아이템1: 상속보다는 컴포지션을 사용하라

> 상속은 관계가 명확하지 않을 때 사용하면, 여러가지 문제가 발생할 수 있습니다. 따라서 **단순하게 코드 추출 또는 재사용을 위해 상속을 하려고 한다면**, 조금 더 신중하게 생각해야 합니다. 일반적으로 이러한 경우에는 상속보다 컴포지션을 사용하는 것이 좋습니다.

간단한 행위 재사용

“프로그래스 바 출력 → 로직 처리 → 프로그래스 바 숨기기”의 유사한 동작을 하는 클래스가 있는 경우

```
class ProfileLoader{
    fun load(){
        // show progress bar
        // load profile
        // hide progress bar
    }
}

class ImageLoader{
    fun load(){
        // showing progress bar
        // load image
        // hiding progress bar
    }
}
```

많은 개발자가 슈퍼클래스를 만들어서 공통되는 행위 추출

```
abstract class LoaderWithProgress {
    fun load() {
        //show progress bar
        innerLoad()
        // hide progress bar
```

```
    }
    abstract fun innerLoad()
}

class ProfileLoader: LoaderWithProgress() {
    override fun innerLoad() {        // load profile    }

}

class ImageLoader: LoaderWithProgress(){
    override fun innerLoad() { // read profile    }
}
```

이런 코드는 간단한 경우에는 문제 없이 동작하지만, 복잡한 경우 몇 가지 단점 존재

1. 행위를 추출하다 보면 **거대한 base class**를 만들게 되고 **깊고 복잡한 계층 구조가** 만들어진다.
    - 한 클래스만을 대상으로 할 수 있기 때문 (다중상속 x)
    - 기능을 추가하려면 base class에 넣거나, 계층을 하나 더 만들어야겠죠?
2. **ISP 위반 가능성**이 있다.
    - 클래스의 모든 것을 가져오게 되므로, 불필요한 함수를 가질 수 있다.
3. **상속은 이해하기 어렵다.**
    - 메서드의 작동 방식을 이해하기 위해 슈퍼 클래스를 여러 번 확인해야 한다. 그 자체로 문제가 있는 것임.

상속의 대안: 컴포지션(composition, 합성)

객체를 프로퍼티로 갖고, 함수를 호출하는 형태로 재사용하는 것을 의미한다.

아래 코드에서는 progress 객체를 활용하는 **추가적인 코드**가 필요하다.

- **코드의 동작을 명확하게 예측**할 수 있게 해준다. (이해하기 쉽다)
- **합성한 객체를 자유롭게 사용 가능** (프로그래스 바를 자유롭게 사용 가능)
    - hide가 강제되지 않음
- **하나의 클래스 내부에서 여러 기능을 재사용**할 수 있다.
    - 상속으로 구현하려면, 다른 기능을 슈퍼 클래스에 배치해야 해서 계층 구조가 만들어질 수 있음.
    - hideProgress 이후 알림을 주고 싶은 경우 등

```
class ProfileLoader {
    private val progress = Progress() // 추가적인 코드
    fun load(){
        progress.showProgress() // progress bar사용이 명확하게 드러남
        // load profile
        progress.hideProgress()
    }
}
class ImageLoader {
    private val progress = Progress()
    // 다른 기능을 컴포지션해서 재사용
    // 컴포지션 하지 않으면 계층 구조를 추가해야함
    private val finishedAlert = FinishedAlert()
    fun load(){
        progress.showProgress()
        // load image
        progress.hideProgress()
        finishedAlert.show()
    }
}
```

- bad case
  - 추가적인 계층을 만들지 않고, 합성도 안하고 기능을 추가하는 경우: 분기 추가
  - 서브클래스가 필요하지 않은 기능을 갖고, 차단할 뿐이다.

```
abstract class InternetLoader(val showAlert: Boolean){
  fun load() {
    // show progress bar
    innerLoad()
    // hide progress bar
    if(showAlert){
      //show alert
    }
  }
  abstract fun innerLoad()
}

class ProfileLoder(): InternetLoader(showAlert = true)
class ImageLoder(): InternetLoader(showAlert = false)
```

문제1 : 모든 것을 가져오는 상속

> 상속은 슈퍼클래스의 모든 것을 가져오므로, **계층 구조를 나타낼 때 좋은 도구**(중략) 하지만 **일부분을 재사용하기 위한 목적으로는 컴포지션**

> **을 사용**하는 것이 좋습니다. 원하는 행위만 가져올 수 있기 때문입니다.

강아지는 bark, sniff

로봇 강아지는 bark 기능만 갖고 싶은 경우 (일부만 재사용)

```
abstract class Dog {
  open fun bark(){}
  open fun sniff(){}
}

class Bulldog: Dog()
// robotDog는 필요 없는 메서드를 가지고 (ISP 위반), 슈퍼클래스의 동작을 깬다 (LSP)
class RobotDog: Dog(){
  override fun sniff() {
    throw Error("Operation not supported")
  }
}
```

컴포지션이 반드시 좋은 것은 아니고, 인터페이스 다중상속으로 계층 관계를 표현하는게 좋을 수도 있다.

