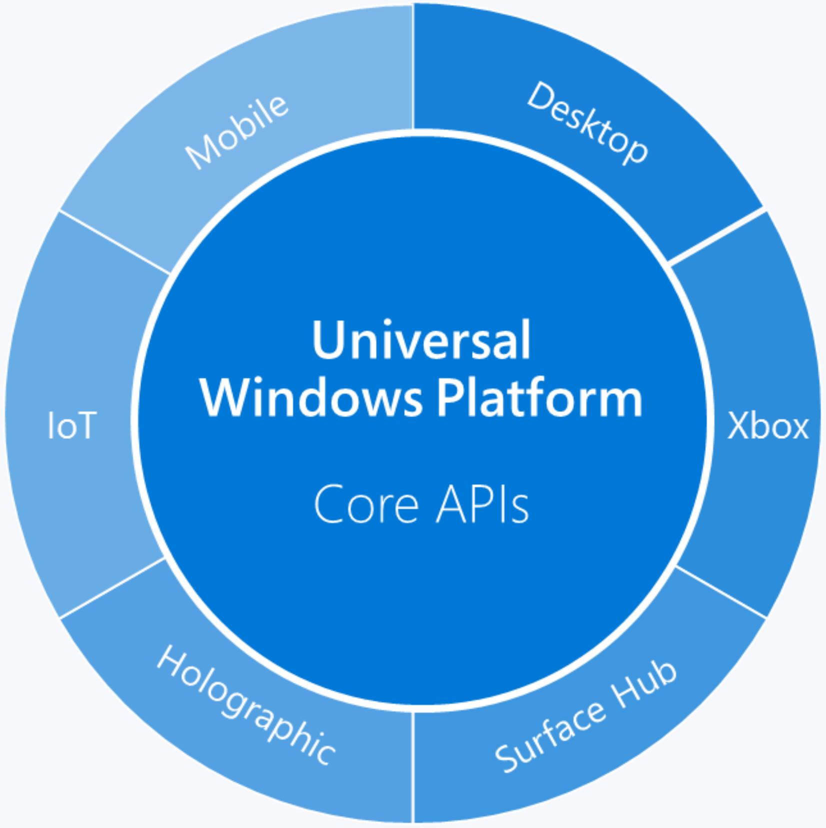
- really well integrated into Windows:
  - installed with Windows;
  - Native UI.
- a comprehensive base class library;

Why .NET? - OS / Platform Support

So, it only runs on **Windows**

Why .NET? - OS / Platform Support

It runs on the entire **Microsoft** ecosystem



Why .NET? - OS / Platform Support

and .NET Core apps also runs on:

Linux

MacOS

- .NET Core allows the development of cross-platform Console and Web applications
- Further reading: https://docs.microsoft.com/en-us/dotnet/core/

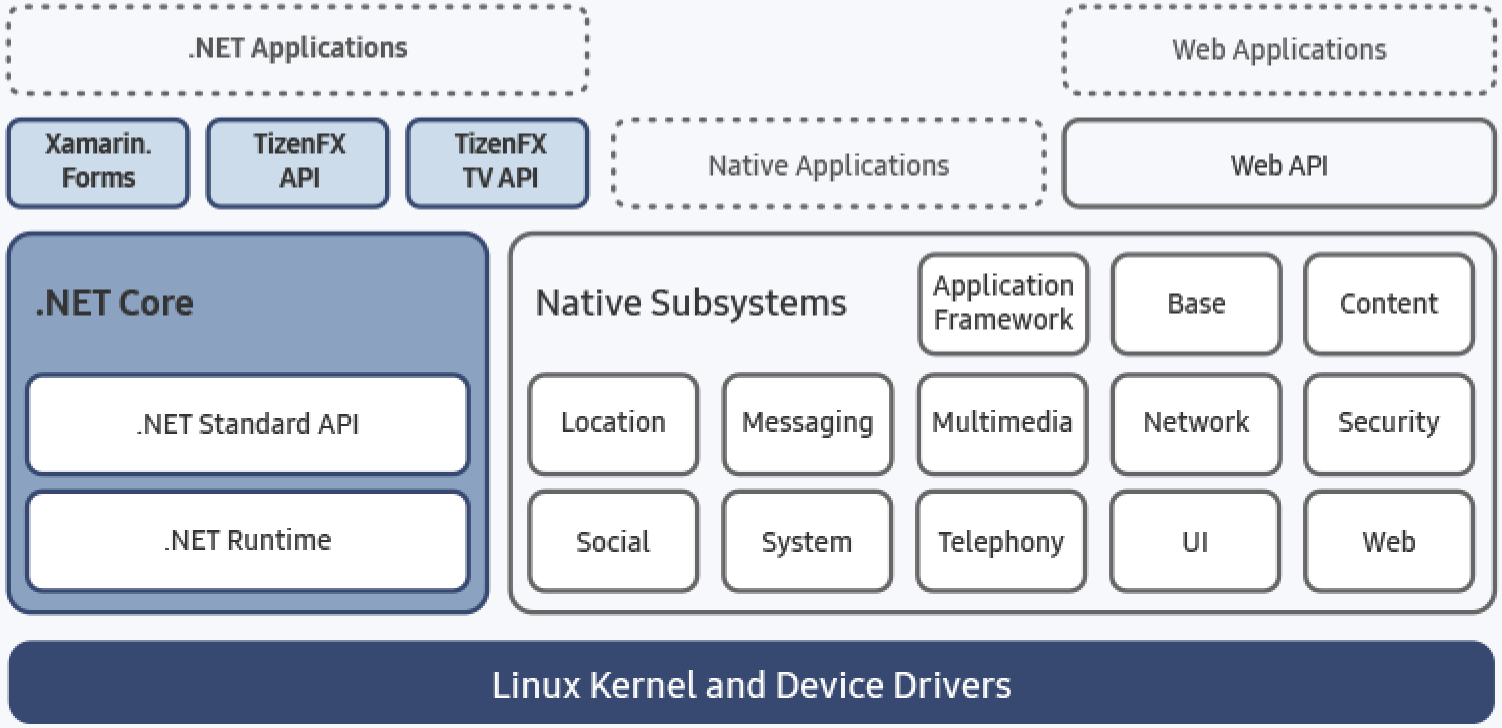
Why .NET? - OS / Platform Support

and **Samsung** Smart TVs

Why .NET? - OS / Platform Support

- The **Tizen .NET** TV framework enables you to build .NET TV applications with **Xamarin.Forms** and the Tizen .NET framework.

- Further reading: https://developer.samsung.com/tv/tizen-net-tv



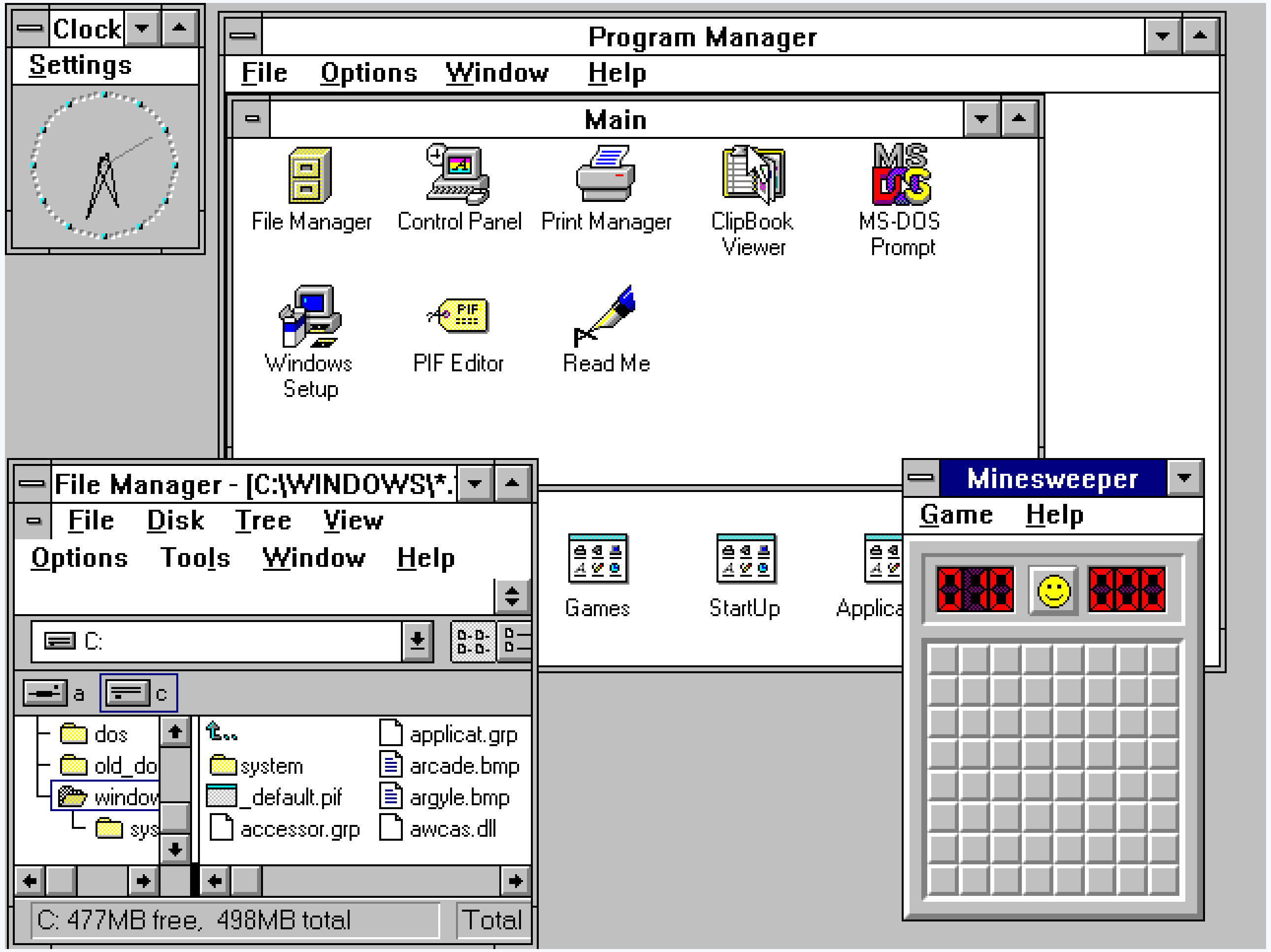
Why .NET? - OS / Platform Support

and using Xamarin **.NET apps** also run on:

android iOS

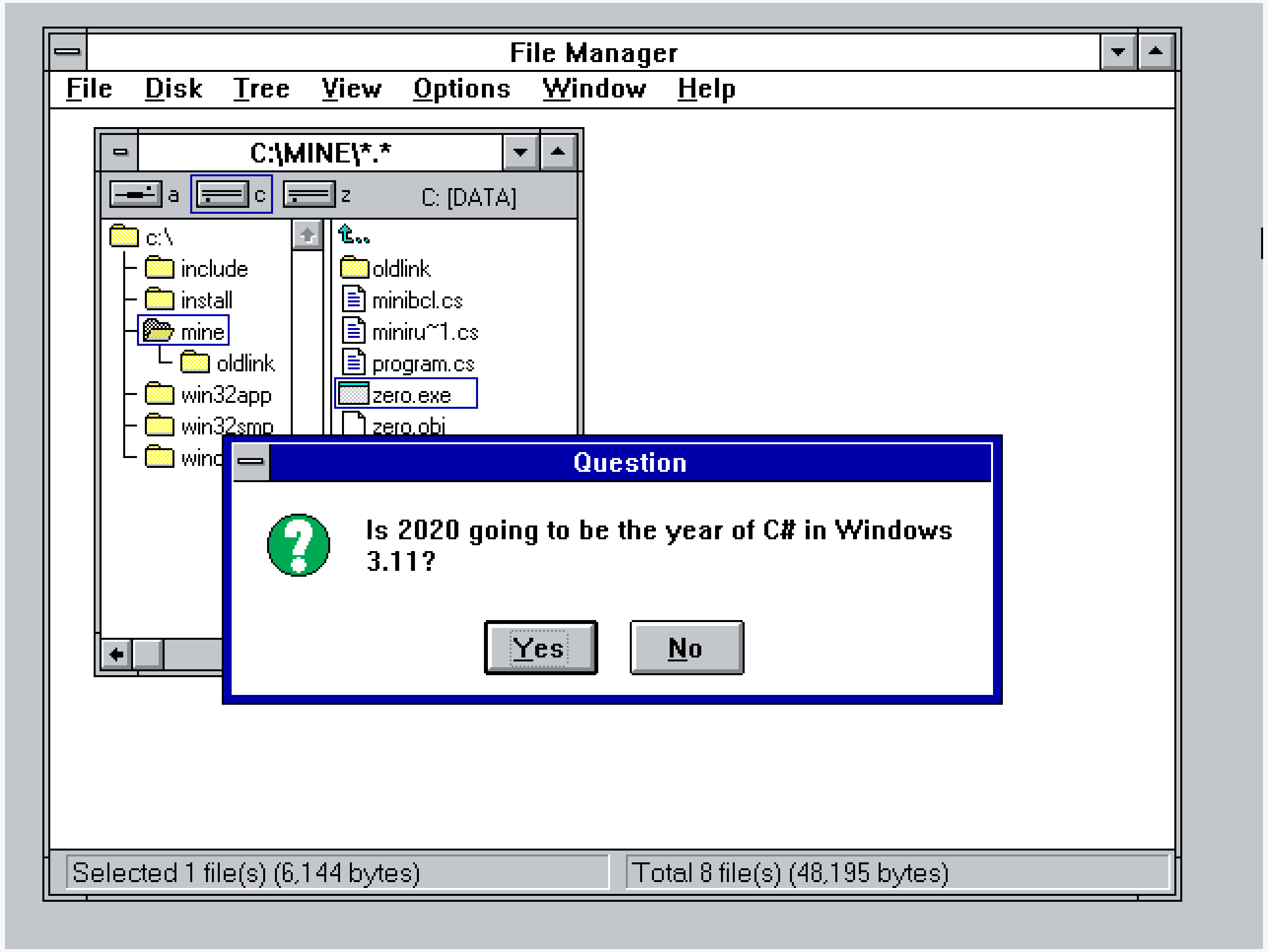
- Further reading: https://docs.microsoft.com/en-us/xamarin

