SOFTWARE DEVELOPMENT I

3rd lecture

Today

- Main OOP pillars
- SOLID

- Type conversions
- Constructors in depth
- Class inheritance
- Interfaces
- Standard .NET interfaces
  - IComparable
  - IComparer
  - IEquatable
  - IEnumerable
  - ICloneable (and cloning)

SOLID

SOLID - design principles for more understandable, flexible and maintainable software.

S – Single responsibility principle (SRP)

O – Open/closed principle (OCP)

L – Liskov substitution principle (LSP)

I – Interface segregation principle (ISP)

D – Dependency inversion principle (DIP)

OCP (Open/closed principle)

“You should be able to extend a classes behavior, without modifying it”

~

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”

Achieved via OOP (e.g. polymorphism)

We start thinking about OCP, as soon as there is a need "add one more..."

Bad example

```
class Customer
{
    public int CustomerType {get; set;}
    public double GetDiscount(double TotalSales){
    {
        if(CustomerType == 1)
        {
            return TotalSales - 100;
        }
        else
        {
            return TotalSales - 50;
        }
    }
}
```

Good example

```
class Customer
{
    public virtual double getDiscount(double TotalSales)
    {
        return TotalSales;
    }
}

class SilverCustomer : Customer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 50;
    }
}

class goldCustomer : SilverCustomer
{
    public override double getDiscount(double TotalSales)
    {
        return base.getDiscount(TotalSales) - 100;
    }
}
```

Bad example

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class Circle
{
    public double Radius { get; set; }
}

public class AreaCalculator
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle)shape;
                area += rectangle.Width * rectangle.Height;
            }
            else
            {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
            }
        }
        return area;
    }
}
```

- AreaCalculator **not closed** for modification – if logic change is needed, code change is needed
- e.g. it is not possible to adress logic change with adding (not changing) the code (not open for **extension**).

Good example

```
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width * Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius * Radius * Math.PI;
    }
}
```

```
public class AreaCalculator
{
    public double Area(Shape[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Area();
        }

        return area;
    }
}
```

LSP (Liskov substitution principle)

"objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program“

~

“ subtype behavior should match base type behavior as defined in the base type specification”

In simple terms: *Derived classes must be substitutable for their base classes.*

Example

```
public class Rectangle
{
   public int Width { get; protected set; }
   public int Height { get; protected set; }

   public void SetWidth(int width) => Width = width;
   public void SetHeight(int height) => Height = height;

   public int GetArea()
   {
       return Width * Height;
   }
}
```

```
public class Square : Rectangle
{
    public void SetWidth(int width)
    {
        Width = width;
        Height = width;
    }

    public void SetHeight(int height)
    {
        Width = height;
        Height = height;
    }
}
```

```
public class LspTest
{
    private static Rectangle CreateRectangle()
    {
        return new Square();
    }

    public static void Main(string[] args)
    {
        Rectangle rect = CreateRectangle();

        rect.SetWidth(5);
        rect.SetHeight(10);
        // User assumes that rect is a rectangle.
        // They assume that they are able to set the width and height as for the base class

        Assert.AreEqual(rect.GetArea(), 50); // This check fails for a square! We get 100
    }
}
```

LSP Checklist

- **No new exceptions should be thrown in derived class**: If your base class threw ArgumentException then your subclasses are only allowed to throw exceptions of type ArgumentException or any exceptions derived from it. Throwing IndexOutOfRangeException is a violation of LSP.
- **Pre-conditions cannot be strengthened**: Assume your base class works with a member int. Now your subtype requires that int to be positive. This is strengthened pre-conditions, and now any code that worked perfectly fine before with negative ints is broken.
- **Post-conditions cannot be weakened**: Assume your base class required all connections to database to be closed before the method returned. In your subclass you overrode that method and left connection open for further reuse.

ISP (Interface segregation principle)

"Clients should not be forced to implement interfaces they do not use"

Bad example

```
class Machine : IMachine
{
    public Machine() { }

    public void DoPrint(List<Item> item)
    {
        Console.WriteLine("All Items printed" + item.Count());
    }

    public void DoStaple(List<Item> item)
    {
        Console.WriteLine("Items stapled" + item.Count());
    }

    public void DoFax(List<Item> item)
    {
        Console.WriteLine("All Items Faxed" + item.Count());
    }

    public void DoScan(List<Item> item)
    {
        Console.WriteLine("All Items Scanned" + item.Count());
    }

    public void DoPhotoCopy(List<Item> item)
    {
        Console.WriteLine("All Items Photo copied" + item.Count());
    }
}
```

