Chapter03. 함수 정의와 호출

3장에서 다루는 내용
코틀린에서 컬렉션 만들기
함수를 호출하기 쉽게 만들기
    개선1. 이름 붙인 인자
    개선2. 디폴트 파라미터값
    개선 3. 정적인 유틸리티 클래스 없애기: 최상위 함수와 프로퍼티
메서드를 다른 클래스에 추가: 확장 함수와 확장 프로퍼티
    확장 함수
    임포트와 확장 함수
    자바에서 확장 함수 호출
    개선 4. 확장 함수로 유틸리티 함수 정의
    확장 함수는 오버라이드할 수 없다.
    확장 프로퍼티
컬렉션 처리: 가변 길이 인자, 중위 함수 호출, 라이브러리 지원
    자바 컬렉션 API 확장
    가변 인자 함수: 인자의 개수가 달라질 수 있는 함수 정의
    쌍(튜플) 다루기: 중위 호출과 구조 분해 선언
문자열과 정규식 다루기
    문자열 나누기
    정규식과 3중 따옴표로 묶은 문자열
    여러 줄 3중 따옴표 문자열
코드 깔끔하게 다듬기: 로컬 함수와 확장
    코드 중복 예제
    로컬 함수를 이용한 코드 중복 제거
    로컬 함수에서 바깥 함수의 파라미터 접근
    검증 로직을 확장 함수로 추출하기
요약
질문정리

3장에서 다루는 내용

- 컬렉션, 문자열, 정규 식을 다루기 위한 함수
- 이름 붙인 인자, 디폴트 파라미터 값, 중위 호출 문법 사용
- 확장 함수와 확장 프로퍼티를 사용해 자바 라이브러리를 코틀린에 맞게 통합

- 최상위 및 로컬 함수와 프로퍼티를 사용해 코드 구조화

코틀린에서 컬렉션 만들기

```
val set = setOf(1, 7, 54)
val list = listOf(1, 7, 54)
val map = hashMapOf(1 to "one", 7 to "seven", 54 to "fifty-four")
```

- 코틀린은 표준 자바 컬렉션 클래스를 사용한다.
  - 서로 변환할 필요가 없다.
  - 자바보다 더 많은 기능 사용 가능
- 하지만 자바와 다르게 코틀린 컬렉션 인터페이스는 디폴트로 읽기 전용이다.
- `1 to “one”` 에서 `to` 는 일반 함수

함수를 호출하기 쉽게 만들기

- 자바 컬렉션의 디폴트 방식(toString)이 아닌 다른 방식으로 출력하는 함수를 정의해보자.

```
fun <T> joinToString(
    collection: Collection<T>,
    seperator: String,
    prefix: String,
    postfix: String
): String {
    val result = StringBuilder(prefix)
    for((index, element) in collection.withIndex()) {
        if(index > 0) result.append(seperator)
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}
```

개선1. 이름 붙인 인자

```
joinToString(collection, seperator=";", prefix="[", postfix="]")
```

- 함수 호출 부분의 가독성을 개선
- 함수를 호출할 때 함수에 전달하는 인자 중 일부의 이름을 명시할 수 있다.
- 모든 인자의 이름을 명시하는 경우 인자 순서도 변경 가능
- 특히 디폴트 파라미터 값과 함께 사용할 때 잘 작동

개선2. 디폴트 파라미터값

```
fun <T> joinToString(
    collection: Collection<T>,
    //디폴트 값이 지정된 파라미터
    seperator: String = ", ",
    prefix: String = "",
    postfix: String = ""
): String
```

- 함수 선언에서 파라미터의 디폴트 값을 지정할 수 있어 과한 오버로딩을 방지할 수 있다.
- 함수의 디폴트 파라미터 값은 함수를 호출하는 쪽이 아니라 함수 선언 쪽에 인코딩 된다.

자바에는 디폴트 파라미터 값이라는 개념이 없어 기본값을 제공하는 코틀린 함수를 자바에서 호출하는 경우 모든 인자를 명시해야 한다. 좀더 편하게 호출하고 싶다면 @JvmOverloads 어노테이션을 함수에 추가하면 코틀린 컴파일러가 자동으로 맨 마지막 파라미터로부터 파라미터를 하나씩 생략한 오버로딩한 자바 메서드를 추가해준다.

