

Chapter05. 람다를 사용한 프로그래밍

5장에서 다루는 내용

람다식

람다식과 멤버 참조

람다 소개 : 코드 블록을 값으로 다루기

람다와 컬렉션

람다식의 문법

람다 호출 단순화

현재 영역에 있는 변수 접근

멤버 참조

값과 익명 호출 가능 참조

자바의 함수형 인터페이스 사용: 단일 추상 메서드

람다를 자바 메서드의 파라미터로 전달

SAM 변환: 람다를 함수형 인터페이스로 명시적 변환

코틀린에서 SAM 인터페이스 정의 : fun interface

함수형 인터페이스는 동적으로 인스턴스화된다.

수신 객체 지정 람다: with, apply, also

with 함수

apply 함수

객체에 추가 작업 수행: also

수신 객체 지정 람다와 DSL

요약

5장에서 다루는 내용

- 람다식과 멤버 참조를 사용해 코드 조각과 행동 방식을 함수에게 전달
- 코틀린에서 함수형 인터페이스를 정의하고 자바의 함수형 인터페이스 사용
- 수신 객체 지정 람다 사용

람다식

- 람다식 또는 람다는 기본적으로 다른 함수에 넘길 수 있는 작은 코드 조각
- 람다를 사용하면 쉽게 공통 코드 구조를 라이브러리 함수로 뽑아낼 수 있다.
 - 코틀린 표준 라이브러리는 람다를 아주 많이 사용한다.
 - 멤버 참조와 람다의 관계
- 함수형 인터페이스
- 수신 객체 지정 람다
 - 람다 선언을 둘러싸고 있는 환경과는 다른 맥락에서 람다 본문을 실행할 수 있는 특별한 람다

람다식과 멤버 참조

- 자바 8의 람다는 아주 중요한 부분
 - 왜 그럴까?
- 람다의 유용성을 살펴보고 코틀린 람다식 구문을 학습해보자

람다 소개 : 코드 블록을 값으로 다루기

- 일련의 동작을 변수에 저장하거나 다른 함수에 넘겨야 하는 경우
 - 자바에서는 익명 내부 클래스를 통해 해결했지만 상당히 번거롭다.
- 익명 내부 클래스 대신 해결하는 다른 접근 방법
 - 함수를 값처럼 다루기
- 람다식
 - 클래스의 인스턴스를 함수에 넘기는 대신, 함수를 직접 다른 함수에 전달할 수 있다.
 - 코드가 간결해지고 함수를 선언할 필요가 없다.
 - 함수형 프로그래밍 특성 중 함수를 일급 시민으로 다룰 수 있게 해줌

```
button.setOnClickListener(object: OnClickListener {
    override fun onClick(v: View) {
        println("clicked")
    }
})
```

```

    }
})

button.setOnClickListener {
    println("clicked")
}

```

- 람다를 메서드가 하나뿐인 익명 객체 대신 사용할 수 있다.

람다와 컬렉션

- 람다로 인해 코틀린은 컬렉션을 다룰 때 강력한 기능을 제공하는 표준 라이브러리를 제공할 수 있었다.

컬렉션을 for 루프로 직접 검색하기

```

data class Person(
    val name: String,
    val age: Int
)

fun findTheOldest(people: List<Person>) {
    var maxAge = 0
    var theOldest: Person? = null
    for (person in people) {
        if (person.age > maxAge) {
            maxAge = person.age
            theOldest = person
        }
    }
    println(theOldest)
}

fun main() {
    val people = listOf(Person("Alice", 29), Person("Bob", 31))
    findTheOldest(people)
}

```

```
// The oldest is: Person(name=Bob, age=31)
```

- 루프에는 많은 코드가 들어 있기 때문에 실수를 저지르기 쉽다.
 - 연산자 잘못 사용 등

maxByOrNull 함수를 사용해 컬렉션 검색하기

```
data class Person(  
    val name: String,  
    val age: Int  
)  
  
fun main() {  
    val persons = listOf(  
        Person("Alice", age = 29),  
        Person("Bob"),  
    )  
    val oldest = persons.maxByOrNull { it.age }  
    println("The oldest is: $oldest")  
}  
  
// The oldest is: Person(name=Bob, age=31)
```

- `{ it.age }`
 - 선택자 로직 구현
- 코드가 더 짧고 이해하기 쉬우며, 루프 기반의 구현보다 코드의 목적을 더 잘 드러낸다.