```
interface IMachine
{
    public void DoPrint(List<Item> item);
    public void DoStaple(List<Item> item);
    public void DoFax(List<Item> item);
    public void DoScan(List<Item> item);
    public void DoPhotoCopy(List<Item> item);
}
```

- All code needs to be recompiled for even the smallest changes.
- What if device wants only to print?
- This is a fat interface.

Better example

```
interface IPrinter
{ void DoPrint(List<Item> item);    }

interface IStaple
{ void DoStaple(List<Item> item);    }

interface IFax
{ void DoFax(List<Item> item);    }

interface IScan
{ void DoScan(List<Item> item);    }

interface IPhotoCopy
{ void DoPhotoCopy(List<Item> item);    }

interface IMachine : IPrinter, IFax, IScan, IPhotoCopy, IStaple { }

class Machine : IMachine
{
    private IPrinter printer { get; set; }
    private IFax fax { get; set; }
    private IScan scan { get; set; }
    private IPhotoCopy photocopy { get; set; }
    private IStaple staple { get; set; }

    public Machine(IPrinter printer, IFax fax, IScan scan, IPhotoCopy photoCopy, IStaple staple)
    {// (constructor dependency injection)
        this.printer = printer;
        this.fax = fax;
        this.scan = scan;
        this.photocopy = photocopy;
        this.staple = staple;
    }
```

ISP summary

- We favor:
  - Composition instead of Inheritance
    - Separating by roles (responsibilities)
  - Decoupling over Coupling
    - Not coupling derivative classes with unneeded responsibilities inside a monolith

DIP (Dependency inversion principle)

"Depend on abstractions, not on concretions"

– *High level modules should not depend upon low level modules. Both should depend upon abstractions.*

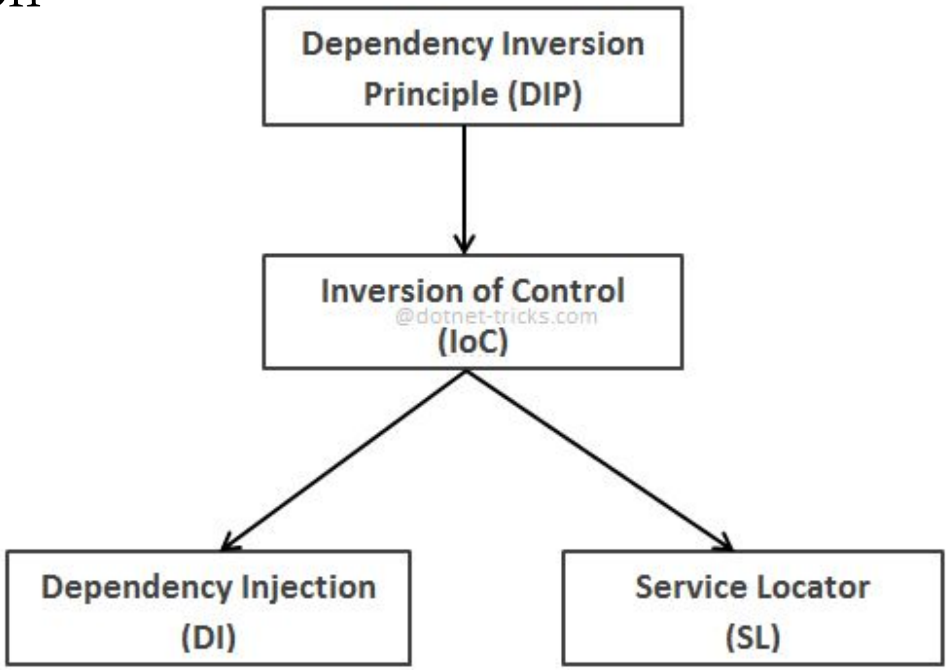
– *Abstractions should not depend upon details. Details should depend upon abstractions.*

SOLID: DIP

Dependency injection – most common way to implement DIP.

Others:

- Service Locator
- Delegates
- Events
- Etc.



Types conversion (1)

- Widening vs narrowing:
  - Widening: type that we are converting to can store more values than type from which we are converting (short -> int).
  - Narrowing: vice versa(int -> short).
- C# does not throw an exception, if narrowing conversion fails for integers or floating point numbers.
  - For integer values value is decreased
  - For floating point numbers infinity value is set.

Converting integer values

```csharp
// short max value is 32767
var first = 32769;
short second = (short)first;
Console.WriteLine(first);
Console.WriteLine(second);
Console.ReadKey();
```

```
32769
-32767
```

Types conversion (2): solutions

- Integers: using **checked** statement, which throws *OverflowException*

```
checked
{
    int big = 1000000;
    short small = (short)big;
}
```

- Integers: project settings conf
  - Properties -> Build tab -> Advanced -> Check For Arithmetic Overflow (true).
  - Disadvantage – code does not reflect program behavior.
- Floating point numbers:

```
double big = -1E40;
float small = (float)big;
if (float.IsInfinity(small)) throw new OverflowException();
```

Implicit vs explicit conversion

- Implicit: conversion without using additional code.
- Explicit: using additional code (like *cast or parsing*) methods.

