Chapter04. 클래스, 객체, 인터페이스

4장에서 다루는 내용

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

클래스 계층 정의

코틀린에서 클래스 계층을 정의하는 방식을 살펴보자.

코틀린 인터페이스

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
}

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

- 추상 메서드뿐만 아니라 구현이 있는 메서드도 정의할 수 있다.
- 하지만 아무런 상태도 들어갈 수 없다.
- 상속이나 구성에서 모두 클래스 이름 뒤에 콜론을 붙이고 인터페이스나 클래스 이름을 적는 방식 사용
- 원하는 만큼 개수 제한 없이 구현가능하지만 클래스는 오직 하나만 확장할 수 있다.
- 상위 프로퍼티나 메서드 오버라이드할 경우 변경자 필수
    - `override`
    - 실수 방지

다이아몬드 문제: 같은 메서드를 구현하는 다른 인터페이스 정의

```
interface Clickable {
    fun click()

    fun showOff() = println("I'm clickable!")
}

interface Focusable {
    fun setFocus(b: Boolean) = println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm Focusable!")
}

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

}
```

- `showOff` 메서드는 어느 쪽도 선택되지 않고 대체할 오버라이딩 메서드를 직접 제공해야 한다.
- 그렇지 않으면 컴파일 에러

상속한 인터페이스의 메서드 구현 호출 하기

```
interface Clickable {
    fun click()

    fun showOff() = println("I'm clickable!")
}

interface Focusable {
    fun setFocus(b: Boolean) = println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm Focusable!")
}
```

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

    override fun showOff() {
        super<Clickable>.showOff()
        super<Focusable>.showOff()
    }
}
```

- 자바와 문법이 다르다.
    - `Clickable.super.showOff()`
- 코틀린은 디폴트 메서드가 있는 인터페이스를 일반 인터페이스와 디폴트 메서드 구현이 정적 메서드로 들어있는 클래스를 조합해 구현한다.
- 따라서 자바 클래스에서 코틀린 인터페이스를 구현하면 디폴트 구현을 제공하는 메서드여도 모든 메서드에 대한 구현을 제공해야한다.
    - (java6과 호환되므로)

open, final, abstract 변경자 : 기본적으로 final

- 코틀린에서 모든 클래스와 메서드는 기본적으로 final이다.
    - 클래스에 대해 하위 클래스 만들 수 없다.
    - 기반 클래스의 메서드를 하위 클래스가 오버라이드할 수 없다.
    - 자바와 반대, 왜일까?
- 편리한 반면 취약한 기반 클래스(fragile base class)
    - 기반 클래스 구현을 변경함으로써 하위 클래스가 잘못된 동작을 하게 되는 경우를 뜻한다.
    - 이펙티브 자바: 상속을 위한 설계와 문서를 갖춰라. 그럴 수 없다면 상속을 금지하라
- `open`
    - 클래스의 상속을 허용
    - 메서드나 프로퍼티의 오버라이드 허용
- `abstract`

- 추상클래스, 추상 멤버는 당연히 상속받아서 사용해야하기 때문에 open을 명시할 필요 없다.
- 항상 열려있다.

열린 메서드를 포함하는 열린 클래스

```
open class RichButton : Clickable {

    fun disable() {}

    open fun animate() {}

    override fun click() {}
}

class ThemedButton : RichButton() {
        override fun animate() { }
        override fun click() { }
        override fun showOff() { }

}
```

- `RichButton` : 열린 메서드를 포함하는 열린 클래스
- `ThemedButton` : 열린 메서드를 오버라이드하는 열린 클래스의 하위 클래스
- 기반 클래스나 인터페이스의 멤버를 오버라이드한 경우에는 기본적으로 open으로 간주된다.
  - 이것을 금지하려면 오버라이드된 메서드를 명시적으로 final로 표현해야함

오버라이드 금지하기

```
open class RichButton : Clickable {
    final override fun click() {}
}
```

💡 열린 클래스와 스마트 캐스트

클래스의 기본적인 상속 가능 상태를 final로 함으로써 다양한 경우에 스마트 캐스트가 가능하다. 스마트 캐스트는 타입 검사 뒤에 변경될 수 없는 변수에만 적용 가능하다.

클래스 프로퍼티의 경우 val이면서 커스텀 접근자가 없는 경우에만 스마트 캐스트를 쓸 수 있다는 의미다. 즉 프로퍼티가 final이어야 한다는 뜻

추상클래스 정의하기

