4. 클래스, 객체, 인터페이스

> - 클래스와 인터페이스
> - 뻔하지 않은 생성자와 프로퍼티
> - 데이터 클래스
> - 클래스 위임
> - object 키워드 사용

4.1 클래스 계층 정의

코틀린에서 클래스 계층을 정의하는 방식과 자바 방식을 비교한다.

4.1.1 코틀린 인터페이스

- 자바 8 인터페이스와 비슷하다
- 추상메소드 뿐만 아니라 구현이 있는 메소드도 정의할 수 있다(자바 8의 `default`와 비슷하다)
- 인터페이스에는 아무런 상태(필드)도 들어갈 수 없다

```
interface Clickable{
    fun click()
}
```

인터페이스를 구현하는 방법이다

```
class Button : Clickable{
    override fun click() = println("I was clicked")
}

>> Button().click()
I was clicked
```

코틀린에서는 클래스 이름 뒤에 콜론(`:`)을 붙이고 인터페이스 또는 클래스 이름을 적는것으로 클래스 확장과 인터페이스 구현을 모두 처리한다.

	자바	코틀린
상속	extends	:
구현	implements	:

자바와 마찬가지로 클래스확장(상속)은 한개만, 인터페이스는 개수 제한이 없이 마음대로 구현할 수 있다.

다음은 오버라이드를 하는 방법이다

```
// Java
class Button implements Clickable{
    @Override
    public void click(){
        ...
```

```
    }
}

// Kotlin
// override 변경자를 사용하여 구현한다
class Button : Clickable{
    override fun click(){
        ...
    }
}
```

코틀린의 `override` 변경자는 자바의 @Override 어노테이션과 비슷하다.

- 상위 클래스나 상위 인터페이스에 있는 프로퍼티나 메소드를 오버라이드한다는 표시이다
- `override` 변경자를 꼭 사용해야한다
    - 실수로 상위 클래스의 메소드를 오버라이드하는 경우를 방지해준다

인터페이스 메소드도 디폴트 구현을 제공할 수 있다. 따로 특별한 키워드 없이 메소드 본문을 뒤에 작성하면 된다

```
interface Clickable{
    fun click()  // 일반 메소드 선언
    fun showOff() = println("I'm clickable!")   // 디폴트 구현이 있는 메소드
}
```

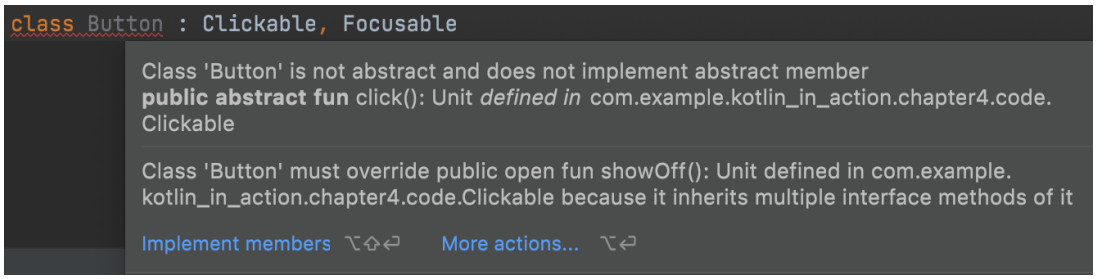
- 이 인터페이스를 구현하는 클래스는 click에 대한 구현을 제공해야한다
- showOff 메소드의 경우 새로운 동작은 정의할수도, 생략해서 디폴트로 사용할 수 있다.

위의 인터페이스와 동일하게 showOff 메소드를 정의하는 인터페이스가 있다고 한다면 어떻게 될까?

```
interface Focusable {
    fun setFocus(b:Boolean) = println("I ${if (b) "got" else "lost"} focus.")
    fun showOff() = println("I'm focusable!")
}
```

- [리스트 4.3]과 [리스트 4.4]는 동일한 디폴트 구현이 있는 showOff 메소드 중 어느쪽도 선택되지 않는다.

- 두 상위 인터페이스에 정의된 showOff 구현을 대체할 오버라이딩 메소드를 직접 제공해야 한다.
    - 그렇지 않다면 컴파일러 오류가 발생한다



`showOff` 메소드에 대해서 반드시 `override`하라고 나온다

다음과 같이 하위클래스에서 직접 구현하게 강제한다

```
class Button : Clickable, Focusable{
    override fun click() = println("I was clicked")

    // 이름과 시그니처가 같은 멤버 메소드에 대해
    // 둘 이상의 디폴트 구현이 있는 경우
    // 인터페이스를 구현하는 하위 클래스에서 명시적으로
    // 새로운 구현을 제공해야한다
    override fun showOff() {
        // 상위 타입의 이름을 꺽쇠 괄호(<>) 사이에 넣어서 "super"를 지정하면
        // 어떤 상위 타입의 멤버 메소드를 호출할지 지정할 수 있다
        super<Clickable>.showOff()
        super<Focusable>.showOff()
    }
}
```

Button 클래스는 이제 두 인터페이스를 구현하며, 상속한 두 상위 타입의 showOff 메소드를 호출하는 방식으로 구현했다.

자바	코틀린
Clickable.super.showOff	super<Clickable>.showOff

상속화 구현중 단 하나만 호출되도록 하려면 다음과 같이 쓰면 된다

```
override fun showOff() = super<Clickable>.showOff()
```

위의 작성한 인터페이스를 검증해 볼수 있다.

```
fun main(args: Array<String>) {

    val button = Button()
    button.showOff()
//  I'm clickable!
//  I'm focusable!
    button.setFocus(true)
//  I got focus.
    button.click()
//  I was clicked
}
```

4.1.2 open, final, abstract 변경자: 기본적으로 final

- 자바는 `final`로 클래스의 상속이나 메소드의 오버라이드를 금지 시킬 수 있다 (자바는 기본으로 Open 되어 있다)
- 기본적으로 상속이 가능하면 문제가 생기는 경우도 있다
  - **취약한 기반 클래스**
    - 기반클래스를 변경함으로써 하위 클래스가 깨져버리는 경우
    - 상속하는 방법에 대해 정확한 규칙을 제공하지 않는다면 작성한 사람의 의도와 다른 방식으로 메소드를 오버라이드할 위험이있다.

