3.6 함수

함수는 키워드 **fun**에 이어 함수 이름, 형식 매개변수(formal parameter)와 반환 타입을 지정하여 함수를 선언합니다[1]. 함수 본문은 중괄호를 사용하여 블록 문으로 정의합니다. 아래 함수 add()는 2개의 형식 파라미터(a, b)가 있으며, 반환 값의 타입은 Int입니다.

```
fun add(a:Int, b:Int): Int {
    return a + b
}
```

코틀린에서는 함수 본문의 코드가 한 줄이면 중괄호와 return 문 없이 수식으로 나타낼 수 있습니다.

```
fun add(a:Int, b:Int) = a+b
```

■ 함수 본문을 수식으로 표현할 때 유의할 점

함수 max()는 a, b 중에 큰 값을 반환합니다. if 문의 조건식에 따라 a 또는 b가 함수 max()에 할당됩니다.

```
fun main() {
    var larger = max(7, 12)
    println("The greater value is $larger")
}

fun max(a:Int, b:Int) = if (a > b) a else b
```

max() 함수를 아래처럼 바꾸면 어떻게 될까요?

```
fun max(a:Int, b:Int) =
    if ( a > b ) {
        a
        println("$a is chosen")
    } else {
        b
        println("$b is chosen")
    }
```

if 블록이나 else 블록에서 마지막 변수가 함수의 반환 값입니다. 블록의 마지막 문장인 println()은 실행문이어서 값을 반환하지 않습니다. 반환받은 값이 없으니 Unit가 출력됩니다.

```
12 is chosen
The greater value is kotlin.Unit
```

if 문의 조건식에 따라 실행되는 블록 문에서 맨 마지막에 있는 변수를 반환합니다. 아래 예제의 실행 결과 "The greater value is **98**" 이 출력됩니다. 왜 그럴까요?

---

1) 이를 함수의 signature라고 부릅니다.

```
fun main() {
    var larger = max(7, 12)
    println("The greater value is $larger")
}

fun max(a: Int, b: Int) =
    if (a > b) {
        println("$a is chosen")
        a
    } else {
        println("$a is chosen")
        b
        98
    }
```

■ 함수의 반환 값이 없을 때

함수의 반환 값이 없을 때 반환 타입은 Unit입니다. 자바의 void와 유사하지만, 차이점은 Unit은 타입입니다. Unit은 생략할 수 있습니다.

```
fun main() {
    max(4, 6)
}

fun max(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a+b}")
}
```

■ 실 매개변수에서 이름 지정

함수를 호출할 때 전달하는 파라미터를 실 매개변수(actual parameter)라고 합니다. 함수를 정의할 때 선언하는 파라미터는 형식 매개변수입니다. 형식 매개변수를 선언할 때 기본(default)값을 지정할 수 있습니다.

```
fun main() {
    println(findVolume(2, 3))
    println(findVolume(2, 3, 30))
    println(findVolume(width=20, length = 10))
    println(findVolume(height=30, length = 10, width = 10))
}

fun findVolume(length: Int, width: Int, height:Int=10)
    = length * width * height
```

함수를 호출할 때 실 매개변수 height 값을 전달할 때, 이 파라미터를 생략하면 기본값인 10이 할당됩니다. 함수를 호출할 때 형식 매개변수의 이름을 직접 지정해 호출할 수 있습니다. 형식 매개변수 이름을 사용해 실 매개변수를 전달할 경우, 형식 매개변수 순서와 달라도 상관없습니다.

실행 결과입니다.

```
60
180
2000
3000
```

■ 가변 매개변수를 갖는 함수

함수의 형식 매개변수 이름 앞에 키워드 **vararg**를 붙이면, 형식 매개변수의 개수는 미정입니다. 즉, 실 매개변수가 몇 개이든 상관없이 함수를 호출할 수 있습니다.

```
fun main() {
    add(1, 2, 3)
    add(1, 2, 3, 4)
    add(1, 2, 3, 4, 5)
}

fun add(vararg values:Int) {
    var sum = 0
    for (num in values)
        sum += num
    println(sum)
}
```

3.7 함수형 프로그래밍과 람다 식

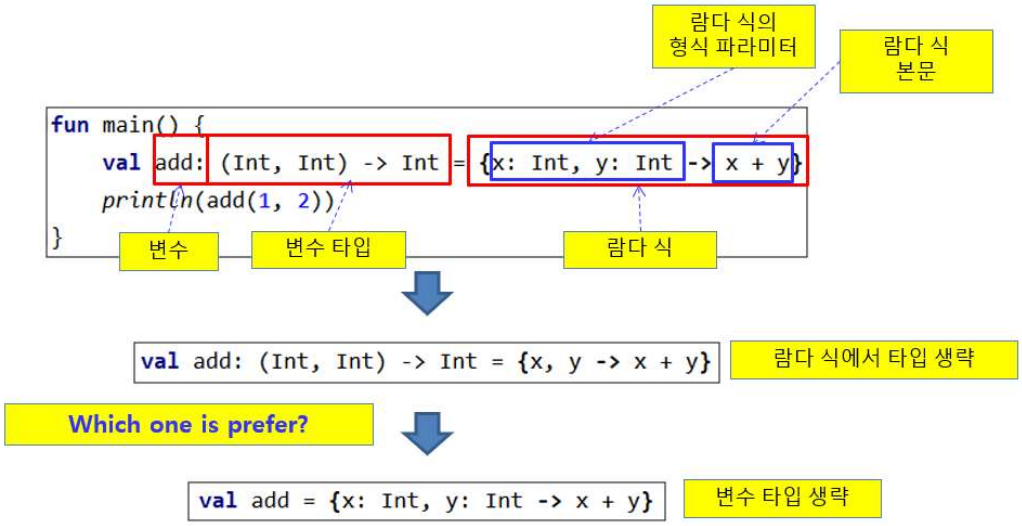
함수 실행 직후에 실 매개변수의 값이 바뀌면 간접 효과(side effect)가 발생했다고 합니다. 물론 의도적으로 실 매개변수의 값을 변경하기도 하지만, 부정적 간접 효과는 의도치 않았는데 값이 바뀐 경우를 말합니다. 비유하자면 "**새 옷을 입고 식당에 갔다가 식사 후에 옷에 얼룩을 묻히고 나온 셈**"입니다. 함수형 프로그래밍(functional programming)이란 이상적으로는 전체 코드를 순수 함수로만 작성하여 함수 호출 직후에 발생할지 모르는 간접 효과를 차단하려는 프로그래밍 기법입니다. 이상적 함수형 프로그래밍이란 "**새 옷을 입고 어느 식당에 가더라도 식사 후에 옷에 얼룩이 묻는 일은 없습니다.**"

이를 위해 함수형 프로그래밍에서는 순수 함수(pure function)만을 사용합니다. 순수 함수란 같은 실 매개변수에 대해 항상 같은 값을 반환하며, 실 매개변수를 포함하여 함수 바깥의 어떤 상태도 바꾸지 않습니다. 그렇다면 순수 함수는 어떻게 만들면 될까요? 아이러니하지만 함수를 아주 간단하게 만들면 됩니다. 그 함수가 바로 람다 식(lambda(λ) expression)입니다. 람다 식 본문은 한 줄 이내만 가능합니다.