```
abstract class Animated {
    abstract val animationSpeed: Double
    val keyFrames: Int = 20
    open val frames: Int = 60

    abstract fun animate()
    open fun stopAnimating() {}
    fun anumateTwice() {}

}
```

클래스 내에서 접근 변경자의 의미

변경자	이 변경자가 붙은 멤버는...	설명
final	오버라이드할 수 없음	클래스 멤버의 기본 변경자다.
open	오버라이드할 수 있음	반드시 open을 명시해야 오버라이드할 수 있다.
abstract	반드시 오버라이드해야 함.	추상 클래스의 멤버에만 이 변경자를 붙일 수 있다. 추상 멤버에는 구현이 있으면 안 된다.

override	상위 클래스나 상위 인스턴스의 멤버를 오버라이드하는 중	오버라이드하는 멤버는 기본적으로 열려 있다. 하위 클래스의 오버라이드를 금지하려면 final을 명시해야 한다.

가시성 변경자: 기본적으로 공개

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.	-
private	같은 클래스 안에서만 볼 수 있다.	같은 파일 안에서만 볼 수 있다.

- visibility modifier: 코드 기반에 있는 선언에 대한 클래스 외부 접근을 제어한다.
  - 패키지 전용 가시성 개념이 없다.
  - 코틀린은 패키지를 네임스페이스를 관리하기 위한 용도로만 사용
- public, protected, private 제공
  - 자바의 경우에 대응
  - 하지만 바이트 코드상 private class는 패키지 전용 클래스로 컴파일
  - 코틀린은 기본 public이다.
- protected
  - 자바에서는 같은 패키지 안에서 접근할 수 있지만 코틀린에서는 어떤 클래스나 그 클래스를 상속한 클래스안에서만 접근 가능하다.
- internal
  - 모듈안으로 한정된 가시성
    - 모듈은 한 번에 한꺼번에 컴파일되는 코틀린 파일들을 의미
    - 그레이들 메이븐 프로젝트나 인텔리제이 IDEA 모듈이 모듈일 수 있다.
    - gradle의 경우 테스트 소스 세트가 main의 internal 선언에 액세스할 수 있다는 점은 제외
  - 모듈 구현에 대한 진정한 캡슐화 제공
    - 자바의 package-private는 패키지가 같은 클래스를 선언하기만 하면 프로젝트 외부에 있는 코드라도 패키지 내부에 있는 패키지 전용 선언에 쉽게 접근할

수 있어 캡슐화가 쉽게 깨질 수 있다.

    - 바이트 코드상으로는 public
- 코틀린에서는 최상위 선언에 대해 private 가시성을 허용한다.
    - 클래스, 함수, 프로퍼티 등 포함
- 따라서 코틀린에서는 접근할 수 없지만 자바 코드에서는 접근 가능한 경우가 있다.
    - protected, internal

가시서 변경자 예제

```
internal open class TalkativeButton : Focusable{
  private fun yell() = ...
  protected fun whisper() = ...
}

fun TalkativeButton.giveSpeech(){// 오류: public 멤버가 자신의 internal 수신 타입
  yell() // private 접근 오류
  whisper() // protected 접근 오류
}
```

- 컴파일 오류를 없애려면
    - TalkativeButton 클래스를 public으로 변경
    - giveSpeech 확장 함수를 internal로 변경

내부 클래스와 내포된 클래스: 기본적으로 내포 클래스

클래스 B 안에 정의된 클래스 A	자바에서는	코틀린에서는
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

- 클래스 안에서 다른 클래스 선언
    - 도우미 클래스 캡슐화
    - 코드 정의를 그 코드를 사용하는 곳 가까이 두고 싶을 때

- 자바와 달리 nested class는 명시적으로 요청하지 않는 한 바깥쪽 클래스 인스턴스에 대한 접근 권한이 없다.
  - java의 `static inner class`
- 바깥쪽 클래스에 대한 참조를 포함하게 만들고 싶다면 inner 변경자
  - java의 `inner class`

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

봉인된 클래스: 확장이 제한된 클래스 계층 정의

- sealed 클래스는 상위 클래스를 상속한 하위 클래스의 가능성을 제한할 수 있다.
  - 하위 클래스들은 반드시 컴파일 시점에 알려져야 한다.
  - 하위 클래스들은 봉인된 클래스가 정의된 패키지와 같은 패키지에 속해야 한다.
  - 모든 하위 클래스들은 같은 모듈안에 위치해야 한다.
- sealed 변경자는 클래스가 추상 클래스임을 명시한다.
- 인터페이스도 가능하다.
  - 봉인된 인터페이스가 속한 모듈이 컴파일되고 나면 해당 인터페이스에 대한 새로운 구현을 밖에서 추가할 수 없다.