Fun facts :)



Windows 3.1X (1992)

Fun facts :)

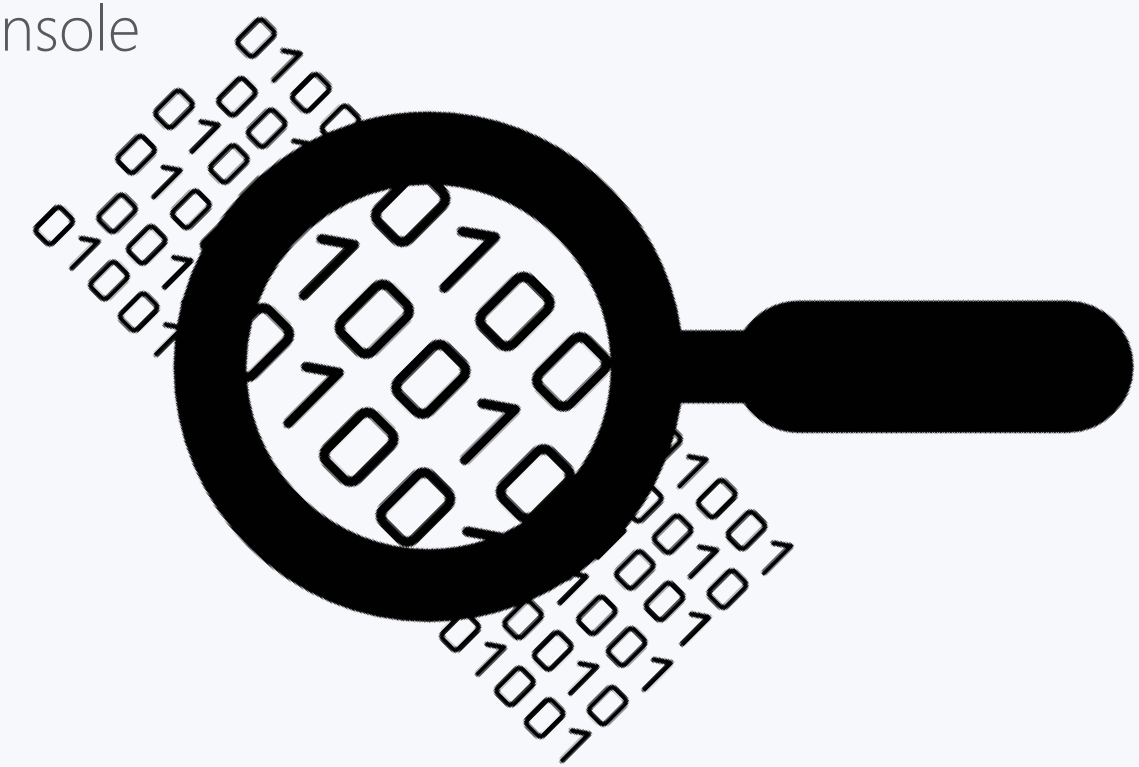


- Further reading: https://www.hanselman.com/blog/NETEverywhereApparentlyAlsoMeansWindows311AndDOS.aspx

Demo

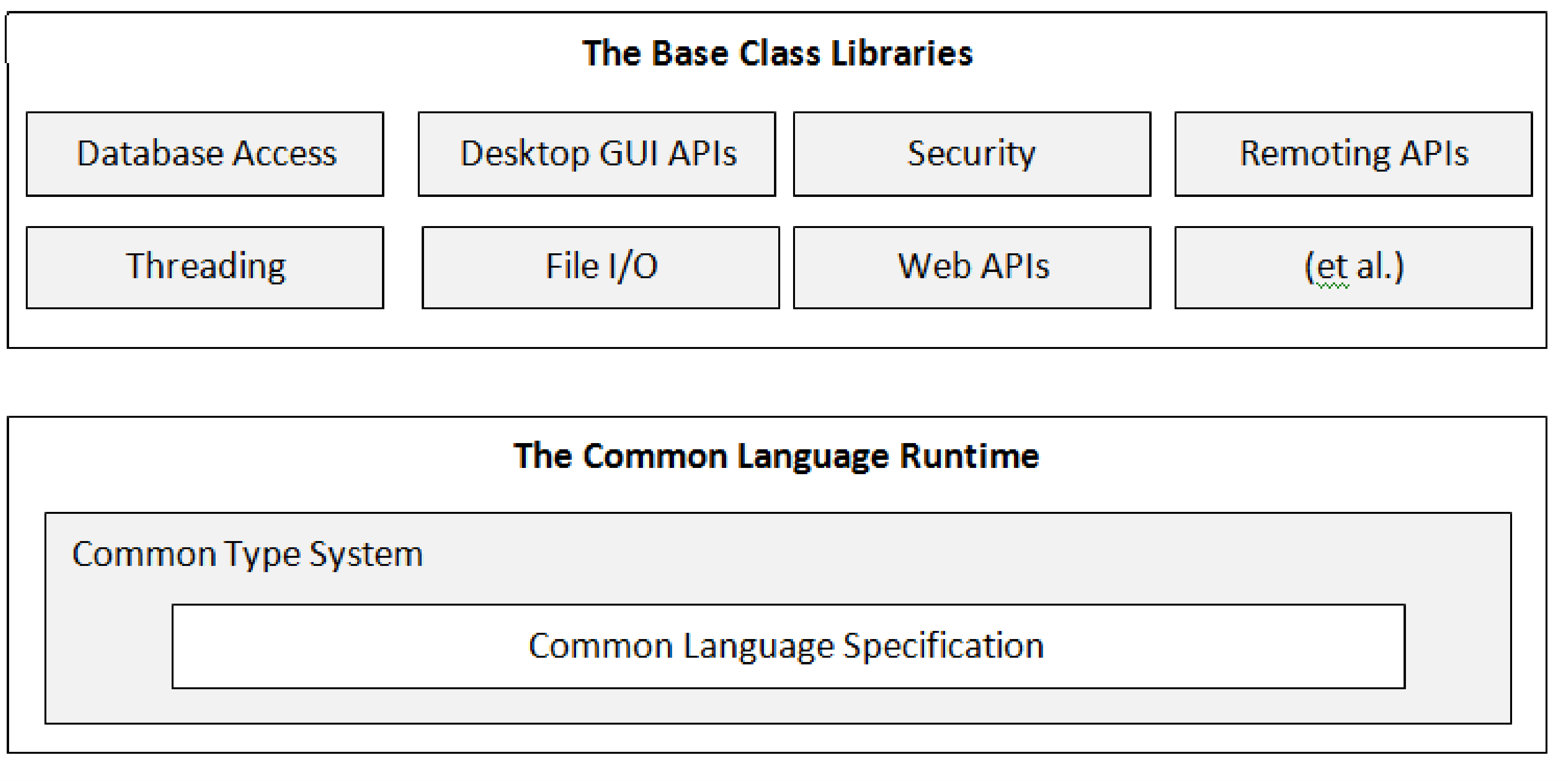
.NET Core on Ubuntu (Linux)

- https://code.visualstudio.com/docs/other/dotnet
- sudo snap install dotnet-sdk --classic
- dotnet new console
- dotnet run



Building Blocks of the .NET Platform

From a programmer's point of view, .NET can be understood as a **runtime environment** and a **comprehensive base class library**.



Common Language Runtime

The **CLR** manages the code being executed and provides for a layer of abstraction between the code and the operating system. Built into the CLR are mechanisms for the following:

- Loading code into memory and preparing it for execution.
- Converting the code from the intermediate language to native code.
- Managing code execution.
- Managing code and user-level security.
- Automating deallocation and release of memory.
- Debugging and tracing code execution.
- Providing structured exception handling.

Common Type System and Common Language Specification

- Common Type System
  - describes all possible data types and all programming constructs supported by the runtime,
- Common Language Specification
  - Defines a subset of common types and programming constructs that all .NET programming languages can agree on.

Framework Base Class Library

- available to all .NET programming languages.

- define types that can be used to build any type of software application:
  - ASP.NET to build web sites,
  - WCF to build networked services,
  - Windows Forms / WPF to build desktop GUI applications

- provide types to interact with XML documents, the local directory and file system on a given computer, communicate with a relational databases (via ADO.NET), and so forth.

Managed languages

- Managed languages depend on services provided by a runtime environment.
- C# is one of many languages that compile into managed code.
- Managed code is executed by the Common Language Runtime (CLR).

Why use C#?

- C# (pronounced "C sharp") is a general-purpose, **type-safe** language, **object-oriented** programming language, designed for building a variety of applications
- Focused on developer **productivity**.
- C# syntax is based on the C & C++ syntax.
- Platform-neutral, but it was written to work well with the Microsoft .NET Framework.

Constantly evolving language

.Net 1	.Net 1.	.Net 2	.Net 3	.Net 3.5	.Net 4	.Net 4.5
C# 1	C# 1.2	C# 2	N/A	C# 3	C# 4	C# 5
		GENERICS		VAR	DYNAMICS	ASYNC
		PARTIAL CLASSES		LINQ	OPTIONAL ARGS	CALLER ATTR
		ANONYMOUS		LAMBDA	COVARIANCE	
		NULLABLE		INITIALIZERS		
				AUTO PROPS		
				EXTENSIONS		
				PARTIAL METHODS		

Compiling and Executing Managed Code

- When .NET code is compiled, it is converted into a **.NET portable executable (PE)** file. The compiler translates the source code into **Microsoft intermediate language (MSIL)** format.

- MSIL is **CPU independent code**, which means it needs to be further converted into native code before executing.

- Decompiles: http://ilspy.net/, https://www.jetbrains.com/decompiler/

Compiling and Executing Managed Code

- Before the MSIL code in the PE file is executed, a .NET Framework **just-in-time (JIT) compiler** converts it into CPU-specific native code. To improve efficiency, the JIT compiler does not convert all the MSIL code into native code at the same time. MSIL code is converted on an as-needed basis. When a method is executed, the compiler checks to see if the code has already been converted and placed in cache. If it has, the compiled version is used; otherwise, the MSIL code is converted and stored in the cache for future calls.

- Because JIT compilers are written to target different CPUs and operating systems, **developers are freed** from needing to rewrite their applications to target various platforms.

Compiling and Executing Managed Code

- Because the source code for the various .NET-compliant languages is compiled into the same MSIL and metadata format based on a common type system, the .NET platform supports language integration. With .NET language integration, a class written in VB could inherit a class written in C#.

Differences between C++ and .NET

Globals

- In C#, **global** methods and variables are **not supported**. Methods and variables must be contained within a class or struct.

- Further reading: link

```
using System; //referenced namespace

namespace NameSpaceProgram
{
        internal class Program
        {
                private static void Main(string[] args)
                {
                        //HelloWorld application
                        Console.WriteLine("Hello World!");
                        Console.ReadLine();
                } //end main
        }//end class
} //end namespace
```

Main

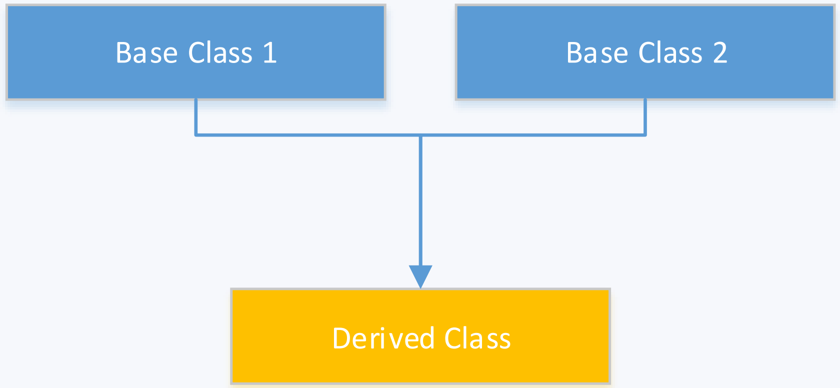
- The Main method in C# is declared differently from the main function in C++. In C# it is **capitalized**, and **always static**.

- Support for processing of command-line arguments is much more robust in C#.

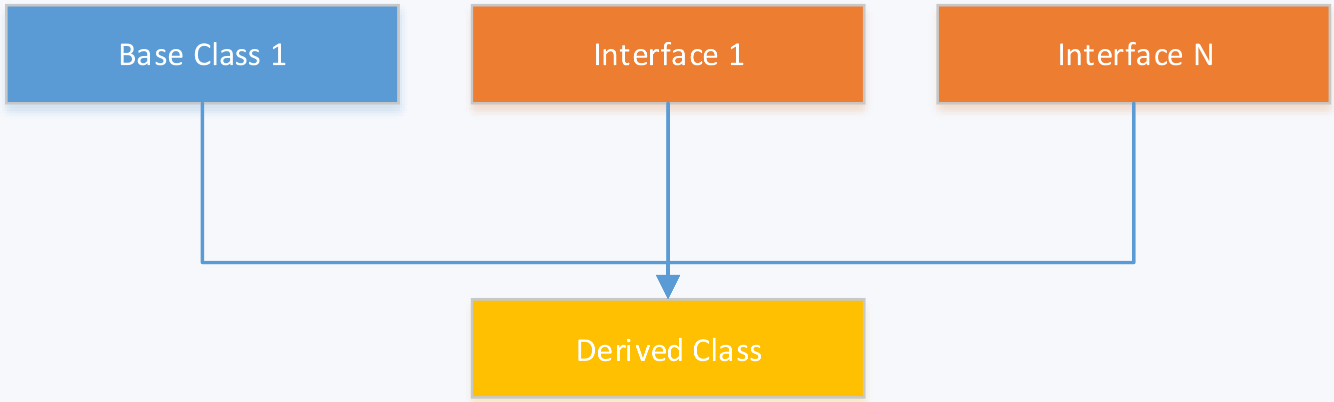
```
private static void Main(string[] args)
{
        //HelloWorld application
        Console.WriteLine("Hello World!");
        Console.ReadLine();
} //end main
```

Inheritance

- C# **classes** can implement any number of interfaces, but can inherit from only one base class => Multiple Inheritance is **not supported**!
- C# **structs** do not support inheritance, and do not support explicit default constructors (one is provided by default).