재미있는 사실은 람다 식이 "**작은 고추가 맵다**"를 확실히 보여준다는 겁니다. 람다 식은 일급 함수(first class function)이자 일급 객체(first class citizen)입니다. 일급 함수는, 앞에서 본 것처럼, 함수가 변수 역할을 할 수 있습니다. 일급 객체란 함수의 실 매개변수로 사용할 수 있으며, 함수의 반환 값으로도 사용할 수 있습니다. 람다 식이 이 모든 것을 할 수 있습니다. 안드로이드 앱 코딩에서 전체 코드 길이가 짧아진 것도 다 람다 식 때문입니다.

함수형 프로그래밍에 대한 기본 개념과 이 절의 내용과 관련 있는 용어만 간단히 소개했으며, 자세한 내용은 전문 서적을 참고하기를 바랍니다.

람다(lambda(λ)) 식은 이름이 없는 함수이지만, 뒤에서 설명할 익명 함수(anonymous function)와는 다릅니다. 람다 식은 함수의 본문을 한 줄로 짧게 작성할 수 있을 때 사용합니다. 화살표(->)를 사용하며, 화살표 왼쪽에는 람다 식의 형식 매개변수를 선언하며, 화살표 오른쪽에는 람다 식 본문을 작성합니다. 전체 람다 식은 중괄호로 묶어야 합니다.



람다 식의 실행 결과는 변수에 할당할 수 있습니다. 변수는 람다 식의 형식 매개변수 타입과 반환 타입을 지정해야 하며, 람다 식처럼 화살표를 사용합니다. 코틀린은 타입 추론이

가능하므로, 꼭 필요한 타입만 남기고 대부분 생략할 수 있습니다. 단, 람다 식의 형식 매개변수 타입은 생략할 수 없습니다.

최종 축약된 문장에서 알 수 있듯, 변수에 람다 식의 연산 결과를 할당하는 형태입니다.

■ 고차함수

함수형 프로그래밍에서 고차함수(high order function)란, $x^n (n \geq 2)$의 고차함수가 아니라, 2가지 조건을 만족하는 함수를 말합니다. 실 매개변수로써 함수를 사용할 수 있으며, 함수를 반환할 수 있는 함수가 고차함수입니다. 이 중 한 가지만 만족해도 고차함수라고 합니다.

아래 예에서 함수 multiply()의 실 매개변수로 함수 add()를 사용했습니다.

```
fun main() {
    multiply(add(4, 5), 6)
}

fun add(a:Int, b:Int) = a + b
fun multiply(a:Int, b:Int) = println(a * b)
```

아래 예에서 함수 highOrderFun()은 함수 add()를 반환합니다. 고차함수 예에서 알 수 있는 것처럼, 복잡한 함수를 매개변수로 전달하거나 반환하기는 쉽지 않습니다. 아주 간단한 함수, 람다 식이 이런 역할에 적격인 일급 객체입니다.

```
fun main() {
    var result = highOrderFun(2,3)
    println(result)
}

fun add(a:Int, b:Int) = a + b
fun highOrderFun(a:Int, b:Int):Int {
    return add(a, b)
}
```

■ 람다 식을 갖는 고차함수

람다 식을 갖는 변수 myLambda를 정의합니다. 람다 식은 단순히 파라미터 s를 출력합니다. 함수 addTwoNumbers()는 이 변수를 전달받을 수 있도록, 같은 입출력 서식을 갖는 람다 식을 형식 매개변수(myFun)로 선언하였습니다. 실행 결과 9가 출력됩니다.

```
fun main() {
    var myLambda: (Int) -> Unit = {s: Int -> println(s)}
    addTwoNumbers(2, 7, myLambda)
}

fun addTwoNumbers(a:Int, b:Int, myFun:(Int) -> Unit) {
    val sum = a + b
    myFun(sum)
}
```

람다 식을 조금 수정해 보겠습니다. 이번에는 반환 값이 있는 람다 식입니다. 람다 식을 할당한 변수를 실 파라미터로 전달해도 되지만, 람다 식을 통째로 실 매개변수로 전달해도 됩니다. 실행 결과 9, 9가 연속 출력됩니다.

```
fun main() {
    var myLambda: (Int, Int) -> Int = {x, y -> x + y}
    addTwoNumbers(2, 7, myLambda)
    addTwoNumbers(2, 7, { x, y -> x+y } )
}

fun addTwoNumbers(a:Int, b:Int, myFun:(Int, Int)-> Int) {
    val sum = myFun(a, b)
    println(sum)
}
```

마지막 매개변수가 람다 식이면, 람다 식을 함수 괄호 밖으로 옮길 수 있습니다.

```
addTwoNumbers(2, 7) { x, y -> x+y }
```

■ anonymous function(익명 함수)

익명 함수도 람다 식처럼 이름이 없습니다. 람다 식과 비슷하지만, 다른 점은 일반 함수이기 때문에 제어 문장(return, break, continue)을 사용할 수 있습니다. 또, 함수 본문을 중괄호로 묶지 않으며, 화살표(->) 대신 할당 기호(=)를 사용합니다.

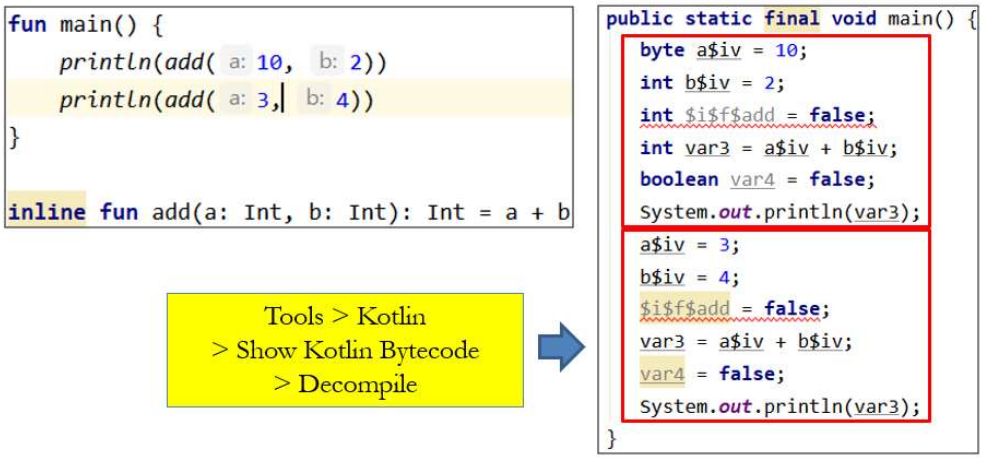
```
fun main() {
    var add: (Int, Int) -> Unit = fun(a, b) = println(a + b)
    add(10, 2)
}
```

■ inline 함수

Inline 함수를 만들려면, 함수 이름 앞에 키워드 inline을 붙여야 합니다. inline 함수는 이 함수를 호출하는 곳에 함수 본문을 그대로 복사합니다. 일반 함수보다 빨리 처리되기 때문에 성능 개선에 도움이 됩니다. 그러나, inline 함수 본문은 짧아야 하며, 너무 많이 사용하면 오히려 성능을 저하합니다.

```
fun main() {
    println(add(10, 2))
    println(add(3, 4))
}

inline fun add(a:Int, b:Int):Int = a + b
```

"도구 > Kotlin > Kotlin 바이트코드 표시"를 선택하면 바이트코드가 표시됩니다. 이를 역 컴파일(Decompile)하면 오른쪽과 같은 코드를 볼 수 있습니다. inline 함수 add()를 사용한 곳에 코드가 복사되었음을 알 수 있습니다.