      > 상속을 위한 설계와 문서를 갖추거나, 그럴수 없다면 상속을 금지하라
      > Effective Java

    - 특별히 하위 클래스에서 오버라이드하게 **의도된 클래스와 메소드**가 아니라면 모두 final로 만들라는 뜻이다
- **코틀린의 클래스와 메소드는 기본적으로 `final` 이다**
  - 상속을 허용하려면 클래스 앞에 open 변경자를 붙여야한다
  - 메소드, 프로퍼티도 오버라이드를 허용하려면 open을 붙여야한다.

```
// 이 클래스는 열려있다. 다른 클래스가 이 클래스를 상속할 수 있다
open class RichButton : Clickable {
    // 이 함수는 final이다. 하위 클래스가 이 메소드를 오버라이드 할 수 없다
    fun disable(){}
    // 이 함수는 열려있다. 하위 클래스에서 이 메소드를 오버라이드 할 수 있다
    open fun animate(){}
    // 상위 클래스에서 선언된 메소드를 오버라이드 한다. 오버라이드한 메소드는 기본적으로 open이다
```

```
    override fun click() {}
}
```

- 기반 클래스나 인터페이스의 메소드를 오버라이드하는 경우 그 메소드는 기본적으로 open이다.
- 오버라이드한 메소드를 금지하려면 메소드 앞에 final을 명시해야한다

```
open class RichButton : Clickable {
    // 하위클래스에서 오버라이드를 금지 시킨다
    final override fun click() {}
}
```

추상클래스(abstract)

- 코틀린에서도 클래스를 abstract로 선언할 수 있다
- 추상 멤버는 항상 열려있다
  - 추상 멤버앞에 open 변경자를 명시할 필요가 없다

```
// 이 클래스는 추상클래스다. 이 클래스의 인스턴스를 만들 수 없다
abstract class Animated {
    // 이 함수는 추상함수이며, 구현이 없다
    // 하위 클래스에서는 이 함수를 반드시 오버라이드 해야한다
    abstract fun animate()

    // 비추상 함수는 기본적으로 파이널이지만
    // 원한다면 open으로 오버라이드를 허용할 수 있다
    open fun stopAnimating(){}
    fun animateTwice(){}
}
```

인터페이스

- 인터페이스 멤버의 경우 `final`, `open`, `abstract`를 사용하지 않는다
- 인터페이스 멤버는 항상 열려 있으며 `final`로 변경할 수 없다
- 인터페이스 멤버에게 본문이 없으면 자동으로 추상 멤버가 되지만, 그렇더라도 `abstract` 키워드를 덧붙일 필요가 없다

[표 4.1] 클래스 내에서 상속 제어 변경자의 의미

변경자	오버라이드 가능여부	설명
final	오버라이드 할 수 **없음**	클래스 멤버의 기본 변경자
open	오버라이드 할 수 **있음**	반드시 open을 명시해야 오버라이드 할 수 있다
abstract	반드시 오버라이드해야 함	추상 클래스의 멤버에만 이 변경자를 붙일 수 있다
override	상위 클래스나 상위 인스턴스의 멤버를 오버라이드하는 중	오버라이드하는 멤버는 기본적으로 열려있다. 하위 클래스의 오버라이드를 금지하려면 `final`을 명시해야 한다

4.1.3 가시성 변경자: 기본적으로 공개

코틀린의 기본 가시성은 아무 변경자도 없는 경우에 모두 `public`이다

코틀린은 패키지를 네임스페이스를 관리하기 위한 용도로만 사용한다. 패키지를 가시성 제어에 사용하지 않는다

패키지 전용 가시성에 대한 대안으로 코틀린에는 `internal` 이라는 새로운 가시성 변경자를 도입했다

`internal`은 모듈 내부에서만 볼 수 있음이라는 뜻이다

코틀린에서는 최상위 선언에 대해 `private`를허용한다

- 클래스, 함수, 프로퍼티등이 포함된다

`private`인 최상위 선언은 그 선언이 들어있는 파일 내부에서만 사용할 수 있다

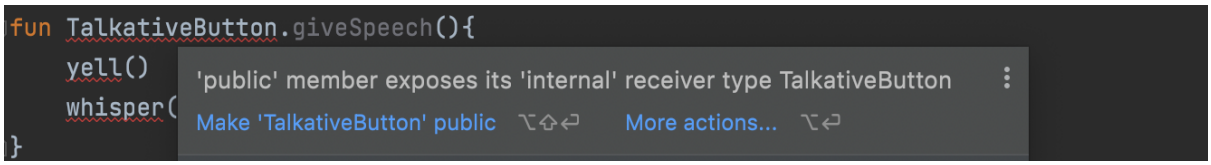
[표 4.2] 코틀린의 가시성 변경자

변경자	클래스 멤버	최상위 선언
public	모든 곳에서 볼 수 있다	모든 곳에서 볼 수 있다
internal	같은 모듈 안에서만 볼 수 있다	같은 모듈 안에서만 볼 수 있다
protected	하위 클래스 안에서만 볼 수 있다	사용 불가
private	같은 클래스 안에서만 볼 수 있다	같은 파일 안에서만 볼 수 있다

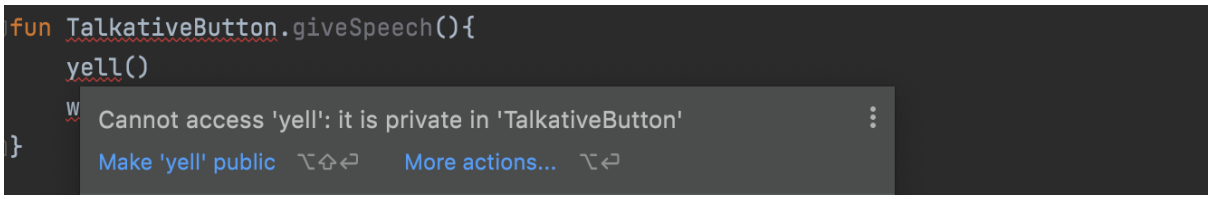
```
internal open class TalkativeButton : Focusable {
    private fun yell() = println("Hey!")
    protected fun whisper() = println("Let's talk!")
}

fun TalkativeButton.giveSpeech(){
    yell()
    whisper()
}
```

```
fun TalkativeButton.giveSpeech(){
    yell()
    whisper()
}
```

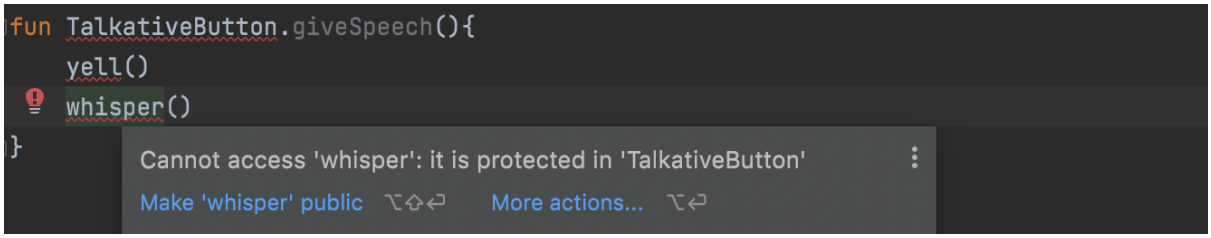
오류가 발생한다



public 멤버가 자신의 internal 수신타입인 "TalkativeButton"을 노출함



yell에 접근할 수 없음 : TalkativeButton의 private 멤버



whisper에 접근할 수 없음 : TalkativeButton의 protected 멤버

어떤 클래스의 기반 타입목록에 들어있는 타입이나 제네릭 클래스의 타입 파라미터에 들어있는 타입의 가시성은 그 클래스 자신의 가시성과 같거나 더 높아야한다

코틀린의 protected

자바는 같은 패키지 안에서 protected 멤버에 접근할 수 있지만, 코틀린은 그렇지 않다

코틀린 `protected` **멤버는 오직 어떤 클래스나 그 클래스를 상속한 클래스 안에서만 보인다**

- 클래스를 확장한 함수는 그 클래스의 `private` 나 `protected` 멤버에 접근할 수 없다 [참고]