```
sealed class 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)
    }
```

- when 식이 모든 하위 클래스를 검사하므로 별도의 else 분기가 없어도 된다.
  - 강력한 when 식
- 근데 이게 왜 내포 클래스? top level 아닌가?
  - 오타같음

뻔하지 않은 생성자나 프로퍼티를 갖는 클래스 선언

- 코틀린은 주생성자, 부 생성자를 구분한다.
- 주 생성자
  - 본문 밖에서 정의
- 부 생성자
  - 본문 안에서 정의
- 초기화 블록으로 초기화 로직을 추가할 수 있다.

클래스 초기화: 주 생성자와 초기화 블록

```
class User constructor(_nickname: String) { // 파라미터 하나만 있는 주 생성자
    val nickname = String
    init {
        nickname = _nickname // 초기화 블록
    }
}

class User constructor(_nickname: String) {
    val nickname = _nickname
}

class User(val nickname: String) // val은 파라미터에 상응하는 프로퍼티 생성된다는
```

```
open class User(val nickname: String)
class SocialUser(nickname : String) : User(nickname){...} // 기반 클래스 초기화
open class Button // 인자가 없는 디폴트 생성자가 만들어진다.
```

```
class RadioButton : Button() {} // 상속받으면서 디폴트 생성자 호출
class Secretive private constructor() {} // 유일한 주 생성자는 private
```

- `constructor`: 주 생성자나 부 생성자 정의를 위해 사용한다.
- `init`: 인스턴스 생성시 실행되는 초기화 코드, 로직이 없는 주 생성자를 보완한다.
- 모든 생성자 파라미터에 디폴트 값을 지정하면 컴파일러가 자동으로 파라미터가 없는 생성자를 만들어준다.

부 생성자: 상위 클래스를 다른 방식으로 초기화

```
class MyDownloader: Downloader {
  constructor(url: String?): this((URI(url)) {
     // ...
   }
  }
  constructor(uri: URI?) : super(uri){
     // ...
  }
}
```

- 클래스 바디 내부에서 constructor 키워드를 사용해서 주 생성자 없이 부생성자만 여러 개 생성할 수 있다.
  - super()를 통해 자신에 대응하는 상위 클래스 생성자를 호출한다.
  - this()를 이용해서 클래스 자신의 다른 생성자를 호출할 수 있다.
- 부생성자는 자바와의 상호운용성, 파라미터 목록이 다른 생성 방법이 필요할 때 사용한다.
- 인자에 대한 기본값을 제공하기 위해 부 생성자를 여럿 만들지 말자.

인터페이스에 선언된 프로퍼티 구현

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

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

class SubscribingUser(val email: String) : User {
    override val nickname: String
        get() = email.substringBefore('@') // 커스텀 게터, 매번 계산
}

class SocialUser(val accountId: Int) : User {
    override val nickname = getNameFromSocialNetwork(accountId) // 프로퍼티
}

fun getNameFromSocialNetwork(accountId: Int) = "kodee$accountId"


fun main() {
    println(PrivateUser("kodee").nickname) // kodee
    println(SubscribingUser("test@kotlinlang.org").nickname) // test
    println(SocialUser(123).nickname) // kodee123
}
```

- 인터페이스에 추상 프로퍼티 선언 가능하다.

```
interface EmailUser {
    val email: String
    val nickname: String
        get() = email.substringBefore('@')
}
```

- 게터와 세터가 있는 프로퍼티 선언 가능하다.
    - 뒷받침하는 필드를 참조할 수 없다.
- email: 반드시 오버라이드해야한다.
- nickname: 상속할 수 있다.

언제 함수 대신 프로퍼티를 사용할까?