개선 3. 정적인 유틸리티 클래스 없애기: 최상위 함수와 프로퍼티

- 코틀린에서는 함수를 클래스 안에 선언할 필요가 없다.
- 따라서 정적 메서드를 모아두는 역할만 담당하는 유틸리티 클래스가 필요없다.

최상위 함수

```
/*
* strings 패키지에 직접 함수 정의
* 파일 : join.kt
*/
package strings

fun <T> joinToString(...): String {...}
```

- 함수를 직접 소스파일의 최상위 수준(다른 클래스들의 밖)에 위치시키면, 가장 앞의 패키지의 멤버 함수이다.

```
/*
* 위의 코틀린 코드를 자바로 변환
*/
package strings;

public class JoinKt {
    public static String jointToString(...) {...}
}

public static void main(String args[]) {
    JoinKt.joinToString(collection, "", "", "");
}
```

- 코틀린 컴파일러가 생성하는 클래스 이름은 최상위 함수가 들어있던 코틀린 소스 파일의 이름과 대응한다.
- 파일에 대응하는 클래스 이름을 변경하고 싶을 땐 `@file:JvmName( "원하는 이름" )`을 파일의 맨 앞, 패키지 이름 선언 이전에 위치 시키면 된다.

최상위 프로퍼티

```
var opCount = 0
val UNIX_LINE_SEPARATOR = "\n"
const val UNIX_LINE_SEPARATOR = "\n" // == public static final String

fun performOperation() {
    opCount++
}
```

- 프로퍼티도 파일 최상위 수준에 놓을 수 있다.
- 이런 프로퍼티의 값은 정적 필드에 저장된다.
- 최상위 프로퍼티도 다른 모든 프로퍼티처럼 접근자 메서드를 통해 자바 코드에 노출된다.
    - 해당 상수를 자바 코드에게 `public static final` 필드로 노출하고 싶다면 `const` 변경자를 추가한다.
    - 기본 타입과 String 타입의 프로퍼티만 const로 지정 가능한다.
        - 컴파일 시점에 결정되어야 하기 때문

메서드를 다른 클래스에 추가: 확장 함수와 확장 프로퍼티

확장 함수

```
package strings

/*
 * 확장함수
 * println("Kotlin".lastChar())의 결과는 n
 */
fun String.lastChar(): Char = this.get(this.length - 1)
```

- 어떤 클래스의 멤버 메소드인 것처럼 호출할 수 있지만, 선언은 그 클래스의 밖에서 된 함수

- 수신 객체 타입: 확장이 정의될 클래스 타입
  - String
- 수신 객체 : 수신 객체 타입의 인스턴스 객체
  - this
  - 확장 함수 본문에서 생략가능
- 확장 함수 내부에서는 수신 객체의 메서드나 프로퍼티를 바로 사용할 수 있다.
  - 하지만 캡슐화를 깨지 않기 때문에 private 멤버와 protected 멤버를 직접 사용할 수 없다.

임포트와 확장 함수

```
import strings.lastChar
import strings.*
import strings.lastChar as last
```

- 확장 함수를 사용하려면 해당 함수를 임포트 해야한다.
  - 이름 충돌을 막기 위해
- `as`를 사용하면 이름을 바꿀 수 있다.
  - 한 파일 안에서 이름이 같은 함수들을 함께 써야할 때 편리하다.
  - 일반 클래스나 함수라면 전체 이름을 써서 사용할 수도 있다.
  - 하지만 확장 함수는 코틀린 문법상 반드시 짧은 이름을 써야 하기 때문에 이름을 바꾸는 것이 충돌 해결하는 유일한 방법

자바에서 확장 함수 호출

```
char c = StringUtilKt.lastChar("Java");
```

- 내부적으로 확장 함수는 수신 객체를 첫 번째 인자로 받는 정적 메서드다.
  - 다른 최상위 함수와 마찬가지로 확장 함수가 들어있는 자바 클래스 이름도 확장 함수가 들어있는 파일 이름에 따라 결정된다.
- 따라서 확장 함수를 호출해도 다른 어댑터 객체나 실행 시점 부가 비용이 들지 않는다.

개선 4. 확장 함수로 유틸리티 함수 정의

```
fun <T> Collection<T>.joinToString(
    seperator: String = ", ",
    prefix: String = "",
    postfix: String = ""
): String {
    val result = StringBuilder(prefix)

    for((index, element) in this.withIndex()) {
        if(index > 0) result.append(seperator)
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}
```

- jointToString 함수의 최종 버전
- 원소로 이뤄진 컬렉션에 대한 확장을 만듬
- 확장 함수는 단지 정적 메서드 호출에 대한 문법적인 편의일 뿐이기 때문에 더 구체적인 타입을 수신 객체 타입으로 지정할 수도 있다.
    - eg) `fun Collextion<String>.join()`
- 확장 함수는 정적 메서드와 같은 특성을 가지므로 확장 함수를 하위 클래스에서 오버라이드할 수 없다.

