*기능을 확장하는 방법 : 1. 기존클래스를 변경

| Car |
|---|
| model |
| maker |
| capacity |
| car() |
| car(~생성자) |
| run() |
| sunroof() |
| auto() |
| Car(선루프 생성자 추가) |

>기존 코드에 새 기능을 위한 코드 추가<

*문제점
1. 코드에 계속 기능을 덧붙이다보면 누더기 코드가 된다.
2.★기존 코드를 사용하는 프로젝트에 영향을 끼친다.★

*기능을 확장하는 방법 : 2. 기존소스를 복제해서 기능 확장

| Car |
|---|
| model |
| maker |
| capacity |
| car() |
| car(~생성자) |
| run() |

→ 복제 →

| Car2 | |
|---|---|
| - model | |
| - maker | |
| - capacity | |
| - car() | |
| - car(~생성자) | |
| - run() | |
| - sunroof | 추가 |
| - auto | |
| - car( - - - ) | |

↑
A 프로젝트
B 프로젝트

↑
C 프로젝트

*특징
1. 기존 코드에 손대지 않는다.
>> 기존 프로젝트에 영향을 끼치지 않는다.

단점
1. 복제 ... 중복코드가 생성된다.
>> 중복코드가 생성되면 기능 추가시 전체에 적용
>> 버그 수정 시 전체에 적용
>> "유지 보수가 힘들다."

*기능을 확장하는 방법 : 3. 상속을 이용

*원본에서 버그 발생 = 원본만 수정하면 된다.

| Car |
|---|
| model |
| maker |
| capacity |
| car() |
| car(~생성자) |
| run() |

← 기존 기능 사용권 획득 ←

| Sedan |
|---|
| - sunroof |
| - auto |
| - Sedan( - - - ) |

↑ ↑
A 프로젝트 B 프로젝트

↑
C 프로젝트

특징
- 기존 코드를 손대지 않는다.
>> 이전 프로젝트에 영향을 끼치지 않는다.
- 기존 코드를 재사용한다. --> 비용 절감
>> 버그 추가 가능성을 낮추는 효과

재사용 : 코드 중복을 없앤다. -> 버그 수정이 용이

단점
- 여러 단계를 거쳐 필요없는 기능도 강제로 상속받는 경우가 있다.

*기능을 확장하는 방법 : 3-1. 상속을 이용

Super class
Parent class
A - m1 ()
A의 사용권을 획득한다. = 상속
↑
Sub class
child class
B - m2 ()
↑
C - m3 ()
↑
D - m4 ()

A

B
A

A와 코드를 그대로 포함한게 아니다!

B obj = new B () ;
obj.m2() ; // OK
obj.m1() ; <-- B 클래스를 통해 A 클래스의 멤버를 사용할 수 있다.
A 의 m1()을 호출

*상속과 인스턴스 필드(변수)

A class - v1
- static { - - - }
↑
B - v2
- static { - - - }

B obj = new B() ;
↓
*인스턴스 필드 생성
-> 수퍼클래스의 인스턴스 변수 먼저 생성됨.

*클래스 로딩
↓
수퍼 클래스 부터 로딩 (A클래스부터 로딩 된 후에 B클래스 실행)

| obj | 200 | v1 | v2 |
|---|---|---|---|
| 200 | | | |

B obj2 = new B() ;

| obj | 300 | v1 | v2 |
|---|---|---|---|
| 300 | | | |

*생성자 호출 순서

Object - Object() {- - -} 모든 클래스 생성자는 슈퍼 클래스 생성자를 호출해야한다.
↑ 아무것도 없으면 Object를 상속한다.
A class - v1
- A() {- - -}
↑ super() <== 생성자의 "첫 문장"은 반드시 슈퍼클래스의 생성자 호출 문장이어야 한다. 예) super() // this() <= 둘다 동시에 호출할 순 없다.
B class - v2 <== 또는 같은 클래스의 다른 생성자를 호출하는 문장이어야 한다.
- B() {- - -}
↑
C class - v3
- C() {- - -} <== 슈퍼클래스의 생성자 호출을 생략하면 컴파일러가 자동으로 추가한다.

C obj = new C () ; ==> C의 생성자 호출 →

| Object | ← 4 | A class | ← 3 | B class | ← 2 | C class | 1. 호출 |
|---|---|---|---|---|---|---|---|
| Object() { super() } | | A() { super() } | | B() { super() } | | C() { super() } | |
| ↓ | | | | | | 생성자 | |
| 5 | → | 6 | → | 7 | → | 8 | → 리턴 |

*생성자 호출 2

A class - v1 B obj = new B(); ==> B의 기본 생성자 호출
↑ - A(int)
B class - v2
- B()

*다중 상속

| A class | b class | C obj = new C () ; | v1 | v2 | v2 | v3 |
|---|---|---|---|---|---|---|
| v1 : int | v2 : int | 200 | A | A | B | C |
| v2 : float | | | | | | |

↑ ↑

b class
v3 : int

obj.v2 = ? ;
A의 변수인지 B의 변수인지 구분할 수 없다. => 구분하려면 문법을 추가 => 언어가 복잡 => 컴파일러가 복잡
↓
그래서 자바는 다중상속을 허락하지 않는다.

