Lec 16: Interfaces

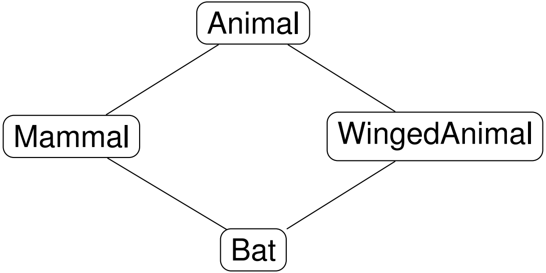
CS220: Programming Principles

Sang Kil Cha

Multi-Inheritance

Multiple Class Inheritance

There are cases where we want to create an object inherited from multiple parents.



In F#?

```
[<AbstractClass>]
type Animal () =
  abstract Breathe: unit -> unit

[<AbstractClass>]
type Mammal () =
  inherit Animal ()
  abstract MakeSound: unit -> unit

[<AbstractClass>]
type WingedAnimal () =
  inherit Animal ()
  abstract Fly: unit -> unit
```

```
type Bat () =
  inherit Mammal ()
  inherit WingedAnimal ()
  override __.Breathe () = ()
  override __.MakeSound () = ()
  override __.Fly () = ()
```

Can't Compile?

Types cannot inherit from multiple concrete types.

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don’t accept late responses.

The Diamond Problem

Suppose both `Mammal` and `WingedAnimal` implemented `Breathe`:

```
[<AbstractClass>]
type Animal () =
  abstract Breathe: unit -> unit

[<AbstractClass>]
type Mammal () =
  inherit Animal ()
  abstract MakeSound: unit -> unit
  override __.Breathe () = printfn "Mammal breathe"

[<AbstractClass>]
type WingedAnimal () =
  inherit Animal ()
  abstract Fly: unit -> unit
  override __.Breathe () = printfn "WingedAnimal breathe"
```

The Diamond Problem (cont'd)

If `Bat` can inherit from both classes, which `Breathe` function should we invoke?

```
(Bat ()).Breathe ().
```

Can we avoid the diamond problem?

Does F# Have the Diamond Problem?

No. Because multiple inheritance is not allowed in F#. But what if we need multiple inheritance?

Interfaces

Interfaces

Interfaces are similar to abstract classes, but it is possible to create a class extended from ***multiple*** interfaces.

How can we avoid the diamond problem then?

Interface?

Interface is a point where two systems, subjects, organizations, etc. meet and interact.

For example,

1. FSI (FSharp Interface) file provides an ***interface***.
2. API (Application Programming Interface) provides an ***interface***.

F#'s Interface

Does not have a ***constructor***: we cannot instantiate it! It purely provides an ***interface***[1].

Abstract Class

```
[<AbstractClass>]
type MyAbstractClass () =
  abstract Foo: int -> int
  // Can have a concrete member.
  member _.Bar = 42
```

Interface

```
type IMyInterface =
  abstract Member Foo: int -> int
  // This is not allowed.
  // member _.Bar = 42
```

[1] We often use a prefix 'I' for interfaces.

Implementing Interfaces

We say we “implement” an interface (instead of saying “inherit from”).

```
type MyClass () =
  interface IMyInterface with
    member __.Foo n = n + 1
```

Implementing Multiple Interfaces

```
type IMammal =
  abstract MakeSound: unit -> unit

type IWingedAnimal =
  abstract Fly: unit -> unit

type Bat () =
  interface IMammal with
    member __.MakeSound () = printfn "sound"

  interface IWingedAnimal with
    member __.Fly () = printfn "I'm flying"

  member __.BatSpecificMember () = ()
```

Interfaces in Practice

Example: Set of Student Objects

Suppose we have the following student object definition.

```
type Student (id) =
  member __.ID = id
```

Can we create a set of students using the above object?

The Student type does not support the comparison constraint.

Comparison Type Constraint?

What's the type of a comparison operator?

```
val (>):  'a -> 'a -> bool when 'a:  comparison
```

If the type implements the `IComparable` interface then it can be compared.

IComparable Interface

Has a single abstract method: `CompareTo`[2].

```
member IComparable.CompareTo:  obj -> int
```

The return value indicates the relative order of the objects being compared. The return value has these meanings:

1. (Less than zero): this instance precedes `obj` in the order.
2. (Zero): this instance occurs in the same order as `obj`.
3. (Greater than zero): this instance follows `obj` in the order.

---

[2] https://docs.microsoft.com/en-us/dotnet/api/system.icomparable.compareto

Make Student Object Comparable

```
type Student (id) =
  member __.ID = id

  interface IComparable with
    member __.CompareTo obj =
      match obj with
      | :? Student as s -> compare s.ID __.ID
      | _ -> failwith "Can't compare"
```

Warnings after Implementing `CompareTo`

`warning FS0343: The type ‘Student’ implements ‘System.IComparable’ explicitly but provides no corresponding override for ‘Object.Equals’. An implementation of ‘Object.Equals’ has been` ***`automatically provided`***`, implemented via ‘System.IComparable’. Consider implementing the override ‘Object.Equals’ explicitly.`

Equality?

```
type Student(name) =
  member _.Name with get(): string = name

let a = Student "Alice"
let b = Student "Bob"
let c = Student "Alice" // is this same as a?
a = c // true or false?
```

GetHashCode?