Multiple inheritance – C++



Simple inheritance – C#

Inheritance

```
class DepositAccount : Account, IComparable, IComparable<Account>, ICloneable
{
        public int CompareTo(object obj)
        {
                throw new NotImplementedException();
        }

        public int CompareTo(Account other)
        {
                throw new NotImplementedException();
        }

        public object Clone()
        {
                throw new NotImplementedException();
        }
}
```

Pointers

- Pointers are allowed in C#, but only in **unsafe** mode.
- Further reading: link

Strings

- In C++ a string is simply an array of characters. In C#, strings are objects that support robust searching methods.

- The C# string class is located in the System namespace (**System.String**).
  - Sealed – the class cannot be inherited
  - Immutable - after the initial value is assigned to a string object, the character data cannot be changed. A brand new string is created each time we modify the initial string

- Further reading: link (string documentation)

Garbage Collection

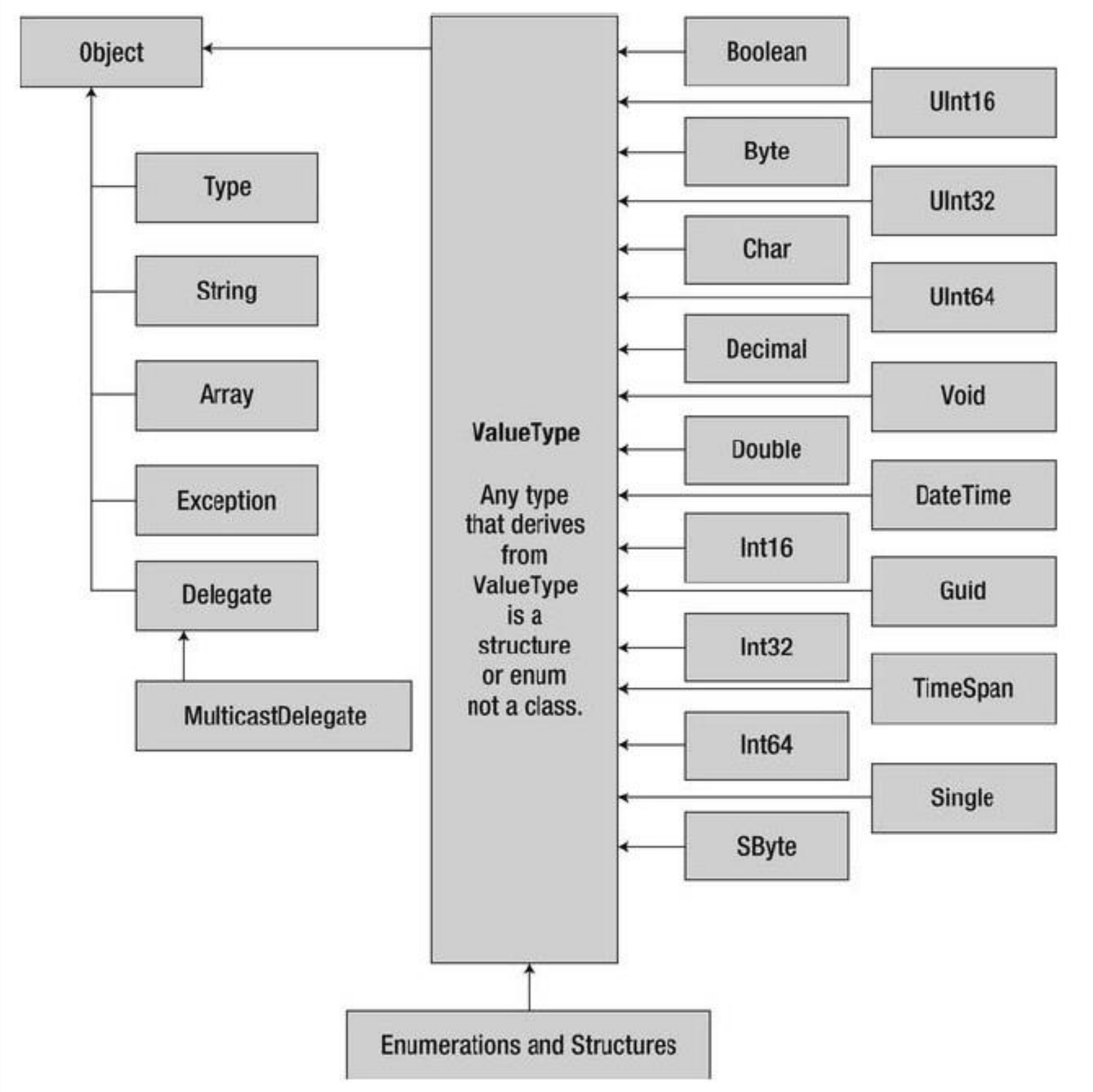
- Memory management: C++ is not a garbage collected language; memory that is not explicitly release remains allocated until the process terminates.
- C# is a garbage collected language.
- Further reading: link

Further reading

- More differences: https://msdn.microsoft.com/en-us/library/yyaad03b%28v=vs.90%29.aspx

System Data Types

Inheritance between the System Data Types



Object

- All types have a common base – **Object**
  - defines base methods that will be available for any type
  - any type variable can be cast to Object (upcast).

Name	Description
Equals(Object)	Determines whether the specified object is equal to the current object.
ReferenceEquals	Determines whether the specified Object instances are the same instance.
GetHashCode()	Serves as the default hash function.
GetType()	Gets the Type of the current instance.
ToString()	Returns a string that represents the current object.

Object

```
public class Object
{
        public Object();
        public extern Type GetType();
        public virtual bool Equals (object obj);
        public static bool Equals (object objA, object objB);
        public static bool ReferenceEquals (object objA, object objB);
        public virtual int GetHashCode();
        public virtual string ToString();
        protected virtual void Finalize();
        protected extern object MemberwiseClone();
}
```

Categories

- Categories of standard types:
  - **Value types** - variables directly store values
  - **Reference types** - or objects, store a reference to data
  - Pointer types –only available in unsafe code

Value types

- Documentation: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types
- Two kinds of value types: structure, enum
- Examples: primitive types (derived from System.ValueType) – char, int, double, float, bool, etc.
- **Allocated:** on the stack;
- **Lifetime:** can be created and destroyed very quickly, as its lifetime is determined by the defining scope;

Reference types

- Documentation: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types
- Keywords are used to declare reference types: **class**, **delegate**, **interface**
- **Allocated:** in the heap;
- **Lifetime:** have a lifetime that is determined by a large number of factors

Question

```
var foo = new int();
```

When the code above is executed, where is foo allocated?

Choice 1: in the **heap**

Choice 2: on the **stack**

Comparison between Value Types and Reference Types

	Value type	Reference Type
Where are objects allocated?	Allocated on the stack.	Allocated on the managed heap.
How is a variable represented?	Value type variables are local copies.	Reference type variables are pointing to the memory occupied by the allocated instance.
What is the base type?	Implicitly extends System.ValueType.	Can derive from any other type (except System.ValueType), as long as that type is not "sealed".
Can this type function as a base to other types?	No. Value types are always sealed and cannot be inherited from.	Yes. If the type is not sealed, it may function as a base to other types.
Default parameter passing behavior	Variables are passed by value (i.e., a copy of the variable is passed into the called function).	For value types, the object is copied-by-value. For reference types, the reference is copied-by-value.
Own constructor for this type	Yes, but the default constructor is reserved (i.e., the custom constructors must all have arguments).	Yes
When do variables of this type die?	When they fall out of the defining scope.	When the object is garbage collected.

String / StringBuilder

Url: https://github.com/liviucotfas/ase-windows-applications-programming/blob/master/01%20-%20C%23%20%20%26%20.NET%20Framework%20Basics.md#working-with-strings

Section: Working with Strings

Array

Url: https://github.com/liviucotfas/ase-windows-applications-programming/blob/master/01%20-%20C%23%20%20%26%20.NET%20Framework%20Basics.md#arrays

Section: Arrays

Classes

Syntax

```
[attributes]
[internal / public] [abstract] [sealed] [static] [partial] class ClassName [:BaseClass, Interface1, …
InterfaceN]
{
        …
}
```

Example:

```
[Serializable]
public class Employee : Person, IComparable<Person>
{
        …
}
```

Class - [accessModifier]

- internal
  - without specifying public the class is **implicitly** internal. This means that the class is only visible inside the same assembly (project).
- public
  - the class is visible outside the assembly (project).

Discussion

Why would we want to make a class **internal**, in other words not visible outside the current assembly (project)?

Fields

```
class ClassName
{
        [accessModifier] [static] [new] [readonly] [const] type field;
}
```

```
class Person
{
        public int _age;
}
```

Modifiers:

- access: private (default), public, protected, internal, protected internal
- static: static
- read-only: readonly
- const: const
- inheritance: new

Fields - Access Modifiers

- public: the type or member can be accessed by any other code in the same assembly or another assembly that references it.

- private: the type or member can be accessed only by code in the same class or struct.

- protected: the type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.

- internal: the type or member can be accessed by any code in the same assembly, but not from another assembly.

Fields - Access Modifiers

- protected internal:
  - The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. Access from another assembly must take place within a class declaration that derives from the class in which the protected internal element is declared, and it must take place through an instance of the derived class type.
  - Documentation: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/protected-internal

Fields - static

- declare a member, which belongs to the type itself rather than to a specific object.
- a static member cannot be referenced through an instance. Instead, it is referenced through the type name.
- It is not possible to use **this** to reference static methods or property accessors.

Fields - readonly

- assignments to the fields can only occur as part of the **declaration** or in a **constructor** in the same class

```
class Age
{
    readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

Fields - const

- constant fields aren't variables and may not be modified. Constants can be numbers, Boolean values, strings, or a null reference.

- assignments to the fields can only occur as part of the **declaration**

```
class SampleClass
{
		public const int c1 = 5;
		public const int c2 = c1 + 5;
}
```

Fields – readonly & const

- The **readonly** keyword is different from the **const** keyword. A **const** field can only be initialized at the declaration of the field. A **readonly** field can be initialized either at the declaration or in a constructor.

- Therefore, **readonly** fields can have different values depending on the constructor used.

Methods

Modifiers:

- access: private (default), public, protected, internal, protected internal
- static: static
- inheritance: virtual, abstract, new, override, sealed
- partial: partial

Constructors

- run initialization code on a class or struct.
- a constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type

```
public class Wine
{
        public decimal Price;
        public int Year;
        public Wine (decimal price) { Price = price; }
        public Wine (decimal price, int year) : this (price) { Year = year; }
}


……
var wine = new Wine(10);
…..
var wine = new Wine(10, 2010);
```

Constructors

- Constructors do not need to be public. A common reason to have a nonpublic constructor is to control instance creation via a static method call. The static method could be used to return an object from a pool rather than necessarily creating a new object.

```
public class Class1
{
        Class1() {} // Private constructor
        public static Class1 Create (...)
        {
        // Perform custom logic here to return an instance of Class1
        ...
        }
}
```

this Reference

- the **this** reference refers to the instance itself.
- the **this** reference also disambiguates a local variable or parameter from a field:

```
public class Test
{
		string name;
		public Test (string name) { this.name = name; }
}
```

Properties

- A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called **accessors**.

Modifiers:

- access: private (default), public, protected, internal, protected internal
- static: static
- inheritance: new, virtual, abstract, override, sealed

Properties

- Encapsulation using accessor methods:

```
#region Accessor Methods
private int _age;
public int GetAge()
{
		return _age; // "this._age" is implicit
}
public void SetAge(int value)
{
		_age = value; // "this._age" is implicit
}
#endregion
```

Properties

```
#region Name - Using properties
private string _name;
//Read/Write property
public string Name
{
        get { return _name; }
        set { _name = value; }
}

//Readonly property
public string Name2
{
        get { return _name; }
}
#endregion
```

Auto-property

- The **get** and **set** accessors can have different access levels. The typical use case for this is to have a public property with an internal or private access modifier on the setter.

```
#region Occupation - Using auto-property
public OccupationEnum Occupation { get; set; }
#endregion

#region Occupation - Using auto-property
public OccupationEnum Occupation { get; private set; }
#endregion
```

CLR property implementation

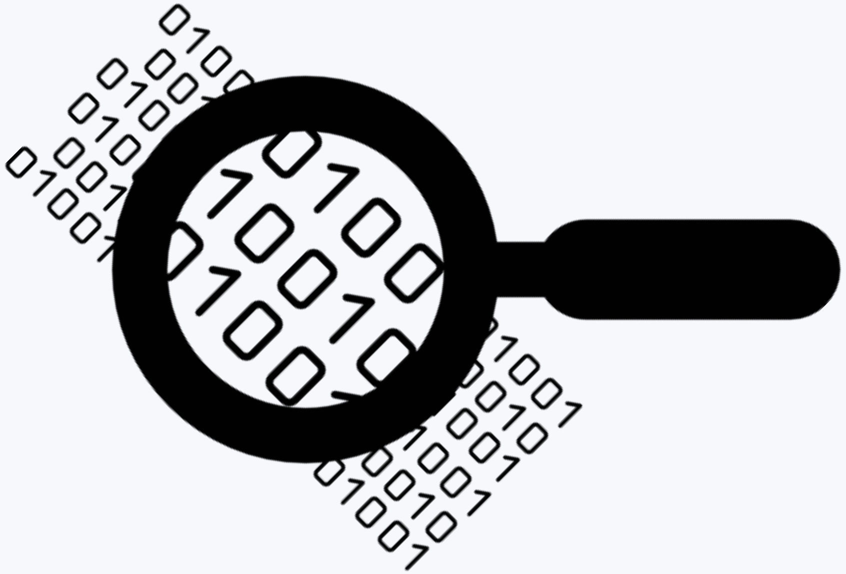
- C# property accessors internally compile to methods called get_*XXX* and set_*XXX*:

```
public decimal CurrentPrice {get;set;}
```

↓

```
public decimal get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}
```

CLR property implementation



1. Create a console application
2. Declare a class named Person
3. Add an automatic property to the class
4. Compile the application
5. Decompile the executable using ILSpy and check the **IL code** for the property

Sealed class

- prevents other classes from inheriting from it
- can also be used on methods or properties that overrides a virtual method or property in a base class. Classes can be derived from the base class, but they are prevented from overriding specific virtual methods or properties.

