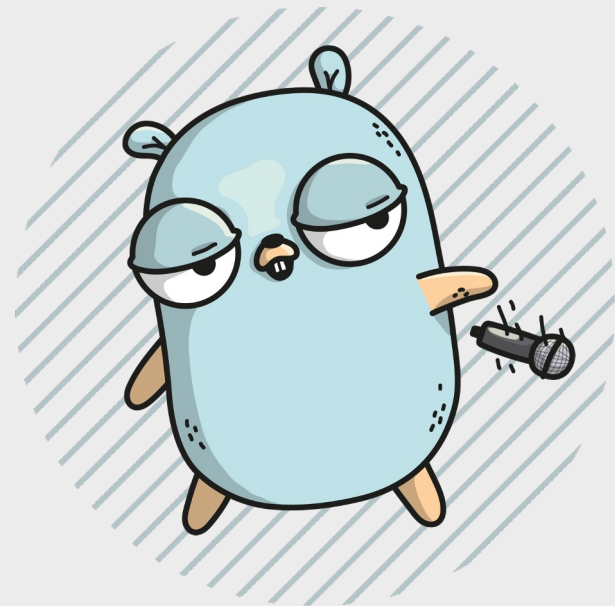


Object Oriented Go

객체지향 언어로서의 Go

정진우



Experience in Go

- 메인 프로젝트 & 사이드 프로젝트 모두 **Go** 언어 기반
- 다른 언어에서의 경험이 더 많고, **OOP**도 **Java**로 학습



Index



- Object Oriented Programming
- Message Passing
- Encapsulation
- Inheritance & Composition
- Abstraction & Polymorphism



Object Oriented Programming



OOP

객체지향 패러다임은 지식을 추상화하고 추상화된 지식을
객체 안에 캡슐화함으로써 실세계 문제에 내재된 복잡성을 관리하려고 한다.
객체를 발견하고 창조하는 것은 지식과 행동을 구조화하는 문제다.

Wirfs-Brock, R. et al., *Designing Object-Oriented Software*, 1990

객체지향이란 시스템을 상호작용하는 자율적인 객체들의 공동체로 바라보고
객체를 이용해 시스템을 분할하는 방법이다.

조영호, *객체지향의 사실과 오해*, 2015

OOP



객체에 메시지를 전달하는 방식으로 비즈니스 로직을 구현하는 것


- OOP의 핵심은 Message Passing
- 프로그램 전체에 OOP를 강제하는 것이 아니라
특정 비즈니스 로직을 객체지향적으로 구현하는 것



Message Passing



Message Passing



객체지향의 세계에서 객체들이 서로 협력하기 위해 사용할 수 있는 유일한 방법은 메시지를 전송하는 것이다. 다른 객체와 협력할 필요가 있는 객체는 메시지를 전송하고, 메시지를 수신한 객체는 미리 정의된 방법에 따라 수신된 메시지를 처리한다.

조영호, *객체지향의 사실과 오해*, 2015

데이터 중심적인 접근

1. 회원의 이름을 수정할 때 기존 이름과 동일한 이름으로는 수정할 수 없다.
2. 회원의 이름은 **2글자 ~ 10글자** 사이여야 한다.

```
package application
```

```
func ChangeMemberName(member *domain.Member, name string) {  
    // member의 name 필드에 접근하여 값 비교  
    if member.Name == name {  
        panic(errors.New("is same name as before"))  
    }  
    nameLength := len(name)  
    if nameLength < 2 || nameLength > 10 {  
        panic(errors.New("name must be 2 to 10 characters"))  
    }  
    // member의 name 필드에 접근하여 값 수정  
    member.Name = name  
}
```

```
package domain
```

```
type Member struct {  
    Name string  
}
```

객체의 책임으로

Member 구조체에 ChangeName 메서드(=수단, 행동) 구현

=> Name에 대한 접근 및 수정 + 입력값에 대한 검증을 Member의 책임화

```
package domain

type Member struct {
    Name string
}

func (member *Member) ChangeName(name string) error {
    if member.Name == name {
        return errors.New("is same name as before")
    }
    nameLength := len(name)
    if nameLength < 2 || nameLength > 10 {
        return errors.New("name must be 2 to 10 characters")
    }
    member.Name = name
    return nil
}
```

Message Passing

```
package application
```

```
func ChangeMemberName(member *domain.Member, name string) {  
    err := member.ChangeName(name)  
    if err != nil {  
        panic(err)  
    }  
}
```

이름을 수정하라는
메시지 전송

```
package domain
```

```
type Member struct {  
    Name string  
}
```

수신한 메시지의
처리 수단(method)

```
func (member *Member) ChangeName(name string) error {  
    if member.Name == name {  
        return errors.New("is same name as before")  
    }  
    nameLength := len(name)  
    if nameLength < 2 || nameLength > 10 {  
        return errors.New("name must be 2 to 10 characters")  
    }  
    member.Name = name  
    return nil  
}
```

Message Passing

≈ Tell, Don't Ask

≈ Method Calling?

Go struct와 객체

Go에는 class 문법이 없다.

그러나 Go struct의 인스턴스는 객체로써 활용 가능하다.

객체의 기본 성질

- 상태(state)를 지닌다.
- 행동(method)을 통해 상태를 변경할 수 있다.

```
package domain
```

```
type Member struct {  
    Name string  
}
```

```
func (member *Member) ChangeName(name string) error {  
    if member.Name == name {  
        return errors.New("is same name as before")  
    }  
    nameLength := len(name)  
    if nameLength < 2 || nameLength > 10 {  
        return errors.New("name must be 2 to 10 characters")  
    }  
    member.Name = name  
    return nil  
}
```



Encapsulation



Encapsulation

구현을 외부로부터 감추는 것을 캡슐화라고 한다... 객체는 상태와 행위를 한데 묶은 후 외부에서 반드시 접근해야만 하는 행위를 골라 공용 인터페이스를 통해 노출한다... 객체는 외부의 객체가 자신의 내부 상태를 직접 관찰하거나 제어할 수 없도록 막기 위해 의사소통 가능한 특별한 경로(**공용 인터페이스**)만 외부에 노출한다.