■ 확장 함수(extension function)

확장 함수는 기존 클래스에 새로운 멤버 함수를 추가합니다. 클래스를 다시 정의하지 않고, 멤버 함수만 쉽게 추가할 수 있습니다. 새롭게 추가한 멤버 함수는 해당 클래스의 정적(static) 멤버 함수로 동작합니다. 정적 멤버 함수란 객체마다 있는 멤버 함수가 아니라 해당 클래스에 하나만 있습니다. 확장 함수의 특징은 새롭게 추가한 멤버 함수가 클래스와 분리되어 있으므로, 코드가 단순해지는 점과 코드를 이해하기 쉽다는 점입니다.

아래 예에서 Student 클래스는 종합시험 통과 여부만 알려줍니다.

```
class Student {
    fun hasPassed(score:Int): Boolean {
        return score > 60
    }
}

fun main() {
    var student = Student()
    println("Pass status: " + student.hasPassed(78))
}
```

장학금 수혜 대상인지를 알 수 있도록 Student 클래스의 확장 함수로 isScholar()를 아래처럼 추가합니다. Student 클래스를 수정하지 않고 필요한 기능만 간단히 추가할 수 있습니다.

```
class Student {
    fun hasPassed(score:Int): Boolean {
        return score > 60
    }
}

fun Student.isScholar(score:Int): Boolean {
    return score > 85
}

fun main() {
    var student = Student()
    println("Pass status: " + student.hasPassed(78))
    println("Scholarship status: " + student.isScholar(78))
}
```

코틀린에서 제공하는 기본 클래스에도 필요한 멤버 함수를 추가할 수 있습니다. String 클래스의 확장 함수 add()에서 this는 이 확장 함수를 호출한 객체입니다. 즉, str1.add(str2)에서 str1이 바로 this입니다. 확장 함수 add()는 객체 자신(str1)과 형식 매개변수로 전달된 문자열(str2)을 연결한 새 문자열(“Hello, Kotlin!”)을 만듭니다.

```
fun main() {
    var str1:String = "Hello, "
    var str2:String = "Kotlin!"

    var tmp = str1.add(str2)
    println(tmp)
}

fun String.add(s:String): String {
    return this + s
}
```

3.8 클래스(Class)

클래스(class)는 객체지향언어의 출발점입니다. 클래스를 정의해야만, 클래스에 속한 객체(object)를 생성할 수 있습니다. 클래스는 메모리가 할당되지 않지만, 클래스에 속한 객체를 생성하면 메모리가 할당됩니다. 클래스는 구조(scheme)만 갖고 있습니다. 클래스 구조는 속성(property)과 메소드(멤버 함수)로 이루어집니다. 클래스에 속한 객체를 생성할 때 속성값을 지정해야 합니다. 클래스에 속한 객체는 클래스에서 정의한 기능(메소드)을 공유합니다.

안드로이드 앱은 객체지향 프로그램입니다. 따라서 이 절의 내용을 잘 숙지해야 합니다.

■ 클래스와 객체

클래스는 본체 없이도 이름만으로 정의할 수 있습니다.

```
class Person {  }
```

클래스를 정의하고 나면 객체(object)를 생성할 수 있습니다. 객체를 클래스의 인스턴스(instance)라고도 부릅니다.

```
val hong = Person( )
```

객체 hong을 생성했다는 것은 이 객체를 저장할 메모리가 할당되었으며, 이제 객체의 속성이나 메소드를 참조할 준비가 되었다는 뜻입니다. 클래스 이름 뒤의 괄호는 객체를 생성하는 역할을 담당하는 생성자(constructor)를 호출합니다. 생성자는 특별한 멤버 함수입니다. 자바와 달리 객체를 생성할 때 키워드 new를 사용하지 않습니다.

■ 생성자가 없는 클래스 Person

클래스 Person은 속성 3개와 메소드 1개를 정의합니다. 이 클래스는 생성자가 없어 속성을 선언할 때 초깃값을 할당해야 합니다. 객체를 생성하면, 점(.) 표기법을 사용하여 객체의 속성이나 메소드를 참조할 수 있습니다.

```
class Person {
    var name:String = "Hong"
    var age:Int = 23
    var isMarried:Boolean = true

    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person()
    hong.getName()          // 출력 "The name is Hong"
    println(hong.age)       // 출력 23
    println(hong.isMarried) // 출력 "true"
}
```

■ 보조 생성자를 정의한 클래스 Person

대개 클래스에서 직접 속성의 초깃값을 지정하지 않습니다. 그렇게 되면 클래스에 속한 객체가 사실상 하나밖에 없기 때문입니다. 클래스 속성이 어떤 값인가에 따라 전혀 다른 객체가 만들어지기 때문에, 속성 초기화는 보기보다 중요한 작업입니다. 이를 위해 코틀린에서는 특별한 메소드인 생성자(constructor)를 제공합니다.

생성자가 특별한 것은 이름이 constructor로 정해져 있기 때문이기도 하지만, 기능이 속성을 초기화하는 역할로 제한되어 있기 때문입니다. 이번에는 클래스 안에 생성자를 정의하겠습니다. 앞의 예와 달리, 3개 속성은 선언만 하고 초기화하지 않았습니다. 초기화를 생성자가 담당하기 때문입니다. 생성자는 객체를 생성할 때 호출됩니다.

```
class Person {
    var name:String
    var age:Int
    var isMarried:Boolean

    constructor(name:String, age:Int, isMarried:Boolean) { // 보조 생성자
        this.name = name
        this.age = age
        this.isMarried = isMarried
    }

    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person("Hong", 23, true) // 생성자 호출
    hong.getName()
    println(hong.age)
    println(hong.isMarried)
}
```