```
public sealed class String : IComparable, ICloneable, I....
{
}
```

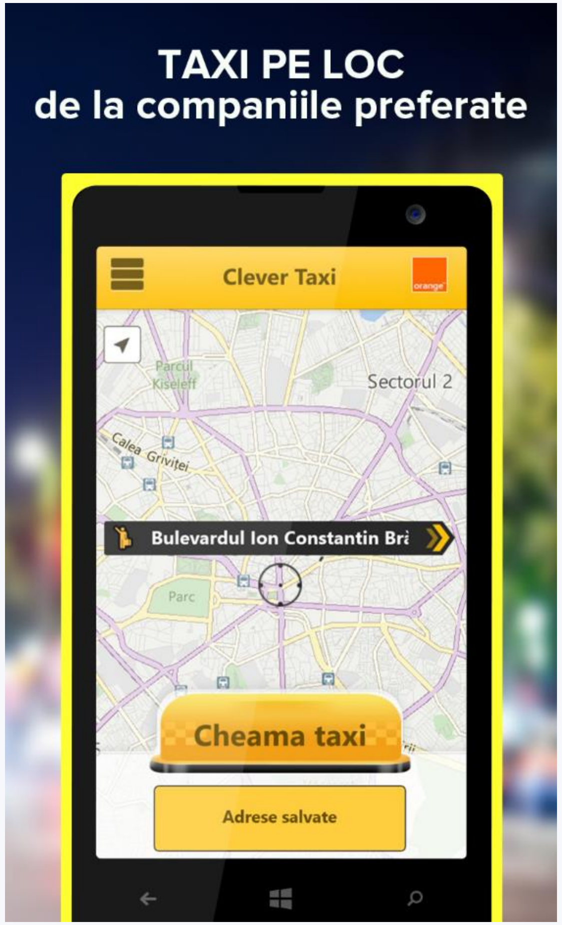
Static class

- has the **static** modifier in the class declaration. All the members of the class need to be **static**.
- can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields.
- a static class cannot be instantiated.
- further reading: link

Static class

```
/// <summary>
/// Contains device hardware related methods
/// </summary>
public static class DeviceInfo
{
		public static string GetDeviceUniqueId()
		{
				var token = HardwareIdentification.GetPackageSpecificToken(null);
				.....
				return hashedString;
		}

		public static String GetDeviceName()
		{
				var deviceInfo = new EasClientDeviceInformation();
				return String.Format("{0} {1} {2}", deviceInfo.SystemManufacturer, deviceInfo.SystemProductName,
deviceInfo.FriendlyName);
		}
}
```

Static class



Operators

Overloading

- can be overload by defining **static** member functions using the operator keyword.

- not all operators can be overloaded and others have restrictions

- further reading: link

List of operators

<table>
<tr><th>Operators</th><th>Overloadability</th></tr>
<tr><td>+, -, !, ~, ++, --, true, false</td><td>These unary operators can be overloaded.</td></tr>
<tr><td>+, -, *, /, %, &, |, ^, <<, >></td><td>These binary operators can be overloaded.</td></tr>
<tr><td>==, !=, <, >, <=, >=</td><td>The comparison operators can be overloaded</td></tr>
<tr><td>&&, ||</td><td>The conditional logical operators cannot be overloaded, but they are evaluated using & and |, which can be overloaded.</td></tr>
<tr><td>[]</td><td>The array indexing operator cannot be overloaded, but you can define indexers.</td></tr>
<tr><td>(T)x</td><td>The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).</td></tr>
<tr><td>+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=</td><td>Assignment operators cannot be overloaded, but +=, for example, is evaluated using +, which can be overloaded.</td></tr>
<tr><td>=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof</td><td>These operators cannot be overloaded.</td></tr>
</table>

operator== and operator!=

```
public sealed class String : IComparable, ICloneable, I....
{
        ............

        public static bool operator == (String a, String b) {
                return String.Equals(a, b);
        }

        public static bool operator != (String a, String b) {
                return !String.Equals(a, b);
        }
        ...........
}
```

Source code: http://referencesource.microsoft.com/#mscorlib/system/string.cs

operator+, operator* , cast

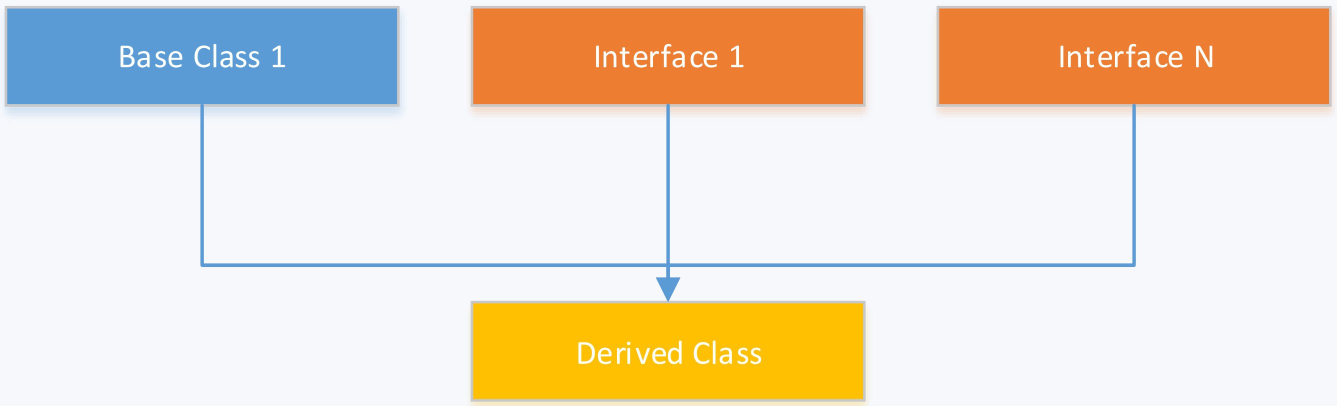
```
class Fraction
{
    // overload operator +
    public static Fraction operator +(Fraction a, Fraction b)    {
        return new Fraction(a.num * b.den + b.num * a.den,
           a.den * b.den);
    }
    // overload operator *
    public static Fraction operator *(Fraction a, Fraction b)    {
        return new Fraction(a.num * b.num, a.den * b.den);
    }
    // user-defined conversion from Fraction to double
    public static implicit operator double(Fraction f)    {
        return (double)f.num / f.den;
    }
......
}
```

Inheritance

Simple Inheritance

C# implements Inheritance in two ways:

- a class may inherit from a single base class
- a class may implement zero or more Interfaces

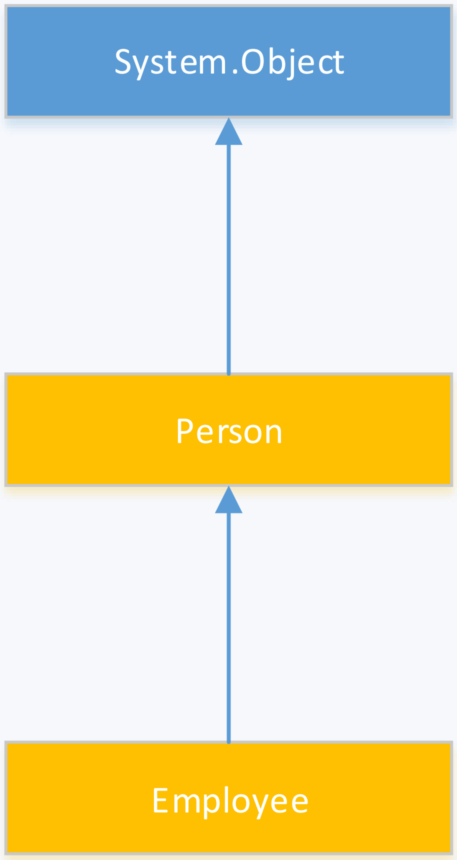


Simple inheritance – C#

Example

```
internal class Person
{
	public string Name { get; set; }
	public int Age { get; set; }
}

internal abstract class Employee : Person
{
	public double Wage { get; set; }
}
```

Example – Inheritance Chain



Upcasting

```
private static void Main()
{
   var employee = new Employee();
   Person person = employee; // Upcast

   Console.WriteLine (person == employee); // True

   Console.WriteLine (person.Name); // OK
   Console.WriteLine (person.Wage); // Error: Wage undefined

}
```

Downcasting

- requires an explicit cast because it can potentially fail at runtime. If a downcast fails, an InvalidCastException is thrown.

```
private static void Main()
{
    var employee = new Employee();
    Person person = employee; // Upcast
    Employee employee2 = (Employee) person; // Downcast

    Console.WriteLine (employee2.Wage); // <No error>
    Console.WriteLine (person == employee); // True
    Console.WriteLine (person == employee2); // True
}
```

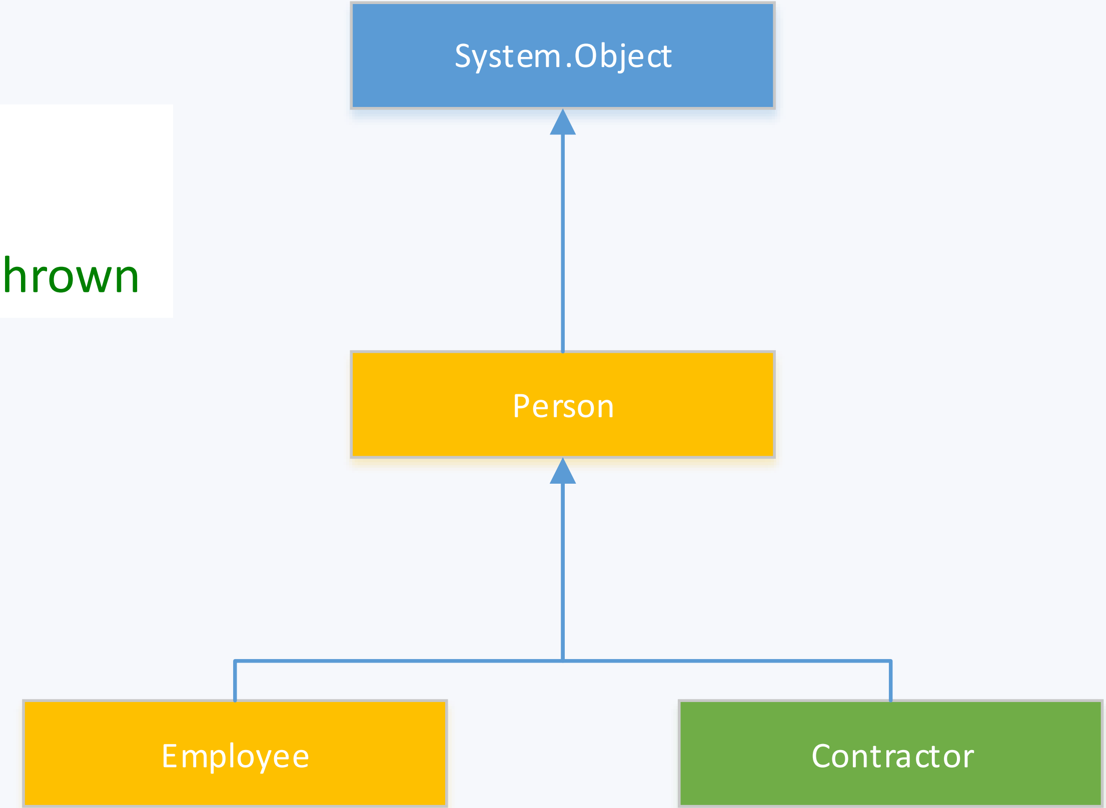
The as operator

- performs a downcast that evaluates to null (rather than throwing an exception) if the downcast fails:

```
Employee e = new Employee();
// Contract and Employee are derived from Person
Contractor c = e as Contractor; // c is null; no exception thrown
```

- test whether the result is null

```
if (c!=null){
        Console.WriteLine(c.Rate)
}
```



The is operator

- tests whether a reference conversion would succeed (whether an object derives from a specified class or implements an interface)
- it is often used to test before downcasting

```
if(e is Contractor)
{
    Contractor c = (Contractor) e;
    Console.WriteLine(c.Rate);
}
```

Abstract classes

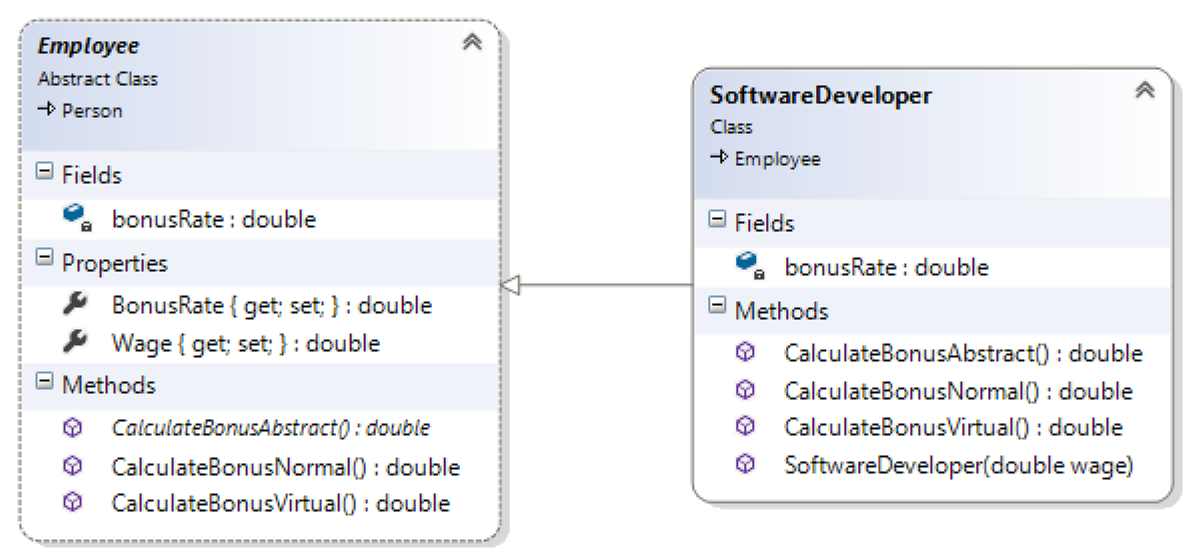
- can never be instantiated. Instead, only its concrete subclasses can be instantiated.

- are able to define *abstract members*. Abstract members are like virtual members, except they don't provide a default implementation. That implementation must be provided by the subclass, unless that subclass is also declared abstract.

Abstract classes

```
internal abstract class Employee : Person{
        private static double bonusRate = 1.1;
        public double Wage { get; set; }

        public abstract double CalculateBonusAbstract(); //Abstract method

        public double CalculateBonusNormal(){//Normal method
                Console.WriteLine("Employee - CalculateBonusNormal");
                return bonusRate * Wage;
        }

        public virtual double CalculateBonusVirtual()  {//Virtual method
                Console.WriteLine("Employee - CalculateBonusVirtual");
                return bonusRate * Wage;
        }
}
```

Virtual Function Members



Virtual Function Members

```
internal class SoftwareDeveloper : Employee
{
        private static double bonusRate = 1.2;

        public override double CalculateBonusAbstract()     {
                Console.WriteLine("SoftwareDeveloper - CalculateBonusAbstract");
                return bonusRate * Wage;
        }


        public override double CalculateBonusVirtual() {
                Console.WriteLine("Employee - CalculateBonusVirtual");
                return bonusRate * Wage;
        }
}
```

Hiding Inherited Members

- using the **new** modifier;
- We can hide attributes, methods and properties.

