5부

1. 인터페이스

- 개념은 다른 객체 지향 프로그래밍 언어와 다르지 않다.
  - 인터페이스는 클래스가 무엇을 하는지 기술하지만, 어떻게 하는지는 기술하지 않는다. (이는 구현체인 클래스가 한다)
  - 인터페이스는 형태를 제시하지만, 구현을 포함하지 않는다.
  - 인터페이스는 객체의 동작을 지정하지만, 그 동작을 어떻게 수행하는지에 대한 세부 사항은 제시하지 않는다.
  - 인터페이스는 존재의 목표나 임무를 기술하며, 클래스는 세부적인 구현 사항을 포함한다.
- 코틀린에서 사용하는 방법

```
interface Computer {
    val num: Int
    fun prompt(): String
    fun calculateAnswer(): Int
}

class Desktop: Computer {
    override val num = 1
    override fun prompt() = "Hello D"
    override fun calculateAnswer() = 51
}

class Notebook: Computer {
    override val num get() = 4
    override fun prompt() = "Hello N"
    override fun calculateAnswer() = 53
}

class Mac: Computer (override val num: Int){
    override fun prompt() = "Hello M"
    override fun calculateAnswer() = 55
}

fun main() {
    val computer = Desktop()
    computer.prompt() eq "Hello"
    computer.calculateAnswer() eq 55
}
```

- 클래스 이름 뒤에 클론(`:`) 과 인터페이스 이름을 넣으면 interface 를 구현한다는 의미
- 인터페이스 Computer 는 선언만 하고 인터페이스를 구현하는 클래스는 해당 함수를 반드시 `override` 해서 구체화 한다.
- 함수나 프로퍼티를 오버라이드 할 수 있다.
  - 프로퍼티는 직접 변경, getter, 생성자 인자 로 총 3가지 방법으로 구현 가능하다.
- `enum` 도 인터페이스를 구현 할 수 있다.
- **SAM**(단일 추상 메서드 Single Abstract Method) 변환
  코틀린에서는
  `fun interface` 로 SAM 인터페이스를 정의할 수 있다.

```
fun interface ZeroArg{
    fun f(): Int
}

fun interface TwoArg{
    fun f(i: Int, j: Int): Int
}

// 클래스 방식
class VerboseZero: ZeroArg {
    override fun f() = 11
}
val verboseZero = VerboseZero()
// 람다 방식
val samZero = ZeroArg { 11 }

// 클래스 방식
class VerboseTwo: TwoArg {
    override fun f(i: Int, j: Int) = i + j
}
val VerboseTwo = VerboseTwo()
// 람다 방식
val samTwo = TwoArg { i, j -> i + j }

fun interface Action {
    fun act()
}

// SAM 인터페이스가 필요한 곳에 넘긴다.
fun delayAction(action: Action) {
    trace("Delaying...")
    action.act()
}

fun main() {
    verboseZero.f() eq 11
```

```
    samZero.f() eq 11
    VerboseTwo.f(11, 47) eq 58
    samTwo.f(11, 47) eq 58
    // 객체 대신 람다를 바로 전달
    delayAction { trace("Hey!") }
}
```

- SAM 인터페이스를 일반적인 번거로운 방식(클래스)로 구현할 수 있고 람다를 넣는 방식으로도 구현 가능하다.
- SAM 인터페이스를 파라미터로 넘길 수도 있다.

2. 복잡한 생성자

- 생성자를 만들 때 더 복잡하게 만들고 싶다면 `init` 를 사용해라

```
private var counter = 0

class Message (text: String) {
    private val content: String
    init {
        counter += 10
        context = "[$counter] $text"
    }
    override fun toString() = content
}

fun main() {
    val m1 = Message("m1")
    m1 eq "[10] m1"
    val m2 = Message("m2")
    m2 eq "[20] m2"
}
```

  - `var` 나 `val` 이 붙어 있지 않더라도 `init` 블록에서 사용할 수 있다.
  - `content` 변수는 `val` 로 정의되어 있지만, 정의 시점에 초기화하지 않았다.
    이런 경우 코틀린은 생성자 안의 어느 시점에서 한 번 초기화가 일어나도록 보장한다.