람다식의 문법

```
{ x: Int, y: Int → x + y }  
// 파라미터   //본문
```

- 람다는 항상 중괄호로 싸여 있으며, 파라미터를 지정하고 실제 로직이 담긴 본문을 제공
 - 인자 목록 주변에 괄호가 없다.

- 화살표가 인자 목록과 람다 본문을 구분해준다.
- 람다의 본문이 여러 줄로 이뤄진 경우 본문의 맨 마지막에 있는 식이 람다의 결과값
 - 명시적 리턴이 필요하지 않다.
 - Q) 가독성 측면에서 어떨지??

- 람다 변수에도 저장 가능

```
val sum = { x: Int, y: Int → x + y }
println(sum(1,2)) // 3
```

- 하지만 보통 함수에 인자로 바로 넘김
- 람다식 직접 호출

```
{ println(42) }() // 42
```

- 읽기 어렵고 쓸모없다.
- run의 인자로 람다를 전달해 람다를 실행

```
run { println(42) } // 42
```

- 식이 필요한 부분에서 코드 블록을 실행하고 싶을 때 run이 유용하다.
- run을 쓰는 호출에는 부가 비용이 들지 않는다.

람다 호출 단순화

1. 정식 람다 문법

```
people.maxByOrNull({ p: Person → p.age })
```

2. 함수 호출시 맨 뒤에 있는 인자가 람다 식이라면 그 람다를 괄호 밖으로 꺼낼 수 있음

```
people.maxByOrNull() { p: Person → p.age }
```

- 둘 이상의 람다를 인자로 받는 경우는 모두 뺄 수 없다.
- 차라리 모든 람다를 괄호에 넣자

3. 람다가 어떤 함수의 유일한 인자이고 괄호뒤에 인자를 썼다면 호출시 **괄호 생략 가능**

```
people.maxByOrNull { p: Person → p.age }
```

- 가장 가독성이 좋다.

4. 람다 파라미터의 타입도 추론할 수 있다.

```
people.maxByOrNull { p → p.age }
```

- 람다를 변수에 저장할 때는 파라미터의 타입을 추론할 문맥이 없으므로 파라미터 타입을 명시해야한다.

5. 파라미터 **디폴트 이름은 it** 이다.

```
people.maxByOrNull { it.age }
```

- **it**을 남용하면 가독성이 떨어진다.
- 중첩 람다인 경우 파라미터를 명시하는게 낫다.

6. 람다가 단순히 함수나 프로퍼티에 위임할 경우에는 **멤버 참조 사용** 가능하다.

```
persons.maxByOrNull(Person::age)
```

현재 영역에 있는 변수 접근

```
fun printProblemCounts (responses: Collection<String>) {  
    var clientErrors = 0  
    var serverErrors = 0  
  
    responses.forEach {  
        if (it.startsWith("4")) {  
            clientErrors++  
        } else if (it.startsWith("5")) {  
            serverErrors++  
        }  
    }  
}
```

```

    }
    println("$clientErrors client errors, $serverErrors server errors")
}

fun main() {
    val responses = listOf("200 OK", "418 I'm a teapot", "500 Internal Server Error")
    printProblemCounts(responses)
}

```

- 람다를 함수안에서 정의하면 함수의 파라미터뿐 아니라 람다 정의보다 앞에 선언된 로컬 변수까지 모두 사용할 수 있다.
- 자바와 다른점은 코틀린 람다 안에서는 파이널 변수가 아닌 변수에 접근(변경)할 수 있다는 점
 - 람다 캡처 변수: 람다안에서 접근할 수 있는 외부 변수
- 로컬 변수 생명주기는 보통 함수 종료 시 끝나지만, 람다가 해당 변수를 캡처하면 생명주기가 늘어날 수 있음
 - 캡처한 변수가 있는 람다를 저장해서 함수가 끝난 뒤에 실행해도 람다의 본문 코드는 여전히 캡처한 변수를 읽거나 쓸 수 있다.
- 어떻게 그런 동작이 가능할까?
 - 파이널 변수를 캡처한 경우에는 람다 코드를 변수 값과 함께 저장한다.
 - 파이널이 아닌 변수를 캡처한 경우에는 변수를 특별한 래퍼에 감싸서 나중에 변경하거나 읽을 수 있게 한 다음, 래퍼에 대한 참조를 람다 코드와 함께 저장한다.

```

class Ref<T>(var value: T) // 변경 가능한 변수를 캡처를 위한 예시 클래스