4.1.4 내부 클래스와 중첩된 클래스: 기본적으로 중첩 클래스

- 자바와 마찬가리로 클래스안에 다른 클래스를 선언할 수 있다

- 자바와의 차이는 코틀린의 중첩 클래스(nested class)는 명시적으로 요청하지 않는 한 바깥쪽 클래스 인스턴스에 대한 접근권한이 없다

```
interface State : Serializable

interface View {
    fun getCurrentState() : State
    fun restoreState(state:State)
}

class Button : View {
    override fun getCurrentState(): State = ButtonState()
    override fun restoreState(state: State) {}
    class ButtonState : State  // 자바의 정적(static) 중첩 클래스와 대응한다
}
```

- 코틀린 중첩 클래스에 아무런 변경자가 붙지 않으면 자바 `static` 중첩 클래스와 같다
- 바깥쪽 클래스에 대한 참조를 포함하게 만들고 싶다면 `inner` 변경자를 붙여야한다

클래스 B안에 정의된 클래스 A	자바	코틀린
중첩 클래스(바깥쪽 클래스에 대한 참조를 저장하지 않음)	static class A	class A
내부 클래스(바깥쪽 클래스에 대한 참조를 저장함)	class A	inner class A

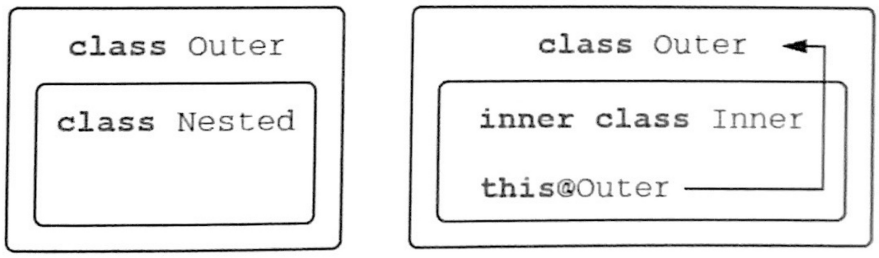


그림 4.1 중첩 클래스 안에는 바깥쪽 클래스에 대한 참조가 없지만 내부 클래스에는 있다.

- 내부 클래스 `Inner` 안에서 바깥쪽 클래스 `Outer`의 참조에 접근하려면 this@Outer라고 써야한다

```
class Outer{
    inner class Inner{
        fun getOuterReference() : Outer = this@Outer
    }
}
```

4.1.5 봉인(sealed)된 클래스: 클래스 계층 정의 시 계층 확장 제한

- 앞서 만들었던 식을 표현하는 계층을 참고

```
interface Expr
class Num(val value : Int):Expr
class Sum(val left:Expr, val right:Expr):Expr

fun eval(e: Expr) : Int =
    when(e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        // else 분기가 꼭 있어야한다
        else -> throw IllegalArgumentException("Unknown expression")
    }
```

코틀린 컴파일러는 when을 사용해 Expr 타입의 값을 검사할 때, 꼭 디폴트 분기인 else 분기를 덧붙이게 강제한다. **왜냐하면 컴파일러가 Expr를 구현한 모든 클래스를 알지 못하기 때문이다.**

새로운 하위 클래스가 추가 된다고 해도 컴파일러가 when의 모든 경우를 처리 하는지 확인 할 수 없다. 오히려 새로운 클래스 처리를 잊어버린다면 디폴트 분기가 선택되기 때문에 심각한 버그가 발생할 수 있다.

```
// 아래와 같이 새로운 하위클래스가 생성
class Minus(val left:Expr, val right:Expr):Expr

// 아래의 when에 Minus의 클래스 처리를 잊었다
fun eval(e: Expr) : Int =
    when(e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        // Minus의 처리를 잊었다
        // is Minus -> evel(e.right) - eval(e.left)
        else -> throw IllegalArgumentException("Unknown expression")
    }

>>>
// 디폴트 분기로 처리되어버린다.
java.lang.IllegalArgumentException: Unknown expression
```

- 이런 문제를 해결하기 위해서는 `sealed` 클래스를 사용하면 된다

  sealed로 기반 클래스를 봉인한 상태에서 eval 함수를 보면 else에 다음과 같이 안내가 뜬다

when은 (빠진것없이) 철저하기때문에 else는 여기서 불필요합니다

```
sealed class Expr{  // 기반 클래스를 sealed로 봉인한다
    // 하위 클래스를 중첩 클래스로 나열한다
    class Num(val value : Int):Expr
    class Sum(val left:Expr, val right:Expr):Expr
    class Minus(val left:Expr, val right:Expr):Expr
}

fun eval(e: Expr) : Int =
    // "when"식이 모든 하위 클래스를 검사하므로 별도의 else 분기가 없어도 된다
    when(e) {
        is Expr.Num -> e.value
        is Expr.Sum -> eval(e.right) + eval(e.left)
        is Expr.Minus -> eval(e.right) - eval(e.left)
    }
```

- `when`식에서 `sealed` 클래스의 모든 하위 클래스를 처리한다면 디폴트 분기가 필요 없다
- `sealed`로 표시된 클래스는 자동으로 `open`이 된다
- 내부적으로 Expr 클래스는 private 생성자를 가진다. (클래스 내부에서만 호출할 수 있다)
- 같은 패키지의 자식 클래스만 상속 가능하다
- sealed 클래스는 추상 클래스로 직접 인스턴스화가 불가능하다

Seald types cannot be instantiated

4.2 뻔하지 않은 생성자와 프로퍼티를 갖는 클래스 선언

- 코틀린은 **주(primary) 생성자**와 **부(secondary) 생성자**를 구분한다.
- **초기화 블록**(initializer block)을 통해 초기화로직을 추가할 수 있다

4.2.1 클래스 초기화

- 클래스를 선언하는 방법

```
class User(val nickname : String)
```

- 중괄호(`{}`)가 없고 괄호 사이에 val 선언만 존재한다 → ***주 생성자***
- 위의 클래스를 명시적으로 풀어서 실제로 아래와 같다

```
class User constructor (_nickname : String){
    val nickname: String

    // 초기화 블록
    init {
        nickname = _nickname
    }
}
```

constructor 키워드

주 생성자나 부 생성자 정의를 시작할 때 사용한다

init 키워드

초기화블록을 시작하며, 클래스의 객체가 만들어질때(인스턴스화 될때) 실행될 초기화 코드가 들어간다

주 생성자와 함께 사용된다

주 생성자는 제한적이기 때문에 별도의 코드를 포함할 수 없으므로 초기화 블록이 필요하다

이 예제는 nickname 프로퍼티를 초기화하는 코드를 nickname 프로퍼티 선언에 포함시킬 수 있어서 초기화 코드를 넣을 필요가 없다.

```
class User(_nickname: String){  // 파라미터가 하나뿐인 주 생성자
    val nickname = _nickname    // 프로퍼티를 주 생성자의 파라미터로 초기화 한다
}

// val은 이 파라미터에 상응하는 프로퍼티가 생성된다는 뜻이다
class User(val nickname: String)
```