```
internal class SoftwareDeveloper : Employee{
        private static double bonusRate = 1.2;

        public new double CalculateBonusNormal()
        {
                Console.WriteLine("SoftwareDeveloper - CalculateBonusNormal");
                return bonusRate * Wage;
        }
}
```

Abstract / Hide / Virtual

```
private static void AbstractNormalVirtualMethods(){
        var softwareDeveloper = new SoftwareDeveloper(2000);

        //Abstract method
        Console.WriteLine("\n###Abstract");
        Console.WriteLine(softwareDeveloper.CalculateBonusAbstract());
        Console.WriteLine(((Employee)softwareDeveloper).CalculateBonusAbstract());

        //Normal method
        Console.Write("\n###Hide");
        Console.WriteLine(softwareDeveloper.CalculateBonusNormal());
        Console.WriteLine(((Employee)softwareDeveloper).CalculateBonusNormal());

        //Virtual method
        Console.Write("\n###Override");
        Console.WriteLine(softwareDeveloper.CalculateBonusVirtual());
        Console.WriteLine(((Employee)softwareDeveloper).CalculateBonusVirtual());
}
```

Abstract / Hide / Virtual

C:\WINDOWS\system32\cmd.exe

```
###Abstract
SoftwareDeveloper - CalculateBonusAbstract
2400
SoftwareDeveloper - CalculateBonusAbstract
2400

###HideSoftwareDeveloper - CalculateBonusNormal
2400
Employee - CalculateBonusNormal
2200

###OverrideEmployee - CalculateBonusVirtual
2400
Employee - CalculateBonusVirtual
2400
Press any key to continue . . .
```

The base Keyword

- access an overridden function member from the subclass
- calling a base-class constructor

```
public class House : Asset
{
        ...
        public override decimal Liability
        {
                get { return base.Liability + Mortgage; }
        }
}
```

Constructors and Inheritance

- A subclass must declare its own constructors. The base class's constructors are *accessible* to the derived class, but are never automatically *inherited*.

```
public class Baseclass
{
public int X;
public Baseclass () { }
public Baseclass (int x) { this.X = x; }
}

public class Subclass : Baseclass { }
```

- the following is illegal:

```
Subclass s = new Subclass (123);
```

Constructors and Inheritance

```
public class Subclass : Baseclass
{
        public Subclass (int x) : base (x)
        {

        }
}
```

Implicit calling of the parameterless base-class constructor

- If a constructor in a subclass omits the base keyword, the base type's parameterless constructor is implicitly called

```
public class BaseClass
{
		public int X;
		public BaseClass() { X = 1; }
}

public class Subclass : BaseClass
{
		public Subclass() { Console.WriteLine (X); } // 1
}
```

Constructor and field initialization order

When an object is instantiated, initialization takes place in the following order:

1. From subclass to base class:
   a) Fields are initialized.
   b) Arguments to base-class constructor calls are evaluated.

2. From base class to subclass:
   a) Constructor bodies execute.

Constructor and field initialization order

```
public class B{
        int x = 1; // Executes 3rd
        public B (int x)
        {
                ... // Executes 4th
        }
}
public class D : B{
        int y = 1; // Executes 1st
        public D (int x)
                : base (x + 1) // Executes 2nd
        {
                ... // Executes 5th
        }
}
```

Interfaces

Interfaces

- An interface declaration is like a class declaration, but it provides **no implementation** for its members, since all its members are **implicitly abstract**. **These members will be implemented by the classes and structs that implement the interface.**

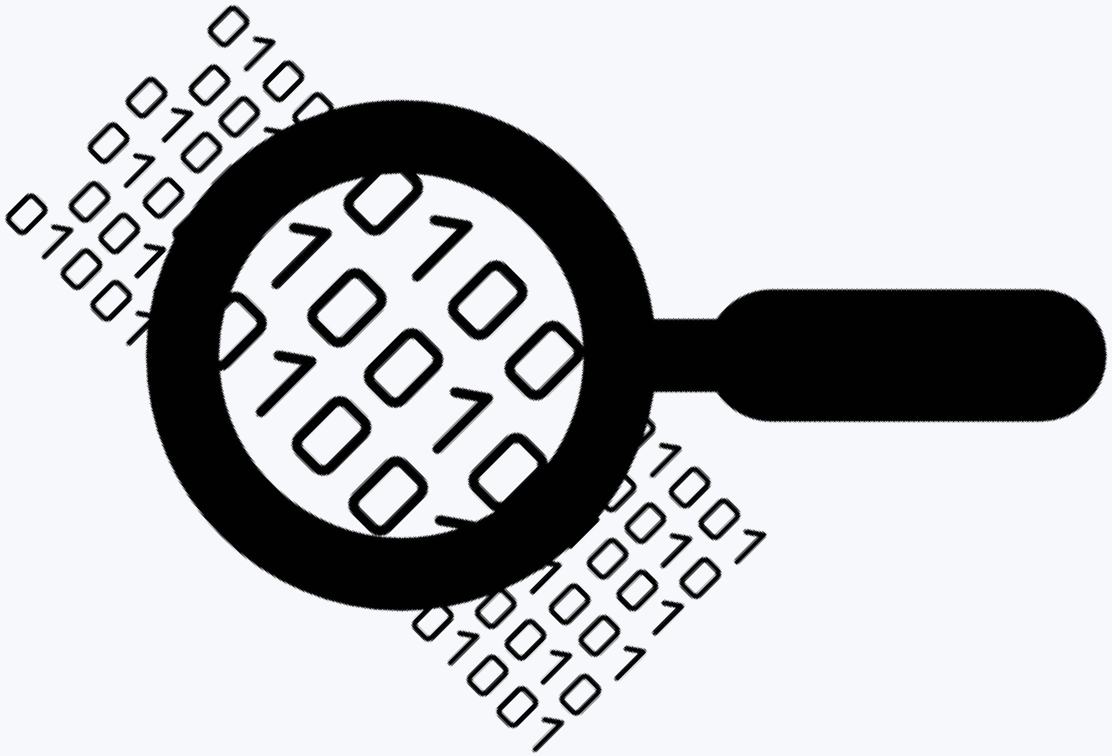
- An interface can contain only **method prototypes**, **properties**, **events**, and **indexers**.

```
public interface IComparable<in T>
{
        int CompareTo(T other);
}
```

```
var p1 = new Person("Name3", 42);
var p2 = new Person("Name1", 23);
var p3 = new Person("Name2", 32);

var pArray = new Person[] { p1, p2, p3 };

Array.Sort(pArray);
```

Example

- Check the source code for the **IComparable** and **IComparable<T>** interfaces
  - http://referencesource.microsoft.com/#mscorlib/system/icomparable.cs,3911354c40e0c30e



IComparable<T>

- Defines a generalized comparison method that a value type or class implements to create a type-specific comparison method for ordering or sorting its instances.

- CompareTo(T) method must return an Int32

Value	**Meaning**
Less than zero	This object precedes the object specified by the CompareTo method in the sort order.
Zero	This current instance occurs in the same position in the sort order as the object specified by the CompareTo method argument.
Greater than zero	This current instance follows the object specified by the CompareTo method argument in the sort order.

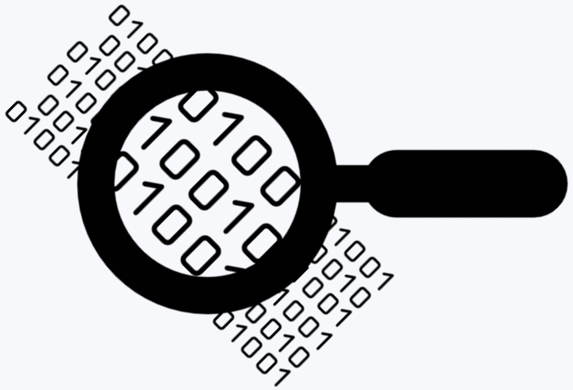
IComparable<T>

```
internal class Person : IComparable<Person>
{
        #region Properties
        public string Name { get; set; }
        #endregion

        public int CompareTo(Person other)
        {
                return Name.CompareTo(other.Name);
        }
}
```

Question: why is it possible to call Name.CompareTo(...) ?

IComparer<T>

- Defines a method that a type implements to compare two objects.



Check the source code for the IComparer and IComparer <T> interfaces:

- https://referencesource.microsoft.com/#mscorlib/system/collections/generic/icomparer.cs,6bf8828f51320498

- Further reading: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1

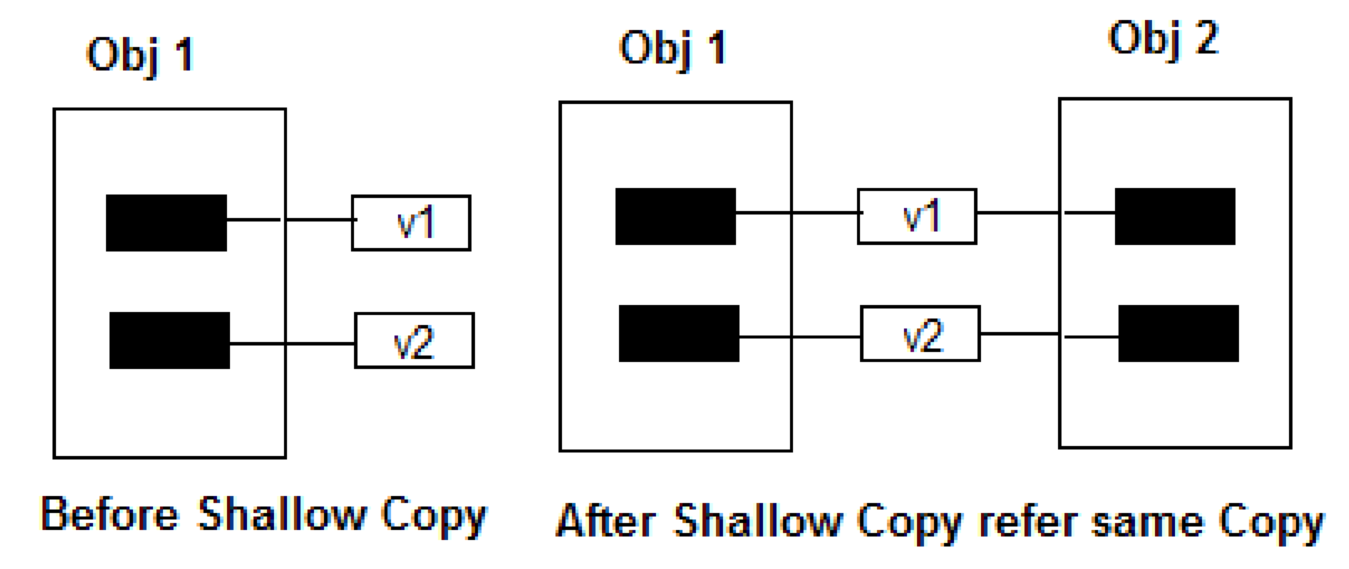
Custom interfaces

```
internal interface IDeveloper
{
	//Property
	string[] Languages { get; set; }

	//Method
	bool Knows(string language);
}
```

Shallow Copy and Deep Copy

Shallow Copy

- Copying an object's value type fields into the target object and the object's reference types are copied as references into the target object but not the referenced object itself. It copies the types bit by bit. The result is that both instances are cloned and the original will refer to the same object.



Further reading: www.c-sharpcorner.com/UploadFile/56fb14/shallow-copy-and-deep-copy-of-instance-using-C-Sharp

Operator=

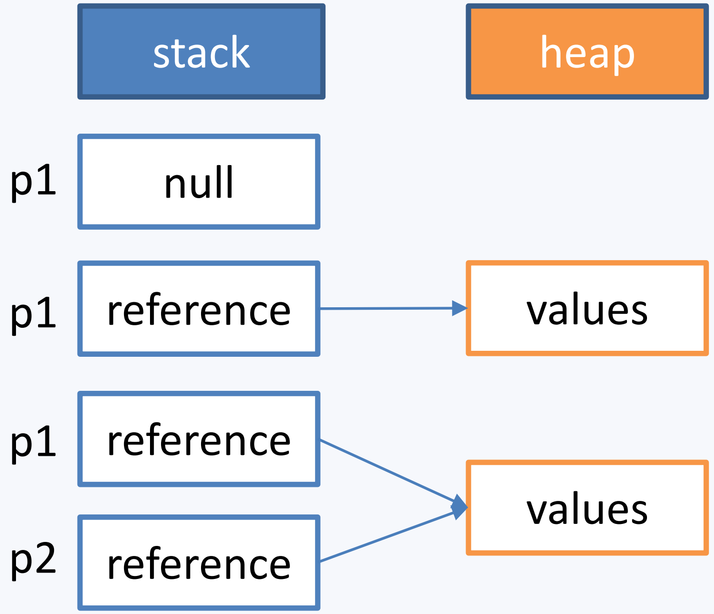
- Does a **shallow copy** - copies the reference and not the object

- Works fine for a Value Type (ex: enum, struct)