```
// Narrowing conversion so explicit conversion is required.
double value1 = 10;
float value2 = (float)value1;

// Widening conversion so implicit conversion is allowed.
int value3 = 10;
long value4 = value3;
```

- Converting floating point numbers to integers, everything after “.” is cut:
  - (int)10.9 returns 10.

Reference types conversion

- Reference types conversion to a base class or interface is possible implicitly.
- If **Employee** class inherits from Person class, then Employee object can be converted to Person object implicitly:

```
Employee employee1 = new Employee();
Person person1 = employee1;
```

Reference types conversion

- Reference types conversion to a base class or interface does not change the actual value, just makes it look as a new type.

```
Employee employee1 = new Employee();
Person person1 = employee1;
```

  - **person1** is **Person** type variable, but points to **Employee** object.
  - Code can use **person1** object as **Person** type, but in memory it stays as **Employee** type object.
- See ref-conversions.ipynb

IS

- **is** returns **true**, if objects are compatible (if casting/conversion is possible)
- „**person is Employee**“ returns **true** not only when person is **Employee** type, but also when person is **Manager** type (because **Manager** is **Employee**)

```
if(person1 is Employee)
{
    Employee emp = (Employee)person1;
    // we can do stuff with Employee here
}
```

AS

- **as** operator work as *cast*. If conversion fails, **as** returns **null** instead of throwing an exception.
- Syntactic sugar

```
Employee emp = person1 as Employee;
if(emp != null)
{
    // we can do stuff with Employee here
}
```

- Arrays conversion: **array-casts.ipynb**
  - **cast** does not create new arrays!
  - As this would not create as well:

```
int[] array1, array2;
array1 = new int[10];
array2 = array1;
```

Parse and tryParse

- All primitive C# data types (int, bool, double, *and so forth*) has **Parse** method.
- bool.Parse("yes") will throw FormatException
- bool.Parse("true") returns **bool** type **true** value.
- **parse** throws exceptions, tryParse returns **out** parameter containing parse result or null (if parse failed).
- Parse requires pre-validation of data.
- Difficult to work with different culture information.
- See parsing.ipynb

System.Convert

- "bankers rounding":
  - Rounds to the closest integer value.
  - If it ends with .5, then it rounds to closest even number.For example below would result in 10:

```
Console.WriteLine(Convert.ToInt32(9.5));
Console.WriteLine(Convert.ToInt32(10.5));
```

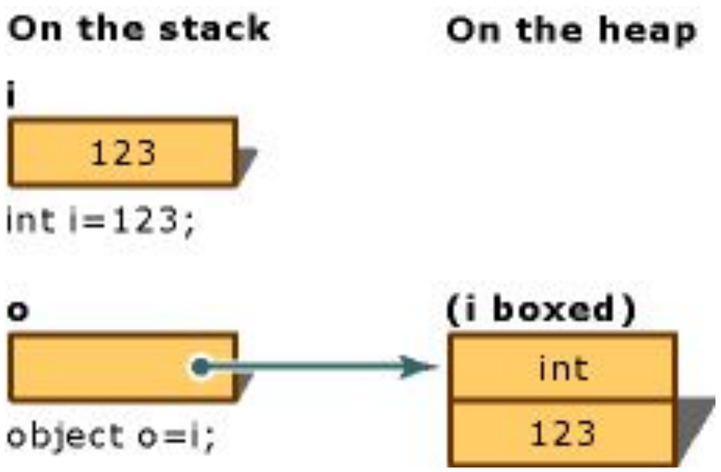
- You can also do it like that:

```
Console.WriteLine(Convert.ToInt32(9.5));
Console.WriteLine((int)Convert.ChangeType(9.5, typeof(int)));
```

ToBoolean
ToByte
ToChar
ToDateTime
ToDecimal
ToDouble
ToInt16
ToInt32
ToInt64
ToSByte
ToSingle
ToString
ToUInt16
ToUInt32
ToUInt64

Boxing/unboxing (1)

- Process when value type is converted to **object** or **interface** type, which value types implements.
  - Lets say we are converting **int** or **bool** (or similar) to **object** type, or to **interface**, which is supported by that value type (e.g. **struct**).
- Unboxing is a process, when boxed value is converted back from reference type to value type.
- Both processes are slow:
  - Boxing – because of heap usage
  - Unboxing – because of casting



Boxing/unboxing (2)

- Boxing is implicit; unboxing is explicit.