생성자가 없었을 때와 달리, 객체를 생성할 때마다 생성자를 사용해 전혀 다른 속성을 갖는 객체를 만들 수 있습니다.

```
val hong = Person("Hong", 23, true)
val kim = Person("Kim", 27, false)
val park = Person("Park", 25, false)
```

constructor에서 this.name과 같이 속성 이름 앞에 this를 붙인 이유는 생성자의 형식 매개변수와 속성 이름이 같아 구분하기 위한 것입니다. 형식 매개변수 이름을 변경하면, this는 사용하지 않아도 됩니다. 키워드 this는 생성자를 호출한 객체입니다.

```
constructor(_name:String, _age:Int, _isMarried:Boolean) {
    name = _name
    age = _age
    isMarried = _isMarried
}
```

■ 보조 생성자 2개를 정의한 클래스 Person

클래스 안에서 정의하는 생성자를 보조 생성자(secondary constructor)라고 합니다. 보조 생성자는 여러 개 정의할 수 있습니다. 보조 생성자가 여러 개이면, 보조 생성자의 형식 매개변수 개수는 모두 달라야 합니다. 아래 예에서 추가로 정의한 보조 생성자는 2개의 파라미터만을 전달받아 2개의 속성만 초기화합니다. 남은 1개의 속성(isMarried)은 기본값(true)을 할당합니다.

```
class Person {
    var name:String
    var age:Int
    var isMarried:Boolean

    constructor(_name:String, _age:Int, _isMarried:Boolean) {
        name = _name
        age = _age
        isMarried = _isMarried
    }

    constructor(_name:String, _age:Int) {
        name = _name
        age = _age
        isMarried = true // default
    }

    fun getName() = println("The name is $name")
}

fun main() {
    val choi = Person("Choi", 33, false)
    choi.getName()

    val kim = Person("Kim", 37)
    println("The age is ${kim.age}")
    if (kim.isMarried) {
        println("${kim.name} is already married.")
    } else {
        println("${kim.name} is not married yet.")
    }
}
```

3개의 속성을 전부 지정해 객체를 생성하면 첫 번째 생성자가 호출되며, 2개 속성만을 정의해 객체를 생성하면 두 번째 생성자가 호출됩니다.

```
val choi = Person("Choi", 33, false)
val kim = Person("Kim", 37)
```

■ 주 생성자를 정의한 클래스 Person

클래스를 정의할 때 속성 초기화가 중요하다고 강조했습니다. 클래스 속성에 초깃값을 할당하기 위해 앞에서 설명한 보조 생성자도 자주 사용하지만, 가장 많이 사용하는 건 주 생성자(primary constructor)입니다.

클래스 이름 뒤에 생성자를 정의할 수 있는데, 이 생성자가 바로 주 생성자입니다. 보조 생성자와 달리, 주 생성자는 클래스 본문 밖에서 정의합니다. 또, 주 생성자의 파라미터가 바로 클래스의 속성입니다. 주 생성자를 정의하면 속성을 따로 설정할 필요가 없습니다. 단, 속성에 전달되는 값이 객체를 생성할 때마다 달라지기 때문에, 속성 이름 앞에 키워드 **var**을 붙여야 합니다. 물론 **val** 타입의 속성을 선언할 수도 있습니다. 단, 메소드는 클래스 본문 안에서 정의해야 합니다.

```
class Person constructor(var name:String,
                         var age:Int,
                         var isMarried:Boolean) {
    fun getName() = println("The name is $name")
}
```

객체가 생성될 때 주 생성자를 호출합니다. 파라미터를 정의한 순서대로 클래스 속성의 초깃값이 할당된 객체가 생성됩니다. 주 생성자를 정의할 때 키워드 constructor는 생략할 수 있습니다.

```
class Person (var name:String,
              var age:Int,
              var isMarried:Boolean) {
    fun getName() = println("The name is $name")
}
```

아래 코드는 주 생성자를 사용해 클래스를 정의한 코드입니다. 생성자 없이 직접 클래스에서 속성을 정의했던 코드와 비교해 보기 바랍니다. 이 코드를 제대로 이해했다면, 객체지향프로그래밍을 시작할 준비를 완벽하게 마친 셈입니다.

```
class Person (var name:String,
              var age:Int,
              var isMarried:Boolean) {
    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person("Hong", 23, true)
    hong.getName()
    println(hong.age)
    println(hong.isMarried)
}
```

■ init 블록이 있는 주 생성자

주 생성자는 속성을 정의하며, 객체를 생성할 때 속성에 초깃값을 할당하는 역할을 합니다. 속성 초기화가 아닌 다른 작업을 위한 코드를 추가하려면 init 블록이 필요합니다. init 블록은 클래스 안에서 정의합니다. init 블록은 객체를 생성할 때 주 생성자를 호출한 직후 호출됩니다.

```
class Person (var name:String,
              var age:Int,
              var isMarried:Boolean){
    init {
        println("Beginning of init block")
        println("결혼여부=$isMarried, 나이=$age")
        println("End of init block")
    }
    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person("Hong", 23, true)
    hong.getName()
}
```

실행 결과입니다.

```
Beginning of init block
결혼여부=true, 나이=23
End of init block
The name is Hong
```

■ 주 생성자와 보조 생성자를 함께 정의한 클래스 Person

보조 생성자는 init 블록이 실행된 다음에 호출됩니다. 즉, 실행 순서는 주 생성자 → init 블록 → 보조 생성자 순입니다. 주 생성자에서 정의하지 않은 속성 nickname을 클래스 블록 안에서 추가로 정의하였습니다.

```
class Person (var name:String,
              var age:Int,
              var isMarried:Boolean){
    var nickname:String = ""
    init { ... }
    . . . . . .
}
```

보조 생성자는 4개의 속성을 모두 초기화할 필요는 없습니다. 기존 3개 속성의 초기화는 주 생성자를 사용하면 됩니다. 이를 위해 this(...) 구문을 사용합니다. this(...)는 클래스의 주 생성자를 호출합니다. 주 생성자에서 초기화한 3개 속성을 매개변수로 전달합니다.

```
constructor(_name:String, _age:Int, _isMarried:Boolean, _nickname:String)
                : this(_name, _age, _isMarried) {
    nickname = _nickname
}
```

아래 클래스 Person은 주 생성자와 보조 생성자를 함께 갖고 있습니다.

```
class Person (var name:String,
                var age:Int,
                var isMarried:Boolean){
    var nickname:String = ""
    init {
        println("결혼여부=$isMarried, 나이=$age")
    }
    constructor(_name:String, _age:Int, _isMarried:Boolean, _nickname:String)
            :this(_name, _age, _isMarried) {
        nickname = _nickname
    }
    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person("Hong", 23, true, "불어라 봄바람")
    println("The nickname is \"${hong.nickname}\"")
    hong.getName()
}
```