```
Person p1;
```

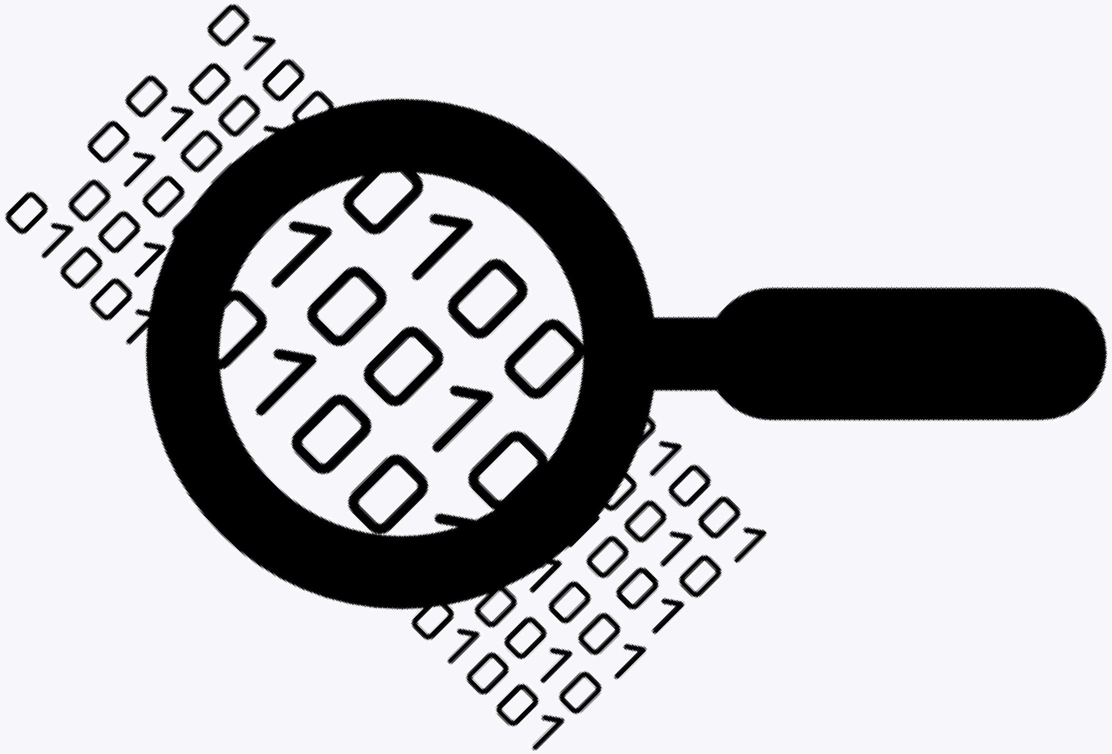
```
p1 = new Person("Name 1", 21);
```

```
var p2 = p1;
```



Operator=

- "EnumStructClass" project – comparison between value and reference types



Operator=

- C# **does not allow** the **assignment operator** to be **overloaded** … and neither does Java ☺ (have any doubts? check the documentation: link)

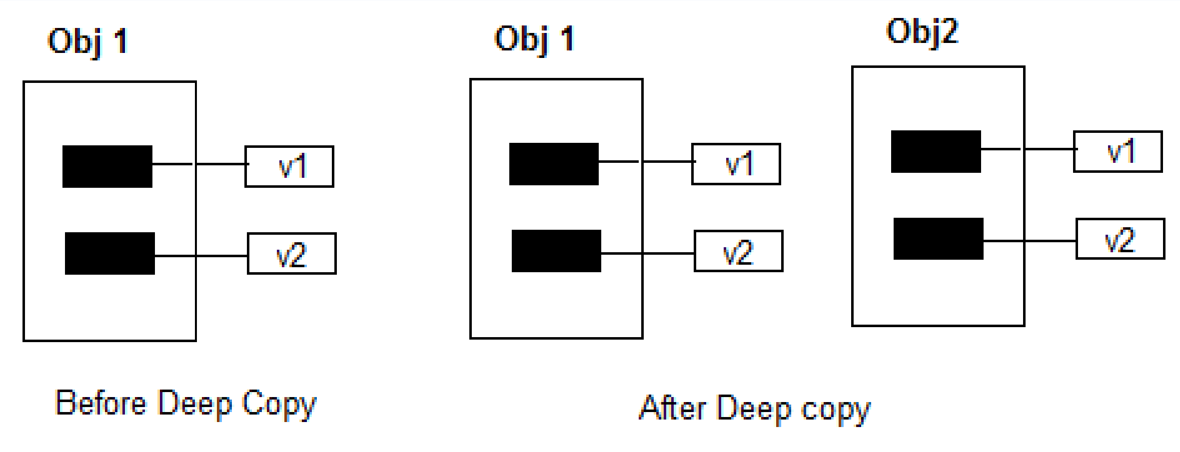
- It's a **language design choice,** likely meant to simplify code understanding. Think what would happen otherwise:

```
var c1 = new ClassFromALibrary();
var c2 = c1;
```

- If we wouldn't have the source code for ClassFromALibrary, it would be difficult to tell whether a **deep** or a **shallow** copy has been made.

Deep Copy

- used to make a complete deep copy of the internal reference types
- If a field is a **value type**, a bit by bit copy of the field is performed. If a field is a **reference type**, a new copy of the referred object is performed.
- A deep copy of an object is a new object with entirely new instance variables, it does not share objects with the old.



Deep Copy

- Implemented using the:
  - Copy Constructor
  - **ICloneable** interface **(most frequently)**
  - Your **own** properly named **interface** (ex: IDeepCopy, IDeepClone, etc.)

Copy Constructor

- C# doesn't provide a copy constructor for objects, but you can write one yourself.

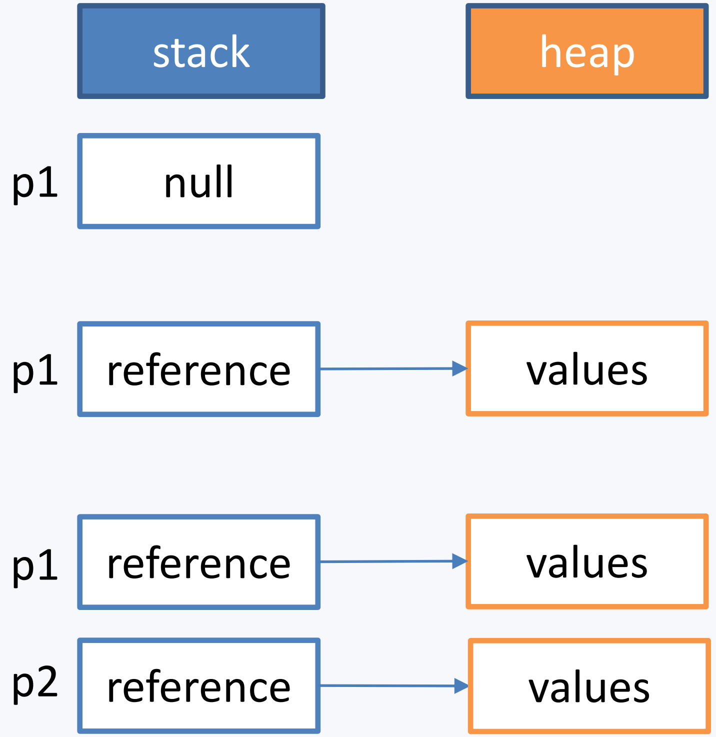
```
//Copy Constructor
public Person(Person person): this(person.Name, person.Age)
{

}

//Constructor
public Person(string name, int age)
{
        Name = name;
        Age = age;
}
```

Copy Constructor

```
Person p1;
```

```
//Create an instance
p1 = new Person("Name 1", 21);
```

```
//Create an instance using the copy constructor
var p2 = new Person(p1);
```



Further reading: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/how-to-write-a-copy-constructor

ICloneable

- Supports cloning, which creates a new instance of a class with the same value as an existing instance.

- **Warning:** as shown in the source code of the framework, the copy can be either shallow or deep (depending on the implementation)

```
public interface ICloneable
{
        // Interface does not need to be marked with the serializable attribute
        // Make a new object which is a copy of the object instanced.
        // This object may be either deep copy or a shallow copy depending on
        // the implementation of clone.
        // The default Object support for clone does a shallow copy.

        Object Clone();
}
```

ICloneable

- Note in **MSDN Documentation**:

*"The **ICloneable** interface simply requires that your implementation of the **Clone()** method return a copy of the current object instance. It does not specify whether the cloning operation performs a deep copy, a shallow copy, or something in between. Nor does it require all property values of the original instance to be copied to the new instance.*

Because callers of Clone() cannot depend on the method performing a predictable cloning operation, we recommend that ICloneable not be implemented in public APIs."

- Further reading: https://docs.microsoft.com/en-us/dotnet/api/system.icloneable

ICloneable

- The **MemberwiseClone** method creates a shallow copy by creating a new object, and then copying the nonstatic fields of the current object to the new object.

```
public class Object
{
    ........

        protected extern object MemberwiseClone();
}
```

- If a field is a **value type**, a bit-by-bit copy of the field is performed.

- If a field is a **reference type**, the reference is copied but the referred object is not; therefore, the original object and its clone refer to the same object (still a **shallow clone**).

ICloneable

```
internal class Person :ICloneable
{
        ...........

        public object Clone(){
                //Get a shallow copy
                var newPerson = (Person)MemberwiseClone();

                return newPerson;
        }
}
```

- Further reading: https://docs.microsoft.com/en-us/dotnet/api/system.object.memberwiseclone

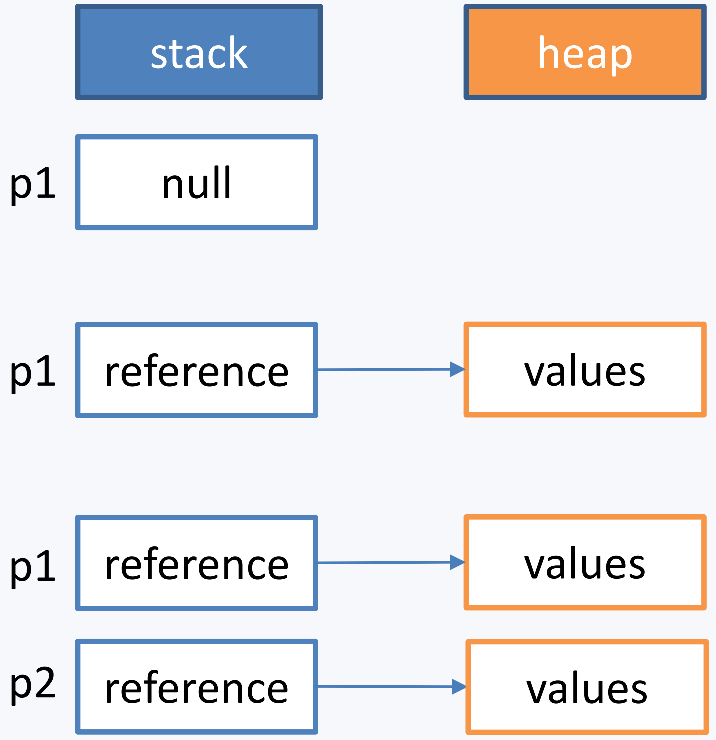
Deep Copy

ICloneable

```
Person p1;
```

```
p1 = new Person("Name 1", 21);
```

```
var p2 = p1.Clone();
```



ICloneable

What happens if the class contains a reference type member (other than System.String) ….?

```
internal class Student : Person, ICloneable
{
    public int[] Marks { get; set; }
}
```

Copy Constructor

```
public Student(Student other)
// First call the copy constructor in the Person class
: base(other)
{
    Console.WriteLine(“Student - Copy Constructor");

    // Then fill in the gaps.
    Marks = new int[other. Marks.Length];

    for (var i = 0; i < other.Marks.Length; i++)
        Marks[i] = other.Marks[i];

    // Marks = (int[])other.Marks.Clone();
}
```

Would we need to change the code if the members of the `Marks` array would be a reference type?

Deep Copy

ICloneable

```
public override object Clone()
{
        // First get a shallow copy.
        var newPerson = (PersonLuckyNumbers)MemberwiseClone();

        // Then fill in the gaps.
        newPerson.LuckyNumbers = new int[LuckyNumbers.Length];
        for (var i=0; i< LuckyNumbers.Length; i++)
        {
                newPerson.LuckyNumbers[i] = LuckyNumbers[i];
        }

        // newPerson.Marks = (int[])Marks.Clone();

        return newPerson;
}
```

System.Object

Boxing and Unboxing

- Boxing is the act of converting a value-type instance to a reference-type instance.

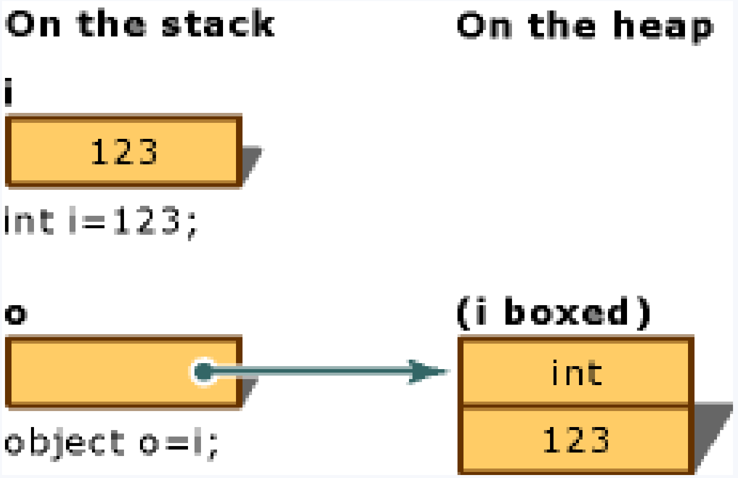
```
int i = 123;
object o = i; // Box the int
```

- Unboxing reverses the operation, by casting the object back to the original value type.