조영호, 객체지향의 사실과 오해, 2015

Encapsulation

Java: 접근제어자를 통한 인스턴스/클래스/패키지 수준에서의 캡슐화 제공
Go: Exported/unexported 구분을 통한 패키지 수준에서의 캡슐화만 제공

```
package example

const ExportedConst = 1
const unexportedConst = 2

var ExportedVar = 1
var unexportedVar = 2

func ExportedFunc() {}
func unexportedFunc() {}

type ExportedStruct struct {
    ExportedField int
    unexportedField int
}

func (m *ExportedStruct) ExportedMethod() {}
func (m *ExportedStruct) unexportedMethod() {}

type unexportedStruct struct {}
```

```
println(example.ExportedConst)
println(example.unexportedConst) // Unexported constant

println(example.ExportedVar)
println(example.unexportedVar) // Unexported variable

example.ExportedFunc()
example.unexportedFunc() // Unexported function

instance := example.ExportedStruct{
    ExportedField: 10,
    unexportedField: 20, // Unexported field
}
instance.ExportedMethod()
instance.unexportedMethod() // Unexported method

example.unexportedStruct{} // Unexported type
```

문제1. 내부 구현 노출

Member 구조체의 Name 필드는 Exported = 내부 구현이 그대로 노출

```
package domain

type Member struct {
    Name string // Exported
}

func (member *Member) ChangeName(name string) error {
    if member.Name == name {
        return errors.New("is same name as before")
    }
    nameLength := len(name)
    if nameLength < 2 || nameLength > 10 {
        return errors.New("name must be 2 to 10 characters")
    }
    member.Name = name
    return nil
}
```


문제1. 내부 구현 노출

내부 구현 노출의 문제점

- **Member** 구조체의 내부 구현을 자유롭게 설정 및 수정 가능
- 객체와 상호작용할 수 있는 수단이 **Message Passing** 이외에 존재
- (필드명 변경, 타입 변경 등) 내부 구현이 수정되었을 때 변경의 여파 통제 불가

```
// 생성 시점의 Name 필드 값 자유롭게 초기화 가능
```

```
member := &domain.Member{Name: ""}
```

```
// ChangeName 메서드 사용할 수도 있음
```

```
_ := member.ChangeName("validName")
```

```
// Name에 대한 직접 수정 허용
```

```
member.Name = "ReallyLongNameOver10Characters"
```

문제1. 내부 구현 노출

Member 구조체의 내부 구현을 자유롭게 설정 및 수정 가능
“회원의 이름은 **2 ~ 10**글자 사이여야 한다” 등의 규칙 강제 불가

```
package application

func CreateMember(name string) error {
    // 생성 시점의 Name 필드 값 자유롭게 초기화 => 2~10글자 보장 불가
    member := &domain.Member{Name: name}
    // ...
}

func UpdateMember(member *domain.Member, name string) error {
    // Name에 대한 직접 수정 허용 => 2~10글자 보장 불가
    member.Name = name
    // ...
}

func ChangeMemberName(member *domain.Member, name string) {
    err := member.ChangeName(name) // Member 내부에서 검증 로직 적용 => 2~10글자 보장
    // ...
}
```

퍼블릭 인터페이스

외부로 제공된 퍼블릭 인터페이스 뒤에 내부 구현을 숨기는 구조

- 내부 구현은 **unexported** (\approx private 필드)
- 퍼블릭 인터페이스는 **exported** (\approx getter/setter 메서드)

```
package domain
```

```
type Member struct {  
    name string // Java Bean 규약의 private 멤버 변수 역할  
}
```

```
func (member *Member) Name() string {  
    return member.name // Java Bean 규약의 GetName 역할  
}  
  
func (member *Member) ChangeName(name string) error {  
    // ...  
    member.name = name // Java Bean 규약의 SetName 역할  
    return nil  
}
```

퍼블릭 인터페이스

unexported 필드에는 직접 접근 불가
=> 외부에 제공된 퍼블릭 인터페이스(Name/ChangeName) 사용 강제

```
package application

func CreateMember(name string) error {
    member := &domain.Member{name: name} // Unexported field 'name' usage
    // ...
}

func UpdateMember(member *domain.Member, name string) error {
    member.name = name // Unexported field 'name' usage
    // ...
}

func ChangeMemberName(member *domain.Member, name string) {
    err := member.ChangeName(name) // Member 내부에서 검증 로직 적용 => 2~10글자 보장
    // ...
}
```

문제 2. Zero Value Struct

Member 구조체는 exported

- 생성 시점에 내부 구현을 자유롭게 지정하는 것은 불가
- 다만, 모든 필드가 **zero value**인 객체는 자유롭게 생성 가능
- 초기화 시점에 **name** 필드 길이 제한 등의 규칙 강제 불가

```
member := &domain.Member{name: name} // Unexported field 'name' usage
```

```
// zero value struct : 모든 필드가 zero value인 구조체 생성 가능
```

```
member := &domain.Member{} // &{name:}
```

```
member = new(domain.Member) // &{name:}
```

```
... // 2~10글자 규칙 위배. 내부 구현이 불완전한 상태
```

```
// 2~10글자 규칙 충족 가능
```

```
_ := member.ChangeName(name)
```