함수 파라미터와 마찬가지로 생성자 파라미터에도 디폴트 값을 정의할 수 있다

```
class User(val nickname: String,
           val isSubscribed: Boolean = true)

>>> val hyun = User("현석") // isSubscribed은 디폴트(true)값이 사용된다
>>> println(hyun.isSubscribed)
true

>>> val gye = User("계영", false) // 모든 인자를 파라미터 선언 순서대로 지정할 수도 있다.
>>> println(gye.isSubscribed)
false

>>> val hey = User("혜원", isSubscribed = false) // 생성자 인자 중 일부에 대해 이름을 지정할 수 있다
>>> println(hey.isSubscribed)
false
```

클래스에 기반 클래스가 있다면 주 생성자에서 기반 클래스의 생성자를 호출해야할 필요가 있다

```
open class User(val nickname: String) {...}
class TwitterUser(nickname: String) : User(nickname) {...}
```

클래스를 정의할 때 별도로 생성자를 정의하지 않으면 컴파일러가 자동으로 인자가 없는 디폴트 생성자를 만들어준다.

```
open class Button // 인자가 없는 디폴트 생성자가 만들어진다

// Button의 생성자는 아무인자도 받지 않지만 하위클래스는 반드시 Button 클래스의 생성자를 호출해야한다
class RadioButton: Button()
```

클래스 정의에 있는 상위 클래스 및 인터페이스 목록에서 이름뒤에 괄호 유무로 기반클래스와 인터페이스를 구분가능하다

```
//Button은 기반 클래스, View는 인터페이스다
class RadioButton: Button(), View
```

어떤 클래스를 클래스 외부에서 인스턴스화하지 못하게 막고 싶다면 모든 생성자를 private로 만들면된다

```
// 이 클래스의 (유일한) 주 생성자는 비공개다.
class Secretive private constructor() {}
```

4.2.2 부 생성자: 상위 클래스를 다른 방식으로 초기화

자바에서 오버로드한 생성자가 필요한 상황중 상당수는 코틀린의 **디폴트 파라미터 값**과 **이름 붙인 인자 문법**을 사용해 해결할 수 있다.

그래도 생성자가 여러개 필요한 경우가 가끔있다. 일반적인 상황은 프레임워크 클래스를 확장해야 하는데 여러 가지 방법으로 인스턴스를 초기화 할 수 있게 다양한 생성자를 지원해야 하는 경우다

```
open class View{
    constructor(ctx: Context){
        // 코드
    }

    constructor(ctx: Context, attr: AttributeSet){
        // 코드
    }
}
```

주 생성자를 선언하지 않고 부 생정자만 2가지 선언한다.

부 생성자는 constructror 키워드로 시작하며, 필요에 따라 얼마든지 부 생성자를 선언해도 된다.

위의 클래스를 확장하면서 똑같이 부 생성자를 정의할 수 있다

```
class MyButton: View{
    // 상위 클래스의 생성자를 호출한다
```

```
    constructor(ctx: Context)
    :super(ctx){
        // 코드
    }
    // 상위 클래스의 생성자를 호출한다
    constructor(ctx: Context, attr: AttributeSet)
    :super(ctx, attr){
        // 코드
    }
}
```

자바와 마찬가지로 생성자에서 this()를 통해 클래스 자신의 다른 생성자를 호출할 수 있다

```
class MyButton : View{
    // 이 클래스의 다른 생성자에게 위임한다
    constructor(ctx: Context): this(ctx, MY_STYLE){...}
}
```

클래스에 주 생성자가 없다면 모든 부 생성자는 반드시 상위 클래스를 초기화 하거나 다른 생성자에 생성을 위임해야한다.

4.2.3 인터페이스에 선언된 프로퍼티 구현

코틀린에서는 **인터페이스에 추상 프로퍼티 선언**을 넣을 수 있다

```
interface User {
    val nickname: String
}
```

이는 User 인터페이스를 구현하는 클래스가 nickname의 값을 얻을 수 있는 방법을 제공해야한다는 뜻이다

```
// nickname을 저장하기만 한다.
// 주생성자에 있는 프로퍼티
class PrivateUser(override val nickname: String) : User

// email을 함께 저장한다
// 커스텀 Getter
class SubscribingUser(val email: String) : User {
    override val nickname: String
        get() = email.substringBefore('@')
}

// 페이스북 계정 ID를 저장한다
```

```
// 프로퍼티 초기화 식, getFacebookName(이 함수는 다른곳에 정의되어있다고 가정)
class FacebookUser(val accoundId: Int) : User{
    override val nickname = getFacebookName(accoundId)
}
```

`SubscribingUser`의 nickname은 매번 호출될 때마다 `substringBefore`를 호출해 계산하는 커스텀 Getter를 사용하고 `FacebookUser`의 nickname은 객체 초기화 시 계산한 데이터를 뒷받침하는 필드에 저장했다가 불러오는 방식이다.

인터페이스에는 게터와 세터가 있는 프로퍼티를 선언할 수 있다.

- 게터와 세터를 뒷받침하는 필드를 참조할 수 없다. 인터페이스는 상태를 저장할 수 없기 때문이다

```
interface User {
    val email: String

    // 프로퍼티에 뒷받침하는 필드가 없다
    // 대신 매번 결과를 계산하여 돌려준다
    val nickname: String
      get() = email.substringBefore('@')
}
```

하위 클래스는 추상 프로퍼티인 `email`을 **반드시 오버라이드 해야**하며, `nickname`은 오버라이드하지 않고 상속 할 수 있다.

4.2.4 게터와 세터에서 뒷받침하는 필드에 접근

프로퍼티에 저장된 값의 변경 이력을 로그에 남기려는 경우를 생각해 볼때 변경 가능한 프로퍼티를 정의하되 세터에서 프로퍼티 값을 바꿀 때마다 약간의 코드를 추가로 실행해야 한다

```
class User(val name: String){
    var address: String = "unspecified"
        // 뒷받침하는 필드 값 읽기
        set(value: String){
            println("""
                Address was changed for $name : "$field" -> "$value
            """.trimIndent())
            // 뒷받침하는 필드 값 변경하기
            field = value
        }
}

>>> val user = User("Alice")
>>> user.address = "Elsenheimerstrasse 47, 80687 Muenchen"
Address was changed for Alice : "unspecified" -> "Elsenheimerstrasse 47, 80687 Muenchen
```

이 예제는 커스텀 세터를 정의해서 추가 로직을 실행한다

접근자의 본문에는 field라는 특별한 식별자를 통해 뒷받침하는 필드에 접근 할 수 있다.

`Getter`에서는 `field` 값을 읽을 수만 있고, `Setter`에서는 `field` 값을 읽거나 쓸수 있다.