```
int i = 10;
object iObject = i; // Box i.
i = (int)iObject;   // Unbox
```

- Sometimes is happens silently:

```
Console.WriteLine(string.Format("i is: {0}", i));
```

Boxing/unboxing (3)

- What will be printed out?

```
int i = 10;
object iObject = i;

i = 1;
iObject = 2;

Console.WriteLine(i);
Console.WriteLine(iObject);
```

```
object iObject1 = 1;
object iObject2 = 2;

iObject1 = iObject2;
iObject2 = 5;

Console.WriteLine(iObject1);
Console.WriteLine(iObject2);
```

1 and 2

2 and 5

Constructors

- Constructor – it is a method that is being called first when an instance of a class or struct being created.
- Same for static constructor – but only for the first time.
- What constructors can do:
  1. Overload

```
class Demo
{
    public Demo() { }
    public Demo(int param) { }
    public Demo(string param) { }
}
```

Constructor

- What constructors can do:

  2. Call base class constructor using keyword : **base** (see constructors.ipynb)

```
public Employe(string firstName, string lastName, string departament)
    : base(firstName, lastName)
    // base class constructor will be called first
```

  3. If there are no explicit constructor defined – the default constructor is being created implicitly:

     - There are no parameters.
     - Field values are initialized to default value.
     - Causes problems when changes are needed.

Constructor

- What constructors can do:
  4. Call same class different constructors using: **this**

```
class SomeClass
{
    // Parameterless constructor
    public SomeClass() : this(1)
    {
        Console.WriteLine("Hello from parameterless constructor");
    }

    // Paramterized constructor
    public SomeClass(int number) : this(number.ToString())
    {
        Console.WriteLine("Hello from parameterized constructor (int number)");
    }

    // Paramterized constructor
    public SomeClass(string str)
    {
        Console.WriteLine("Hello from parameterized constructor (string sr)");
    }
}
```

Constructor

- What constructors can't do:
  5. Cannot call multiple other constructors.

```
public Customer(string email)
{
    Email = email;
}
public Customer(int age)
{
    Age = age;
}
public Customer(string email, int age)
    : this(age), this(email)
{
    //ain't gonna work.
}
```

```
public Customer(string email)
{
    StoreEmail(email);
}
public Customer(int age)
{
    StoreAge(age);
}
public Customer(string email, int age)
{
    StoreAge(age);
    StoreEmail(email);
}

private void StoreEmail(string email){}
private void StoreAge(string age){}
```

Can constructor be non-public?

Constructors

What constructors can do:

6. **Private/public** constructors:

   1. **Public** is standard
   2. **Private** constructors are not allowed to be called from other classes, so if we want to create an instance of such class, there is a special implementation that we have to provide.

See singleton.ipynb

```
class Program
{
    static void Main(string[] args)
    {
        Singleton SingletonObject = Singleton.GetObject();
        SingletonObject.Print("Hello World");
        Console.ReadLine();
    }
}

public class Singleton
{
    protected static Singleton _obj;
    private Singleton()
    {

    }
    public static Singleton GetObject()
    {
        if (_obj == null)
        {
            _obj = new Singleton();
        }
        return _obj;
    }
    public void Print(string s)
    {
        Console.WriteLine(s);
    }
}
```

Singleton can be implement with Lazy<>

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public static Singleton Instance { get { return lazy.Value; } }

    private Singleton()
    {
    }
}
```

Constructors

What constructors can do:

7. Static constructor (see static.ipynb)
   - Is called implicitly when:
     - Class instance is created
     - Class static fields or methods are used for the first time
   - Class can have only one static constructor
   - Has to be parameterless, because CLR is calling it
   - Can access only static fields/methods of this class
   - Static constructor does not have access modifiers
   - Slow

```
class A
{
    static A()
    {
        Console.WriteLine("Init A");
    }
    public static void F()
    {
        Console.WriteLine("A.F");
    }
}
class B
{
    private static string smth;
    static B()
    {
        Console.WriteLine("Init B");
    }
    public static string Smth
    {
        get
        {
            return smth;
        }
        set
        {
            smth = value;
            Console.WriteLine("Smth B");
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        A.F();
        B.Smth = "a";

        Console.ReadLine();
    }
}
```