생성자 함수

생성자 함수 **NewMember**를 외부에 제공

- 함수 내부에서 검증 로직을 실행하여 **2~10**글자 규칙 보장

```
package domain

type Member struct {
    name string
}

func NewMember(name string) (*Member, error) {
    if err := validateName(name); err != nil {
        return nil, err
    }
    return &Member{name: name}, nil
}

func (member *Member) Name() string {
    return member.name
}
```

```
func (member *Member) ChangeName(name string) error {
    if member.name == name {
        return errors.New("is same name as before")
    }
    if err := validateName(name); err != nil {
        return err
    }
    member.name = name
    return nil
}

// 검증 로직
func validateName(name string) error {
    nameLength := len(name)
    if nameLength < 2 || nameLength > 10 {
        return errors.New("name must be 2 to 10 characters")
    }
    return nil
}
```

생성자 함수

생성자 함수 **NewMember**를 외부에 제공

- 함수 내부에서 검증 로직을 실행하여 **2~10**글자 규칙 보장

```
import (  
    "testing"  
  
    "github.com/stretchr/testify/assert"  
    ...  
)  
  
func TestNewMember(t *testing.T) {  
    m1, err := domain.NewMember("John") // (*domain.Member, nil) 반환  
    assert.NotNil(t, m1)  
    assert.NoError(t, err)  
  
    m2, err := domain.NewMember("") // (nil, error) 반환  
    assert.Nil(t, m2)  
    assert.Error(t, err, "name must be 2 to 10 characters")  
}
```

문제 2. Zero Value Struct (... again)

Member 구조체는 여전히 exported

- 생성자 함수 이외의 수단으로 Member 구조체 생성 가능
- 생성자 함수 NewMember의 사용 강제 불가

```
func TestNewMember(t *testing.T) {  
    m1, err := domain.NewMember("John") // (*domain.Member, nil) 반환  
    assert.NotNil(t, m1)  
    assert.NoError(t, err)  
  
    m2, err := domain.NewMember("") // (nil, error) 반환  
    assert.Nil(t, m2)  
    assert.Error(t, err, "name must be 2 to 10 characters")  
  
    m3 := &domain.Member{  
        assert.Equal(t, m3.Name(), "")  
    }  
}
```


인터페이스: 추상화 뒤에 내부 구현 캡슐화하기

- member 구조체를 Member 인터페이스의 구현체로 정의
 - 추상화(Member 인터페이스)만 패키지 외부에 노출
- => 외부에는 구체적인 구현체가 아닌 추상적인 인터페이스만 제공

```
package domain
```

```
type Member interface {  
    Name() string  
    ChangeName(string) error  
}
```

```
func NewMember(name string) (Member, error) {  
    if err := validateName(name); err != nil {  
        return nil, err  
    }  
    return &member{name: name}, nil  
}
```

```
type member struct {  
    name string  
}
```

```
func (member *member) Name() string {  
    return member.name  
}
```

```
func (member *member) ChangeName(name string) error {  
    // ...  
    return nil  
}
```

```
func validateName(name string) error {  
    // ...  
}
```

인터페이스: 추상화 뒤에 내부 구현 캡슐화하기

- member 구조체를 Member 인터페이스의 구현체로 정의
 - 추상화(Member 인터페이스)만 패키지 외부에 노출
- => 외부에는 구체적인 구현체가 아닌 추상적인 인터페이스만 제공

```
// Usage of the unexported type 'member'  
m := &domain.member{}
```

```
var m domain.Member  
m, _ = domain.NewMember("John")  
m.Name()  
_ = m.ChangeName("Jake")
```



Inheritance & Composition



배경지식 1. 서브타이핑과 서브클래싱

상속은 서브타이핑(subtyping)과 서브클래싱(subclassing)의 두 가지 용도로 사용될 수 있다. 서브클래스가 슈퍼클래스를 대체할 수 있는 경우 이를 서브타이핑이라고 한다. 서브클래스가 슈퍼클래스를 대체할 수 없는 경우에는 서브클래싱이라고 한다. 서브타이핑은 설계의 유연성이 목표인 반면 서브클래싱은 코드의 중복 제거와 재사용이 목적이다.

조영호, 객체지향의 사실과 오해, 2015

배경지식2. 상속 대신 조합

(코드 재사용을 위해서는) 객체 합성이 클래스 상속보다 더 좋은 방법이다... 객체 합성은 클래스 상속의 대안이다.

Gamma E. et al., *GoF의 디자인 패턴: 재사용성을 지닌 객체지향 소프트웨어의 핵심요소*, 2015

상속을 제대로 활용하기 위해서는 부모 클래스의 내부 구현에 대해 상세하게 알아야하기 때문에 자식클래스와 부모 클래스 사이의 결합도가 높아질 수밖에 없다... 합성은 구현에 의존하지 않는다는 점에서 상속과 다르다. 합성은 내부에 포함되는 객체의 구현이 아닌 퍼블릭 인터페이스에 의존한다. 따라서 합성을 이용하면 객체의 내부 구현이 변경되더라도 영향을 최소화할 수 있기 때문에 변경에 더 안정적인 코드를 얻을 수 있게 된다.

조영호, *오브젝트*, 2019

코드 재사용 목적의 상속

Stack & Queue 클래스는 각자 **extends** 키워드로 **List** 클래스를 상속
=> pushLeft, push, popLeft, pop 등의 퍼블릭 메서드들 상속

```
public class List {  
  
    private int[] elements;  
  
    // 맨 처음에 요소 추가  
    public void pushLeft(int value) { ... }  
  
    // 맨 끝에 요소 추가  
    public void push(int value) { ... }  
  
    // 첫 번째 요소 제거 후 반환  
    public int popLeft() { ... }  
  
    // 마지막 요소 제거 후 반환  
    public int pop() { ... }  
}
```

```
// push & pop으로 선입후출 구현 가능  
public class Stack extends List {}
```

```
// push & popLeft로 선입선출 구현 가능  
public class Queue extends List {}
```