4.2.5 접근자의 가시성 변경

접근자의 가시성은 기본적으로 프로퍼티의 가시성과 같다

원한다면 get이나 set 앞에 가시성 변경자를 추가해서 접근자의 가시성을 변경할 수 있다

```
class LengthCounter {
    var counter: Int = 0
      private set  // 이 클래스 밖에서 이 프로퍼티의 값을 바꿀 수 없다

    fun addWord(word: String) {
      counter += word.length
    }
}


>>> val lengthCounter = LengthCounter()
>>> lengthCounter.addWord("Hi!")
>>> println(lengthCounter.counter)
3
```

4.3 컴파일러가 생성한 메소드: 데이터 클래스와 클래스 위임

- 자바는 equals, hashCode, toString 등의 메소드를 구현해야한다

  → 보통 IDE를 통해서 자동으로 만들어 줄 수있다

  → 자동으로 생성한다고해도 코드가 복잡해지는것은 동일하다

- 코틀린 컴파일러는 이런 메소드를 보이지 않는 곳에서 해준다

  → 복잡함 없이 소스코드를 깔끔하게 유지할 수 있다

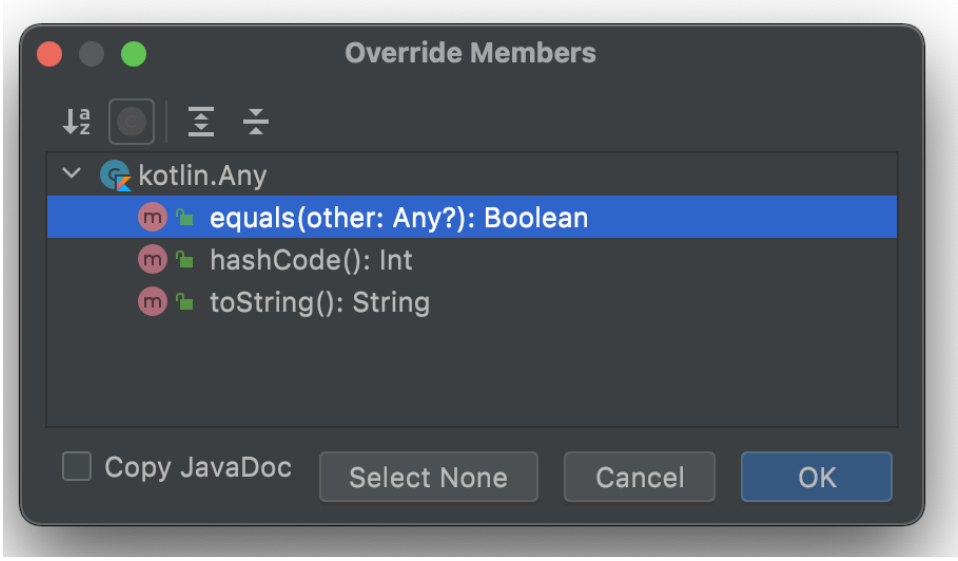
4.3.1 모든 클래스가 정의해야 하는 코드

- 코틀린도 toString, equals, hashCode 등을 오버라이드 할 수있다

```
class Client(val name: String, val postalCode: Int)
```

문자열 표현: toString()

- 주로 디버깅과 로깅 시 이 메소드를 사용한다
- 기본 제공되는 객체의 문자열 표현은 Client@5e9f23b4 같이 나타난다
- 이 기본 구현을 바꾸려면 toString 메소드를 오버라이드 해야한다



IDE를 쓴다면 쉽게 쓰자..

```
class Client(val name: String, val postalCode: Int){
    override fun toString() = "Client (name=$name, postalCode=$postalCode)"
}

>>> val client1 = Client("라이언", 4122)
>>> println(client1)
Client (name=라이언, postalCode=4122)
```

객체의 동등성: equals()

- 서로 다른 두 객체가 내부에 동일한 데이터를 포함하는 경우 그 둘을 **동등한 객체**로 간주해야 할 수도 있다

```
>>> val client1 = Client("라이언", 4122)
>>> val client2 = Client("라이언", 4122)
// 코틀린에서 == 연산자는 참조 동일성을 검사하지 않고 객체의 동등성을 검사한다
// 따라서 == 연산은 equals를 호출하는 식으로 컴파일된다
>>> println(client1 == client2)
false
```

※ 자바와 코틀린의 비교연산

- 자바의 ==
  - 원시타입일때는 값이 같은지 비교(동등성)
  - 참조타입(객체)일때는 주소가 같은지 비교(참조비교)
    - 두 객체의 동등성 비교시에는 `equals`를 사용해야한다
- 코틀린의 ==
  - 원시타입, 참조타입(객체) 둘다 동등성을 비교한다
    - 참조타입(객체)일때는 내부적으로 `equals`를 호출해서 비교한다
  - 참조비교를 하기 위해서는 `===` 연산자를 사용한다

```
class Client(val name: String, val postalCode: Int){
    // "Any"는 java.lang.Object에 대응하는 클래스
    // 코틀린의 모든 클래스의 최상위 클래스다
    override fun equals(other: Any?): Boolean {
        if (other == null || other !is Client)
            return false
        return name == other.name &&
                postalCode == other.postalCode
    }

    override fun toString() = "Client (name=$name, postalCode=$postalCode)"
}

>>> val client1 = Client("라이언", 4122)
>>> val client2 = Client("라이언", 4122)
>>> println(client1 == client2)
true
```

- 코틀린의 `is` 검사는 자바의 `instanceof`와 같다
- 오버라이드 후 프로퍼티의 값이 모두 같은 두 고객 객체는 동등하다고 예상할 수 있다
  - 하지만 더 복잡한 작업을 수행해보면 제대로 작동하지 않는 경우가 있다(~~hashSet이라던가..~~)
  - hashCode정의가 없어서 그렇다

해시 컨테이너: hashCode()

- 자바에서는 equals를 오버라이드할 때 반드시 hashCode도 함께 오버라이드 해야한다
- 다음 예제를 보면 `true`가 나올것이라 예상하지만 `false`가 튀어나온다

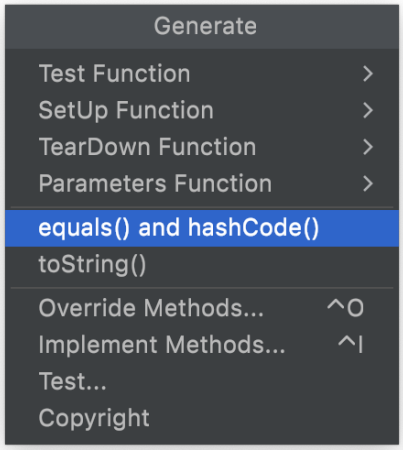
```
>>> val processed = hashSetOf(Client("라이언", 4122))
>>> println(processed.contains(Client("라이언", 4122)))
```

```
false
```

- Client 클래스가 hashCode 메소드를 정의하지 않았기 때문이다
- JVM 언어에서는 hashCode가 지켜야하는 제약조건이 있다

> equals()가 true를 반환하는 두객체는 반드시 같은 hashCode()를 반환해야 한다

- 그래서 보통 IDE에서 다음과 같이 같이 구현하라고도 이야기해준다



- `HashSet`은 원소를 비교할 때 비용을 줄이기 위해서
  1. 객체의 해시 코드를 비교하고
  2. 해시코드가 같은 경우에만 실제 값을 비교한다
- 이 문제를 해결하기 위해서는 Client 클래스에 `hashCode`를 구현해야 한다

```
class Client(val name: String, val postalCode: Int){
    ...
    override fun hashCode(): Int = name.hashCode() * 31 + postalCode
```

```
}

>>> val processed = hashSetOf(Client("라이언", 4122))
>>> println(processed.contains(Client("라이언", 4122)))
true
```

▼ 번외 : 왜 hashCode를 구할때 31을 곱할까???

