Lec 14: Object-Oriented Programming

CS220: Programming Principles

Sang Kil Cha

Object-Oriented Programming

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.

OOP (Object-Oriented Programming)

OOP is a crucial programming paradigm in modern software development, especially when building a large and complex system. It considers a program as a collection of objects that interact with each other, which is similar to how we view the world.

OOP Example: Car

Car consists of several parts: engine, wheels, and etc. Each part can be considered as an object, and they interact with each other to make the car work.

OOP Advantages

By separating a program into objects, we can:

1. Intuitively model the real world.
2. Modularize the program: each object can be developed independently.
3. Easily identify where to fix a bug or add a new feature.
   (Only if you have well designed objects).

Why Learn OOP with F#?

Is F# a functional programming language? Yes, and no. It is indeed a hybrid language, and we often call it “functional-first” language.

When developing a large system, I recommend you write your code in an OOP manner with functional programming principles in mind. Follow F#’s philosophy of “functional-first” programming.

What is an Object?

An object is a data structure encapsulating some internal states, named ***properties***, and offering access to the states to users with a collection of ***methods***[1].

[1] In OOP, we call a function attached to an object a ***method***.

Object vs. Class

An object is an instance of a class.

Instantiation of a class is the creation of a new instance of the class.

Example

Consider a ***Car*** class. We can define several operations (***methods***) for a car:

1. Start.
2. Stop.
3. Accelerate.
4. ...

A car also has its own states (***properties***):

1. Fuel amount.
2. Current speed.
3. ...

OOP Key Concepts

There are several key concepts to understand OOP:

1. Encapsulation.
2. Inheritance.
3. Polymorphism.

Encapsulation

Encapsulation

Bundle data (properties) with functions (methods), while making the data hidden.

Functional Data Abstraction

```
type BankAccount = {
  Balance: int
}
module BankAccount =
  let create () = { Balance = 0 }
  let getBalance account = account.Balance
  let deposit account amount =
    { account with Balance = account.Balance + amount }
```

In OOP, we want a data object to have its own ***state***, and we want to have the state and functions altogether.

Records with Mutable States

```
type BankAccount = {
  mutable Balance: int // in Won
  GetBalance: unit -> int // Function encapsulated
}
```

Data accesses are ***transparent***: we can always directly access the `Balance` field.
Can we hide the data?

Another Attempt

```
type BankAccount = private {
  mutable Balance: int // in Won
  GetBalance: unit -> int // Function encapsulated
}
let myAccount = { Balance = 0; GetBalance = (* ? *) }
```

Two problems remain:

1. GetBalance function is also not accessible! We want to expose only the functions (***methods***).
2. It is not straightforward how to instantiate a `BankAccount` record, because `GetBalance` function cannot directly access the current balance of the instance.

Using a Closure

```
type BankAccount = {
  GetBalance: unit -> int
  Deposit: int -> unit
}
module BankAccount =
  let create () =
    let mutable balance = 0
    { GetBalance = fun () -> balance
      Deposit = fun m -> balance <- balance + m }
```

The function is well attached (encapsulated) to the data object, but `create`?

Class Definition in F#

```
type BankAccount () =
  let mutable balance = 0
  member __.GetBalance () = balance
  member __.Deposit amount =
    balance <- balance + amount
```

1. `__` is a self identifier, referencing the class instance itself, and can be used with other names. Historically, `__`, `this`, or `self` is preferred.
2. Member functions can be called as usual: `instance.GetBalance ()`

Class Signature

```
type BankAccount =
  class
    new : unit -> BankAccount
    member Deposit : amount:int -> unit
    member GetBalance : unit -> int
  end
```

Primary Constructor

The previous class definition automatically creates a primary constructor, which is a function that creates the object instance. We can create an object instance by:

```
let x = BankAccount ().
```

Or, we can use the `new` keyword explicitly to call the constructor:

```
let x = new BankAccount ().
```

But, it is recommended not to use the `new` keyword for simplicity.