코드 재사용 목적의 상속

Stack: push & pop 메서드만 호출시 선입후출 보장

Queue : push & popLeft 메서드만 호출시 선입선출 보장

```
// 선입후출: push & pop만 사용
Stack stack = new Stack();
stack.push(1); // [1]
stack.push(2); // [1, 2]
stack.push(3); // [1, 2, 3]
stack.pop(); // [1, 2]
stack.push(4); // [1, 2, 4]
stack.pop(); // [1, 2]
stack.pop(); // [1]
stack.pop(); // []
```

```
// 선입선출: push & popLeft만 사용
Queue queue = new Queue();
queue.push(1); // [1]
queue.push(2); // [1, 2]
queue.push(3); // [1, 2, 3]
queue.popLeft(); // [2, 3]
queue.push(4); // [2, 3, 4]
queue.popLeft(); // [3, 4]
queue.popLeft(); // [4]
queue.popLeft(); // []
```

리스코프 치환 원칙 위배 (=서브클래싱)

리스코프 치환 원칙 위배: 상위 타입인 **List**로서 사용 불가능 (=서브타이핑X)

=> **Stack & Queue**는 **List**로부터 모든 퍼블릭 메서드들을 전부 상속받음

=> 호출되어서는 안되는 메서드의 존재로 선입후출/선입선출 등 보장 불가

```
// List로 업캐스팅 가능
List stack = new Stack();
stack.push(1); // [1]
stack.pushLeft(2); // [2, 1]
stack.pop(); // [2]
stack.pushLeft(3); // [3, 2]
stack.push(4); // [3, 2, 4]
stack.pop(); // [3, 2]
stack.popLeft(); // [2]
stack.popLeft(); // []
```

```
// List로 업캐스팅 가능
List queue = new Queue();
queue.push(1); // [1]
queue.pushLeft(2); // [2, 1]
queue.pop(); // [2]
queue.pushLeft(3); // [3, 2]
queue.push(4); // [3, 2, 4]
queue.pop(); // [3, 2]
queue.popLeft(); // [2]
queue.popLeft(); // []
```

Golang Korea

상속 대신 조합

Stack & Queue는 내부에 List 객체를 관리하며 List의 메서드를 내부적으로 호출

↳ has-a 관계 (~~is-a~~ 관계) ↳ message passing

=> 필수적인 메서드들만 별도로 구현하여 선입후출/선입선출 강제

```
public class Stack {  
    private final List list;  
  
    public Stack(List list) {  
        this.list = list;  
    }  
  
    public void push(int value) {  
        list.push(value);  
    }  
  
    public int pop() {  
        return list.pop();  
    }  
}
```

```
public class Queue {  
    private final List list;  
  
    public Queue(List list) {  
        this.list = list;  
    }  
  
    public void push(int value) {  
        list.push(value);  
    }  
  
    public int popLeft() {  
        return list.popLeft();  
    }  
}
```

상속 대신 조합

Stack & Queue는 내부에 List 객체를 관리하며 List의 메서드를 내부적으로 호출

↳ has-a 관계 (~~is-a 관계~~) ↳ message passing

=> 필수적인 메서드들만 별도로 구현하여 선입후출/선입선출 강제

```
// 선입후출: push & pop만 사용
List l = new List();
Stack stack = new Stack(l);
stack.push(1); // [1]
stack.push(2); // [1, 2]
stack.push(3); // [1, 2, 3]
stack.pop(); // [1, 2]
stack.push(4); // [1, 2, 4]
```

```
// 선입선출: push & popLeft만 사용
List l = new List();
Queue queue = new Queue(l);
queue.push(1); // [1]
queue.push(2); // [1, 2]
queue.push(3); // [1, 2, 3]
queue.popLeft(); // [2, 3]
queue.push(4); // [2, 3, 4]
```

Go에는 상속이 없다

Is Go an object-oriented language? Yes and no.
Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy...
There are also ways to **embed types in other types to provide something analogous**—but not identical—**to subclassing**.

<https://go.dev/doc/faq#inheritance>

=> 조합을 통해 서브클래싱과 유사한 기능 제공
다만, 조합 문법의 활용법은 다양

Go의 조합 방법

Stack 내부에 **List** 타입을 포함시키는 방법에 따라 미세한 차이 발생

- 익명 필드 여부, 캡슐화 여부 등

```
package list

type List struct {
    Values []int
}

// 맨 처음에 요소 추가
func (l *List) PushLeft(value int64) { , , , }

// 맨 끝에 요소 추가
func (l *List) Push(value int64) { , , , }

// 첫 번째 요소 제거 후 반환
func (l *List) PopLeft() int64 { , , , }

// 마지막 요소 제거 후 반환
func (l *List) Pop() int64 { , , , }
```

```
type Stack1 struct {
    list.List // 익명 필드로 관리
}
```

```
type Stack2 struct {
    List list.List // Exported 필드로 관리
}
```

```
type Stack3 struct {
    list list.List // unexported 필드로 관리
}
```

(1) Anonymous Field

필드명을 생략하는 익명 필드의 경우, 서브클래싱과 유사한 문법적 편의 기능 제공

1) List 타입을 명시하여 **List**의 **Exported** 메서드+필드 접근 가능

2) List 명시 불필요: 간편 문법, 서브클래싱과 유사한 기능 제공

```
stack := Stack1{  
stack.List.Push(1) // [1]  
stack.Push(2)      // [1, 2]  
stack.Push(3)      // [1, 2, 3]  
stack.List.Pop()   // [1, 2]  
stack.Pop()        // [1]  
values := stack.List.Values  
values = stack.Values  
  
// 선입후출 위배: 상속의 문제 그대로 공유  
stack.PushLeft(5) // [5, 1]  
stack.Push(4)     // [5, 1, 4]  
stack.PopLeft()   // [1, 4]
```

```
type Stack1 struct {  
    list.List // 익명 필드로 관리  
}
```

```
var stack list.List = Stack1{  
// Cannot use 'Stack1{ }'  
// (type Stack1) as the type list.List
```