3. 부생성자

- 코틀린에서는 오버로드한 생성자를 **부생성자**라고 부른다.
  생성자 파라미터 목록, 프로퍼티 초기화, init 블록들을 모두 합한 생성자를 **주생성자**라 한다

```
class WithSecondary(i: Int) {
    init {
        trace("Primary: $i")
```

```
    }
    constructor(c: Char): this(c - 'A') {
        trace("Secondary: '$c'")
    }
    constructor(s: String):
        this(s.first()) {
        trace("Secondary: \"$s\"")
    }
    // 부 생성자는 주 생성자를 호출해야만 한다. 그렇지 않으면 컴파일 오류낸다.
    //constructor(f: Float) {
    //  trace("Secondary: $f"
    //}
}

fun main() {
    WithSecondary(1)
    WithSecondary('D')
    WithSecondary("Last")
    trace eq """
    Primary: 1
    Primary: 3
    Secondary: 'D'
    Primary: 11
    Secondary: 'L'
    Secondary: "Last"
"""
}
```

- 부생성자를 만들려면 `constructor` 다음에 주생성자나 다른 부생성자의 파라미터 리스트와 구별되는 파라미터 목록을 넣어야 한다.
- 부생성자에서 `this` 를 사용해 다른 생성자를 호출한다.
- **부생성자는 주생성자를 호출해야만 한다.**
- 주생성자의 파라미터만 `val`, `var` 를 덧붙일 수 있다.
- 부생성자에 반환 타입을 지정할 수 없다.

4. 상속

- java 와 다를것 없이 클래스를 확장하고 싶으면 상속을 사용한다.

```
open class Parent

class Child: Parent()

// 이는 Child 는 open 을 명시하지 않아 상속 실패한다.
// class GrandChild: Child()

open class GreatApe {
```

```
    protected var energy = 0
    open fun call() = "Hoo!"
    open fun eat() { energy += 10 }
    fun climb(x: Int) { energy -= 10 }
    fun energyLevel() = "Energy: $energy"
}

class Bonobo: GreatApe() {
    override fun call() = "Eep!"
    override fun eat() {
        energy += 10
        super.eat()
    }
    fun run() = "Bonobo run"
}

fun talk(ape: GreatApe): String {
    // ape.run() // ape 의 함수가 아님, 사용 불가
    ape.eat()
    ape.climb(10)
    return "${ape.call()} ${ape.energyLevel()}"
}

fun main() {
    talk(GreatApe()) eq "Hoo! Energy: 0"
    talk(Bonobo()) eq "Eep! Energy: 10"
}
```

- 상속 구문은 인터페이스를 구현하는 구문과 비슷하다. 콜론(`:`) 사용 + `부모 클래스()`
- 코틀린에서는 기본적으로 클래스 생성시 `final` 임으로 상속을 허용하기 위해서는(부모 클래스 로 사용 가능하기 위해서는) `open` 키워드를 붙여야 한다.
- 함수 오버라이딩 할 수 있음 (단, 부모 클래스에서 오버라이딩 가능하게 하는 함수 앞에 `open` 을 붙여야 함.)
- 자식 클래스에서 부모 클래스에 함수를 콜 하고 싶으면 `super` 키워드를 사용하면 된다.