```

```

fun main() {
    val count = Ref(0)
    val inc = { counter.value++ }
}

```

// 실제 코드에서는 아래처럼 래퍼를 만들지 않아도 된다. 변수를 직접 바꾼다.

```

fun main() {
    var count = 0
}

```


```
val inc = { counter++ }
}
```

- 파이널이 아닌 변수를 캡처하기 위해 속임수를 사용
- 변경 가능한 값을 저장할 원소가 단 하나뿐인 배열을 선언하거나, 변경 가능한 변수에 대한 참조를 저장하는 클래스를 선언하는 것
 - Ref 인스턴스에 대한 참조를 파이널로 만들면 쉽게 람다로 캡처가 가능하다.
 - 람다 안에서는 Ref 인스턴스의 필드를 변경할 수 있다.
- 하지만 람다를 이벤트 핸들러나 다른 비동기적으로 실행되는 코드로 활용하는 경우 로컬 변수 변경은 람다가 실행될 때만 일어난다.

```
fun tryToCountButtonClicks(button: Button) : Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

- 항상 0을 반환한다.
- onClick 핸들러는 호출될 때마다 clicks의 값을 증가시키지만 값의 변경을 관찰 할 수 없다.
 - tryToCountButtonClicks가 clicks를 반환한 다음에 호출되기 때문
- 클릭 횟수를 세는 카운터 변수를 함수 밖으로 옮겨야 한다.
 - 예를들어 클래스의 프로퍼티 등의 위치로 빼낼 수 있다.

멤버 참조

- 람다를 사용해 코드 블록을 다른 함수에 인자로 넘길 때 이 코드가 이미 함수로 정의되어 있을 수 있다.
- 그 함수를 호출하는 별도의 람다를 만들어도 되지만, 이는 중복이므로 함수를 직접 넘기는 것이 좋다.
- 코틀린에서는 함수를 값으로 바꿀 수 있는데, 를 이용한 멤버 참조를 사용하면 된다.
- 멤버 참조는 정확한 한 프로퍼티나 메소드를 호출하는 함수 값을 만들어 준다.


```
// 이 둘은 같다
people.maxByOrNull(Person::age)
people.maxByOrNull { it.age }
```

- 최상위에 선언된 함수나 프로퍼티를 참조할 수도 있다.

```
fun salute() = println("Salute!")

fun main() {
    run(::salute)
}

// 실행 결과
Salute!
```

- 클래스 이름을 생략하고 바로 참조
- 람다가 인자가 여럿인 다른 함수에게 작업을 위임하는 경우 멤버 참조를 제공하면 편리하다.

```
val action = { person: Person, message: String → sendEmail(person, message)
val nextAction = ::sendEmail // 람다 대신 멤버 참조
```

- 파라미터 이름과 함수를 반복하지 않아도 되기 때문
- 생성자 참조를 사용하면 클래스 생성 작업을 연기하거나 저장해둘 수 있다.

```
fun main() {
    val createPerson = ::Person
    val p = createPerson("name", 23)
    println(p)
}
```

- 확장 함수도 멤버 함수와 똑같은 방식으로 참조할 수 있다.