(2) Exported Field

List라는 필드를 경유하여 **List** 타입의 **Exported** 메서드 및 필드 접근 가능

- 익명 필드에 따른 문법적 편의 활용 불가
- 캡슐화 위배: **List** 필드의 모든 메서드 및 필드를 외부에 그대로 노출

```
stack := Stack2{}
stack.List.Push(1) // [1]
stack.List.Push(2) // [1, 2]
stack.List.Pop()   // [1]
values := stack.List.Values
```

```
stack.Push(0) // Unresolved reference
stack.Pop()   // Unresolved reference
```

```
// 선입후출 위배: 캡슐화 위배로 때문
stack.List.PushLeft(4) // [4, 1]
stack.List.Push(3)     // [4, 1, 3]
stack.List.PopLeft()   // [1, 3]
```

```
type Stack2 struct {
    List list.List
    // Exported 필드로 관리
}
```

(3) Unexported Field

`list`라는 필드를 경유하여 내부적으로 `List` 타입의 **Exported** 메서드 및 필드 접근

- 내부적으로 `list` 필드 활용하여 필수적인 메서드들만 별도로 구현
- (패키지 수준에서의) 캡슐화 보장 + 선입후출 보장 => **Good**

```
stack := stack.Stack3{}
stack.Push(1) // [1]
stack.Push(2) // [1, 2]
stack.Pop()   // [1]
stack.Push(3) // [1, 3]
stack.Push(4) // [1, 3, 4]
stack.Pop()   // [1, 3]
stack.Push(5) // [1, 3, 5]

// Unexported field 'list' usage
stack.list.Values
stack.list.PushLeft()
stack.list.PopLeft()
```

```
package stack

type Stack3 struct {
    list list.List // unexported 필드로 관리
}

// 맨 끝에 요소 추가
func (s *Stack3) Push(value int64) {
    ...
}

// 마지막 요소 제거 후 반환
func (s *Stack3) Pop() int64 {
    ...
}
```

정리

Java에서 코드 재사용 목적으로는 상속 대신 조합이 권장된다.

- 코드 재사용 목적의 상속은 서브타이핑이 아닌 서브클래싱이라고 불린다.

Go는 상속이 지원되지 않는다. 타입들은 다양한 방식으로 조합될 수 있다.

- 타입을 익명 필드로 조합할 경우, 서브클래싱과 유사한 편의 문법이 지원된다.
- 캡슐화를 통해 외부로 노출되는 퍼블릭 인터페이스를 적절히 제한할 수 있다.



Abstraction & Polymorphism



Abstraction

어떤 양상, 세부 사항, 구조를 좀 더 명확하게 이해하기 위해 특정 절차나 물체를 의도적으로 생략하거나 감춤으로써 복잡도를 극복하는 방법이다... 구체적인 사물들 간의 공통점은 취하고 차이점을 버리는 일반화를 통해 단순하게 만드는 것이다. ... 중요한 부분은 강조하기 위해 불필요한 세부사항을 제거함으로써 단순하게 만드는 것이다. 모든 경우에 추상화의 목적은 복잡성을 이해하기 쉬운 수준으로 단순화하는 것이라는 점을 기억하라.

조영호, 객체지향의 사실과 오해, 2015

Member 인터페이스 정의

1. Name 메서드로 회원의 이름 조회 가능
 2. ChangeName 메서드로 회원의 이름 변경 가능
- => Do 함수에서 Member와 상호작용할 수 있는 유일한 방법은
Name & ChangeName 메서드 호출 (=Message Passing)

```
package domain

type Member interface {
    Name() string
    ChangeName(string) error
}
```

```
package application

func Do(member domain.Member, name string) {
    fmt.Printf("My name was %s \n", member.Name())
    err := member.ChangeName(name)
    if err != nil {
        panic(err)
    }
    fmt.Printf("My name is %s \n", member.Name())
}
```

Member 인터페이스의 구현

member 구조체에 Name & ChangeName 메서드 구현

=> member는 Member의 구현체로써 사용될 수 있다고 간주 (별도의 문법X)

```
package domain

func NewMember(name string) (Member, error) {
    // ...
    return &member{name: name}, nil
}

type member struct { // unexported
    name string // unexported
}

func (m *member) Name() string {
    return m.name
}

func (m *member) ChangeName(name string) error {
    m.name = name
    return nil
}
```

```
m, err := domain.NewMember("Jake")
if err != nil {
    panic(err)
}
```

```
application.Do(m, "John")
application.Do(m, "Jason")
```

```
// Unresolved reference 'name'
m.name
```

```
// Usage of the unexported type 'member'
m2 := domain.member{}
```

추상화 뒤에 내부 구현 캡슐화

- 추상화(**Member**)만 패키지 외부에 노출 : **Member**의 메서드 호출만 가능
- 내부 구현(**member**)은 패키지 수준에서 캡슐화

```
package domain

func NewMember(name string) (Member, error) {
    // ...
    return &member{name: name}, nil
}

type member struct { // unexported
    name string // unexported
}

func (m *member) Name() string {
    return m.name
}

func (m *member) ChangeName(name string) error {
    m.name = name
    return nil
}
```

```
m, err := domain.NewMember("Jake")
if err != nil {
    panic(err)
}
```

```
application.Do(m, "John")
application.Do(m, "Jason")
```

```
// Unresolved reference 'name'
m.name
```

```
// Usage of the unexported type 'member'
m2 := domain.member{}
```

Polymorphism

다형성이란 서로 다른 유형의 객체가 동일한 메시지에 대해 서로 다르게 반응하는 것을 의미한다. 좀 더 구체적으로 말해 서로 다른 타입에 속하는 객체들이 동일한 메시지를 수신할 경우 서로 다른 메서드를 이용해 메시지를 처리할 수 있는 메커니즘을 가리킨다.