> **31은 소수이면서 홀수이기 때문에 선택된 값이다.** 만일 그 값이 짝수였고 곱셈 결과가 오버플로되었다면 정보는 사라졌을 것이다. 2로 곱하는 것은 비트를 왼쪽으로 shift하는 것과 같기 때문이다. **소수를 사용하는 이점은 그다지 분명하지 않지만 전통적으로 널리 사용된다.** 31의 좋은 점은 곱셈을 시프트와 뺄셈의 조합으로 바꾸면 더 좋은 성능을 낼 수 있다는 것이다(`31 * i`는 `(i << 5) - i` 와 같다). 최신 VM은 이런 최적화를 자동으로 실행한다

그러나 소수를 사용하는 이점은 분명하지 않으며, "전통적으로 널리 사용된다"고 한다. 즉 관행이라고만 언급하고 있다.

4.3.2 데이터 클래스: 모든 클래스가 정의해야 하는 메소드 자동 생성

데이터 클래스(data class)

- 어떤 클래스가 데이터를 저장하는 역할만을 수행한다면 toString, equals, hashCode를 반드시 오버라이드 해야한다.
- IDE에서 자동으로 만들어주는 기능이 있어 손쉽게 작성할 수 있다
- **코틀린은 data 변경자를 클래스 앞에 붙이면 컴파일러가 자동으로 만들어준다**
- data 변경자가 붙은 클래스를 **데이터 클래스**하고 부른다

```
data class Client(val name:String, val postalCode: Int)
```

위의 Client 클래스는 자바에서 요구하는 모든 메소드를 포함한다

- 인스턴스 간 비교를 위한 `equals`
- HashMap과 같은 해시 기반 컨테이너에서 키로 사용할 수 있는 `hashCode`
- 클래스의 각 필드를 선언 순서대로 표시하는 문자열 표현을 만들어주는 `toString`

`equals`와 `hashCode`는 주 생성자에 나열된 모든 프로퍼티를 고려해 만들어진다

- `equals` 메소드는 모든 프로퍼티 값의 동등성을 확인한다
- `hashCode` 메소드는 모든 프로퍼티의 해시 값을 바탕으로 계산한 해시 값을 반환한다
- **주 생성자 밖에 정의된 프로퍼티는 equals나 hashCode를 계산할 때 고려의 대상이 아니다**

데이터 클래스와 불변성: copy() 메소드

- 데이터 클래스는 모든 프로퍼티를 읽기 전용으로 만들어서 불변(immutable) 클래스로 만들길 권장한다
  - HashMap등의 컨테이너에 데이터 클래스르 객체를 담는 경우에는 불변성이 필수적이다.
  - 변경점이 없으므로 저장된 데이터에 대해 훨씬 쉽게 추론할 수 있다
  - 다중스레드 프로그램의 경우 이런 성질은 더 중요하다
- 데이터 클래스 인스턴스를 불변객체로 더 쉽게 활용할 수 있게 코틀린 컴파일러는 `copy` 메소드를 제공한다
  - 객체를 복사(copy)하면서 일부 프로퍼티를 바꿀 수 있게 해준다
  - 객체를 모메리상에서 직접 바꾸는 대신 복사본을 만드는 편이 더 낫다
  - 원본과 다른 생명주기를 가지며, 프로퍼티 값을 바꾸거나 복사본을 제거해도 원본에 전혀 영향을 끼치지 않는다

copy를 직접 구현했을 경우는 아래와 같다

```
class Client(val name: String, val postalCode: Int){
    ...
    fun copy(name:String = this.name, postalCode: Int = this.postalCode)
      = Client(name, postalCode)
}
```

copy 메소드를 사용하는 방법이다

```
>>> val lee = Client("이계영", 4122)
>>> println(lee.copy(postalCode = 4000))
Client (name=이계영, postalCode=4000)
```

4.3.3 클래스 위임: by 키워드 사용

- 대규모 객체지향 시스템 설계시 시스템을 취약하게 만드는 문제는 보통 **구현 상속에 의해 발생**한다
- 코틀린을 설계하면서 이런 문제를 인식하여 모든 클래스를 `final`로 취급하기로 했다고 한다
  - 기본은 `final`이며 상속을 염두에 두고 `open`변경자로 열어둔 클래스만 확장할 수 있다
  - `open`변경자를 보고 상속했을거라 예상하여 좀 더 조심할 수 있다

- 종종 상속을 허용하지 않는 클래스에 새로운 동작을 추가해야 할 때가 있다.
  - 일반적으로 사용하는 방법이 데코레이터(Decorator) 패턴
  - 이런 접근방법의 단점은 준비 코드가 상당히 많이 필요하다

> 📎 필요한 준비코드가 너무 많기 때문에 IntelliJ IDEA 등의 IDE는 데코레이터 준비코드를 자동으로 생성해주는 기능을 제공한다

  - 예로 Collection 같이 비교적 단순한 인터페이스를 구현하면서 아무 동작도 변경하지 않는 데코레이터를 만들 때 아래와같이 복잡한 코드를 작성해야한다.

```
class DelegatingCollection<T> : Collection<T>{
    private val innerList = arrayListOf<T>()

    override val size: Int
        get() = innerList.size

    override fun contains(element: T): Boolean
      = innerList.contains(element)

    override fun containsAll(elements: Collection<T>): Boolean
      = innerList.containsAll(elements)

    override fun isEmpty(): Boolean = innerList.isEmpty()

    override fun iterator(): Iterator<T> = innerList.iterator()
}
```

- 이런 위임을 언어가 제공하는 일급 시민 기능으로 지원한다는 것이 코틀린의 장점이다

- 인터페이스를 구현할 때 by 키워드를 통해 그 인터페이스에 대한 구현을 다른 객체에 위임 중이라는 사실을 명시할 수 있다
- 위의 코드를 위임을 사용하여 재작성 했을 때 아래와 같이 작성된다

```
class DelegatingCollection<T>(
    innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList
```

- 메소드 중 일부의 동작을 변경하고 싶은 경우 메소드를 오버라이드하면 컴파일러가 생성한 메소드 대신 오버라이드한 메소드가 쓰인다
- 기존 클래스의 메소드에 위임하는 기본 구현으로 충분한 메소드는 따로 오버라이드할 필요가 없다
- 이러한 기법을 이용해 다음과 같은 컬렉션을 구현할 수 있다

```
class CountingSet<T>(
    private val innerSet : MutableCollection<T> = HashSet<T>()
) : MutableCollection<T> by innerSet{

    var objectsAdded = 0

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

    override fun addAll(elements: Collection<T>): Boolean {
        objectsAdded += elements.size
        return innerSet.addAll(elements)
    }
}

>>> val cset = CountingSet<Int>()
>>> cset.addAll(listOf(1,1,2))
>>> println("${cset.objectsAdded} objects were added, ${cset.size} remain")
3 objects were added, 2 remain
```