Initializer

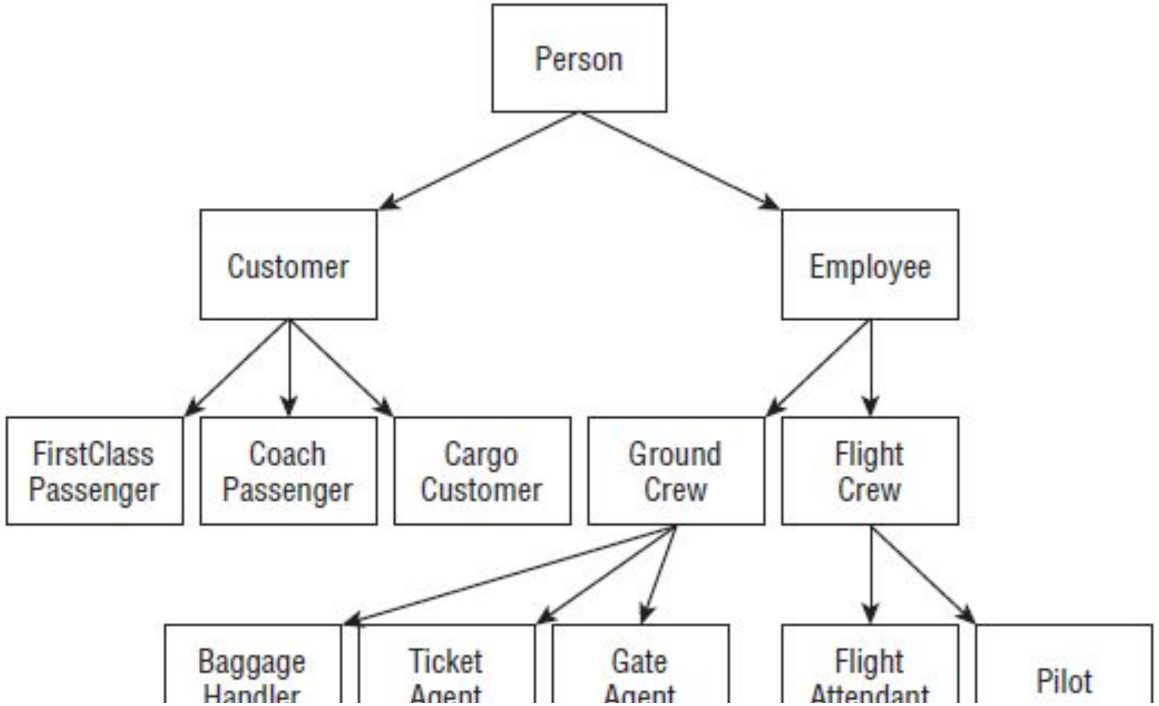
- Explicit creation of an object by setting all the properties manually.
- Only the standard constructor is called
- Example in the next slide

```csharp
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        // new object initialized by calling standard constructor and using explicit initialization
        var studentNameInitializer = new StudentName() { FirstName = "Michael", LastName = "Jordan" };

        // new object created by using constructor with parameters that set properties
        var studentNameConstructor = new StudentName("Michael", "Jordan");
    }
}

4 references
public class StudentName
{
    2 references
    public string FirstName { get; set; }

    2 references
    public string LastName { get; set; }

    1 reference
    public StudentName() { }

    1 reference
    public StudentName(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

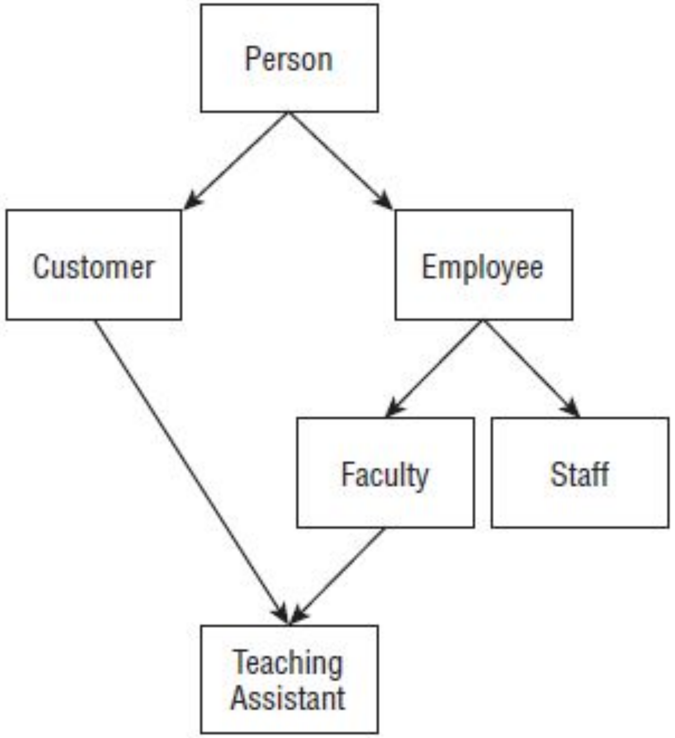
Questions about constructors?

Inheritance

Allowed



Not Allowed



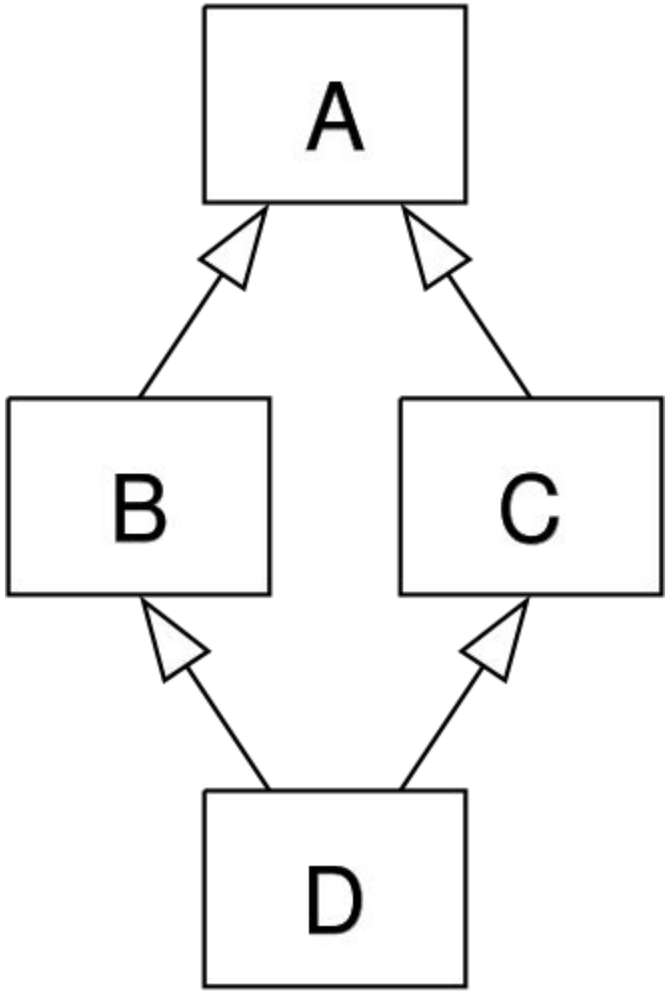
Multiple inheritance

Diamond problem:

If A has a method, which B and C classes have overridden, but D did not, then which method will D inherit – from B or from C?

From A method is called successfully, but from D – not necessarily.

C# solution: **interface**



Interface

- Interface **exposes a contract**, specifying characteristics that a class **must** implement**.**
- Can state required: properties, methods and actions.
- Interface can not contain any **static** members
- Interface **can** have implementation of the methods (different from abstract class, because abstract class can have implementation).

Interface

- Since it is similar to inheritance, sometimes it is being called interface inheritance
- Class can inherit from ONE base class, and MANY interfaces
- Look **interfaces.ipynb – Person**

Can a class implement two interfaces which has methods with same signatures?

Explicit and implicit interface implementation

- If class implements an interface explicitly, then to access implemented method you will need an object of interface type, if an interface is implemented implicitly – then you can access method with class type object.
- Explicitly implementing interface requires to write interface name before method name like:
  - **void Interface.Method**
- In interfaces.ipynb look at **ExplicitImplementor** which implements **InterfaceA** interface explicitly, and **InterfaceB** – implicitly.