조영호, *객체지향의 사실과 오해*, 2015

컴퓨터 과학에서는 다형성을 하나의 추상 인터페이스에 대해 코드를 작성하고 이 추상 인터페이스에 대해 서로 다른 구현을 연결할 수 있는 능력으로 정의한다.

Czarnecki K. et al., *Generative Programming: Methods, Tools, and Applications*, 2000

cf) 상속은 다형성을 위한 것

상속을 사용하는 일차적인 목표는 코드 재사용이 아니라 타입 계층을 구현하는 것이어야 한다... 동일한 메시지에 대해 서로 다르게 행동할 수 있는 다형적인 객체를 구현하기 위해서는 객체의 행동을 기반으로 타입 계층을 구성해야 한다. 상속의 가치는 이러한 타입 계층을 구현할 수 있는 쉽고 편한 방법을 제공한다는 데 있다.

조영호, 오브젝트, 2019

=> 그런데 Go에는 상속이 없으므로
추상화(인터페이스)를 통해서만 다형성 구현 가능

추상화를 통한 다형성

Stack 인터페이스 정의: Push & Pop

=> List, Stack1, Stack3은 구현체로써 활용 가능

```
type Stack interface {  
    Push(int64)  
    Pop() int64  
}
```

```
package list
```

```
type List struct {  
    Values []int64  
}
```

```
// 맨 처음에 요소 추가
```

```
func (l *List) PushLeft(value int64) { ... }
```

```
// 맨 끝에 요소 추가
```

```
func (l *List) Push(value int64) { ... }
```

```
// 첫 번째 요소 제거 후 반환
```

```
func (l *List) PopLeft() int64 { ... }
```

```
// 마지막 요소 제거 후 반환
```

```
func (l *List) Pop() int64 { ... }
```

```
type Stack1 struct {  
    list.List // 익명 필드이므로 Push & Pop 호출 가능  
}
```

```
type Stack2 struct {  
    List list.List // Exported 필드이므로 간편 문법X  
}
```

```
type Stack3 struct {  
    list list.List // 명시적으로 Push & Pop 정의  
}
```

```
func (s *Stack3) Push(value int64) { ... }  
func (s *Stack3) Pop() int64 { ... }   Golang Korea
```


추상화를 통한 다형성

Stack 인터페이스 정의: **Push & Pop**

=> **List, Stack1, Stack3**은 구현체로써 활용 가능

```
type Stack interface {  
    Push(int64)  
    Pop() int64  
}
```

```
// 인터페이스를 매개변수로  
func PushAndPop(stack Stack) {  
    stack.Push(1) // [1]  
    stack.Push(2) // [1,2]  
    stack.Push(3) // [1,2,3]  
    stack.Pop()   // [1,2]  
    stack.Pop()   // [1]  
    stack.Pop()   // []  
}
```

```
PushAndPop(&list.List{})  
PushAndPop(&Stack1{})  
PushAndPop(&Stack3{})  
  
PushAndPop(&Stack2{})  
// Cannot use '&Stack2{}' (type *Stack2)  
// as the type Stack Type does not  
// implement 'Stack'  
// as some methods are missing:  
// Push(int64) Pop() int64
```

Open-Closed Principle

개방 폐쇄 원칙은 다음과 같다... 기능을 변경하거나 확장할 수 있으면서 그 기능을 사용하는 코드는 수정하지 않는다... (사용되는 기능의) 확장에는 열려 있어야 하고, (기능을 사용하는 코드의) 변경에는 닫혀 있어야 한다...

최범균, *개발자가 반드시 정복해야 할 객체 지향과 디자인 패턴*, 2013

이처럼 기존 코드에 아무런 영향도 미치지 않고 새로운 객체 유형과 행위를 추가할 수 있는 객체지향의 특성을 개방-폐쇄 원칙이라고 부른다. 이것이 객체지향 설계가 전통적인 방식에 비해 변경하고 확장하기 쉬운 구조를 설계할 수 있는 이유다.

조영호, *오브젝트*, 2019

OCP : 추상화에 대한 의존

PushAndPop은 추상화(인터페이스)에 의존

=> Push/Pop 메시지 전송하는 코드는 변경에 **Closed**

실질적으로 실행되는 코드는 메시지를 수신한 **Stack** 구현체의 메서드

=> 다형적인 객체들을 추가하는 방식으로 기능 확장에 **Open**

```
// 변경에 Closed = 수정 불필요
func PushAndPop(stack Stack) {
    stack.Push(1) // [1]
    stack.Push(2) // [1,2]
    stack.Push(3) // [1,2,3]
    stack.Pop()   // [1,2]
    stack.Pop()   // [1]
    stack.Pop()   // []
}
```

```
// 기존 기능들
PushAndPop(&list.List{})
PushAndPop(&Stack1{})
PushAndPop(&Stack3{})
```

```
// 기능의 확장에 Open
PushAndPop(&InmemoryStack{})
PushAndPop(&db.DurableStack{})
PushAndPop(&redis.RedisStack{})
```

Dependency Inversion Principle

의존 역전 원칙의 정의는 다음과 같다... 고수준 모듈은 저수준 모듈의 구현에 의존해서는 안 된다. 저수준 모듈이 고수준 모듈에서 정의한 추상 타입에 의존해야 한다.

최범균, 개발자가 반드시 정복해야 할 객체 지향과 디자인 패턴, 2013

상위 수준의 모듈은 하위 수준의 모듈에 의존해서는 안 된다. 둘 모두 추상화에 의존해야 한다. 추상화는 구체적인 사항에 의존해서는 안 된다. 구체적인 사항은 추상화에 의존해야 한다. 이를 의존성 역전 원칙이라고 부른다.