  - `add`와 `addAll`을 오버라이드해서 카운터를 증가시킨다
  - `MutableCollection` 인터페이스의 `add`와 `addAll`을 제외한 나머지 메소드는 내부 컨테이너(innerSet)에 위임한다.
  - 내부 클래스 MutableCollection이 변경되지 않는한 CountingSet코드는 계속 잘 작동할 것임을 확신할 수 있다

4.4 object 키워드: 클래스 선언과 인스턴스 생성

- 코틀린의 `object` 키워드는 다양한 상황에서 사용하며, 모든 경우 클래스를 정의하면서 동시에 인스턴스(객체)를 생성한다는 공통점이 있다
  - 객체 선언(object declaration)은 싱글턴(singleton)을 정의하는 방법 중 하나다
  - 동반 객체(companion object)는 인스턴스 메소드가 아니지만 어떤 클래스와 관련 있는 메소드와 팩토리 메소드를 담을 때 사용한다. 동반 객체 메소드에 접근할 때는 동반 객체가 포함된 클래스의 이름을 사용할 수 있다
  - 객체 식은 자바의 무명 내부 클래스(anonymous inner class) 대신 쓰인다

4.4.1 객체 선언 : 싱클턴을 쉽게 만들기

코틀린은 **객체 선언** 기능을 통해 싱글턴을 언어에서 기본 지원한다

- 객체 선언은 클래스 선언과 그 클래스에 속한 단일 인스턴스의 선언을 합친 선언이다

```
object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary(){
        for (person in allEmployees) {
            ...
        }
    }
}
```

- 객체 선언은 object 키워드로 시작한다
- 객체 선언은 클래스를 정의하고 그 클래스의 인스턴스를 만들어서 변수에 저장하는 모든 작업을 단 한문장으로 처리한다
- 객체 선언안에도 프로퍼티, 메소드, 초기화 블록 등이 들어갈 수 있다
- 하지만 생성자(주 생성자, 부 생성자)는 쓸 수 없다
  - 객체 선언문이 있는 위치에서 생성자 호출 없이 즉시 만들어진다
  - 그러므로 객체 선언에는 생성자 정의가 필요 없다.
- 변수와 마찬가지로 객체에 속한 메소드나 프로퍼티에 접근할 수 있다

```
Payroll.allEmployees.add(Person(...))
Payroll.calculateSalary()
```

객체 선언도 클래스나 인터페이스를 상속할 수 있다

- 구현 내부에 다른 상태가 필요하지 않은 경우에 이런 기능이 유용하다
- 두 파일 경로를 대소문자 관계없이 비교해주는 Comparator를 구현

```
object CaseInsensitiveFileComparator : Comparator<File>{
    override fun compare(file1: File, file2: File): Int {
        return file1.path.compareTo(file2.path, ignoreCase = true)
    }
}

>>>println(CaseInsensitiveFileComparator.compare(File("/User"), File("/user")))
0
```

일반 객체(클래스 인스턴스)를 사용할 수 있는 곳에서는 항상 싱글턴 객체를 사용할 수 있다

- 위의 객체를 Comparator를 파라미터로 받는 함수에게 파라미터로 넘길 수 있다

```
>>> val files = listOf(File("/Z"), File("/a"))
>>> println(files.sortedWith(CaseInsensitiveFileComparator))
[/a, /Z]
```

클래스 안에서 객체를 선언할 수도 있다.

- 이러한 객체도 인스턴스는 단 하나뿐이다
- 바깥 클래스의 인스턴스마다 중첩 객체 선언에 해당하는 인스턴스가 하나씩 따로 생기는 것이 아니다

예를 들어 어떤 클래스의 인스턴스를 비교하는 Comparator를 클래스 내부에 정의하는 것이 더 바람직하다

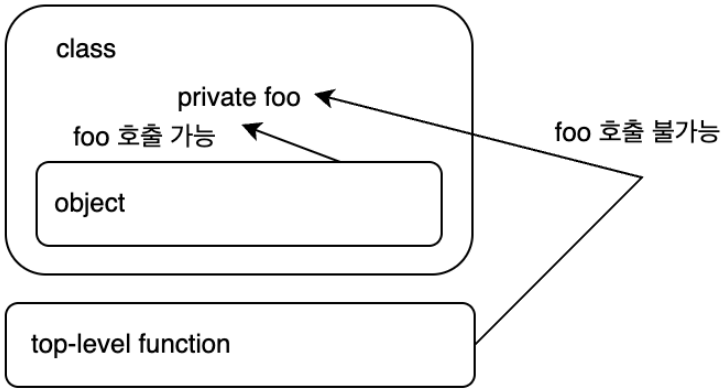
```
data class Person(val name:String){
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
          p1.name.compareTo(p2.name)
    }
}

>>> val persons = listOf(Person("Bob"), Person("Alice"))
>>> println(persons.sortedWith(Person.NameComparator))
[Person(name=Alice), Person(name=Bob)]
```

4.4.2 동반 객체: 팩토리 메소드와 정적 멤버가 들어갈 장소

코틀린 클래스 안에는 정적(`static`) 멤버가 없다

- 코틀린 언어는 자바 `static` 키워드를 지원하지 않는다
- 대신 패키지 수준의 최상위 함수와 객체 선언을 활용한다

하지만 최상위 함수는 `private`로 표시된 클래스 비공개 멤버에 접근할 수 없다.



클래스의 인스턴스와 관계 없이 호출해야할 경우에는 클래스 안에 정의된 객체중 하나에 `companion`이라는 특별한 표시를 붙이면 그 클래스를 동반 객체로 만들 수 있다.

- 동반 객체의 프로퍼티나 메소드에 접근하려면 동반 객체가 정의된 클래스 이름을 사용한다.
- 객체의 이름을 따로 지정할 필요가 없다
- 동반 객체의 멤버를 사용하는 구문은 자바의 정적 메소드 호출이나 정적 필드사용 구문과 같아진다

```
class A{
    companion object{
        fun bar(){
            println("Companion object called")
        }
    }
}

>>> A.bar()
Companion object called
```

동반 객체가 private 생성자를 호출하기 좋은 위치다

- 자신을 둘러싼 클래스의 모든 private 멤버에 접근할 수 있다
- 동반 객체는 바깥쪽 클래스의 private 생성자도 호출할 수 있다.
- 동반 객체는 팩토리 패턴을 구현하기 가장 적합한 위치다

아래 예제는

▼ 부생성자가 2개있는 클래스를 살펴보고

```
class User {
    private val nickname: String
    constructor(email:String){
        nickname = email.substringBefore('@')
    }
    constructor(facebookAccountId: Int){
        nickname = getFaceBookName(facebookAccountId)
    }

    private fun getFaceBookName(facebookAccountId: Int): String {
        return "Alice"
    }
}
```

▼ 이 클래스를 동반 객체안에서 팩토리 클래스를 정의하는 방식으로 변경한다

```
// 주 생성자를 비공개로 만든다
class User private constructor(val nickname: String){
    // 동반 객체를 선언한다
    companion object{
        fun newSubscribingUser(email:String) = User(email.substringBefore('@'))
        fun newFaceBookUser(accountId:Int) = User(getFacebookUser(accountId))

        private fun getFacebookUser(accountId: Int): String {
            return "Alice"
        }
    }
}

>>> val subscribingUser = User.newSubscribingUser("bob@gmail.com")
>>> val facebookUser = User.newFaceBookUser(4)
>>> println(subscribingUser.nickname)
>>> println(facebookUser.nickname)
bob
Alice
```