■ 클래스 속성

클래스 속성은 멤버 변수라고도 부르며, 값 또는 상태를 저장할 수 있는 필드(field)입니다. 코틀린 클래스는 속성을 가져(getter)오거나 속성을 설정(setter)하는 메소드를 자동 생성합니다. val로 선언한 속성은 getter가 생성되며, var로 선언한 속성은 getter와 setter가 함께 생성됩니다. 물론 직접 속성의 getter 나 setter를 정의할 수도 있습니다.

```
class Rectangle(val shape:String, var height:Int, var width:Int)

fun main() {
    val rect = Rectangle("Rectangle", 30, 30)
    rect.height = 40
    rect.width = 40
    println("${rect.shape}, ${rect.width}, ${rect.height}")
}
```

아래 코드는 자동 생성된 setter를 사용한 속성 변경입니다.

```
rect.height = 40
rect.width = 40
```

아래 코드는 자동 생성된 getter를 사용해 속성을 가져옵니다.

```
println("${rect.shape}, ${rect.width}, ${rect.height}")
```

이제 클래스 속성에 대해 getter와 stter를 어떻게 정의하는지 살펴보겠습니다. 참고로 shape는 val로 선언했으며, width와 height는 var로 선언했습니다. shape는 getter만 정의할

수 있으며, width와 height는 getter와 setter 모두 정의할 수 있습니다. **field**는 속성을 참고하는 변수이며, **value**는 setter로 전달된 형식 매개변수입니다.

```
class Rectangle(_shape:String, _height:Int, _width:Int) {
    val shape:String = _shape
        get() = field
    var height:Int = _height
        get() = field
        set(value) {
            field = value
        }
    var width:Int = _width
        get() = field
        set(value) {
            field = value
        }
}
```

■ with를 사용한 객체 속성 참조

클래스에 속한 객체를 생성한 후 객체의 속성을 참조할 때 매번 **객체.속성**을 사용하는 것보다는 with나 apply 구문을 사용하는 것이 낫습니다. with나 apply를 범위 지정함수(scope function)라고 부릅니다.

```
class Person {
    var name:String = ""
    var age:Int = -1
    var isMarried:Boolean = false
}

fun main() {
    val hong = Person()
    hong.name = "Hong"
    hong.age = 23
    hong.isMarried = true
    println("name = ${hong.name}, age = ${hong.age}")
    println("marriage = ${hong.isMarried}")
}
```

with 메소드에서 블록에 전달하는 객체(hong)는 null이 아닌 객체이어야 합니다. with 블록에 객체가 전달되기 때문에 속성만 지정하면 됩니다. 단, with 블록은 전달된 객체를 반환하지는 않습니다.

```
class Person { ... }

fun main() {
    val hong = Person()
    with (hong) {
        name = "Hong"
        age = 23
        isMarried = true
        println("name = $name, age = $age, marriage = $isMarried")
    }
}
```

apply 메소드도 마찬가지로 블록 안으로 전달되는 객체는 null이 아닌 객체이어야 합니다. apply 블록에 객체가 전달되기 때문에 속성만 지정하면 됩니다. 단, apply 블록은 전달된 객체를 다시 반환합니다. with나 apply나 사용법은 거의 같습니다.

```
class Person { ... }

fun main() {
    val hong = Person()
    hong.apply {
        name = "Hong"
        age = 23
        isMarried = true
        println("name = $name, age = $age, marriage = $isMarried")
    }
}
```

3.9 상속(inheritance)

객체지향언어의 꽃 상속(inheritance)을 다뤄보겠습니다. 상속이란 자식 클래스가 부모 클래스로부터 속성이나 메소드를 물려받는 것입니다. 프로그래밍 언어에서 상속받는다는 것은 부모 클래스의 속성이나 메소드를 참조할 수 있다는 뜻입니다. 상속하는 클래스를 부모 클래스, 슈퍼 클래스(super class), 기본 클래스(base class)라고 부릅니다. 상속받는 클래스를 자식 클래스, 하위 클래스(sub class), 파생 클래스(derived class)라고 부릅니다. 그림으로 상속 관계를 나타낼 때는 화살표가 자식 클래스(class B)로부터 부모 클래스(class A)로 향하도록 표시합니다.

코틀린에서 상속은 클래스 상속과 인터페이스 상속 2가지가 있습니다. 클래스 상속은 단일 상속(single inheritance)이며, 인터페이스 상속은 다중 상속(multiple inheritance)입니다. 단일 상속은 부모 클래스가 하나인 상속입니다. 다중 상속은 여러 부모로부터 물려받을 수 있지만, 부모는 반드시 클래스가 아닌 인터페이스(interface)이어야 합니다. 다중 상속은 구현 상속이라고도 부릅니다. 인터페이스는 아직 설명하지 않았기 때문에, 클래스 상속부터 살펴보겠습니다.

자식 클래스가 부모 클래스를 상속받으면 어떤 장점이 있을까요? 자식 클래스가 부모 클래스의 속성이나 메소드를 참조할 수 있으면, 코드를 중복해서 정의하지 않고 코드를 재사용(reuse)할 수 있습니다. 또, 자식 클래스는 필요에 따라 부모 클래스의 속성이나 메소드를 목적에 맞게 재정의(overriding)할 수 있습니다.

실제 예를 보면 이해가 쉽습니다. 이제 Animal, Dog, Cat의 3개 클래스를 독립적으로 만들 것입니다. Dog 클래스와 Cat 클래스는 속성 color와 메소드 eat()를 공통으로 갖고 있습니다.

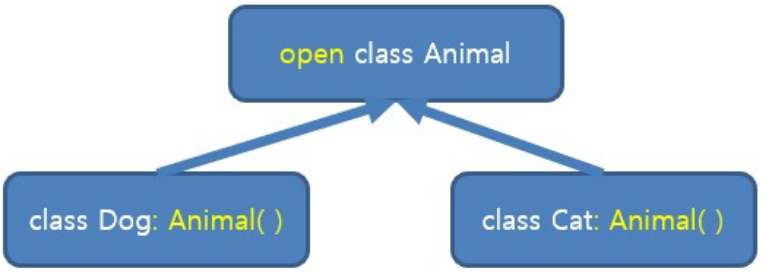
```
class Dog {
    var color:String = ""
    var breed:String = ""
    fun bark() {
        println("Bark")
    }
    fun eat() {
        println("Eat")
    }
}

class Cat {
    var color:String = ""
    var age:Int = -1
    fun meow() {
        println("Meow")
    }
    fun eat() {
        println("Eat")
    }
}
```

부모 클래스에 해당하는 Animal 클래스를 아래처럼 정의합니다. Animal 클래스는, Dog 클래스와 Cat 클래스에서 공통으로 갖고 있던, 속성 color와 메소드 eat()를 정의합니다.

```
class Animal {
    var color:String = ""
    fun eat() {
        println("Eat")
    }
}
```

이제 부모 클래스인 Animal 클래스가 상속하기 위해서는 상속 가능한 클래스로 만들어야 합니다. 이를 위해 Animal 클래스를 open class로 바꿔야 합니다. 키워드 class 앞에 키워드 open을 붙이면, Animal 클래스는 open class로 바뀌면서 상속할 수 있게 됩니다.