```csharp
interface InterfaceA
{
    void Method();
}

interface InterfaceB
{
    void Method();
}

class ExplicitImplementor : InterfaceA, InterfaceB
{
    void InterfaceA.Method()
    {
        Console.WriteLine("Hi from InterfaceA");
    }

    public void Method()
    {
        Console.WriteLine("Hi from InterfaceB");
    }
}
```

```csharp
var instance = new ExplicitImplementor();

instance.Method();

InterfaceA a = instance;
a.Method();

InterfaceB b = instance;
b.Method();
```

Results in ⬇️

```
Hi from InterfaceB
Hi from InterfaceA
Hi from InterfaceB
```

Explicit and implicit interface implementation

- **Implicit implementation** is the default way of implementing interfaces, because the code **looks more intuitive** this way.
- Explicit implementation is required in some cases:
  - When class and interface methods or properties have the same signature.
  - When multiple interfaces asks to implement method or property with same signature.
- When there is a need to “hide” method or property behind interface.

Explicit and implicit interface implementation

- If you are implementing interface implicitly then the methods will be available for class that implements this interface type objects, and for interface type objects. Sometimes this is not a desired functionality.
- If you are implementing interface explicitly, then access modifier must be **private**, because your method can only be accessed via interface.
- When implementing explicitly, we don’t have duplicate names problems.
- In reality – 90% of implicit implementation.

Interface delegation

- If both **Student** and **SomePerson** implements **IStudent** interface, then both have a code, which ensures that contract is fulfilled.
- However, if their implementation is identical - we have a **code duplication** problem.
- Duplication of code can be avoided by using interface delegation.
  - This basically means that **proxy the implementation** of interface to the other class.

Interface delegation

- In the delegation process an object of type **Student** is being created in **SomePerson** class.
  - When **SomePerson** object has to perform methods, which are in **IStudent** interface, then **Student** object is called to do that.
- Look interfaces.ipynb cell with **SomePerson** class in it.