```
fun Person.isAdult = age >= 20
val predicate = Person::isAdult
```

값과 엮인 호출 가능 참조

```
fun main() {
    val seb = Person("Seb", 26)
    val personAgeFunction = Person::age // 멤버 참조
    println(personAgeFunction(seb)) // 사람을 인자로 받는다

    val sebsAgeFunction = seb::age // 특정 사람의 나이를 돌려주는, 값과 엮인 호출 가능 참조
    println(sebsAgeFunction()) // 특정 값과 엮여 있기 때문에 아무 파라미터를 지정하지 않음
}
```

- 값과 엮인 호출 가능 참조(bounded callable reference)를 사용하면 같은 멤버 참조 구문을 사용해 특정 객체 인스턴스에 대한 메서드 호출에 대한 참조를 만들 수 있다.
- `sebsAgeFunction` 은 { `seb.age` } 라는 람다를 적은 것과 같지만 더 간결하다.

자바의 함수형 인터페이스 사용: 단일 추상 메서드

- 코틀린 람다는 자바 API와 완전히 호환된다.
- 인터페이스안에 추상 메서드가 단 하나뿐 인터페이스
 - 함수형 인터페이스 or 단일 추상 메서드(SAM, Single Abstract Method) 이라고 부른다.
 - eg) Runnable, Callable

람다를 자바 메서드의 파라미터로 전달

```
void process(int delay, Runnable computation);
```

```
process(1000) { println(42) } // 전체 프로그램에 Runnable 인스턴스 하나만 생성
process(1000, object : Runnable {
    override fun run() {
        println(42)
    }
})
```

```
}
})
```

- 함수형 인터페이스를 파라미터로 받는 모든 자바 메서드에 람다를 전달할 수 있다.
- 컴파일러는 자동으로 람다를 **함수형 인터페이스의 인스턴스**로 변환해준다.
 - 익명 클래스 인스턴스로 만들고 람다를 그 인스턴스의 유일한 추상 메서드의 본문으로 만들어주는 것
- 명시적으로 구현하는 익명 객체를 만드는 것이라 동일하다 (object)
 - 하지만 명시적으로 객체를 선언하면 매번 호출할 때마다 새 인스턴스가 생긴다.

```
fun handleComputation(id: String) {
    process(1000) { // handleComputation 호출마다 새 Runnable 인스턴스가 만들어
        println(id) // id를 캡처한다.
    }
}
```

- 하지만 람다를 사용하면 람다가 자신이 정의된 함수의 변수에 접근하지 않는다면 함수가 호출될 때마다 람다에 해당하는 익명 객체가 재사용된다.
 - 람다가 자신을 둘러싼 환경의 변수를 캡처하면 각 함수 호출에 대해 같은 인스턴스를 재사용할 수 없다.
 - 호출마다 새로운 인스턴스를 만들고, 그 객체 안에 캡처한 변수를 저장한다.
- 하지만 코틀린 확장 함수를 사용하는 컬렉션에 대해서는 매번 익명 클래스의 인스턴스를 생성하지 않는다.
 - 람다를 **inline** 표시가 돼 있는 코틀린 함수에 전달하면 익명 클래스가 생성되지 않는다.
 - 대부분의 라이브러리 함수에는 **inline** 이 붙어있다.

SAM 변환: 람다를 함수형 인터페이스로 명시적 변환

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") } // SAM 생성자 사용
}
```

```
fun main() {
    createAllDoneRunnable().run() // ALL done!
}
```

- SAM 생성자
 - 컴파일러가 생성한 함수로 람다를 단일 추상 메서드 인터페이스의 인스턴스로 명시적으로 변환해준다.
 - 컴파일러가 변환을 자동으로 수행하지 못하는 맥락에서 사용할 수 있다.
 - 즉 람다만 있으면 안 되고, **"이 람다를 이 인터페이스로 써줘!"** 하고 **명시적으로** 말해야 할 때 SAM 생성자가 필요하다.
- 예를 들어 함수형 인터페이스의 인스턴스를 반환하는 함수가 있다면 람다를 직접 변환할 수 없다.
 - 대신 람다를 SAM 생성자로 감싸야 한다.
- SAM 생성자의 이름은 사용하려는 함수형 인터페이스의 이름과 같다.

코틀린에서 SAM 인터페이스 정의 : fun interface

```
fun interface IntCondition {
    fun check(i: Int): Boolean // 추상 메서드 정확히 하나만 존재
    fun checkString(s: String) = check(s.toInt()) // 비추상 메서드
    fun checkChar(c: Char) = check(c.digitToInt()) // 비추상 메서드
}

fun main() {
    val isOdd = IntCondition { it % 2 != 0 }
    println(isOdd.check(1))
    println(isOdd.checkString("2"))
    println(isOdd.checkChar('3'))
}
```

- **fun interface** 를 정의하면 함수형 인터페이스를 정의할 수 있다.