5. 기반 클래스 초기화

- 코틀린은 클래스가 다른 클래스를 상속할 때, 두 클래스가 모두 초기화되도록 보장한다.
  - 멤버 객체들의 생성자
  - 부모 클래스에 추가된 객체의 생성자
  - 자식 클래스의 생성자
- 부모 클래스에 생성자 파라미터가 있다면, 자식 클래스가 생성되는 동안 반드시 기반 클래스 인자를 제공해야 한다.

```
open class GreatApe(
    val weight: Double,
    val age: Int
)

open class Bonobo(weight: Double, age: Int): GreatApe(weight, age)
class Chimpanzee(weight: Double, age: Int): GreatApe(weight, age)
class BonoboB(weight: Double, age: Int): Bonobo(weight, age)

// 부모 클래스 생성자 파라미터가 없어도 코틀린은 부모 클래스의 생성자를 인자 없이 호출해야 한다
open class SuperClass1(val i: Int)
class SubClass1(val i: Int): SuperClass1(i)
open class SuperClass2()
class SubClass2(): SuperClass1()
```

- `GreatApe` 를 상속하는 클래스는 반드시 생성자 인자를 `GreatApe` 클래스에 전달해야한다.
  안하면, 컴파일 에러난다.
- 코틀린 객체 사용시 과정
  - 객체에 사용될 메모리 확보
  - 부모 클래스 생성자 호출
  - 다음 자식 클래스 생성자 호출
  - 마지막 자식 클래스 생성자 호출
- 꼭 주생성자를 호출할 필요는 없다
  - 자식 클래스의 책임은 부모 클래스 생성자를 호출할 때 제대로 인자를 제공하면 되는 것임으로 부모 클래스가 부생성자를 호출해도 된다.

6. 추상 클래스

> 추상 클래스는 하나 이상의 프로퍼티나 함수가 불완전하다는 점을 제외하면 일반 클래스와 다를 게 없다. 본문이 없는 초기값 대입을 하지 않는 프로퍼티 정의가 불완전한 정의다. 인터페이스는 추상클래스와 비슷하지만, 인터페이스에는 추상 클래스와 달리 상태가 없다.

- 클래스 멤버에서 본문이나 초기화를 제거하려면 `abstract` 변경자를 해당 멤버 앞에 붙여야 한다.

```
// 아무 초기값도 없는 x 선언
// 초기화 코드가 없으면 코틀린이 해당 참조의 타입을 추론할 방법이 없음으로
// abstract 참조에는 반드시 타입이 지정되어야 한다.
abstract class WithProperty {
    abstract val x: Int
}

// 함수의 내용이 없는 f, g 선언
```

```
// abstract 함수에는 반환 타입을 적지 않으면 반환 타입을 Unit 로 간주한다.
abstract class WithFunctions {
    abstract fun f(): Int
    abstract fun g(n: Double)
}
```

- 추상 클래스를 상속하는 클래스는 `abstract` 멤버, 함수를 구체화(구상화) 를 해야한다.
- 인터페이스에 정의된 함수나 프로퍼티는 모두 기본적으로 추상 함수, 프로퍼티이다.
  - 인터페이스에 함수나 프로퍼티 정의에는 `abstract` 이 필요 없다. (기본값임으로)
  - 인터페이스도 프로퍼티를 선언할 수 있지만, 값을 저장하는 것은 금지되어 있다.
- 인터페이스와 추상 클래스 모두 구현이 있는 함수를 포함할 수 있다.
  - 인터페이스가 함수 구현을 포함할 수 있음으로, 내부에 정의된 프로퍼티가 상태를 바꿀 수없는 경우에는 인터페이스도 프로퍼티의 커스텀 게터를 포함할 수 있다.
- 인터페이스는 다중 상속, 가상 클래스는 단일 상속 된다.
  - 만약, 인터페이스 다중 상속시 프로퍼티나, 함수 시그니처가 충돌하면 개발자가 직접 해결해야 한다.

```
interface A {
    fun f() = 1
    val n: Double
        get() = 1.1
}

interface B {
    fun f() = 2
    val n: Double
        get() = 2.2
}

class C: A, B {
    override fun f() = 0
    override val n:Double
        get() = super<A>.n + super<B>.n
}
```