```
class SomePerson : IStudent
{
    private readonly Student _innerStudent = new Student();

    public void Study() => _innerStudent.Study();
}
```

Interface is a TYPE

- You can specify it as a parameter to a method

```
void DoSound(IMakeSound soundMaker)
{
    soundMaker.MakeSound();
}
```

  - If you are passing a class object, that implement an interface, then this object is implicitly being casted to a interface type.
- Return type can be an interface:

```
IMakeSound CreateSoundMaker()
{
    var random = new Random();

    return random.Next() % 2 == 0 ?
        new Cat() :
        new Dog();
}
```

Interface is a TYPE

- Casting operators, to check if interface type is implemented:

```
var obj = new object();

// Will be null because it's not actually cat
// but the code is legit.
IStudent student = obj as IStudent;
student.Display();

// And in turn this is false
var isStudent = obj is IStudent;
isStudent.Display();
```


ME: I SHOULD BUILD AN INTERFACE
ALSO ME: NAA MAKE A SIMPLE CLASS AND GET IT DONE
memegenerator.net

Interfaces advantages

- Question: why should we define interface, implement it in a class and then create interface type of object, instead of class type object?
- Answer:

```
void DoSound(IMakeSound soundMaker)
{
    soundMaker.MakeSound();
}

var list = new List<IMakeSound> { new Cat(), new Dog(), };

foreach (var item in list)
{
    DoSound(item);
}
```

Generic Interface

- Interface can be generic (have a type passed as parameter)

```
interface IGenericInterface<T>
{
    T FirstMethod();
    void SecondMethod(T param);
}

class GenericInterfaceImplementor : IGenericInterface<int>
{
    public int FirstMethod() { return 0; }
    public void SecondMethod(int param) { }
}
```

Generic constraints

- **where** is used to specify constraints of the types
- **new()** specifies that class has public parameterless constructor

```
class ClassWithConstraints<T1, T2>
    where T1 : new()
    where T2 : IEnumerable
{
    private readonly T1 _t1;
    private readonly T2 _t2;

    public ClassWithConstraints(T1 t1, T2 t2)
    {
        _t1 = t1;
        _t2 = t2;
    }
}
```

Standard interface implementations

- Benefit – contract implementation
- .NET behaves **“better”** with types, that implement:
  - **IComparable** interface, Array.Sort() method can sort an array of that class members.
  - **IEquatable** interface, then list.Contains() can check, whether an object is really in the list(instead of checking if same pointer is in the list)

IComparable

- Used for comparing **this** object to a given object.
- Has one method: **CompareTo** (one param., obj)
- Has both **simple** and **generic** version

Value	Meaning
Negative	This instance precedes obj in the sort order.
Zero	This instance occurs in the same position in the sort order as obj.
Greater than zero	This instance follows obj in the sort order.

IComparable

Simple

```csharp
class SimpleComparable : IComparable
{
    public int SomeProperty { get; set; }

    public int CompareTo(object other)
    {
        var comparable = other as SimpleComparable;
        if (comparable == null)
        {
            throw new ArgumentException(
                "Must be non null SimpleComparable.",
                nameof(other));
        }

        return SomeProperty - comparable.SomeProperty;
    }
}
```

Generic

```csharp
class GenericComparable : IComparable<GenericComparable>
{
    public int SomeProperty { get; set; }

    public int CompareTo(GenericComparable other)
    {
        if (other == null)
        {
            throw new ArgumentNullException(
                nameof(other));
        }

        return SomeProperty - other.SomeProperty;
    }
}
```

IComparer

- IComparable<T> says **I'm comparable.**
- IComparer<T> says **I'm comparer**.
- Method: **compare(two params)**

Value	**Meaning**
Less than zero	First object is less than the second.
Zero	Both object are equal.
Greater than zero	First object is more than the second.

IComparer

```
class SomeClass
{
    public int SomeProperty { get; set; }
}

class Comparer : IComparer<SomeClass>
{
    public int Compare(SomeClass left, SomeClass right)
    {
        return left.SomeProperty - right.SomeProperty;
    }
}
```

- See **icomparer.ipynb**
- If you want to use Linq ordering on custom types, then you should use **IComparer** or **IComparable.**

IEquatable

- Is used for comparing if two objects are equal.
- Has method **Equals**.
- Generic collections: List, Dictionary, Stack, Queue (etc.) has **Contains** method, which compares objects for equality.
- If **IEquatable** interface is implemented then List.Contains check by using our implemented **Equals** method.
- Microsoft recommends that every class that has a possibility to be added to a list would implement **IEquatable** interface.

IEquatable

