# IOP, Interface Oriented Programming

an introduction

## > Greetings!

Hi! I'm Héctor, father of the upcoming PHATASS Studios and programmer since I was 16. Over the last couple of years I've been designing and refining a design paradigm I've come to name **"Interface Oriented Programming"**, or **IOP.** It's a set of ideas and parameters designed around enforcing a very clean code architecture.

**Now I'd like to share it with you.**

While the vast majority of my experiments have been in C#, the ideas contained herein should apply to any interface-capable object-oriented language. My favourite candidate would be TypeScript.

**A note on examples:** Examples are based on C# but the general ideas herein should apply to any interface-apt language.

**> Greetings!** 1

**> What's an interface? The strengths of not knowing too much.** 2

**> IOP's core: you ONLY know about interfaces.** 3

**> The evil temptation of Public class members** 4

**> Constrain yourself to gain freedom: Analize with IOP, implement however you please** 4

**> Interface Inheritance** 5

**> Polymorphism through Interfaces** 6

**> Generic types are your friend, generic constraints are your savior.** 6

**> Interface Implicit vs Explicit implementation (Softcore vs Hardcore IOP)** 8

**> IOP: Putting the LID on SOLID Programming** 11

**> Recap** 12

**> Wrap-up** 12

## > What's an interface? The strengths of not knowing too much.

Simply put, an **interface is a contract**: When an object implements an interface it agrees to do what the interface says it must do.
It can be requested whatever the interface says it can be requested, and when requested something through an interface it must do **exactly that and only that** as defined by the interface, no side effects.

E.G.: For a car to be considered a proper vehicle, it must fulfill all of a vehicle's capabilities: you can put things and/or people in it, and move around.

```
public interface IVehicle
{
        //We define the type and accessibility of a Contents property
        //Offers the client access to the vehicle cargo/passengers
        IList<IThing> Contents { get; } //Read-only

        //Method with a defined return type and parameter list
        //we can make the vehicle go somewhere - taking its cargo with it
        void TravelTo (IPlace destination);
}

//Both Car and Bike implement IVehicle
public class Car : IVehicle {/*...*/}
public class Bike : IVehicle {/*...*/}
```

Notice how **interfaces** themselves **only refer** to other **interfaces**, **never classes**.

Obviously a car fits more people and has a spacious trunk, whereas a bike fits only you and maybe a small basket.
Regardless of those petty differences in **implementation** details, your car and your bike perfectly **fulfil** the **IVehicle contract**, the very same way a spaceship does. They both have contents and can go places taking their content with them.

Using a proper naming convention is capital, as usual. I can't stress enough how important it is to distinguish interfaces by name - I prefer prepending an "I" to represent it is an **I**nterface. This way, if you somewhere mistakenly refer to something that's **NOT** an interface you can quickly notice and fix it.

**Interfaces allow us to easily separate logical structure from implementation details.**

## > IOP's core: you ONLY know about interfaces.

Interface Oriented Programming is an **interface-first** analysis and design paradigm that revolves around **very heavy** use of **interfaces**. The core Philosophy boils down to a single phrase:

**You think, analyze, and design ONLY with interfaces.**

**Only afterwards** will you program concrete components that **implement** said **interfaces**. This offers **GREAT flexibility**.

Imagine you wake up in the morning and walk up to your parking lot expecting to climb into your **car** and drive to work, as usual. You'll be in a lot of trouble if that morning your daughter took your car, leaving instead your **bike** for you to go to work.

You just can't climb on top of a **bike**, grab the wheel, step on the gas, and drive as if it was the **car** you expected.

```
//This breaks…
Car myRide = new Bike();

//...but it's good that it breaks because this wouldn't work either
myRide.gear = 1;              //Your city bike may or may not have a gear plate
myRide.gasPedal = true;       //But certainly it doesn't have a gas pedal
```

If, on the other hand, you wake up, go to the **IVehicle** in your parking lot, and use it to travel to your workplace, your routine will work without a hitch, regardless of it being with a car or with a bike.

```
IVehicle myRide = garage.GetVehicle();

myRide.Contents.Add(myThings);
myRide.Contents.Add(me);

myRide.TravelTo(workplace);
```

Put things in vehicle. Put myself in vehicle. Go to work. **Simple** and functional.

Heck, worst-case-scenario, if there is **nothing** in your parking lot, you could even consider **yourself** a IVehicle (you can carry things in your pockets, and walk to places), use yourself as a fall-back mechanism, and just TravelTo() your workplace using your natural walking capabilities, no fuss, no muss.