이어서 Animal 클래스와 Dog 클래스, Animal 클래스와 Cat 클래스를 각각 상속 관계로 묶을 것입니다. 이들의 상속 관계는 아래 그림과 같습니다.



```
open class Animal {
    var color:String = ""
    fun eat() {
        println("Eat")
    }
}

class Dog : Animal( ) {
    var breed:String = ""
    fun bark() {
        println("Bark")
    }
}

class Cat : Animal( ) {
    var age:Int = -1
    fun meow() {
        println("Meow")
    }
}

fun main() {
    var dog = Dog()
    with (dog) {
        breed = "labra"
        color = "black"
        bark()
        eat()
    }

    var cat = Cat()
    with (cat) {
        age = 3
        color = "white"
        meow()
        eat()
    }
}
```

Dog 클래스의 인스턴스 dog와 Cat 클래스의 인스턴스 cat를 생성하면, 각 객체는 부모 클래스로부터 상속받은 color 속성과 메소드 eat()를 참조할 수 있습니다. 상속 전에 공통으로 갖고 있던 속성과 메소드가 상속을 통해 각 클래스 정의에서 빠지면서, 자식 클래스만의 고유한 특징을 표현하는 데 집중할 수 있게 되었습니다. 공통적인 특징은 부모 클래스로부터 물려받아 코드 재사용성이 증가하며, 동시에 불필요한 코드가 사라지면서 코드가 단순해집니다. 이것이 클래스 상속 이 갖는 장점입니다.

■ **부모 속성과 메소드를 재정의**

상속 관계라고 해도 자식 클래스는 부모 클래스에서 정의한 특징을 그대로 물려받지 않아도 됩니다. 필요에 따라 자식 클래스에서 속성이나 메소드를 재정의(overriding)할 수 있습니다. 부모 클래스가 자식 클래스에 속성이나 메소드 재정의를 허용하려면, 부모 클래스에서 속성이나 메소드 앞에 키워드 **open**을 붙이면 됩니다.

아래 예는 Animal 클래스의 메소드 eat()에 대한 재정의를 허용했습니다. Dog 클래스와 Cat 클래스에서 eat() 메소드를 재정의하려면 키워드 **override**를 붙여야 합니다.

```
open class Animal {
    var color:String = ""
    open fun eat() {
        println("An animal eats food.")
    }
}

class Dog : Animal() {
    var breed:String = ""
    fun bark() {
        println("Bark")
    }
    override fun eat() {
        println("A dog eats food.")
    }
}

class Cat : Animal() {
    var age:Int = -1
    fun meow() {
        println("Meow")
    }
    override fun eat() {
        println("A cat eats food.")
    }
}

fun main() {
    var dog = Dog()
    dog.eat()
    var cat = Cat()
    cat.eat()
}
```

실행 결과입니다.

```
A dog eats food.
A cat eats food.
```

이번에는 속성을 재정의하는 예를 살펴보겠습니다. Animal 클래스에서 속성 color에 대한 재정의를 허용했습니다. Dog 클래스에서만 속성 color 앞에 키워드 **override**를 붙여 재정의합니다. 즉 기본 color 속성은 "white"이지만, Dog 클래스는 color 속성을 "black"으로 재정의했습니다. Cat 클래스는 color 속성을 재정의하지 않았기 때문에 "white" 값을 갖습니다.

Dog 클래스의 eat() 메소드에서 super.eat()는 슈퍼 클래스인 Animal 클래스의 eat() 메소드를 호출합니다. 즉, 키워드 **super**는 부모 클래스를 참조합니다.

```
open class Animal {
    open var color:String = "white"
    open fun eat() {
        println("An animal eats food.")
    }
}

class Dog : Animal() {
    var breed:String = ""
    override var color:String = "black"
    fun bark() {
        println("Bark")
    }
    override fun eat() {
        super.eat()
        println("A dog eats food.")
    }
}

class Cat : Animal() {
    var age:Int = -1
    fun meow() {
        println("Meow")
    }
    override fun eat() {
        println("A cat eats food.")
    }
}

fun main() {
    var dog = Dog()
    println("color = ${dog.color}")
    dog.eat()

    var cat = Cat()
    println("color = ${cat.color}")
}
```

실행 결과입니다.

```
color = black
An animal eats food.
A dog eats food.
color = white
```

■ **생성자를 포함한 상속: 부모 클래스의 주 생성자 호출**

이번에는 상속 관계에 있는 클래스에서 속성 초깃값 설정을 다뤄보겠습니다. 속성 초깃값 설정에만 초점을 맞추도록, 클래스는 Animal 클래스와 Dog 클래스로 제한하며, 메소드는 모두 없앴습니다. 아래 예는 생성자를 전혀 사용하지 않은 클래스 상속입니다.

```
open class Animal {
    open var color:String = "white"
}

class Dog : Animal( ) {
    var breed:String = ""
}

fun main() {
    var dog = Dog()
    dog.color = "Black"
    dog.breed = "Pug"

}
```

이제 클래스 Animal에 대해 속성 color를 파라미터로 갖는 주 생성자를 정의합니다. init 블록은, 앞서 설명한 것처럼, 주 생성자 실행 직후에 호출됩니다.

```
open class Animal(var color:String) {
    init {
        println("at init of Animal class: $color")
    }
}
```

클래스 Animal을 상속받은 Dog 클래스도 주 생성자를 정의합니다. 클래스 상속 관계에서 **Animal( )**이 **Animal(color)**로 바뀌었습니다. 부모 클래스에서 주 생성자를 정의했으며, 주 생성자에서 color 속성을 초기화하기 때문입니다.

```
class Dog(color:String, var breed:String): Animal(color) {
    init {
        println("at init of Dog class: $breed")
    }
}
```

Dog 클래스도 주 생성자를 정의합니다. 2개 속성 중에서 속성 color는 **var**이 없고 속성 breed에만 **var**이 있습니다. 왜 그랬을까요? 속성 color는 자식 클래스인 Dog에서 정의한 속성이 아니라 부모 클래스로부터 물려받았기 때문입니다. 따라서 color 속성은 부모 클래스의 생성자가 초기화한 결과를 전달받아야 합니다.

아래에 전체 코드가 있으며, 실행 결과입니다.

```
at init of Animal class: Black
at init of Dog class: Pug
```

```
open class Animal(var color:String) {
    init {
        println("at init of Animal class: $color")
    }
}

class Dog(color:String, var breed:String)
    : Animal(color) {
    init {
        println("at init of Dog class: $breed")
    }
}

fun main() {
    var dog = Dog("Black", "Pug")
}
```

■ **생성자를 포함한 상속: 부모 클래스의 보조 생성자 호출**

주 생성자를 주로 사용하긴 하지만, 보조 생성자를 사용하는 사례도 꽤 많습니다. 이번에는 부모 클래스에서 보조 생성자를 정의했을 때, 상속 관계에 있는 자식 클래스의 속성 초기화 문제를 다뤄보겠습니다.