- 앞장에서 나온 이야기 추가로
- 아래와 같은 특징을 만족하는 경우라면 함수 대신 프로퍼티
  - 예외를 던지지 않는다.
  - 계산 비용이 적게 든다 혹은 최초 실행 후 결과를 캐시해 사용할 수 있다.
  - 객체 상태가 바뀌지 않으면 여러 번 호출해도 항상 같은 결과를 돌려준다.
- 아니라면 함수를 사용하라

게터와 세터에서 뒷받침하는 필드에 접근

```
class User(val name: String) {
    var address: String = "unspecified"
        set(value: String) {
            println(
                """
                Address was changed for $name:
                "$field" → "$value".
                """.trimIndent()
            )
            field = value
        }
}

fun main() {
    val user = User("Alice")
    user.address = "Christoph-Rapparini-Bogen 23, 80639 Muenchen"
}
```

- 접근자의 본문에서는 field라는 식별자를 이용해서 뒷받침 필드에 접근할 수 있다.
  - 게터에서는 field 값을 읽을 수만 있고 세터에서는 읽거나 쓸 수 있다.
- field를 사용하지 않는 커스텀 접근자 구현을 정의하면 뒷받침하는 필드는 존재하지 않는다.

접근자의 가시성 변경

```
class LengthCounter {
    var counter: Int = 0
        private set

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

fun main() {
    val lengthCounter = LengthCounter()
    lengthCounter.addWord("Hi!")
    println(lengthCounter.counter)
}
```

- 접근자의 가시성은 기본적으로 프로퍼티의 가시성과 같다.
- get이나 set 앞에 변경자를 추가해서 가시성을 변경할 수 있다.
- `lateinit 변경자`
    - 널이 될 수 없는 프로퍼티에 지정해서 프로퍼티 초기화를 미룰 수 있다.
- `lazy initilized`
    - 프로퍼티 초기화를 요청이 들어올 때까지 미룬다.
    - 위임 프로퍼티의 일종

컴파일러가 생성한 메서드: 데이터 클래스와 클래스 위임

- Constructor나 getter/setter를 코틀린 컴파일러가 만들어준 것처럼 다른 필수 메소드도 컴파일러가 만들어준다.

모든 클래스가 정의해야 하는 메소드

- toString(), equals(), hashCode() 오버라이드 필요하다.

- 객체 동등성 : equals()
  - 코틀린은 내부적으로 ==를 사용할 때 equals를 호출해서 객체를 비교한다.
- 참조 동일성
  - 참조 비교를 위해서는 ===를 사용한다.
- 해시 컨테이너: hashCode()
  - jvm에서는 equals() true 이면 반드시 같은 hashCode()를 반환해야한다.

데이터 클래스 : 필수 정의 메서드 자동 생성

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

- 주 생성자 밖에서 정의된 프로퍼티는 eqauls나 hashCode 계산의 고려대상이 아니다.
- 불변성
  - copy() 메서드: 객체를 복사하면서 일부 프로퍼티를 바꿀 수 있다.
- 자바 레코드 vs
  - 모든 프로퍼티가 private이며 final 이어야한다.
  - 레코드는 상위 클래스를 확장할 수 없다.
  - 클래스 본문 안에서 다른 프로퍼티를 정의할 수 없다.
  - 상호운용을 위해 코틀린 데이터 클래스에 @JvmRecord 가능

클래스 위임: by 키워드 사용

- 구현 상속을 우회하는 방법으로 데코레이터 패턴을 사용한다.
  - 상속을 허용하지 않는 클래스에서 새로운 동작을 추가해야 할 때
- 데코레이터 패턴은 준비 코드가 상당히 많이 필요
  - by 키워드를 통해 언어가 제공하는 일급 시민 기능으로 지원

```
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)
    }
}

fun main() {
    val cset = CountingSet<Int>()
    cset.addAll(listOf(1, 1, 2))
    println("Added ${cset.objectsAdded} objects, ${cset.size} uniques.")
}
```

- 메서드 일부 동작을 변경하고 싶은 경우 오버라이드 하면 컴파일러가 생성한 메서드 대신 사용된다.
- MutableCollection의 구현 방식에 대한 의존관계가 생기지 않는다.

object 키워드 : 클래스 선언과 인스턴스 생성을 한꺼번에 하기

- object 키워드는 클래스를 정의하는 동시에 인스턴스를 생성한다.
  - 객체 선언
    - 싱글턴을 정의하는 방법 중 하나
  - 동반 객체

        - 어떤 클래스와 관련이 있지만 호출하기 위해 그 클래스의 객체가 필요하지 않은 메서드와 팩토리 메서드를 담을 때 쓰인다.
        - 동반 객체의 멤버에 접근할 때는 동반 객체가 포함된 클래스의 이름을 사용한다.
    - 객체 식
        - 자바의 익명 내부 클래스 대신 쓰인다.