- 코틀린 함수형 인터페이스는 정확히 하나의 추상 메서드만 포함하지만 다른 비 추상 메서드를 여러개 가질 수 있다.
 - 이를 통해 함수 타입의 시그니처에 들어맞지 않는 여러 복잡한 구조를 표현할 수 있다.
- 자바 SAM과 마찬가지로 SAM 변환을 사용해 이 인터페이스를 구현하는 인스턴스를 만들 수 있다.

함수형 인터페이스는 동적으로 인스턴스화된다.

```
fun checkCondition(i: Int, condition: IntCondition): Boolean {
    return condition.check(i)
}

fun main() {
    checkCondition(1) { it % 2 != 0 } // 람다 직접 사용

    val isOdd: (Int) → Boolean = { it % 2 != 0 }
    checkCondition(1, isOdd) // 시그니처가 일치하는 람다에 대한 참조를 사용할 수 있다
}
```

- 함수형 인터페이스 타입의 파라미터를 받는 함수가 있을 때
 - 람다 구현이나 람다에 대한 참조를 직접 넘길 수 있다.
 - 두 경우 모두 동적으로 인터페이스 구현을 인스턴스화 해준다.

수신 객체 지정 람다: with, apply, also

- 수신 객체 지정 람다(lamda with receiver)
 - 수신 객체를 명시하지 않고 람다의 본문 안에서 다른 객체의 메서드를 호출할 수 있게 하는 것

with 함수

```

fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
        result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}

```

```

fun main() {
    println(alphabet())
}

```

```

// ABCDEFGHIJKLMNOPQRSTUVWXYZ
// Now I know the alphabet!

```

- 어떤 객체의 이름을 반복하지 않고도 그 객체에 대해 다양한 연산을 수행하는 기능
 - 코틀린에서는 with 라이브러리 함수를 통해 제공

```

fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) { // 메서드를 호출하려는 수신 객체를 지정한다.
        for (letter in 'A'..'Z') {
            this.append(letter) // stringBuilder가 this가 된다.
        }
        this.append("\nNow I know the alphabet!")
        this.toString()
    }
}

```

```

fun main() {
    println(alphabet())
}

```

- with를 사용하기

- with 문은 파라미터가 2개 있는 함수
 - 첫번째: stringBuilder
 - 두번째: 람다
- with 함수는 첫 번째 인자로 받은 객체를 두 번째 인자로 받은 람다의 수신 객체로 만든다.
 - 람다 안에서 this를 꼭 명시할 필요는 없다.
- 수신 객체? 확장 함수?
 - 일반 함수 & 일반 람다
 - 람다는 일반 함수와 비슷한 동작을 정의하는 하나의 방법
 - 확장 함수 & 수신 객체 지정 람다
 - 수신 객체 지정 람다는 확장 함수와 비슷한 동작을 정의하는 하나의 방법

```
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
    toString()
}

fun main() {
    println(alphabet())
}
```

- with와 식을 본문으로 하는 함수
- with 함수가 반환하는 값은 람다 코드를 실행한 결과이며, 그 결과는 람다식의 본문에 있는 마지막 식의 값이다.
- 하지만 때로는 람다의 결과 대신 수신 객체가 필요한 경우도 있다.
 - 그럴때는 apply 함수

apply 함수

```

fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()

fun main() {
    println(alphabet())
}

```

- apply 함수는 with 함수와 동일하게 작동한다.
- 유일한 차이는 apply는 항상 자신에게 전달된 수신 객체를 반환한다.
 - apply를 임의의 타입의 확장 함수로 호출할 수 있다.
 - apply를 호출한 객체는 apply에 전달된 람다의 수신 객체가 된다.
- 인스턴스를 만들면서 즉시 프로퍼티 중 일부를 초기화해야 하는 경우 유용하다.
 - 자바에서는 보통 별도의 빌더 객체가 이런 역할을 담당한다.

```

fun alphabet() = buildString {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}

fun main() {
    println(alphabet())
}