주 생성자와 보조 생성자의 차이점은 뭘까요? 가장 중요한 차이점은 주 생성자는 속성도 함께 정의할 수 있다는 점입니다. 반면, 보조 생성자는 클래스 안에서 정의하기 때문에, 이미 정의된 속성만을 초기화할 수 있습니다.

클래스 Animal에서 보조 생성자를 아래처럼 정의하겠습니다. Animal 객체를 생성할 때는 보조 생성자로 속성 color를 초기화하기 위해 초깃값(=빈 문자열)을 지정해야 합니다.

```
open class Animal {
    var color:String = ""
    constructor(color: String) {
        this.color = color
    }
}

fun main() {
    var anim = Animal("")
}
```

위 코드를 아래와 같이 바꿀 수 있습니다. 뭔가 많이 달라졌죠! 3곳이 달라졌습니다.

```
open class Animal( ) {
    var color:String = ""
    constructor(color: String): this( ) {
        this.color = color
    }
}

fun main() {
    var anim = Animal( )
}
```

코드가 달라진 이유는 Animal 이름 뒤에 괄호를 붙였기 때문입니다. 괄호가 있으면 뭐가 달라질까요? 바로 주 생성자를 정의한 것입니다. 괄호 앞에 키워드 constructor를 붙여볼까요? 이제 이해되나요? 속성을 정의하지 않았고, 당연히 아무런 초기화도 하지 않았지만, 엄연히 주 생성자입니다.

```
open class Animal( ) → open class Animal constructor( )
```

주 생성자가 있으면 보조 생성자를 정의할 때 주 생성자를 this()로 참조해야 합니다. 주 생성자는 전혀 초기화를 하지 않지만, 반드시 this()로 참조해야 합니다.

```
open class Animal( ) {
    var color:String = ""
    constructor(color: String): this( ) { ... }
}
```

Animal 객체를 생성할 때 주 생성자를 호출하지만, 전달해야 할 속성은 없습니다.

```
fun main() {
    var anim = Animal( )
}
```

아래 코드는 보조 생성자를 정의한 클래스를 상속받는 전체 코드입니다. 클래스 Dog에서도 보조 생성자를 정의합니다. super(color)는 부모 클래스의 보조 생성자를 호출합니다.

```
open class Animal() {
    var color:String = ""
    constructor(color: String): this( ) {
        this.color = color
    }
}

class Dog : Animal {
    var breed:String = ""
    constructor(color: String, breed:String): super(color) {
        this.breed = breed
    }
}

fun main() {
    var anim = Animal()
    println("color = ${anim.color}")

    var dog = Dog("black", "pug")
    println("color = ${dog.color}, breed = ${dog.breed}")
}
```

실행 결과입니다.

```
color =
color = Black, breed = Pug
```

■ **퀴즈: 어떤 값이 출력될까요?**

클래스 상속 관계를 정의하는 것보다 생성자를 사용한 초깃값 설정이 조금 더 이해하기 어렵죠! 여러분이 앞에 설명을 정말 잘 이해했는지 아래 퀴즈를 풀어보세요. 정답은 실행 결과에서 확인할 수 있습니다. 아래 코드는 부모 클래스와 자식 클래스에서 모두 주 생성자를 정의하고 있습니다. 또 속성 color를 자식 클래스에서 재정의하고 있습니다.

```
open class Animal(open var color:String) {
    init {
        println("at init of the Animal class: $color")
    }
}

class Dog(override var color:String, var breed:String) : Animal(color) {
    init {
        println("at init of the Animal class: $breed")
    }
}

fun main() {
    var anim = Animal("White")
    var dog = Dog("Black", "Pug")

    println("color = ${anim.color}")
    println("color = ${dog.color}, breed = ${dog.breed}")
}
```

코드 실행 결과는 아래와 같습니다. 왜 이런 결과가 나왔는지 코드를 분석해보세요.

```
at init of the Animal class: White
at init of the Animal class: null
at init of the Animal class: Pug
color = White
color = Black, breed = Pug
```

■ Polymorphism(다형성) : overriding 과 overloading

객체지향언어를 대표하는 용어 중에 다형성(polymorphism)이 있습니다. 다형성은 이름은 하나이지만 여러 가지(poly) 모양(morph)을 갖는다는 뜻입니다. 찰흙으로 이 모양, 저 모양 만들면, 다양한 모양이 바로 찰흙의 다형성입니다. 다형성에는 이름도 비슷한 overriding과 overloading이 있습니다. 재정의(overriding)는, 앞에서 본 것처럼, 메소드나 속성 이름은 같지만 다른 클래스에서 이를 재정의해서 사용하는 것입니다. 반면, 함수 overloading이란 함수의 기능은 같지만, 파라미터 타입이나 개수가 다른 경우입니다. 여기서는 함수 overloading에 대한 예제만 보겠습니다.

```
class Calc {
    fun add(x:Int, y:Int):Int = x + y
    fun add(x:Float, y:Float):Float = x + y
    fun add(x:Float, y:Float, z:Float):Float = x + y + z
    fun add(x:Double, y:Double):Double = x + y
    fun add(x:String, y:String):String = x + y
}

fun main() {
    val calc = Calc()
    println(calc.add(2, 3))
    println(calc.add(4.1f, 3.5f))
    println(calc.add(4.1f, 3.5f, 6.9f))
    println(calc.add(4.0, 5.0))
    println(calc.add("Hello, ", "Kotlin"))
}
```

실행 결과입니다.

```
5
7.6
14.5
9.0
Hello, Kotlin
```

■ 가시성 수식어(visibility modifier)

가시성(visibility)이란 클래스의 멤버(속성, 메소드)에 대한 접근 제한을 지정하는 것입니다. 객체지향언어 개념에서 흔히 언급하는 정보 은닉 (information hiding)이 바로 가시성 수식어를 사용하는 것입니다. 가시성 수식어는 public, private, protected, internal 4가지가 있습니다. 코틀린에서는 가시성 수식어를 생략하면 public입니다[2].

public은 클래스 외부에서 멤버 참조를 허용하며, private은 클래스 내부에서만 접근할 수 있고 외부 접근을 차단합니다. protected는 상속 관계에 있는 클래스에서는 멤버 참조가 가능합니다. internal은 같은 모듈 내에서 멤버 참조가 가능합니다. UML(unified modeling language)에서 가시성 수식어는 +(public), #(protected), −(private) 기호를 사용합니다. internal은 코틀린에서만 사용하기 때문에, UML에서 정의하고 있지 않습니다.

2) 자바는 가시성 수식어를 생략하면 기본 수식어는 private입니다.