- 클래스를 확장해야만 하는 경우에는 **동반 객체 멤버를 하위 클래스에서 오버라이드할 수 없으므로** 여러 생성자를 사용하는 편이 더 나은 해법이다

4.4.3 동반 객체를 일반 객체처럼 사용

동반 객체에 이름을 붙이거나, 인터페이스를 상속하거나, 동반 객체 안에 확장 함수와 프로퍼티를 정의할 수 있다.

```
class Person(val name:String){
    companion object Loader {
        // 동반객체에 이름을 붙인다
        fun fromJSON(jsonText:String): Person{ ... }
    }
}

>>> val person = Person.Loader.fromJSON("{name:'Dmitry'}")
>>> person.name
Dmitry

>>> val person2 = Person.fromJSON("{name:'Brent'}")
>>> person.name
Brent
```

- 특별히 이름을 지정하지 않으면 동반 객체 이름은 자동으로 `Companion`이 된다

동반 객체에서 인터페이스 구현

- 동반 객체도 인터페이스를 구현할 수 있다
- 인터페이스를 구현하는 동반 객체를 참조할 때 객체를 둘러싼 클래스의 이름을 바로 사용할 수 있다

```
interface JSONFactory<T>{
    fun fromJSON(jsonText: String) : T
}

class Person(val name:String){
    companion object : JSONFactory<Person>{
        override fun fromJSON(jsonText: String): Person {
            return Person("Person $jsonText")
        }
    }
}

class Animal(val name:String){
    companion object : JSONFactory<Animal>{
        override fun fromJSON(jsonText: String): Animal {
            return Animal("Animal $jsonText")
        }
    }
}

fun <T> loadFromJSON(factory: JSONFactory<T>) : T {
    return factory.fromJSON("Bob")
}

// 동반 객체의 인스턴스를 함수에 넘긴다
```

```
>>> println(loadFromJSON(Person).name)
>>> println(loadFromJSON(Animal).name)
Person Bob
Animal Bob
```

동반 객체 확장

```
// 비즈니스 로직 모듈
class Person(val firstName:String, val lastName:String){
    // 비어있는 동반 객체를 선언한다
    companion object{}
}

// 클라이언트/서버 통신 모듈
// 확장 함수를 선언한다
fun Person.Companion.fromJSON(json:String) : Person{
    return Person("firstName $json", "lastName $json")
}

>>> val p = Person.fromJSON("Alice")
>>> println(p.firstName)
>>> println(p.lastName)
firstName Alice
lastName Alice
```

- 마치 동반 객체 안에서 fromJSON 함수를 정의한 것처럼 보이지만 **실제로 클래스 밖에서 정의한 확장 함수다**
- 동반객체에 대한 확장 함수를 작성하려면 **원래 클래스에 빈 객체라도 동반 객체를 꼭 선언해야 한다**

4.4.4 객체 식: 무명 내부 클래스를 다른 방식으로 작성

- 무명 객체(anonymous object)를 정의할 도 `object`키워드를 쓴다
- 자바의 무명 내부 클래스를 대신한다

아래는 이벤트 리스너를 코틀린에서 구현한것이다

```
window.addMouseListner(
    object : MouseAdapter(){
        override fun mouseClicked(e: MouseEvent?) {...}
        override fun mouseEntered(e: MouseEvent?) {...}
    }
)
```

- 객체 선언과 같지만 객체 이름이 빠졌다는 점이 다르다

- 객체 식은 클래스를 정의하고 그 클래스에 속한 인스턴스를 생성하지만, 그 클래스나 인스턴스에 이름을 붙이지는 않는다
- 보통 함수를 호출하면서 인자로 무명 객체를 넘기기 때문에 이름이 필요하지 않다. 이름을 붙여야 한다면 변수에 묶여 객체를 대입하면 된다

```
val listener = object : MouseAdapter(){
    override fun mouseClicked(e: MouseEvent) { ... }
    override fun mouseEntered(e: MouseEvent) { ... }
}
```

- 코틀린의 무명 클래스는 여러 인터페이스를 구현하거나 클래스를 확장하면서 인터페이스를 구현할 수 있다.

객체 선언과 달리 무명 객체는 **싱글턴이 아니다**. 객체 식이 쓰일 때마다 새로운 인스턴스가 생성된다

자바의 무명 클래스와 같이 객체 식 안의 코드는 그 식이 포함된 함수의 변수에 접근할 수 있다

하지만 자바와 다르게 `final`이 아닌 변수도 객체 식 안에서 사용할 수 있다. 따라서 객체 식 안에서 그 변수의 값을 변경할 수 있다

```
fun countClicks(window: Window){
    var clickCount = 0
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent?) {
            clickCount++
            super.mouseClicked(e)
        }
    })
}
```

객체 식은 무명 객체 안에서 여러 메소드를 오버라이드해야하는 경우에 훨씬 유용하다
메소드가 하나뿐인 인터페이스를 구현해야 한다면 코틀린의 SAM 변환 지원을 활용하는것이 좋다

4.5 요약

- 코틀린의 인터페이스는 디폴트 구현을 포함할 수 있고(자바8도 가능), 프로퍼티도 포함할 수 있다
- 모든 코틀린 선언은 기본적으로 `final`이며 `public`이다
- 상속과 오버라이딩이 가능하게 하려면 `open`을 붙여야한다
- `internal` 선언은 같은 모듈 안에서만 볼 수 있다
- 중첩 클래스는 내부 클래스가 아니다. 바깥쪽 클래스에 대한 참조를 중첩 클래스 안에 포함시키려면 inner 키워드를 중첩 클래스 선언 앞에 붙여서 내부 클래스로 만들어야한다
- ~~sealed 클래스를 상속하는 클래스를 정의하려면 반드시 부모 클래스 정의 안에 중첩(또는 내부) 클래스로 정의 해야한다.~~ (코틀린 1.1부터는 같은 파일 안에만 있으면 된다)
- 초기화 블록과 부 생성자를 활용해 클래스 인스턴스를 더 유연하게 초기화 가능하다
- field 식별자를 통해 프로퍼티 접근자(`Getter`, `Setter`) 안에서 프로퍼티의 데이터를 저장하는 데 쓰이는 뒷받침하는 필드를 참조할 수 있다.
- 데이터 클래스를 사용하면 `eqauls`, `hashCode`, `toString`, `copy` 등의 메소드를 자동으로 만들어 준다
- 클래스 위임을 사용하면 위임 패턴을 구현할 때 필요한 수많은 준비코드를 줄일 수 있다
- 객체 선언은 사용하면 코틀린 답게 싱글턴 클래스를 정의할 수 있다
- 동반 객체는 자바의 정적 메소드와 필드 정의를 대신한다
- 동반 객체도 다른 객체와 마찬가지로 인터페이스를 구현할 수 있다. 외부에서 동반 객체에 대한 확장 함수와 프로퍼티를 정의할 수 있다.
- 코틀린의 객체 식은 자바의 무명 내부 클래스를 대신한다. 하지만 자바의 무명 내부 클래스보다 더 많은 기능을 제공한다