```

- buildString: StringBuilder를 활용해 String을 만드는 경우 사용가능 함수

```

fun main() {
    val fibonacci = buildList {

```



```

    addAll(listOf(1, 1, 2))
    add(3)
    add(index = 0, element = 3)
}

val shouldAdd = true

val fruits = buildSet {
    add("Apple")
    if(shouldAdd) {
        addAll(listOf("Apple", "Banana", "Cherry"))
    }
}

val medals = buildMap<String, Int> {
    put("Gold", 1)
    putAll(listOf("Silver" to 2, "Bronze" to 3))
}
}

```

- 읽기 전용 컬렉션을 생성하지만 가변 컬렉션인 것 처럼 다루고 싶을 때 도움되는 빌더 함수

객체에 추가 작업 수행: also

```

fun main() {
    val fruits = listOf("Apple", "Banana", "Cherry")
    val uppercaseFruits = mutableListof<String>()
    val reversedLongFruits = fruits
        .map { it.uppercase() }
        .also { uppercaseFruits.addAll(it) }
        .filter { it.length > 5 }
        .also { println(it) }
        .reversed()
    println(uppercaseFruits)
}

```

```
println(reversedLongFruits)
}
```

- `apply`와 마찬가지로 `also` 함수 함수도 수신 객체를 받으며, 수신 객체에 대해 어떤 동작을 수행한 후 수신 객체를 돌려준다
 - 주된 차이는 `also`의 람다 안에서는 수신 객체를 인자로 참조한다는 점이다.
 - 따라서 파라미터 이름을 부여하거나 디폴트 이름 사용해야 한다.
 - Q) 왜 이런 차이가 있을까? `it` vs `this`
- 원래의 수신 객체를 인자로 받는 동작을 실행할 때 유용하다.
 - 어떤 효고라를 추가로 수행하는 것으로 보통 해석

수신 객체 지정 람다과 DSL

- 수신 객체 지정 람다는 DSL을 만들 때 아주 유용한 도구다.

요약

- 람다를 사용하면 코드 조각을 다른 함수에게 인자로 넘길 수 있다.
 - 따라서 공통 코드 구조를 쉽게 추출할 수 있다.
- 코틀린에서는 람다가 함수 인자인 경우 괄호 밖으로 람다를 빼내서 코드를 더 깔끔하고 간결하게 만들 수 있다.
- 람다의 인자가 단 하나뿐인 경우 인자 이름을 지정하지 않고 `it`이라는 디폴트 이름으로 부를 수 있다.
 - 이를 통해 짧고 간단한 람다 내부에서 파라미터 이름을 붙이느라 노력할 필요가 없다.
- 람다는 외부 변수를 캡처할 수 있다.
- 메서드, 생성자, 프로퍼티의 이름 앞에 `::` 을 붙이면 각각에 대한 참조를 만들 수 있다.
 - 그런 참조를 람다 대신 다른 함수에 넘길 수 있다.
- `filter`, `map`, `all`, `any` 등의 함수를 활용하면 직접 원소를 이터레이션하지 않고 컬렉션에 대한 대부분의 연산을 수행할 수 있다.

- 추상메서드가 단 하나뿐인 인터페이스(SAM)를 구현하기 위해 그 인터페이스를 구현하는 객체를 생성하는 대신 람다를 직접 넘길 수 있다.
- 수신 객체 지정 람다를 사용하며 람다 안에서 미리 정해둔 수신 객체의 메서드를 직접 호출 할 수 있다.
 - 이 람다의 본문은 그 본문을 둘러싼 코드와는 다른 컨텍스트에서 작동하기 때문에 코드를 구조화할 때 도움이 된다.
- 표준 라이브러리의 with 함수를 사용하면 어떤 객체에 대한 참조를 반복해서 언급하지 않으면서 그 객체의 메서드를 호출할 수 있다.
- apply를 사용하면 어떤 객체이든 빌더 스타일의 API를 사용해 생성하고 초기화할 수 있다.
- also를 사용하면 객체에 대해 추가 작업을 수행할 수 있다.