아래 예에서 상속 관계인 클래스 Derived에서 protected로 선언된 메소드 baseFun()에 접근할 수 있습니다. 이 메소드를 사용해 슈퍼 클래스 Base에서 private로 선언한 속성 a의 값을 변경할 수 있습니다.

```
open class Base {
    private var a = 1
    protected fun baseFun( ) {
        a += 1
        println("a is $a")
    }
}

class Derived : Base() {
    fun deriveFunc() {
        super.baseFun( )
    }
}

fun main() {
    val derived = Derived()
    derived.deriveFunc()
}
```

3.10 추상 클래스와 인터페이스

추상 클래스와 인터페이스는 같은 듯 다릅니다. 같은 점은 모두 추상 메소드를 갖고 있다는 점이지만, 다른 점은 인터페이스는 클래스처럼 속성을 갖지 않는다는 점입니다. 인터페이스는 오로지 추상 메소드만 갖고 있습니다. 자식 클래스가 인터페이스를 상속받는 것을 구현 상속이라고 합니다.

■ 추상 클래스(abstract class)

추상 클래스는 미완성 클래스입니다. 추상 클래스는 추상 메소드, 즉 미완성 메소드를 갖고 있습니다. 미완성 메소드란 메소드를 선언만 했을 뿐, 메소드 블록을 전혀 구현하지 않았습니다. 따라서, 추상 클래스는 상속 관계에 있는 하위 클래스에서 이 추상 메소드를 재정의(overriding)해서 구현해야만 객체를 생성할 수 있습니다.

클래스 Foo는 추상 클래스가 아니라 완성된 클래스입니다. 따라서, 클래스 Foo의 객체를 만들 수 있습니다. 아래 코드는 정상 실행됩니다.

```
class Foo

fun main() {
    val foo:Foo = Foo()
}
```

추상 클래스는 class 앞에 키워드 abstract를 붙입니다. Foo가 추상 클래스인 이유는 추상 메소드 bar()를 갖고 있기 때문입니다. 메소드 bar()에도 키워드 abstract를 붙여야 추상 메소드입니다.

```
abstract class Foo {
    abstract fun bar()
}
```

추상 메소드는 선언만 있고 구현은 없습니다. 추상 메소드 bar() 뒤에 빈 블록을 만들어도 에러입니다. 컴파일러가 블록 중괄호를 붙이기만 해도 구현한 것으로 인식합니다.

```
abstract class Foo {
    abstract fun bar() { }  // 에러!!!
}
```

추상 클래스를 상속받은 클래스 Anonymous를 정의합니다. 슈퍼 클래스 Foo의 추상 메소드 bar()를 재정의해서 구현하면 됩니다. 이제 Anonymous 클래스는 당당히 객체를 생성할 수 있습니다.

```
abstract class Foo {
    abstract fun bar()
}

class Anonymous : Foo() {
    override fun bar() {
        println("bar() is implemented.")
    }
}

fun main() {
    val foo = Anonymous()
    foo.bar()
}
```

추상 클래스를 상속받은 하위 클래스의 객체를 한 번만 사용한다면, 제대로 된 클래스를 정의할 필요가 없습니다. 키워드 object를 사용하여 무명 클래스(anonymous class)로 구현하는 것이 낫습니다. 클래스 선언과 함께 객체까지 생성해서 반환하기 때문입니다.

```
abstract class Foo {
    abstract fun bar()
}

fun main() {
    val foo = object : Foo() {
        override fun bar() {
            println("bar() is implemented.")
        }
    }
    foo.bar()
}
```

추상 클래스라고 해서 추상 메소드만 있는 건 아닙니다. 상속 관계에서 재정의를 허용한 메소드나 재정의를 허용하지 않는 메소드도 가질 수 있습니다. 단, 추상 메소드가 아닌 메소드는 구현을 마쳐야 합니다. 아래 예에서 openFunction( )과 publicFunction( )은 구현을 마친 메소드입니다. 물론 추상 속성, 즉 초깃값을 설정하지 않는 속성도 가질 수 있습니다.

```
abstract class Foo {
    abstract var name:String        // 추상 속성 = 초기값 미설정 속성
    abstract fun bar( )             // 추상 메소드
    open fun openFunction( ) { }    // 재정의를 허용하는 메소드
    fun publicFunction( ) { }       // 재정의를 허용하지 않는 메소드
}
```

아래 예처럼 상속을 통해 추상 메소드와 추상 속성을 재정의해서 객체를 생성할 수 있습니다.

```
abstract class Foo {
    abstract var name:String
    abstract fun bar()
    open fun openFunction() {}
    fun publicFunction() {}
}

fun main() {
    val foo = object : Foo() {
        override var name: String = "new name"
        override fun bar() {
            println("bar() is implemented.")
        }
    }
    foo.bar()
    print(foo.name)
}
```

■ 인터페이스(interface)

객체지향언어에서 가장 문제가 많았던 것은 다중 클래스 상속이었습니다. 이 문제를 해결하기 위해 클래스 상속은 단일 상속으로 제한하고, 다중 상속은 인터페이스인 경우에만 허용하였습니다. 인터페이스(interface)와 추상 클래스가 같은 점은 추상 메소드를 갖고 있다는 점이지만, 다른 점은 인터페이스는 오로지 추상 메소드만 갖는다는 점입니다.

인터페이스를 사용하는 목적은 구현 상속 때문입니다. 인터페이스가 바뀌어도 이를 구현하는 하위 클래스는 클래스 상속과 달리 영향을 거의 받지 않기 때문입니다. 계속 강조하지만 인터페이스는 다중 상속이 가능합니다. 자식 클래스가 동시에 여러 개의 인터페이스를 상속받을 수 있습니다.

아래 예에서 인터페이스 Clickable은 2개의 메소드가 있으며, 메소드 click()이 미완성 메소드입니다. 추상 클래스와 달리 click() 메소드에 키워드 abstract를 붙이지 않습니다. Clickable은 인터페이스이며 클래스가 아닙니다. 인터페이스는 미완성 메소드가 있어 생성자를 정의할 수 없습니다. 따라서 상속 관계를 지정할 때 인터페이스 이름 뒤에 생성자를 호출하는 괄호를 붙이지 않습니다. 상속 관계에서 괄호 여부를 보고 클래스 상속인지 인터페이스 상속인지 판단할 수도 있습니다.

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

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

fun main() {
    val button = Button()
    button.click()
    button.showOff()
}
```

이번 예제는 다중 인터페이스 상속입니다. 인터페이스 Clickable은 추상 메소드 click()과 메소드 showOff()를 갖고 있습니다. 한편, 인터페이스 Focusable은 2개의 메소드를 갖고 있습니다. 이 중 showOff()는 인터페이스 Clickable의 메소드와 이름이 같습니다.

클래스 Button은 2개의 인터페이스를 다중 상속받습니다. 먼저 인터페이스 Clickable의 추상 메소드 click()을 구현해야 합니다. 또 메소드 showOff()를 재정의하려면, 2개 인터페이스가 모두 같은 이름의 메소드를 갖고 있어 이를 구분해야 합니다. 어느 인터페이스의 메소드인지 구분하기 위해 T-파라미터처럼 <인터페이스>를 지정해야 합니다.

```
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()
    }
}

fun main() {
    val button = Button()

    button.showOff()
    button.setFocus(true)
    button.click()
}
```