확장 함수는 오버라이드할 수 없다.

```
open class View {
    open fun click() = println("View clicked")
}

class Button: View() {
    override fun click() = println("Button clicked")
}
```

```
fun View.showOff() = println("I'm a view!")
fun Button.showOff() = println("I'm a button!")

fun main() {
    val view: View = Button()
    view.showOff() // I'm a view!
}
```

- 이름과 파라미터가 완전히 같은 확장 함수를 기반 클래스와 하위 클래스에 대해 정의할 수 있지만 실제 Button 타입이었다 해도 View 타입에 해당하는 확장 함수가 호출된다.
- 실제 호출될 함수는 확장 함수를 호출할 때 수신 객체로 지정한 변수의 컴파일 시점의 타입에 의해 결정된다.
    - 실행 시간에 그 변수에 저장된 객체의 타입에 의해 결정되지 않는다.
    - 즉, 확장 함수는 정적으로 결정된다.
- 따라서 확장 함수에 대해서는 오버라이딩이 적용되지 않는다.

어떤 클래스를 확장한 함수와 그 클래스의 멤버 함수의 이름과 시그니처가 같다면 확장 함수가 아니라 멤버 함수가 호출된다. 클래스의 API를 확장할 경우 항상 이를 염두에 두어야 한다.

확장 프로퍼티

- 확장 프로퍼티를 사용하면 함수가 아니라 프로퍼티 형식의 구문으로 사용할 수 있는 API를 추가 가능
- 하지만 상태를 저장할 방법이 없기 때문에 실제로 확장 프로퍼티는 아무 상태도 가질 수 없다.
    - 기존 자바 객체 인스턴스에 필드를 추가할 방법은 없기 때문
- 따라서 확장 프로퍼티는 항상 커스텀 접근자를 정의한다.

```
val String.lastChar: Char
    get() = get(length - 1)
```

- 프로퍼티에 수신 객체 클래스가 추가되었다.
- 뒷받침 필드가 없어 기본 게터 구현을 제공할 수 없으므로 게터는 꼭 정의를 해야 한다.
- 초기화 코드도 쓸수 없다.

```
var StringBuilder.lastChar: Char
    get() = get(length - 1)
    set(value: Char) {
        this.setCharAt(length-1, value)
    }

fun main() {
    val sb = StringBuilder("Kotlin?")
    println(sb.lastChar) // ?
    sb.lastChar = '!'
    println(sb) // Kotlin!
}
```

- StringBuilder 맨 마지막 문자를 변경할 수 있으므로 var로 변경 가능한 확장 프로퍼티 만들수 있다.
    - mutable이라서

컬렉션 처리: 가변 길이 인자, 중위 함수 호출, 라이브러리 지원

- `vararg` 키워드를 사용하면 호출 시 인자 개수가 달라질 수 있는 함수를 정의 할 수 있다.
- `중위(infix)` 함수 호출 구문을 사용하면 인자가 하나뿐인 메서드를 간편하게 호출 할 수 있다.
- 구조 분해 선언을 사용하면 복합적인 값을 분해해서 여러 변수에 나눠 담을 수 있다.

자바 컬렉션 API 확장

어떻게 자바 라이브러리 클래스의 인스턴스인 컬렉션에 대해 코틀린이 새로운 기능을 추가할 수 있을까?

- 확장 함수로 정의되고 항상 코틀린 파일에서 디폴트로 임포트 된다.

가변 인자 함수: 인자의 개수가 달라질 수 있는 함수 정의

```
fun listOf<T>(vararg values: T): List<T> { ... }
```

- 원하는 개수만큼 값을 인자로 넘기면 배열에 그 값들을 넣어주는 언어 기능이 `가변 길이 인자`를 사용한다.
- 자바는 타입 뒤에 `...` 을 붙이는 대신 코틀린에서는 파라미터 앞에 `vararg` 변경자를 붙인다.

```
fun main(args: Array<String>) {
    val list = listOf("args: ", *args)  //스프레드 연산자
}
```

- 배열에 들어있는 원소를 가변 길이 인자로 넘길 때는 `스프레드 연산자`로 배열을 풀어서 각 원소가 인자로 전달되게 해야한다.
    - `spread`
- 자바는 이런 기능을 지원하지 않는다.