해결방법 1: 계층관계 표현 X

```


interface BarkStrategy{}
interface SniffStrategy{}

class BullDog(val barkStrategy: BarkStrategy , val sniffStrategy: SniffStrategy)
class RobotDog(val barkStrategy: BarkStrategy)
```

해결방법 2: 계층관계 표현

```
1. 인터페이스 다중 상속
class BullDog(): Barkable, Sniffable
class RobotDog(): Barkable
2. 위임
```

문제2: 캡슐화를 깨는 상속

자식 클래스의 구현 방법 변경에 의해 클래스의 캡슐화가 깨질 수 있음

```
class CounterSet<T>: HashSet<T>() {
    var elementsAdded: Int = 0
    private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return super.add(element)
    }

    override fun addAll(elements: Collection<T>):Boolean {
        elementsAdded+=elements.size
        return super.addAll(elements) // 내부적으로 add를 호출
    }
}

fun main() = runBlocking {
    val counterList = CounterSet<String>()
    counterList.addAll(listOf("A","B","C"))
    print(counterList.elementsAdded) // 결과는 6
}
```

discussion: 캡슐화의 두가지 관점 - 1.외부에서 정보를 보지 않아도 된다. 2. 내부의 데이터를 보호할 수 있다

해결방법 1: 최대한 구현 방법 변경을 줄여본다

```
class CounterSet<T>: HashSet<T>() {
    var elementsAdded: Int = 0
    private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return super.add(element)
    }
  // 제거!
}
```

- addAll()이 add를 호출하지 않도록 변경되면 문제

해결방법 2: 컴포지션 사용

```
class CounterSet<T> {
    private val innerSet = HashSet<T>()
    var elementsAdded: Int = 0
    private set

    fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    fun addAll(elements: Collection<T>):Boolean {
        elementsAdded+=elements.size
        return innerSet.addAll(elements) // 내부적으로 add를 호출
    }
}
```

- 계층 관계를 표현하지 못함

해결방법 3: 위임 패턴

위임 패턴: 클래스가 인터페이스를 상속받게 하고, **합성된 객체로 인터페이스를 구현하는 방법**
코드 재사용에 있어서, 합성을 상속처럼 강하게 만드는 패턴
- 이 과정에서 구현된 메서드를 포워딩 메서드라고 한다.

- 포워딩 메서드가 많아질 수있지만, 특히 코틀린에서는 위임 패턴을 언어 레벨에서 지원한다.

```
class CounterSet<T>(
    private val innerSet: MutableSet<T> = mutableSetOf()
) : MutableSet<T> by innerSet{

    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>):Boolean {
        elementsAdded += elements.size
        return innerSet.addAll(elements)
    }
/**
* contains, isEmpty 등은 다른 인터페이스가 알아서 구현된다.
```

```
 * override fun contains() = innerSet.contains()
 /

 }
```

소결: 다형성이 필요한데 상속이 위험한 경우는 위임 패턴 사용, 다형성이 필요 없으면 composition 사용.

오버라이딩 제한

- open 키워드로 오버라이드를 허용할 수 있다
- 서브타입에서 오버라이드 한 후 해당 메서드가 더이상 오버라이딩 목적이 아니라면 final 키워드로 제한한다.

정리

1. 컴포지션은 안전하다. 외부에서 관찰되는 동작에만 의존하므로 안전하다. (상속에 의해 캡슐화가 깨질 여지가 없다)
2. 컴포지션은 유연하다.
    - 여러 클래스를 대상으로 할 수 있고,
    - 필요한 것만 받을 수 있고,
    - 변경에 제한적이다.
    - (사견: 컴포지션한 객체를 자유롭게 사용할 수 있다)
3. 컴포지션은 명시적이다
    - 리시버를 명시적으로 활용할 수밖에 없으므로 메서드가 어디에 있는 것인지 확실하게 알 수 있다
    - 코드가 길어지긴 하는데 혼동되지 않으므로 안전함.
4. 컴포지션은 생각보다 번거롭다
    - 객체를 명시적으로 사용해야 하므로 상속보다 코드를 수정하는 경우가 많다

5. 상속은 다형성을 활용할 수 있지만 양날의 검이다. 상속을 사용할 경우 슈퍼클래스와 서브클래스의 규악을 잘 지켜야 한다.
6. 일반적으로는 상속보다 컴포지션이 좋다. 상속은 명확한 IS-A관계일 때 사용한다. (슈퍼클래스의 모든 단위 테스트는 서브클래스로도 통과해야한다, LSP).