객체 선언 : 싱글턴을 쉽게 만들기

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

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

- 싱글턴을 객체 선언 기능을 통해 언어에서 기본 지원
- 객체 선언: 클래스 선언 + 그 클래스에 속한 단일 인스턴스 선언
- 프로퍼티, 메소드, 초기화 블록을 사용할 수 있으나 생성자는 사용할 수 없다.
- 싱글턴과 의존관계 주입
    - 대규모 컴포넌트에는 적합하지 않다.
    - 객체 생성을 제어할 방법이 없고 생성자 파라미터를 지정할 수 없기 때문
    - 객체를 대체하거나 객체의 의존 관계를 바꿀 수 없다는 뜻
        - 일반 코틀린 클래스와 의존관계 주입을 사용해야 한다.

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

```
    }
}

fun main() {
    println(CaseInsensitiveFileComparator.compare(
        File("/User"), File("/user")))
    val files = listOf(File("/Z"), File("/a"))
    println(files.sortedWith(CaseInsensitiveFileComparator))
}
```

- 객체 선언도 클래스나 인터페이스를 상속할 수 있다.
- 구현 내부에 다른 상태가 필요하지 않는 경우 유용하다.

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

fun main() {
    val persons = listOf(Person("Bob"), Person("Alice"))
    println(persons.sortedWith(Person.NameComparator))
}
```

- 클래스 안에서 object를 선언할 수도 있다.
- 바깥 클래스의 인스턴스마다 별도의 인스턴스가 생기는 것이 아니다.
    - 단 하나뿐이다.
- 자바에서는 INSTANCE 필드로 접근하면 된다.

동반 객체: 팩토리 메서드와 정적 멤버가 들어갈 장소

- 코틀린은 정적인 멤버가 없다.
    - static 키워드를 지원하지 않는다는 것

- 대신 패키지 수준의 최상위 함수와 객체 선언으로 대체
  - 최상위 함수 권장
  - 하지만 private 클래스의 비공개 멤버 접근 불가능
- 팩토리 메서드
  - private 클래스의 비공개 멤버 접근해야 하는 대표 함수

    동반 객체의 멤버 함수로 정의하자

```
class User private constructor(val nickname: String) {
    companion object {
        fun newSubscribingUser(email: String) =
            User(email.substringBefore('@'))

        fun newSocialUser(accountId: Int) =
            User(getNameFromSocialNetwork(accountId))
    }
}

fun main() {
    val subscribingUser = User.newSubscribingUser("bob@gmail.com")
    val socialUser = User.newSocialUser(4)
    println(subscribingUser.nickname)
}
```

- 클래스 안에 선언된 object 중 하나에 companion 이라는 키워드를 붙이면 그 객체는 동반 객체가 된다.
- 객체의 이름을 따로 지정할 필요 없이 클래스 이름을 사용한다
  - 자바의 정적 구문이랑 같다.
- 동반 객체는 자신에 대응하는 클래스에 속한다.
  - 인스턴스는 동반 객체의 멤버에 접근 불가능
    - 자바와 다른점
- 동반 객체는 자신을 둘러싼 클래스의 모든 private 멤버 접근 가능

동반 객체를 일반 객체처럼 사용

- 동반 객체도 클래스 안에 정의된 일반 객체다.

```
class Person(val name: String) {
    companion object Loader {
            fun fromJSON(json: String): Person = /* ... */
    }
}


fun main() {
        Person.Loader.fronJSON("")
        Person.fronJSON("")

}
```

- 동반 객체에 이름을 붙일 수 있다.

```
interface JSONFactory<T>

class Person(val name: String) {
    companion object: JSONFactory<Person> {
            fun fromJSON(json: String): Person = /* ... */
    }
}
```

- 동반 객체가 인터페이스를 상속할 수 있다.
    - 일반 객체가 상속 받듯이 `:` 키워드를 사용하면 된다.

동반 객체 확장

```
class Person(val firstName: String, val lastName: String) {
    companion object {
```

```
    }
}

fun Person.Companion.fromJSON(json: String): Person {
    ...
}

val p = Person.fromJSON(json)
```

- 동반 객체 안에 확장 함수와 프로퍼티를 정의할 수 있다.
- `Person.fromJSON()`은 마치 Person의 멤버 함수처럼 보이지만, Person 클래스 밖에서 정의한 확장 함수다.

객체 식: 익명 내부 클래스를 다른 방식으로 작성