Every object has this method, which is a hash function used to map data of arbitrary size to a fixed-size value. This is particularly useful when we use our object as a key in a hash table.

It is important to make sure that if two objects are equal, then their hash values must be equal as well.

Full Implementation

```
open System

type Student (id) =
  member __.ID = id
  override __.Equals obj =
    match obj with
    | :? Student as s -> s.ID = __.ID
    | _ -> false
  override __.GetHashCode () = hash __.ID
  interface IComparable with
    member __.CompareTo obj =
      match obj with
      | :? Student as s -> compare s.ID __.ID
      | _ -> failwith "Can't compare"
```

F#'s Functional Data Types are Comparable

Records, Discriminated Unions, Tuples, etc. use structural equality by default. It uses a lexicographic left-to-right comparison. This is naturally possible because functional data types are immutable and ***transparent***.

In-Class Activity #16

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

Problem

Modify the `isCircleLargerThanRectangle` function to check if the given Circle object is larger than the given Rectangle object.

OOP Design Guidelines

How to Design a Program Using OOP?

It is not always clear how to properly design a program using OOP. How do we design class hierarchies? How do we decide which class should inherit from which class? When do we need to use interfaces?

There is ***no definitive answer*** to these questions, but there are some general design guidelines that we can follow.

SOLID Principles

1. **S**ingle Responsibility Principle
2. **O**pen/Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

(1) Single Responsibility Principle

A class should have only one reason to change. In other words, a class should have only one job.

For example, a class that is responsible for both reading and writing to a file violates this principle.

SRP Violation Example

Suppose we have a class that is responsible for representing invoices.

```
type Invoice () =
  member __.InvoiceNumber = // ...
  member __.IssueDate = // ...
  member __.Amount = // ...
  member __.Customer = // ...
  member __.Save () = // save this invoice to DB
```

This class violates the SRP because it has two responsibilities: representing an invoice and saving it to the database.

SRP Example (Refactored)

```
// Represents an invoice in a transparent way.
type Invoice = {
  InvoiceNumber: int
  IssueDate: DateTime
  Amount: int
  Customer: Customer
}

// Responsible for saving invoices to the database.
type InvoiceRepository () =
  member __.Save (invoice: Invoice) = // save this invoice to DB
```

N.B. Understanding a way to mix functional and OOP design is important.

(2) Open/Closed Principle

A class should be open for extension but closed for modification.

For example, we should be able to add new functionality to a class without changing its source code.

OCP Violation Example

```
type ShapeType =
  | Circle of radius: float
  | Rectangle of width: float * height: float

let area = function
  | Circle r -> Math.PI * r * r
  | Rectangle (w, h) -> w * h
```

This example violates the OCP because we need to modify the `area` function whenever we add a new shape.

OCP Example (Refactored)

```
type Shape =
  abstract Area: unit -> float

type Circle (radius) =
  member __.Area () = Math.PI * radius * radius

type Rectangle (width, height) =
  member __.Area () = width * height

let area (shape: Shape) = shape.Area ()
```

We can add new shapes without modifying the `area` function.

(3) Liskov Substitution Principle

If `S` is a subtype of `T`, then objects of type `T` may be replaced with objects of type `S` without altering any of the desirable properties of the program.

For example, one may consider a square as a subtype of a rectangle, but a square is not a rectangle.

(4) Interface Segregation Principle

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

For example, a class that implements an interface with many methods that it does not use violates this principle.

Try to create small, cohesive interfaces!

ISP Violation Example

```
type IVehicle =
  abstract Run: unit -> unit
  abstract Fly: unit -> unit

type Aircraft () =
  interface IVehicle with
    member __.Run () = ()
    member __.Fly () = ()

type Car () =
  interface IVehicle with
    member __.Run () = ()
    member __.Fly () = failwith "Can't fly"
```

The `Car` class violates the ISP because it is forced to implement the `Fly` method.

ISP Example (Refactored)

```
type IRunnable =
  abstract Run: unit -> unit

type IFlyable =
  abstract Fly: unit -> unit

type Aircraft () =
  interface IRunnable with
    member __.Run () = ()

  interface IFlyable with
    member __.Fly () = ()

type Car () =
  interface IRunnable with
    member __.Run () = ()
```

(5) Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

For example, a class that depends on a concrete implementation of another class violates this principle.

DIP Violation Example

```
type DBService () = // low-level
  member __.Save (data: string) = // save data to DB

type Logger (db: DBService) = // high-level
  member __.Log (message: string) =
    db.Save message
```

The `Logger` class violates the DIP because it depends on a concrete implementation of the `DBService` class. By modifying the `DBService` class, we may need to modify the `Logger` class as well.

DIP Example (Refactored)

```
type IDBService =
  abstract Save: string -> unit

type DBService () =
  interface IDBService with
    member __.Save data = // save data to DB

type Logger (db: IDBService) =
  member __.Log (message: string) =
    db.Save message
```

The `Logger` class now depends on an abstraction instead of a concrete implementation.

Conclusion

1. Interfaces provide a way to avoid the diamond problem.
2. There are some design principles to follow to design a program using OOP, although they do ***not*** provide definitive answers to SW design.
3. Always make your code easy to understand and maintain.
4. Mix functional and OOP design to get the best of both worlds.

Further Readings

- `https://fsharpforfunandprofit.com/posts/interfaces/`
- Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin

Question?