```
class Equatable : IEquatable<Equatable>
{
    public string Property { get; set; }

    public bool Equals(Equatable other)
    {
        return other != null && other.Property == Property;
    }
}
```

- If **IEquatable<>** would be removed – Contains method would not work.
- See **iequatable.ipynb**

IEnumerable

- Allows to iterate (e.g. using **foreach**) through collection
- Has simple and generic version:

```
ArrayList list = new ArrayList();

list.Add("1");
list.Add(2);
list.Add("3");

foreach (object s in list)
{
    Console.WriteLine(s);
}
```

```
List<string> listOfStrings = new List<string>();

listOfStrings.Add("one");
listOfStrings.Add("two");
listOfStrings.Add("three");

foreach (string s in listOfStrings)
{
    Console.WriteLine(s);
}
```

IEnumerable

- Has method **GetEnumerator**, which returns an object, which implements an interface **IEnumerator**.
- **IEnumerator** has:
  - **Current** property, which returns current object from the list
  - **MoveNext** method, which moves enumerator one position forward.
  - **Reset** which moves enumerator to the initial position.
  - **Dispose** (only *generics*) – inherited from IDisposable.
- See **ienumerable.ipynb**

IEnumerable

- Can be simplified with **yield**. **See ienumerable.ipynb**

```
static int SimpleReturn()
{
    return 1;
    return 2;
    return 3;
}

static void Main(string[] args)
{
    Console.WriteLine(SimpleReturn());
    Console.WriteLine(SimpleReturn());
    Console.WriteLine(SimpleReturn());
}
```

```
static IEnumerable<int> YieldReturn()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
static void Main(string[] args)
{
    foreach (int i in YieldReturn())
    {
        Console.WriteLine(i);
    }
}
```

- Must:
  - Return **IEnumerable** type
  - Be called from iteration loop(e.g **foreach**)

ICloneable

- From JAVA lectures: new object copy creations when object is same type as a type that it is being cloned from and has same state.
- Possible:
  - Shallow cloning
  - Deep cloning
- C#: class that implements **ICloneable** interface must implement **Clone** method.
  - Returns cloned object (seriously, object type)

ICloneable

- Deep vs shallow (see **icloneable.ipynb**)

```csharp
// Shallow cloning means we only root level members
public class ShallowCloneable : ICloneable
{
    public string PropertyA { get; set; }
    public string PropertyB { get; set; }

    public object Clone()
    {
        // Or simply
        // return this.MemberwiseClone();

        return new ShallowCloneable
        {
            PropertyA = PropertyA,
            PropertyB = PropertyB,
        };
    }
}
```

```csharp
// Deep clone means we clone members recursively
class DeepCloneable : ICloneable
{
    public string PropertyA { get; set; }
    public Cloneable PropertyB { get; set; }

    public object Clone()
    {
        return new DeepCloneable
        {
            PropertyA = PropertyA,
            PropertyB = (Cloneable)PropertyB?.Clone(),
        };
    }
}
```

ICloneable

- Since **Clone** method returns **object** type object, then whoever called Clone method has to take care of casting returned object to required type.
- Implementation is hidden (deep vs shallow):
  - Microsoft does not recommend to implement Icloneable for exposed APIs, because consumers will not know how your Clone method will behave.
  - More: MSDN ICloneable Interface.

Other popular .NET interfaces

- **IQueryable** (or IQueryProvider): allows to form queries for datasources, that are *queryable*.
- **INotifyPropertyChange**: is used to display data in WPF, Windows Forms and Silverlight applications.
- **IEqualityComparer** (similar to **IEquatable**)
- **IList** and **ICollection**: for collections
- **IDictionary**: for collections, in which you can search using key/value principle.
- **ISerializable** – allows for an object to control how it is being serialized/deserialized.
- **IFormatter** / **IFormatProvider** – used for formatting.

Literature for reading

- A must: C# in depth. Why Properties Matter (online)
- Types (C# Programming Guide). MSDN
- MCSD certification toolkit:
  - 3rd chapter second side
  - 4th chapter. Converting between types.
  - MSDN: When to Use Generic Collections
- More: MSDN - When to Use Generic Collections

Literature for own reading

- MCSD certification toolkit:
  - 5th chapter until "Managing object lifecycle"
- MSDN
- MSDN: Boxing and Unboxing (C# Programming Guide)
- On you own: IEnumerable and IEnumerator
  - How simple and generic version are different?

Next time

- Software system construction.
- Key goals and challenges.
- Business needs analysis.
- Software system modification and maintenance (introduction)

Questions

?