7. 업캐스트

- 객체 참조를 받아서 그 객체의 부모 타입에 대한 참조처럼 취급하는 것
  자식 객체가 부모 클래스 객체로 취급되는것
- (객체지향방법) SOLID 원칙의 리스코프 치환 원칙
  (자식 타입은 언제든 부모 타입으로 교체할 수 있어야 한다.)

```
interface Shape {
    fun draw(): String
```

```
    fun erase(): String
}

class Circle: Shape {
    override fun draw() = "Circle.draw"
    override fun erase() = "Circle.erase"
}

class Square: Shape {
    override fun draw() = "Square.draw"
    override fun erase() = "Square.erase"
    fun color() = "Square.color"
}

fun show(shape: Shape) {
    trace("Show: ${shape.draw()}")
    // 이는 컴파일되지 않는다.
    trace("Color: ${shape.color()}")
}

fun main() {
    listOf(Circle(), Square()).forEach(::show)
    trace eq """
    Show: Circle.draw
    Show: Square.draw
"""
}
```

- `show()` 의 파라미터는 부모 클래스 `Shape` 이므로, 각 타입이 모두 부모 `Shape` 클래스의 객체로 취급 된다. 이런 구체적인 타입이 부모 타입으로 **업캐스트** 됐다고 한다.

8. 다형성

- 객체나 멤버의 여러 구현이 있는 경우를 뜻함

```
open class Pet {
    open fun speak = "Pet"
}

class Dog: Pet() {
    override fun speak = "Bark!"
}

class Cat: Pet() {
    override fun speak = "Meow"
}
```

```
fun talk(pet: Pet) = pet.speak()

fun main() {
    talk(Dog()) eq "Bark!"
    talk(Cat()) eq "Meow"
}
```

- `talk` 메서드에서는 모두 `Pet` 으로 업캐스트 된다. 그럼에도 출력 값은 각자 `override` 된 함수의 값으로 나온다.
- 다형성은 부모 클래스 참조가 자식 클래스의 인스턴스를 가리키는 경우 발생한다.
- 함수 호출을 함수 본문 과 연결 짓는 작업을 **바인딩**이라 한다. 일반적인 바인딩은 정적이다. (컴파일 시점에 일어남) 하지만, 다형성이 사용되는 경우 컴파일러는 해당 어떤 함수를 사용할지 미리 모르기 때문에, 동적 바인딩으로 실행 시점에 동적으로 결정된다. (**동적 바인딩, 동적 디스패치**)
- 동적 바인딩은 당연히 런타임 시점에 동작함으로 성능에 부정적인 영향을 준다.

9. 합성

- 객체 지향을 사용해야 하는 가장 큰 이유는 코드 재사용이다.
  상속 보다 합성을 위주로 사용하자. 합성이 상속 보다 더 단순한 설계를 만들어 줄 수 있다.
  - 상속을 사용시 단점
    - 강한 결합이 된다.
    - 접근자에 대한 제약이 강하게 된다
    - 부모 클래스의 내부 구현을 알아야 자식 클래스를 제대로 구현할 수 있다.
    - 동적 바인딩으로 성능 하락

```
interface Building
interface Kitchen

// 상속
interface House: Building {
    // 합성
    val kitchen: Kitchen
}
```

- 합성은 포함(has-a) 관계 (집은 부엌을 포함한다)
- 상속은 이다(is-a) 관계 (집은 건물이다.)

10. 상속과 확장

- 상속 보다 확장 함수와 합성을 사용하자.
  기존 클래스를 새로운 목적으로 활용하기 위해 새로운 함수를 추가할 때가 있다. 이때 기존 클래스를 변경할 수 없다면, 새 함수를 추가하기 위해 상속을 사용해야 한다. 이로 인해 코드를 이해하고 유지 보수하기 어려워 진다.