쌍(튜플) 다루기: 중위 호출과 구조 분해 선언

```
val map = mapOf(1 to "one", 7 to("seven"))
```

- `to` 라는 단어는 코틀린 키워드가 아니다.
- `infix call` 이라는 특별한 방식으로 to라는 일반 메서드를 호출한 것이다.

- 중위 호출 시에는 수신 객체 뒤에 메서드 이름을 위치시키고 그 뒤에 유일한 메서드 인자를 넣는다.
    - 이때 다른 구분자는 필요 없고 각각을 공백으로 분리 한다.

```
1.to("one") // to 함수를 일반적 방식으로 호출
1 to "one"  // to 함수를 일반적 방식으로 호출
```

- 두 호출은 동일 하다.

```
infix fun Any.to(other: Any) = Pair(this, other)
```

- 인자가 하나뿐인 일반 메서드나 인자가 하나뿐인 확장 함수에만 중위 호출을 사용 할 수 있다.
- 함수를 중위 호출에 사용하게 허용하고 싶으면 infix 변경자를 함수 선언 앞에 추가해야 한다.

```
val (number, name) = 1 to "one"
```

- Pair의 내용을 갖고 두 변수를 즉시 초기화할 수 있다.
- 해당 기능을 구조 분해 선언이라고 한다.
- `to` 함수는 확장 함수로 수신 객체가 제네릭이다.

문자열과 정규식 다루기

- 자바의 문자열과 코틀린의 문자열은 같다.
- 코틀린에서는 다양한 확장 함수로 더 많은 기능을 제공한다.

문자열 나누기

- 자바의 `split` 은 정규식을 구분 문자열로 받아 문자열로 나눈다.

- 코틀린에서는 대신에 여러 가지 다른 조합의 파라미터를 받는 split 확장 함수를 제공함으로써 혼동을 야기하는 메서드를 감춘다.

```
println("12.345-6.A".split("\\.|-".toRegex()))  //toRegex로 정규식으로 반환
```

- 정규식을 파라미터로 받는 함수는 String이 아닌 Regex 타입의 값을 받는다.

```
println("12.345-6.A".split(".", "-"))
```

- 확장 함수를 오버로딩한 버전 중에는 구분 문자열을 하나 이상 인자로 받는 함수가 있다.

정규식과 3중 따옴표로 묶은 문자열

```
fun parsePath(path: String) {
    val directory = path.substringBeforeLast("/")
    val fullName = path.substringAfterLast("/")

    val fileName = fullName.substringBeforeLast(".")
    val extension = fullName.substringAfterLast(".")

    println("Dir: $directory, name: $fileName, ext: $extension")
}

fun main() {
    parsePath("/Users/yole/kotlin-book/chapter.adoc")
}
```

- 코틀린에서는 정규식을 사용하지 않고도 문자열을 쉽게 파싱할 수 있다.

```
fun parsePathRegex(path: String) {
    val regex = """(.+)/(.+)\.(.+)""".toRegex()
    val matchResult = regex.matchEntire(path)
    if (matchResult != null) {
        val (directory, filename, extension) = matchResult.destructured
        println("Dir: $directory, name: $filename, ext: $extension")
```

```
    }
}

fun main() {
    parsePathRegex("/Users/yole/kotlin-book/chapter.adoc")
}
```

- 정규식은 강력하지만 가독성이 떨어진다.
  - 정규식이 필요할 때는 코틀린 라이브러리를 사용하면 도움이 된다.
- 3중 따옴표 문자열을 사용해 정규식을 썼다.
  - 3중 따옴표 문자열에서는 백슬래시를 포함한 어떤 문자도 이스케이프할 필요가 없다.

여러 줄 3중 따옴표 문자열

- 3중 따옴표 문자열에는 줄 바꿈을 포함해 아무 문자열이나 이스케이프 없이 그대로 들어간다.
- 줄 바꿈이 들어있는 텍스트를 쉽게 프로그램에 포함시킬 수 있다.
  - `trimIndent`
    - 모든 입력 라인의 공통 최소 들여쓰기를 감지하고 모든 라인에서 그만큼 제거한다.
    - 비어있는 첫번째와 마지막 라인 제거.
- 테스트 코드의 예상 결과와 비교할 때 용이하다.
  - IDE 문법 하이라이팅도 가능
  - @Language 어노테이션
- 코틀린은 운영체제 관계없이 CRLF, CR, LF를 모두 줄 끝으로 취급한다.