**By completely removing our knowledge of implementation details and replacing EVERYTHING with interfaces, we make the process a lot more robust AND simpler.**

## > The evil temptation of Public class members

The **most important** rule in POI is the one that, not only single-handedly enables proper separation of responsibilities across your project, also ensures your analysis schemes are simple, straightforward, and easy to understand:

**No object must know anything about the implementation details of another object.**

Object meaning any concrete implementation of a class/struct/whatever.
That means you can never access the members of another class or struct, and in fact:

**Any piece of code should only know about itself, and about required interfaces.**

The easiest and safest way to perfectly achieve this is as simple as:

**No object should expose any non-interface public member.**

This way you can completely guarantee your code only flows through appropriately defined and analyzed interfaces. Absolutely no stealth code accessing things that *maybe* it should not.

*But why do I need interfaces for that? I already program this way by properly using public/private memberships across my class hierarchy!*

Of course you do! I haven't invented anything you couldn't do before. But **IOP** lets you analyze your codebase in a way that **forces** you to properly **insulate responsibilities** before you even begin to analyze and design your classes.

**Think only with interfaces.**

## > Constrain yourself to gain freedom: Analize with IOP, implement however you please

Note how most included examples forgo any implementation details, defining **only interfaces** and their **relationships**.
This is intentional. Try to notice how entirely **ignoring implementation** details and **focusing** on the **logical relationships** between abstract designs (**interfaces**) makes every problem way **faster** and **easier** to grasp.

This is why **Interface Oriented Programming** or **IOP** is a programming paradigm **focused** exclusively in the **analysis** of the **relationships** between **objects.**

**IOP** offers absolutely **no** advice on **implementation** decisions, and in fact strongly advises to entirely **ignore implementation** details while designing object relationships.

**When planning, focus on *what* you need to do, NOT on *how* to do it.**

## > Interface Inheritance

Interfaces can inherit from as many other interfaces as they want. This is actually referred to as an interface *implementing* another interface

**Interface implementation trees** are way more **powerful and flexible** than Class inheritance trees as far as **codebase analysis and design** is concerned:

- An interface can **inherit from multiple others**, without concerns for fragile resolution ordering.
- Interfaces let you **properly separate** the various aspects of a specific problem/resolution set in order to **re-use** them.
- Interfaces **force you to think abstractly** about their relationships, lacking implementation. This leads to a **way more maintainable** codebase.

E.G.: in our example, vehicles have a content.
However, every consideration about keeping and exposing a modifiable bunch of contents, is completely independent from the notion of a vehicle.

The car's **trunk** is **independent** from its **engine**.

We can and should **separate those responsibilities** across multiple interfaces:

```
//IContainer inherits (implements) system ICollection interface for even more flexibility
public interface IContainer : ICollection
{
 IList<IThing> Contents { get; }
}

//IMovable defines any form of object that can move by itself
public interface IMovable
{
 void MoveTo (IPlace destination); //Renamed TravelTo as MoveTo so it more
appropriately fits any form of movable item
}

//IVehicle definition is now way neater and clearer
public interface IVehicle : IMoveable, IContainer
{}
```

**We must always properly separate the responsibilities of our interfaces, and what problems and sub-problems they represent.**

## > Polymorphism through Interfaces

Since **interfaces represent a contract**, when communicating exclusively through interfaces it is tremendously **easy** and natural to **replace an object** that **fulfills a contract** with **any other object** as long as it **fulfills the same contract.**

In our initial example we could use our **bike** as a perfect **replacement** for our **car** solely thanks to the use of **interfaces**:

- Car **fulfills** the contract **IVehicle**
- Bike **fulfills** the contract **IVehicle**
- We can **use any IVehicle**, instead of just a specific implementation.

In this manner we **polymorph** specific **implementations** into a common **interface** we can refer to without knowledge about implementation. Thanks to this, it is trivial to change the vehicle we use at any time.

Another form of interface polymorphism is **casting** an object or interface to **a specific interface** representing a **specific responsibility.**

E.G.: Before a big trip, we may want to pack our things into the car the day before.
We generally consider our **car** an **IVehicle**, and we could store things into it directly. But for the sake of just packing our things, we should treat our **car** as an **IContainer** for simplicity:

- Since IVehicle inherits or implements IContainer, **every IVehicle is an IContainer**.
- **IContainer** represents the **specific problem** of **storing** items. This means it's more adequate to expect to **store our things** into an **IContainer** rather than an IVehicle.
- This way, we could then easily **swap** the **target IContainer** from our car to any **other** IContainer.

Thanks to this, we could use a suitcase as a proxy IContainer, put our things into it, then just put our suitcase into the car for the trip. Sounds way neater than just tossing our underwear all over the back seats. It's also easier to unpack.

We could even **re-use** the very same steps we took to store our things anywhere else.
We might use a box in the basement as an IContainer to protect our stuff while we paint the house.

**When referencing an interface, try to pick the one within the interface tree most adequate to the problem at hand.**