Constructor with Parameters

Constructors can take in parameters.

```
type Student(firstName: string, lastName: string) =
  member __.FirstName = firstName
  member __.LastName = lastName
```

Attaching Values to Objects

Properties are members that represent values associated with an object[2].

```
type MyObject () =
  let mutable myValue = 42
  member __.MyReadOnlyProperty = myValue
  member __.MyWriteOnlyProperty with set(v) = myValue <- v
  member __.MyProperty
    with get() = myValue
    and set(v) = myValue <- v
```

[2] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/members/properties

Automatically Implemented Properties

We always love simplicity.

```
type MyObject () =
  member val MyProperty = 42 with get, set
```

Summary: Encapsulation

Encapsulation is a way of bundling the data with the methods that operate on the data, while hiding the data from direct access.

> Encapsulation is an OOP's way of achieving ***data abstraction***.

Transparency vs. Encapsulation

In functional programming, we often prefer ***transparency*** over ***encapsulation*** because every value is immutable and directly accessing the value is inherently safe.

In OOP, we prefer ***encapsulation*** because object states are often mutable and we don't want users to directly access/modify the states.

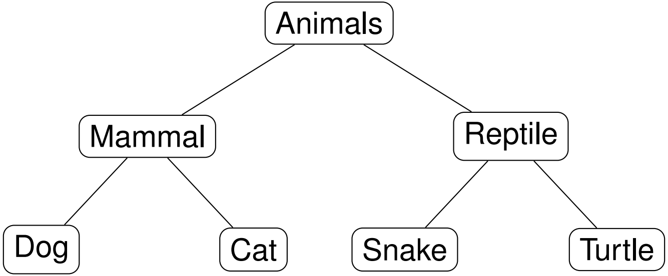
Inheritance

Code Reuse

Can we make new objects by combining existing objects? Thereby, we do not need to write similar code over and over again.

Classifying Objects

Suppose we are writing a program dealing with animals: cat, dog, etc.



Abstract Class

An abstract class is a class that ***cannot be instantiated***, but it represents common functionality of a diverse set of object types.

```
[<AbstractClass>]
type Animal () =
  let mutable x = 0
  let mutable y = 0
  abstract Breathe: unit -> unit // Abstract method
  member __.Move dx dy = // Normal method
    x <- x + dx
    y <- y + dy
```

Inheritance

A class can inherit from an existing class (both regular and abstract class).

```
[<AbstractClass>]
type Mammal () =
  inherit Animal () // Inherit the functionalities of Animal
  abstract MakeSound: unit -> unit

type Dog () =
  inherit Mammal ()
  member __.Run () = printfn "Dog runs"
```

Inheritance

A class can inherit from an existing class (both regular and abstract class).

```
[<AbstractClass>]
type Mammal () =
  inherit Animal () // Inherit the functionalities of Animal
  abstract MakeSound: unit -> unit

type Dog () =
  inherit Mammal ()
  member __.Run () = printfn "Dog runs"
```

No implementation was given for '`abstract member Mammal.MakeSound : unit -> unit`'

Inheritance (cont'd)

We need to provide specific implementation for abstract members[3]! This is often called "***method overriding***".

```
override __.MakeSound () = ...
```

[3] N.B. Abstract functions are often referred to as ***virtual*** methods in OOP.

Inherited Object Instances

Dogs can move, and cats also. And they share the same code: `Move` in `Animal`.

> Can we achieve the same with records?

Why Abstract Class?

A class can be inherited from a normal class too. What's the difference? Why use `abstract` members?

Polymorphism

Polymorphism

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

- From Wikipedia

Subtype Polymorphism

In OOP, we are mostly interested in ***subtype polymorphism***.

Subtype Polymorphism

In OOP, we are mostly interested in ***subtype polymorphism***.

A is inherited from B. Then we say A is a ***subtype*** of B. For example, Dog is a ***subtype*** of Animal, and Animal is a ***supertype*** of Dog.

Subtype polymorphism allows us to create a function that takes in a supertype, but can operate with subtype values.

Subtype Polymorphism Example

```
let speak (m: Mammal) =
  m.MakeSound ()

speak (Dog ()) // What will happen?
speak (Cat ()) // What will happen?
```

Polymorphic List

Can we create a list of Dog or Cat?

```
type DogOrCat =
  | D of Dog
  | C of Cat

[ D (Dog ()); C (Cat ()) ]
// OR
[ Dog () :> Animal; Cat () :> Animal ]
```

The `:>` operator upcast a type to its supertype.

In-Class Activity #14

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 `sumAnimalAges` function to compute the sum of ages of all animals in the list.

Conclusion

Further Readings

- https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/classes
- https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/members/methods

Question?