```
int y = (int)o; // Unbox the int
```

Boxing and Unboxing

```
int i = 123;
object o = i; // Box the int

int y = (int)o; // Unbox the int

long zLong = (long)o; // InvalidCastException
string zString = (string)o; // InvalidCastException

byte w = (byte)(int)o; //succeeds
```



Boxing for Console.WriteLine(string format, object arg0)

- C#

```
int value = 10;
//public static void WriteLine(string format, object arg0);
Console.WriteLine("value: {0}",value);
```

- IL

```
// (no C# code)
IL_0000: nop
// int num = 10;
IL_0001: ldc.i4.s 10
IL_0003: stloc.0
// Console.WriteLine("value: {0}", num);
IL_0004: ldstr "value: {0}"
IL_0009: ldloc.0
IL_000a: box [mscorlib]System.Int32
IL_000f: call void
[mscorlib]System.Console::WriteLine(string, object)
```

Collections

Namespaces

Namespace	Contains
System.Collections	Nongeneric collection classes and interfaces
System.Collections.Specialized	Strongly typed nongeneric collection classes
System.Collections.Generic	Generic collection classes and interfaces
System.Collections.ObjectModel	Proxies and bases for custom collections
System.Collections.Concurrent	Thread-safe collections

System.Collections

- The System.Collections namespace contains **interfaces** and **classes** that define **various collections** of objects, such as **lists**, **queues**, bit arrays, **hash tables** and **dictionaries**.

- https://docs.microsoft.com/en-us/dotnet/api/system.collections

System.Collections.**ArrayList**

- Implements the **IList** interface using an array whose size is dynamically increased as required.

```
var words = new ArrayList();

words.Add("melon");
words.Add("avocado");

string first = (string)words[0];
```

Warning: casts cannot be verified by the compiler; the following compiles successfully but then fails at runtime:

```
int first = (int)words[0];
```

System.Collections.ArrayList

- The **ArrayList** class is designed to hold heterogeneous collections of objects. However, it does not always offer the best performance. Instead, the following are recommend:
  - for a heterogeneous collection of objects, use the **List<Object>**;
  - for a homogeneous collection of objects, use the **List<T>** class.

System.Collections.Generic

- The System.Collections.Generic namespace contains interfaces and classes that define **generic collections**, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.

- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic

System.Collections.Generic.List<T>

- The List<T> class is the generic equivalent of the ArrayList class. It implements the IList<T> generic interface by using an array whose size is dynamically increased as required.

```
var words = new List<string>(); // New string-typed list
words.Add("melon");
words.Add("avocado");
words.AddRange(new[] { "banana", "plum" });
words.Insert(0, "lemon"); // Insert at start
words.InsertRange(0, new[] { "peach", "nashi" }); // Insert at start
words.Remove("melon");
words.RemoveAt(3); // Remove the 4th element
words.RemoveRange(0, 2); // Remove first 2 elements

// Remove all strings starting in 'n':
words.RemoveAll(x => x.StartsWith("n"));
```

System.Collections.Generic.List<T>

```
for (var i=0; i<words.Count; i++)
{
      Console.WriteLine(words[i]);
}

foreach (var word in words)
{
      Console.WriteLine(word);
}
```

System.Collections.Generic.List<T>

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
	public List ();
	public List (IEnumerable<T> collection);
	public List (int capacity);

	// Add+Insert
	public void Add (T item);
	public void AddRange (IEnumerable<T> collection);
	public void Insert (int index, T item);
	public void InsertRange (int index, IEnumerable<T> collection);
```

System.Collections.Generic.List<T>

```
// Remove
public bool Remove (T item);
public void RemoveAt (int index);
public void RemoveRange (int index, int count);
public int RemoveAll (Predicate<T> match);

// Indexing
public T this [int index] { get; set; }
public List<T> GetRange (int index, int count);
public Enumerator<T> GetEnumerator();
```

System.Collections.Generic.List<T>

```
// Exporting, copying and converting:
public T[] ToArray();
public void CopyTo (T[] array);
public void CopyTo (T[] array, int arrayIndex);
public void CopyTo (int index, T[] array, int arrayIndex, int count);
public ReadOnlyCollection<T> AsReadOnly();
public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
converter);
```

System.Collections.Generic.List<T>

```
// Other:
public void Reverse(); // Reverses order of elements in list.
public int Capacity { get;set; } // Forces expansion of internal array.
public void TrimExcess(); // Trims internal array back to size.
public void Clear(); // Removes all elements, so Count=0.
```

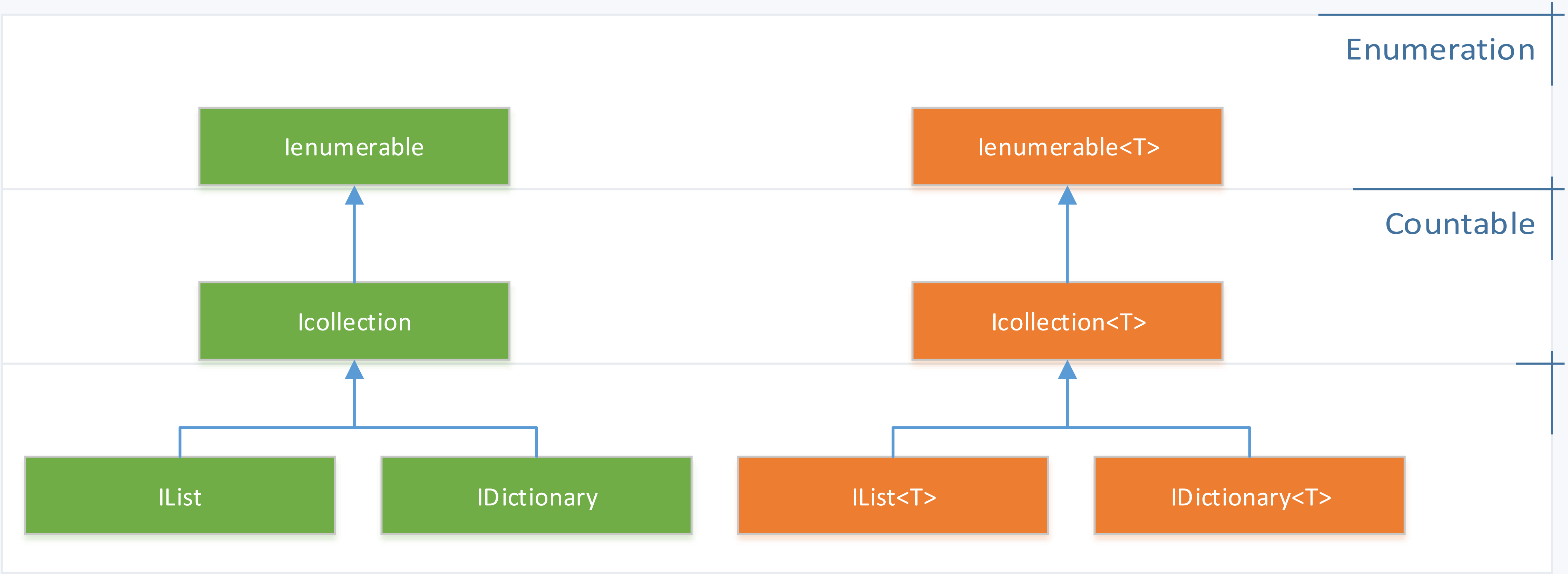
List<T> vs ArrayList

- Internally, List<T> and ArrayList work by maintaining an internal array of objects, replaced with a larger array upon reaching capacity.

- Appending elements is efficient (since there is usually a free slot at the end);

- Inserting elements can be slow (since all elements after the insertion point have to be shifted to make a free slot).

Interfaces



Enumeration
Countable
Ienumerable
Icollection
IList
IDictionary
Ienumerable<T>
Icollection<T>
IList<T>
IDictionary<T>

IEnumerable and IEnumerable<T>

```
public interface IEnumerable{
        IEnumerator GetEnumerator();
}

public interface IEnumerable<T> : IEnumerable{
        IEnumerator<T> GetEnumerator();
}
```

ICollection<T>

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
        int Count { get; }
        bool Contains (T item);
        void CopyTo (T[] array, int arrayIndex);
        bool IsReadOnly { get; }
        void Add(T item);
        bool Remove (T item);
        void Clear();
}
```

IList<T>

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
        T this [int index] { get; set; }
        int IndexOf (T item);
        void Insert (int index, T item);
        void RemoveAt (int index);
}
```

Custom collection

```
public class MyCollection: IEnumerable
{
    int[] items;
    public MyCollection(){
        items = new int[5] {12, 44, 33, 2, 50};
    }

    public MyEnumerator GetEnumerator(){
        return new MyEnumerator(this);
    }

    // Implement the GetEnumerator() method:
    IEnumerator IEnumerable.GetEnumerator(){
        return GetEnumerator();
    }
```

Custom collection

```
public class MyEnumerator: IEnumerator
   {
      int nIndex;

      MyCollection collection;

      public MyEnumerator(MyCollection coll){
         collection = coll;
         nIndex = -1;
      }

      public void Reset(){
         nIndex = -1;
      }
```

Custom collection

```
public bool MoveNext() {nIndex++;
         return(nIndex < collection.items.GetLength(0));
      }

 public int Current
      {
         get
         {
            return(collection.items[nIndex]);
         }
      }
```

Custom collection

```
// The current property on the IEnumerator interface:
    object IEnumerator.Current
    {
        get
        {
            return(Current);
        }
    }
```

Custom collection

```
public class MainClass
{
    public static void Main(string [] args)
    {
        MyCollection col = new MyCollection();
        Console.WriteLine("Values in the collection are:");

        // Display collection items:
        foreach (int i in col)
            Console.WriteLine(i);
    }
}
```

Other Collections

- SortedList<TKey, TValue>
- LinkedList<T>
- Queue<T>
- Stack<T>
- and many others: link

Delegates

Delegates

- A delegate is a reference type that can be used to encapsulate a named or an anonymous method.

- Delegates are similar to function pointers in C++; however, delegates are type-safe and secure.

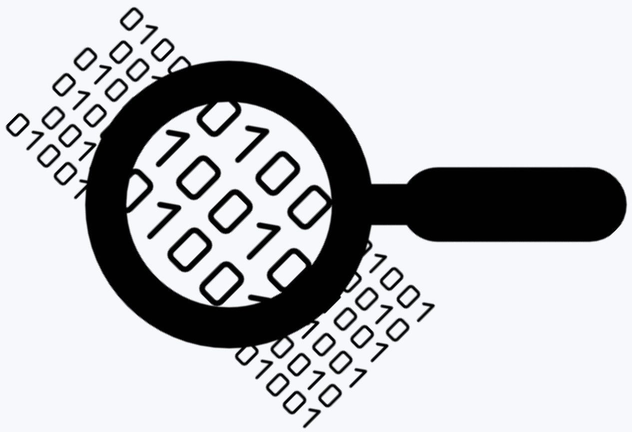
```
public delegate void TestDelegate(string message);

public delegate int TestDelegate(MyType m, long num);
```

Delegates

```
// This delegate can point to any method, taking
two integers and returning an integer.
public delegate int BinaryOp(int x, int y);

//
public class SimpleMath
{
		public static int Add(int x, int y)
		{ return x + y; }
		public static int Subtract(int x, int y)
		{ return x - y; }
}
```

```
private static void Main()
{
		//Definire si instantiere delegat
		BinaryOp b = new BinaryOp(SimpleMath.Add);

		//BinaryOp b = new BinaryOp(SimpleMath.Subtract));

		b += new BinaryOp(SimpleMath.Subtract);
		//Apel prin delegat
		Console.WriteLine("10 + 10 is {0}", b(10, 10));
		Console.ReadLine();
}
```

Sample



- Check the “Delegates” project on GitHub.
- Decompile the code with ILSpy and check the IL code corresponding to:

```
public delegate int BinaryOp(int x, int y);
```

Delegates

- A delegate variable is assigned a method at runtime. This is useful for writing plugin methods.

Plug-in Methods with Delegates

```
public delegate int Transformer (int x);

class Util {
        public static void Transform (int[] values, Transformer t) {
                for (int i = 0; i < values.Length; i++)
                values[i] = t (values[i]);
        }
}

class Program {
        static void Main() {
                int[] values = { 1, 2, 3 };
                Util.Transform (values, Square); // Hook in the Square method
                foreach (int i in values)
                Console.Write (i + " "); // 1 4 9
        }

        static int Square (int x) { return x * x; }
}
```

Multicast Delegates

- All delegate instances have multicast capability. This means that a delegate instance can reference not just a single target method, but also a list of target methods. The + and += operators combine delegate instances.

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2; //d = d + SomeMethod2;
```

- The - and -= operators remove the right delegate operand from the left delegate operand.

```
d -= SomeMethod1;
```

Multicast Delegates

- If a multicast delegate has a nonvoid return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded. In most scenarios in which multicast delegates are used, they have void return types, so this subtlety does not arise.