- 관습에 의한 인터페이스
  확장 함수가 함수가 하나뿐인 인터페이스를 만드는 것으로 보일 수 있다.

```
class X
fun X.f() {}

class Y
fun Y.f() {}

// 오버로드 해야한다.
fun callF(x: X) = x.f()
fun callF(y: Y) = y.f()

fun main() {
    val x = X()
    val y = Y()
    x.f()
    y.f()
    callF(x)
    callF(y)
}
```

  - 코틀린의 철학

    > 필수적인 메서드만 정의해 포함하는 간단한 인터페이스를 만들고, 모든 부가 함수를 확장으로 정의하라

- 어뎁터 패턴
  - interface

```
interface LibType {
    fun f1()
    fun f2()
}

fun utility1(lt: LibType) {
    lt.f1()
    lt.f2()
}

fun utility2(lt: LibType) {
    lt.f2()
    lt.f1()
}
```

  - 상속을 이용한 어뎁터 패턴

```
open class MyClass {
    fun g() = trace("g()")
    fun h() = trace("h()")
}

fun useMyClass(mc: MyClass) {
    mc.g()
    mc.h()
}

class MyClassAdaptedForLib: MyClass(), LibType {
    // 멤버함수는 위 인터페이스를 연결하기 위해 쓰인다.
    override fun f1() = h()
    override fun f2() = g()
}

fun main() {
    val mc = MyClassAdaptedForLib()
    utility1(mc)
    utility2(mc)
    useMyClass(mc)
    trace eq "h() g() g() h() g() h()"
}
```

- MyClass 가 상속할 수 있는 open 클래스라는 점에 의존한다.

- 합성을 통한 어뎁터 패턴

```
class MyClass {
    fun g() = trace("g()")
    fun h() = trace("h()")
}

fun useMyClass(mc: MyClass) {
    mc.g()
    mc.h()
}

class MyClassAdaptedForLib: LibType {
    val field = MyClass()
    override fun f1() = field.h()
    override fun f2() = field.g()
}

fun main() {
    val mc = MyClassAdaptedForLib()
    utility1(mc)
    utility2(mc)
```

```
    useMyClass(mc.field)
    trace eq "h() g() g() h() g() h()"
}
```

- 멤버 함수와 확장 함수 비교
  - 클래스의 맴버 함수가 private 멤버에 접근해야 한다면 멤버 함수를 정의할 수 밖에 없다.
  - **확장 함수에 가장 큰 한계는 오버라이드 할 수 없다는 점이다.**

```
open class Base {
    open fun f() = "Base.f()"
}

class Derived: Base() {
    override fun f() = "Derived.f()"
}

fun Base.g() = "Base.g()"
fun Derived.g() = "Derived.g()"

fun useBase(b: Base) {
    trace("Received ${b::class.simpleName}")
    trace(b.f())
    trace(b.g())
}

fun main() {
    useBase(Base())
    useBase(Derived())
    trace eq """
    Received Base
    Base.f()
    Base.g()
    Received Derived
    Derived.f()
    Base.g()
"""
}
```

    - 멤버 함수 f() 에서는 다형성이 작동하지만 확장함수 g() 에서는 작동하지 않는다.
    - **함수를 오버라이드 할 필요 없고 클래스의 공개 멤버만으로 충분하면 멤버 함수로 구현해도 되고 확장 함수로 구현해도 된다.**
    - 멤버 함수는 타입의 핵심을 반영한다.
      그 함수가 없이는 그 타입을 상상할 수 없어야 한다.
    - 확장 함수는 타입의 존재가 필수적이지 않은, 대상 타입을 지원하고 활용하기 위한 **외부** 연산이나, **편리**를 위한 연산이다.

> 코틀린은 되도록 상속을 사용하지 말라고 확장함수와 open 키워드를 생성한 것이다.
> 상속보다는 확장 함수와 합성을 택하라. (디자인 패턴)