코드 깔끔하게 다듬기: 로컬 함수와 확장

- 함수에서 추출한 함수를 원래의 함수 내부에 내포시킬 수 있다.
- 로컬 함수를 통해 코드 중복을 제거해보자

코드 중복 예제

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
    if (user.name.isEmpty()) {
        throw IllegalArgumentException(
            "Can't save user ${user.id}: empty Name")
    }

    if (user.address.isEmpty()) {
        throw IllegalArgumentException(
            "Can't save user ${user.id}: empty Address")
    }

    // Save user to the database
}

fun main() {
    saveUser(User(1, "", ""))
}
```

- 필드 검증 중복

로컬 함수를 이용한 코드 중복 제거

```
fun saveUser(user: User) {

    fun validate(user: User,
                 value: String,
                 fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user ${user.id}: empty $fieldName")
        }
```

```
    }

    validate(user, user.name, "Name")
    validate(user, user.address, "Address")

    // Save user to the database
}

fun main() {
    saveUser(User(1, "", ""))
}
```

- 로컬 함수로 분리하여 중복을 없애는 동시에 코드 구조를 깔끔하게 유지
  - 하지만 User 객체를 로컬 함수에게 일일이 전달해야 한다.
- 로컬 함수는 자신이 속한 바깥 함수의 모든 파라미터와 변수를 사용할 수 있다.

로컬 함수에서 바깥 함수의 파라미터 접근

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
    fun validate(value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user ${user.id}: " +
                    "empty $fieldName")
        }
    }

    validate(user.name, "Name")
    validate(user.address, "Address")

    // Save user to the database
}

fun main() {
```

```
    saveUser(User(1, "", ""))
}
```

- 추가로 확장 함수로 검증 로직을 만들어 개선할 수 있다.

검증 로직을 확장 함수로 추출하기

```
fun User.validateBeforeSave() {
    fun validate(value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user $id: empty $fieldName")
        }
    }

    validate(name, "Name")
    validate(address, "Address")
}

fun saveUser(user: User) {
    user.validateBeforeSave()

    // Save user to the database
}

fun main() {
    saveUser(User(1, "", ""))
}
```

- User 클래스가 만약 라이브러리에 있는 클래스 혹은 검증 로직은 User를 사용하는 다른 곳에서는 쓰이지 않는 기능이기 때문에 User에 포함시키고 싶지 않다면 확장 함수로 추출하는 것은 매우 유용하다.
- 확장 함수를 로컬 함수로 정의할 수도 있다.
    - 하지만 내포된 함수의 깊이가 깊어지면 코드를 읽기가 상당히 어려워진다.
- 따라서 일반적으로는 한 단계만 함수를 내포시키라고 권장한다.

요약

- 코틀린은 자체 컬렉션 클래스를 정의하지 않지만 자바 클래스를 확장해서 더 풍부한 API를 제공한다.
- 함수 파라미터의 기본값을 정의하면 오버로딩한 함수를 정의할 필요성이 줄어든다.
- 이름 붙인 인자를 사용하면 함수의 인자가 많을 때 함수 호출의 가독성을 더 향상시킬 수 있다.
- 코틀린 파일에서 클래스 멤버가 아닌 최상위 함수와 프로퍼티를 직접 선언 할 수 있다.
    - 이를 통해 코드 구조를 더 유연하게 만들 수 있다.
- 확장 함수와 프로퍼티를 사용하면 외부 라이브러리에 정의된 클래스를 포함해 모든 클래스의 API를 그 클래스의 소스코드를 바꿀 필요 없이 확장할 수 있다.
- 중위 호출을 통해 인자가 하나밖에 없는 메서드나 확장 함수를 더 깔끔한 구문으로 호출할 수 있다.
- 코틀린은 정규식과 일반 문자열을 처리할 때 유용한 다양한 문자열 처리 함수를 제공한다.
- 자바 문자열로 표현하려면 수많은 이스케이프가 필요한 문자열의 경우 3중 따옴표 문자열을 사용하면 더 깔끔하게 표현할 수 있다.
- 로컬 함수를 써서 코드를 더 깔끔하게 유지하면서 중복을 제거할 수 있다.

질문정리

3챕터 질문 정리