조영호, 오브젝트, 2019

비교 : (하위 모듈의) 구현에 대한 의존

하위 모듈(**stack**)의 구현에 의존하면 상위 모듈(**application**)의 변화를 야기함

- 기존 스펙 : 상위 모듈에서 하위 모듈의 **Stack1** 구현체에 의존
- 신규 스펙 : 상위 모듈에서 하위 모듈의 **Stack3** 구현체에 의존

```
package application
```

```
// 기존 스펙
```

```
func PushAndPop(stack stack.Stack1) {  
    // ...  
}
```

```
package application
```

```
// 스펙 변경: 매개변수 수정 필요
```

```
func PushAndPop(stack stack.Stack3) {  
    // ...  
}
```

```
package stack
```

```
type Stack1 struct {  
    list.List  
}
```

```
type Stack3 struct {  
    list list.List  
}
```

```
func (s *Stack3) Push(value int64) { ... }  
func (s *Stack3) Pop() int64 { ... }
```

Golang Korea

DIP : (상위 모듈의) 추상화에 대한 의존

상위 모듈(application)에 추상화 정의 + 내부적으로 추상화에 의존

- 하위 모듈(stack)에서는 상위 모듈의 추상화를 구현 => Stack1, Stack3, ...
- Go에서의 DIP는 두 패키지 간의 의존성이 완전히 끊기는 구조

```
package application

func PushAndPop(stack Stack) {
    // ...
}

type Stack interface {
    Push(int64)
    Pop() int64
}
```

```
package stack

type Stack1 struct {
    list.List
}

type Stack3 struct {
    list list.List
}

func (s *Stack3) Push(value int64) { ... }
func (s *Stack3) Pop() int64 { ... }
```

Go는 인터페이스 구현을 위해 application 패키지를 import하지는 않으므로 의존성 역전이라고 부르기 다소 애매함

DIP : (상위 모듈의) 추상화에 대한 의존

- 상위 모듈(application)에 추상화 정의 + 내부적으로 추상화에 의존
- 하위 모듈(stack)에서는 상위 모듈의 추상화를 구현 => **Stack1, Stack3, ...**
 - **Java**에서는 [상위=>하위] 대신 [하위=>상위]로 모듈 간 의존성이 역전됨

```
package application;

public class Class {
    public void pushAndPop(Stack stack) {
        // ...
    }
}
```

```
package application;

public interface Stack {
    void push(int value);
    int pop();
}
```

```
package stack;

import application.Stack;

public class Stack1 implements Stack {
    public void push(int value) { ... }
    public int pop() { ... }
}
```

stack 패키지 쪽에서
인터페이스 구현을 위해
application 패키지를
import하므로 의존성 역전

Interface Segregation Principle

인터페이스 분리 원칙은 다음과 같다. 인터페이스는 그 인터페이스를 사용하는 클라이언트를 기준으로 분리해야 한다.

최범균, *개발자가 반드시 정복해야 할 객체 지향과 디자인 패턴*, 2013

클라이언트는 자신이 실제로 호출하는 메서드에만 의존해야만 한다.
이것은 이 비대한 클래스의 인터페이스를 여러 개의 클라이언트에 특화된 인터페이스로 분리함으로써 성취될 수 있다... 이렇게 하면 호출하지 않는 메서드에 대한 클라이언트의 의존성을 끊고, 클라이언트가 서로에 대해 독립적이 되게 만들 수 있다.

Martin R., *Agile Software Development, Principles, Patterns, and Practices*, 2013

불필요한 메서드에 의존

Client는 메시지 전송하는 역할 : **Push**만 필요
Server는 메시지 처리하는 역할 : **Pop**만 필요
=> 호출하면 안되는 메서드의 존재 자체가 위험

```
type Queue interface {  
    Push(string) // 데이터 추가  
    Pop() string // 데이터 제거  
}
```

```
type Client struct {  
    queue queue.Queue // Push & Pop 가능  
}  
  
func (c *Client) SendMessage(msg string) {  
    // ...  
    c.queue.Push(msg)  
}  
  
func (c *Client) BadUsage() {  
    msg := c.queue.Pop() // 호출하면 안됨!  
    // ...  
}
```

```
type Server struct {  
    queue queue.Queue // Push & Pop 가능  
}  
  
func (s *Server) HandleMessage() {  
    msg := s.queue.Pop()  
    // ...  
}  
  
func (s *Server) BadUsage(msg string) {  
    s.queue.Push(msg) // 호출하면 안됨!  
    // ...  
}
```

ISP : 소규모 인터페이스들로 분리

Client는 메시지 전송하는 역할 : **Push**만 필요
Server는 메시지 처리하는 역할 : **Pop**만 필요
=> 호출해도 되는 메서드만 제공

```
type Pushable interface {  
    Push(string) // 데이터 추가  
}
```

```
type Popable interface {  
    Pop() string // 데이터 제거  
}
```

```
type Client struct {  
    pushable queue.Pushable // Push만 가능  
}  
  
func (c *Client) SendMessage(msg string) {  
    // ...  
    c.pushable.Push(msg)  
}  
  
func NewClient(p queue.Pushable) Client {  
    return Client{pushable: p}  
}
```

```
type Server struct {  
    popable queue.Popable // Pop만 가능  
}
```

```
func (s *Server) HandleMessage() {  
    msg := s.popable.Pop()  
    // ...  
}  
  
func NewServer(p queue.Popable) Server {  
    return Server{popable: p}  
}
```

ISP : 소규모 인터페이스들로 분리

MessageQueue에 Push & Pop 메서드 정의
=> Pushable & Popable를 다중 구현
=> Client & Queue 모두에서 활용 가능

```
type Pushable interface {  
    Push(string) // 데이터 추가  
}
```

```
type Popable interface {  
    Pop() string // 데이터 제거  
}
```

```
messageQueue := &queue.MessageQueue{}
```

```
// Pushable의 구현체로 활용  
c := NewClient(messageQueue)  
c.SendMessage("Hello")  
c.SendMessage("World")
```

```
// Popable의 구현체로 활용  
s := NewServer(messageQueue)  
s.HandleMessage() // Received Hello  
s.HandleMessage() // Received World
```

```
type MessageQueue struct {  
    list list.List  
}  
  
// Pushable의 구현체  
func (q *MessageQueue) Push(msg string) {  
    q.list.Push(msg)  
}
```

```
// Popable의 구현체  
func (q *MessageQueue) Pop() string {  
    return q.list.PopLeft()  
}
```



Conclusion



Is Go Objected Oriented?