*상속 : specialization(전문화)

Car - model <- 처음에 직접 사용하려고 만든 클래스이다.
- maker <- 필요에 의해 추가로 정의한 클래스이다.
- capacity
- run
↓
상속받은 메서드를 서브클래스에 맞춰 재정의한다 = 오버라이딩
Sedan - run() { - } << 수퍼클래스의 메서드를 현재 클래스의 역할에 맞춰 재정의한다. ="Overriding" (오버라이딩)
- doSunroof() { - } << 승용차에 맞는 기능을 넷볼인다.
↓
*더 특별한 자동차를 만든다 = specialization
Truck - run() { - } <<트럭의 특성에 맞춰 재정의 = Overriding
- dump() { - } <<트럭만의 고유기능

*상속 : Generalization(일반화)

Car - start () << 다른 자동차를 정의할 때 재사용 가능 :generalization
- shutdown () 추출해서 일반적인 기능으로 만든다.
- run

| Sedan | | Truck |
|---|---|---|
| ~~- start()~~ | | ~~- launch()~~ |
| ~~- shutdown()~~ | <-- 유사 --> | ~~- stop ()~~ |
| ~~- run()~~ | | ~~- go ()~~ |
| - doSunroof() | | - dump () |
| run 메서드 재정의 : overriding | | run 메서드 재정의 : overriding |

↑

자동차의 일반적인 기능 : Generalization (일반화)

*상속과 추상메서드

1. 둘 사이의 공통점을 추출해서 Car를 만든다.

| Sedan | Truck |
|---|---|
| ~~- start()~~ | ~~- launch()~~ |
| ~~- shutdown()~~ | ~~- stop ()~~ |
| ~~- run()~~ | ~~- go ()~~ |
| - doSunroof() | - dump () |
| run() { - } | run() { - } |

→ 어짜피 sub 클래스에서 재정의해야 할 메서드라면
슈퍼클래스에서 정의하지 말자!
메서드 형식만 정의하자! : Method Signature(java) = function prototype(c /c++)
> "메서드 body를 만들지 않는다"
> "추상 메서드"

Method Signature
메서드명
파라미터 타입
리턴타입

| Car |
|---|
| - start() { - } |
| - shutdown() { - } |
| - run() { - } <<abstract>> <--- 보조명칭 : stereotype |

*상속과 추상메서드

더 특별한 자동차를 만든다.

추상클래스 => 인스턴스 생성 x

| Car |
|---|
| - start () |
| - shutdown () |
| - run |

<< 소스코드를 유지보수 하기 쉽도록 공통코드를 추출해서 만든 클래스
<< Car 클래스를 따로 쓰려고 만든 클래스가 아님.
"이 경우 직접 사용할 수 없도록, [추상 클래스]로 선언한다."
"abstract"

"구체 클래스" >> 인스턴스 생성 가능
concrete class

| Sedan |
|---|
| - start() |
| - shutdown() |
| - run() |
| - doSunroof() |

| Truck |
|---|
| - launch() |
| - stop () |
| - go () |
| - dump () |

서브클래스에서 재정의할 메서드가 있다면 추상 메서드로 선언하라.

추상 메서드를 쓰려면 상속받은 콘크리트 메서드를 사용하던지, 상속 받아서 구현해서 사용해라.

specilazation
전문화

Car ← Test01
Car ← Sedan ← Test02
Car ← Truck ← Test03

sub

Test01 → Sedan
- ~~start() { - }~~
- ~~shutdown() { - }~~
- ~~run() { - }~~
- dosunroof() { - }
- run() { - } ==> 재정의

Test02 → Truck
- ~~launch() { - }~~
- ~~stop() { - }~~
- ~~go() { - }~~
- dump() { - }
- run() { - } ==> 재정의

Generalization (일반화)

super
<<abstract>>
Car
- on() { - }
- off() { - }
- run() { - } <<abstract>> ==> 추상메서드

A class m1()
↑
B class m2()
↑
C class m3()
↑
D class m4()

D obj = new D () ;
obj.m1() ;
obj.m3() ;
obj.m4() ;
obj.m2() ; =>임의로 삭제할 수 없다. = 상속의 한계

*기능 확장법 : cheat sheet

1. 기존 코드 변경

| Calculator |
|---|
| -plus() |
| -minus() |
| -multple() |

2. 코드 복제 후 기능 추가

| Calculator |
|---|
| -plus() |
| -minus() |

↓ 복제

| Calculator2 |
|---|
| -plus() |
| -minus() |
| -multple() |

3. 상속

| Calculator |
|---|
| -plus() |
| -minus() |

↑ 상속

| Calculator2 |
|---|
| -multple() |

"종속관계"

4. 위임

| Calculator | call "위임" |
|---|---|
| -plus() | |
| -minus() | call |

↑ 포함

| Calculator2 |
|---|
| -plus() |
| -minus() |
| -multple() |

## *클래스 관계 cheat sheet

### 1. 상속(inheritance)

A class <- super class

↑

B class <- sub class

```
class B extend A { ~ }
```

### 2. 연관(association)

B class → A class

지속적인 사용

ex) 사람 — ex) 핸드폰

```
class B {
A obj;
}
```

### 3. 집합(aggregation)

B class ◇→ A class

포함

ex) 컴퓨터 — ex) 키보드 & 마우스

```
class B {
A obj;
}
```

Lifecycle

컴퓨터 ≠ 키보드

≠ 마우스

≠ 모니터

### 4. 합성(composition)

B class ◆→ A class

포함

ex) 컴퓨터 — ex) 메인보드 & cpu

```
class B {
A obj;
}
```

Lifecycle

컴퓨터 = 그래픽카드

= ROM

= CPU

### 5. 의존(depenency)

특정 메서드를 실행할 때 일시적으로 사용하는 관계

B class - - - - - -> A class

ex) 핸드폰 — ex) 충전기

```
class B {
void m(A obj){
}
```