```
interface MouseListener {
    fun onEnter()
    fun onClick()
}

class Button(private val listener: MouseListener) { /* ... */ }

fun main() {
        // 로컬 변수
    var clickCount = 0

    Button(object: MouseListener {
        override fun onEnter() { println("Mouse Enter") }
        override fun onClick() {
            clickCount++  // 로컬 변수의 값 변경
            println("Mouse Click")
        }
    })
}
```

- 익명 객체를 정의할때도 object 키워드를 쓴다.

    - 익명 객체는 자바의 익명 내부 클래스를 대신한다.
- 자바와 다르게 final이 아닌 변수도 객체 식 안에서 사용하거나 변경할 수 있다
- object 키워드가 붙었지만 싱글톤은 아니다.
    - 객체 선언과 동반 객체만 싱글톤임을 주의하자.

인라인 클래스

```
class UsdCent(val amount: Int)
fun addExpense(expense: UsdCent) {
    // 비용을 USD 센트로 저장
}

fun main() {
    addExpense(UsdCent(147))
}
```

- 이 접근 방식을 사용하면 실수로 잘못된 의미의 값을 함수에 전달할 가능성이 훨씬 줄어들지만 몇 가지 성능 고려 사항이 있다.
- 각 `addExpense` 함수 호출마다 새 `UsdCent` 객체를 생성해야 하며, 이 객체는 함수 본문에서 언래핑되어 버려져야 한다는 점이다.
- 이 함수가 많이 호출되는 경우 수명이 짧은 객체를 대량으로 할당하고 가비지 컬렉션을 수행해야 한다.
    - 인라인 클래스가 바로 여기에 적합하다. 인라인 클래스를 사용하면 성능 저하 없이 사용할 수 있다.

```
@JvmInline
value class UsdCent(val amount: Int)
```

- 인라인 클래스는 value 키워드를 사용하고 `@JvmInline` 어노테이션을 사용하여 정의할 수 있다.
- 실행시점에 인스턴스는 감싸진 프로퍼티로 대체된다.
    - 즉, 클래스의 데이터가 사용되는 지점에 인라인된다.

- 인라인 클래스는 프로퍼티 하나, 주 생성자에서 초기화, 클래스 계층에 참여하지 않는다.
    - 그러나 인터페이스를 구현하거나 메서드를 정의하거나 계산된 프로퍼티를 제공할 수는 있다:
- 대부분 기본 타입 값의 의미를 명확하게 하고 싶을 때 사용한다.

요약

- `Kotlin`의 인터페이스는 `Java`와 유사하지만 디폴트 구현과 프로퍼티도 포함할 수 있다.
- 모든 코틀린 선언은 기본적으로 `final`, `public`이다.
- 선언이 `final`이 되지 않게 만들려면 `open`을 사용한다.
- `internal` 선언은 같은 모듈안에서만 볼 수 있다.
- 중첩(내포) 클래스는 기본적으로 내부 클래스가 아니다.
    - 외부 클래스에 대한 참조를 저장하려면 `inner` 키워드를 사용하라.
- `sealed` 클래스를 직접 상속하는 클래스나 sealed 인터페이스에 대한 구현들은 모두 컴파일러에 보여야 한다.
- 초기화 블록과 부 생성자를 활용해 클래스 인스턴스를 더 유연하게 초기화 할 수 있다.
- `field` 식별자를 통해 프로퍼티 접근자 안에서 프로퍼티의 데이터를 저장하는 데 뒷받침하는 필드를 참조할 수 있다.
- 데이터 클래스를 사용하면 컴파일러가 `equals, hashcode, toString, copy` 등의 메서드를 자동으로 생성해준다.
- 클래스 위임을 사용하면 위임 패턴을 구현할 때 비슷한 위임 코드를 반복하는 것을 피할 수 있다.
- 객체 선언은 싱글톤 클래스를 정의하는 `Kotlin`의 방식이다.
- 동반 객체는 `Java`의 `static` 메서드 및 필드 정의를 대체한다.
- 동반 객체는 다른 객체와 마찬가지로 인터페이스를 구현ㅇ할 수 있으며, 동반 객체에 대해서도 확장 함수 및 프로퍼티를 정의할 수 있다.
- 코틀린의 객체 식은 `Java`의 익명 내부 클래스를 대체하는 `Kotlin`의 기능으로, 여러 인터페이스를 구현하거나 객체가 포함된 영역에 있는 변수의 값을 변경할 수 있는 등 더 많은 기능을 제공한다.

- 인라인 클래스를 사용하면 수명이 짧은 객체를 많이 할당하여 발생할 수 있는 성능 저하를 방지하면서 프로그램에 새로운 타입 안정성 계층을 추가할 수 있다.