정의에 따라 객체지향 언어로 분류 가능한지는 논란의 여지 존재
다만, **OOP** 이론 및 개념을 적용할 수 있다는 점은 확실

Is Go an object-oriented language? **Yes and no.**
Although Go has types and methods and **allows an object-oriented style of programming**, there is no type hierarchy...

<https://go.dev/doc/faq#inheritance>

Go is Object Oriented Enough



OOP는 프로그래밍 패러다임일 뿐

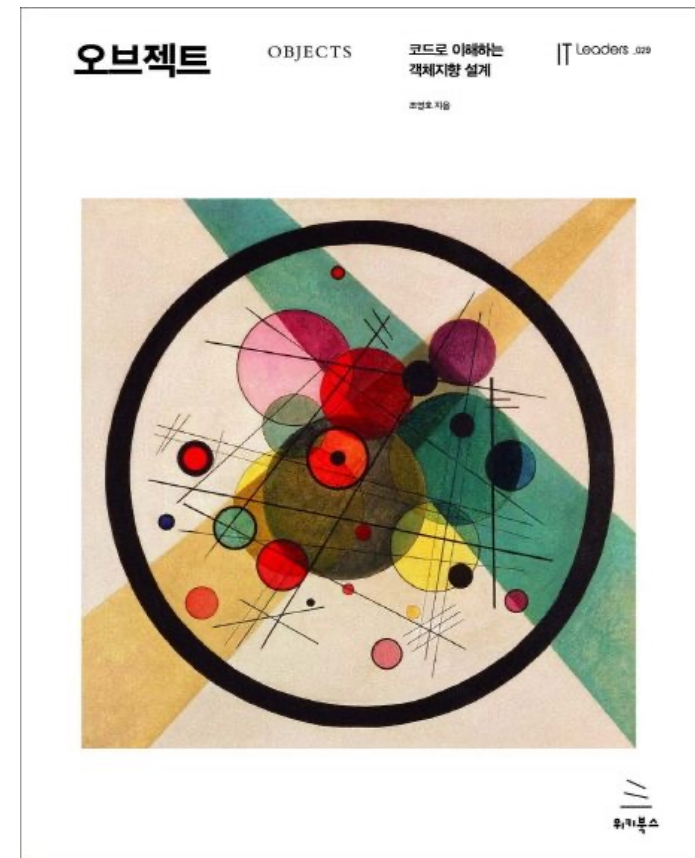
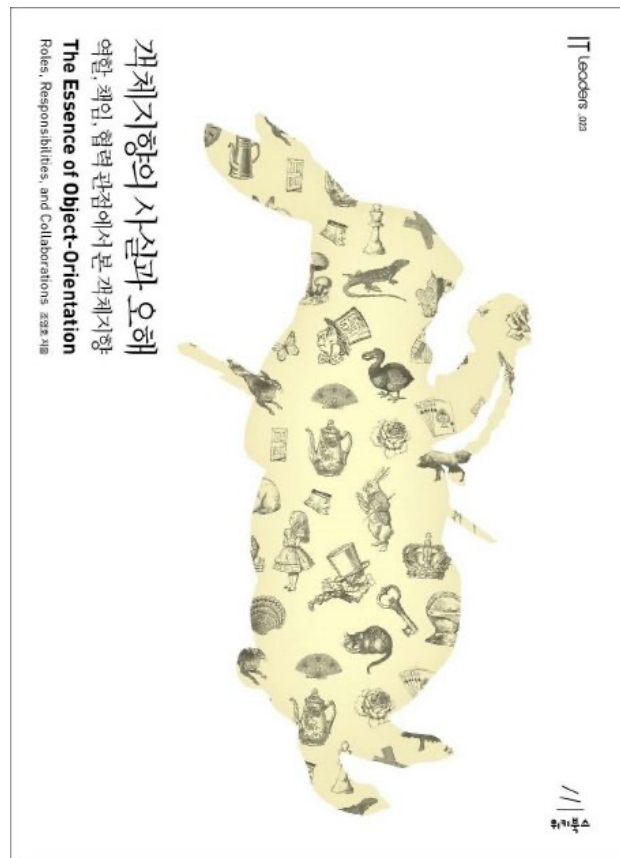
사고의 확장을 위한 지침일 뿐 OOP에 매몰되어서는 안 됨

- 객체지향적이기 때문에 좋은 코드다 (X)
- 객체지향 이론을 적재적소로 적용했더니 구조가 개선되었다 (O)

=> OOP를 적용하는데 필수적인 문법을 충분히 제공

즉, **Go**는 충분히 객체지향적인 언어

Reference



Reference

조영호. (2015). 객체지향의 사실과 오해.

조영호. (2019). 오브젝트.

최범균. (2013). 개발자가 반드시 정복해야 할 객체 지향과 디자인 패턴.

Czarnecki, K., & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. <https://doi.org/10.1604/9780201309775>

Martin, R. (2002). *Agile Software Development, Principles, Patterns, and Practices*. <https://doi.org/10.1604/9780135974445>

Wiener, L., Wirfs-Brock, R., & Wilkerson, B. (1990). *Designing Object-Oriented Software*. <https://doi.org/10.1604/9780136298250>

<https://go.dev/doc>



Thank you