```
public delegate void ProgressReporter (int percentComplete);

public class Util{
        public static void HardWork (ProgressReporter p)        {
                for (int i = 0; i < 10; i++){
                        p (i * 10); // Invoke delegate
                        System.Threading.Thread.Sleep (100); // Simulate hard work
                }
        }
}
```

Multicast Delegates

```
class Program {
        static void Main(){
                ProgressReporter p = WriteProgressToConsole;
                p += WriteProgressToFile;
                Util.HardWork (p);
        }

        static void WriteProgressToConsole (int percentComplete){
                Console.WriteLine (percentComplete);
        }

        static void WriteProgressToFile (int percentComplete){
                System.IO.File.WriteAllText ("progress.txt",
percentComplete.ToString());
        }
}
```

Multicast Delegates

- Further reading: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/delegates

Events

Events

- Events in the .NET Framework are based on the delegate model. The delegate model follows the observer design pattern, which enables a subscriber to register with, and receive notifications from, a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it.

- The *broadcaster* is a type that contains a delegate field. The broadcaster decides when to broadcast, by invoking the delegate

- The *subscribers* are the method target recipients. A subscriber decides when to start and stop listening, by calling += and −= on the broadcaster's delegate. A subscriber does not know about, or interfere with, other subscribers.

Events

- Events are a language feature that formalizes this pattern. An event is a construct that exposes just the subset of delegate features required for the broadcaster/subscriber model.

- The main purpose of events is to ***prevent subscribers from interfering with each other***.

Example

```
public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);

public class Stock {
        string symbol; decimal price;
        public Stock (string symbol) { this.symbol = symbol; }

        public event PriceChangedHandler PriceChanged;
        public decimal Price
        {
                get { return price; }
                set{
                        if (price == value) return; // Exit if nothing has changed
                        decimal oldPrice = price;
                        price = value;
                        if (PriceChanged != null) // If invocation list not
                                PriceChanged (oldPrice, price); // empty, fire event.
                }
        }
}
```

Example

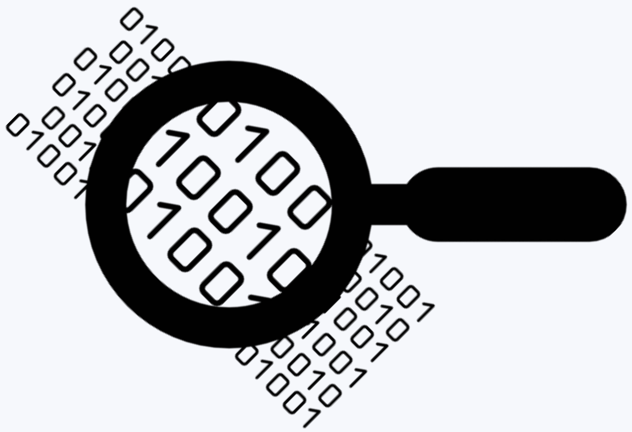
```
private static void Main()
{
        var stock = new Stock("MSFT");
        stock.PriceChanged += Stock_PriceChanged;
        stock.Price = 30;
        stock.Price = 60;
        stock.Price = 90;
}

private static void Stock_PriceChanged(decimal oldPrice, decimal newPrice)
{
        Console.WriteLine("MSFT: {0} {1}", oldPrice, newPrice);
}
```

Example - Discussion

Without the **event** keyword, PriceChanged becomes an ordinary delegate field, making the Stock class less robust. Subscribers could do the following things to interfere with each other:

- replace other subscribers by reassigning PriceChanged (instead of using the += operator).
- clear all subscribers (by setting PriceChanged to null).
- broadcast to other subscribers by invoking the delegate.

Sample



- Check the “EventsPropertyTrigger” project on GitHub.
- Decompile the code with ILSpy and check the IL code corresponding to event.
- Remove the event keyword.
- Decompile the code with ILSpy and check the IL code corresponding to the delegate.

Standard Event Pattern

- The .NET Framework defines a standard pattern for writing events. Its purpose is to provide consistency across both Framework and user code. At the core of the standard event pattern is System.EventArgs

- EventArgs is a base class for conveying information for an event.

```
public class PriceChangedEventArgs : EventArgs
{
        public readonly decimal LastPrice;
        public readonly decimal NewPrice;
        public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)
        {
                LastPrice = lastPrice;
                NewPrice = newPrice;
        }
}
```

Standard Event Pattern

```
public class Stock{
        private string _symbol;  private decimal _price;
        public Stock(string symbol){ _symbol = symbol;    }

        public event EventHandler<PriceChangedEventArgs> PriceChanged;
        protected virtual void OnPriceChanged(PriceChangedEventArgs e)    {
                if (PriceChanged != null) PriceChanged(this, e);
        }
        public decimal Price
        {
                get { return _price; }
                set {
                        if (_price == value) return;
                        decimal oldPrice = _price;  _price = value;

                        OnPriceChanged(new PriceChangedEventArgs(oldPrice, _price));
                }
        }
}
```

Standard Event Pattern

```
private static void Stock_PriceChanged(decimal oldPrice, decimal newPrice)
{
        Console.WriteLine("MSFT: {0} {1}", oldPrice, newPrice);
}
```

```
private static void Stock_PriceChanged1(object sender, PriceChangedEventArgs e)
{
        Console.WriteLine("MSFT: {0} {1}", e.LastPrice, e.NewPrice);
}
```

Events

Further reading: link

Disposal and Garbage Collection

Disposal and Garbage Collection

- Some objects require explicit teardown code to release resources such as open files, database connections, operating system handles, and unmanaged objects. In .NET parlance, this is called **disposal**, and it is supported through the **IDisposable** interface.

- The managed memory occupied by unused objects must also be reclaimed at some point; this function is known as **garbage collection** and is performed by the CLR.

IDisposable, Dispose, and Close

- explicitly initiated;

- the .NET Framework defines a special interface for types requiring a tear-down method:

```
public interface IDisposable
{
        void Dispose();
}
```

using

- provides a syntactic shortcut for calling **Dispose** on objects that implement **IDisposable**, using a **try/finally** block

```
using (FileStream fs = new FileStream ("myFile.txt", FileMode.Open)){
        // ... Write to the file ...
}
```

converted by the compiler into:

```
FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
try{
        // ... Write to the file ...
}
finally{
        if (fs != null) ((IDisposable)fs).Dispose();
}
```

using

- the finally block ensures that the Dispose method is called even when an exception is thrown or the code exits the block early.

IDisposable

```
public interface IDisposable
{
        void Dispose();
}
```

```
sealed class Demo : IDisposable
{
        public void Dispose()
        {
                // Perform cleanup / tear-down.
                ...
        }
}
```

IDisposable

1. Once disposed, an object is beyond redemption. It **cannot be reactivated**, and calling its methods or properties throws an ObjectDisposedException.

2. Calling an object's Dispose method repeatedly causes no error.

3. If disposable **object** ***x*** contains or "wraps" or "possesses" disposable **object** ***y***, ***x*****'s Dispose** method automatically calls ***y*****'s Dispose** method.

Automatic Garbage Collection

- The CLR frees the memory in the heap entirely automatically, via GC – Garbage Collection.
- Garbage collection does not happen immediately after an object is orphaned. The CLR bases its decision on when to collect upon a number of factors, such as:
  - the available memory,
  - the amount of memory allocation,
  - and the time since the last collection.
- There's an indeterminate delay between an object being orphaned and being released from memory. This delay can range from nanoseconds to days.

Automatic Garbage Collection

```
internal class Student {
        public int Age {get;set;} //int is a value type

        public string Name {get;set;} // System.String is a reference type
        public int[] Marks { get; set; } // System.Array is a reference type

        public Student(string name, int age, int[] marks){
                ....
        }
}
```

```
static void Main(){

        {
                var p = new Student("Name1", 21, new []{10, 10, 10});
        }

        //object is orphaned
}
```

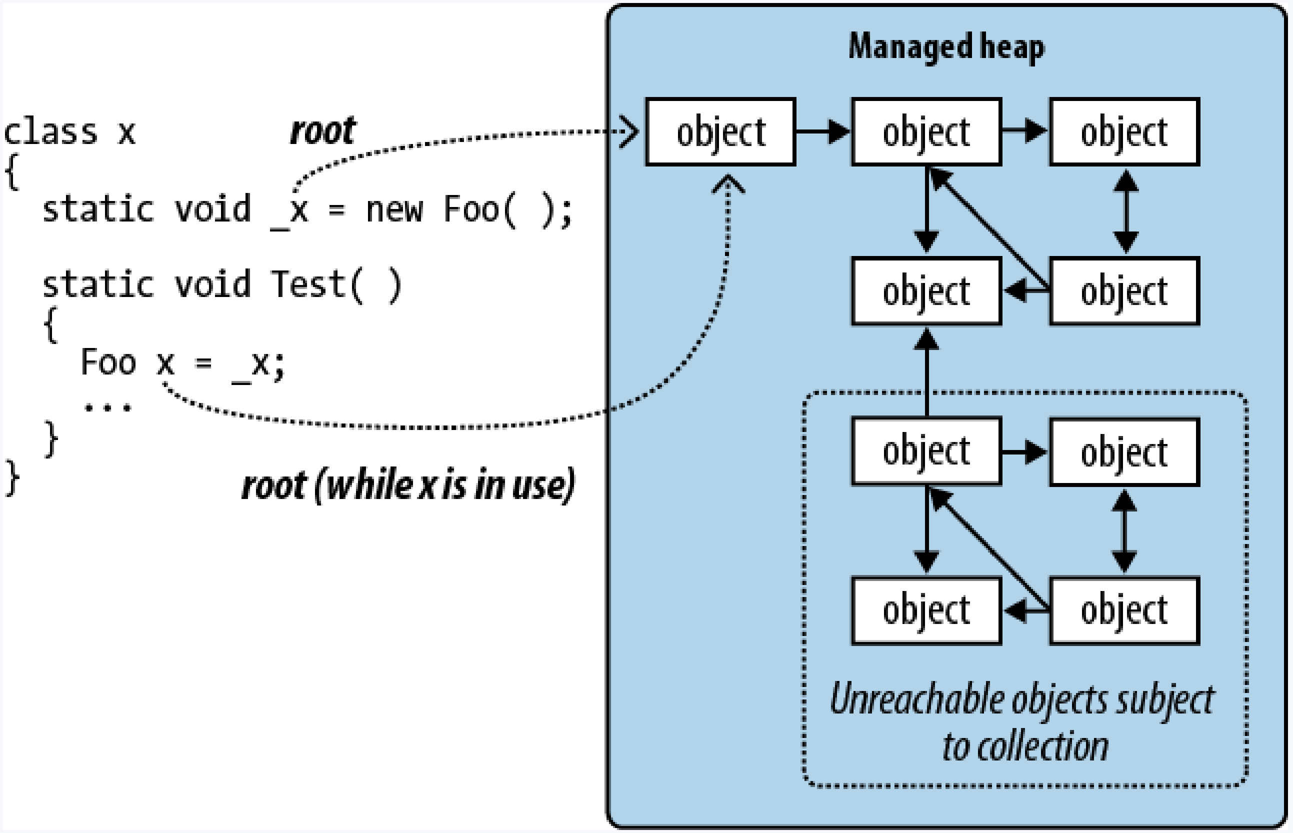
Roots

A root is one of the following:

- a local variable or parameter in an executing method (or in any method in its call stack)
- a static variable
- an object on the queue that stores objects ready for finalization (see next section)

Roots

- objects that cannot be accessed by following the arrows (references) from a root object are *unreachable* —and therefore subject to collection.



Finalizers

- prior to an object being released from memory, its finalizer runs, if it has one.

```
internal class PersonClass
{
        //Finalizer
        ~PersonClass()
        {
                Console.WriteLine("Finalizer");
        }
}
```

- finalizers cannot be declared as public or static, cannot have parameters, and cannot call the base class

Finalizers

Garbage collection works in distinct phases:

1. the GC identifies the unused objects for deletion. Those without finalizers are deleted right away. Those with pending (unrun) finalizers are kept alive (for now) and are put onto a special queue. At that point, garbage collection is complete, and the program continues executing.

2. the *finalizer thread* kicks in and starts running in parallel to the program, picking objects off that special queue and running their finalization methods. Prior to each object's finalizer running, it's still very much alive—that queue acts as a root object. Once it's been dequeued and the finalizer executed, the object becomes orphaned and will get deleted in the next collection.

Finalizers

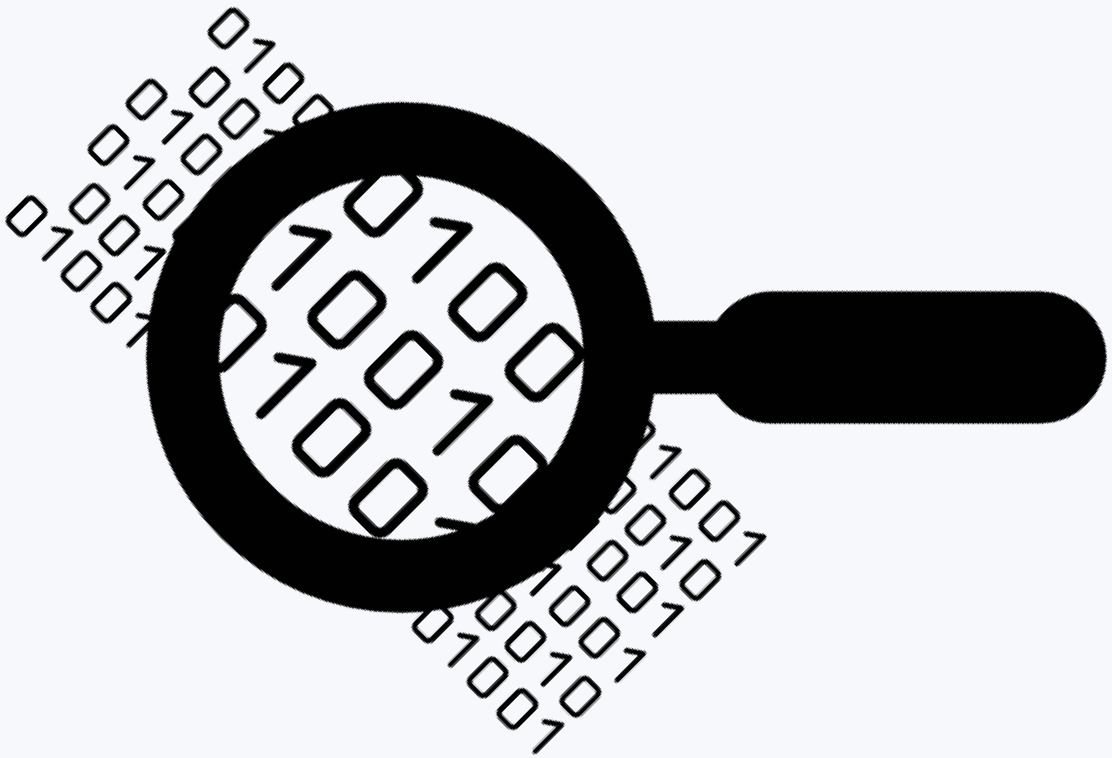
- Finalizers slow the allocation and collection of memory (the GC needs to keep track of which finalizers have run).
- Finalizers prolong the life of the object and any referred objects (they must all await the next garbage truck for actual deletion).
- It's impossible to predict in what order the finalizers for a set of objects will be called.
- You have limited control over when the finalizer for an object will be called.

Finalizers

- If code in a finalizer blocks, other objects cannot get finalized.
- Finalizers may be circumvented altogether if an application fails to unload cleanly.

Demo

- Memory Profiler



Dispose Pattern

- Further reading: link

Globalization