13장 DSL

다루는 내용

- DSL 만들기
- 수신 객체 지정 람다 사용법
- invoke 관례 사용법

표현력이 좋은 커스텀 코드 구조 만들기

- 공동의 목표는 가독성과 유지 보수성을 좋게 유지하는 것
- 개별 클래스에만 집중하는 것이 아닌 클래스 간 상호작용이 일어나는 API를 잘 만들어야 한다

깔끔한 API란?

1. 코드를 읽는 사람이 명확하게 이해할 수 있는 이름과 개념을 담아야 한다
2. 불필요하고 번잡한 구문이 가능한 적어야 한다

깔끔한 API를 위해 코틀린이 지원하는 기능

일반 구문	리팩토링	사용한 기능
`StringUtil.capitalize(s)`	`s.capitailize()`	확장 함수
`1.to("one")`	`1 to "one"`	infix fun
`set.add(2)`	`set += 2`	연산자 오버로딩
`map.get("key")`	`map["key"]`	get 메서드 관례
`file.use( { f -> f.read() } )`	`file.use { f -> f.read() }`	람다 추출 관례

`sb.append("yes")`
`sb.append("no")`
	`with (sb) {`
	`append("yes")`
	`append("no")`
	`}`
		수신 객체 지정 람다

일반 구문	리팩토링	사용한 기능

`val m mutableListOf<Int>()`
`m.add(1)`
`m.add(2)`
`return m.toList()`
	`return buildList {`
	`add(1)`
	`add(2)`
	`}`
		람다를 받는 빌더 함수

이러한 언어적 특성에 더해 도메인 특화 언어인 DSL을 구축해 깔끔한 API를 만들어보자

DSL(Domain-Specific Language): 도메인 특화 언어

- 특정 영역(도메인)에 초점을 맞추고 그 영역에 필요하지 않은 기능을 없앤 언어
  - ex) SQL, 정규식
- 보통의 언어가 연산을 위해 순차적으로 기술하는 명령적 언어라면 dsl은 원하는 결과를 기술하기만 하고 그 결과를 달성하기 위해 필요한 세부 실행은 언어를 해석하는 엔진에 맡기는 선언적 언어이다.
  - ex) SQL의 DELETE 질의를 만들 때 우리는 실제로 테이블의 각 레코드를 어떻게 순회해서 어떻게 추출해서 어떤 동작을 할지 전혀 관여하지 않고, 질의 실행 엔진이 질의를 받아 최적의 방법을 실행하도록 한다

DSL의 단점

- DSL은 자체 문법이 있어 다른 언어의 프로그램 안에 직접 포함 시킬 수가 없기 때문에 별도의 파일이나 문자열 리터럴로 저장해야 한다
- 따라서 범용 언어로 만든 호스트 애플리케이션과 DSL을 조합하기가 어렵다

코틀린 내부 DSL

- 독립적인 문법을 갖는 외부 DSL과 반대로 범용 언어(코틀린)와 동일한 문법을 사용하는 DSL
- 완전히 다른 언어가 아닌 주 언어(코틀린)를 별도의 문법으로 사용해 DSL의 장점을 유지하는 방법
  - ex) 외부 DSL은 SQL을 사용하고 내부 DSL은 코틀린 베이스의 Exposed 프레임워크 사용

DSL의 구조

명령-질의 API(전형적인 라이브러리)

- 여러 메서드로 이루어지며 클라이언트는 한 번에 하나씩 메서드를 호출해 사용
- 함수 호출 시퀀스에는 구조가 없으며 호출과 호출 사이 맥락이 유지되지 않는다

반면, DSL은?

- 람다를 내포시키거나 메서드 호출을 연쇄시키는 등 DSL 문법으로 구조를 만든다
  - ex) SQL 질의를 위해 여러 메서드 호출 조합 필요
- 자연어와 비슷한 구조로 만들어지고 구조(블럭)이 형성되기 때문에 가독성이 좋다

- 같은 맥락을 매 함수 호출시마다 반복하지 않고도 재사용할 수 있다는 장점이 있다

```
// DSL 버전
dependencies {
    testImplementation(kotlin("test"))
    implementation("org.jetbrains.kotlin:kotlin-stdlib")
}

// 일반 버전
project.dependencies.add("testImplementation", kotlin("test"))
project.dependencies.add("implementation", "org.jetbrains.kotlin:kotlin-st
```

- 메서드 호출 연쇄를 통해 같은 맥락 안에서 구조를 만들 수 있다

```
// kotest
str should startWith("kot")

// junit
assertTrue(str.startsWith("kot"))
```

예제) 내부 DSL로 HTML 작성

- 직접 HTML 텍스트를 작성하지 않고 내부 DSL을 사용하면 타입 안전하게 작성할 수 있고 주 언어(코틀린) 코드를 활용할 수 있다

```
fun createAnotherTable() = createHTML().table {
    val numbers = mapOf(1 to "one", 2 to "two")
    for ((num, string) in numbers) {
        tr {
            td { +"$num" }
            td { +string }
        }
    }
}
```

DSL에서 수신 객체 지정 람다 사용

수신 객체 지정 람다

buildString() - 일반 람다 버전

```
fun buildString(
    builderAction: (StringBuilder) → Unit
): String {
    val sb = StringBuilder()
    builderAction(sb)
    return sb.toString()
}

fun main() {
    println(
        buildString {
            it.append("Hello, ")
            it.append("World!")
        }
```

```
    )   // Hello, World!
}
```

- 람다 본문에서 매번 `it`을 사용해 `StringBuilder` 인스턴스를 참조해줘야 한다

buildString() - 수신 객체 지정 람다 버전

```
fun buildString(
    builderAction: StringBuilder.() → Unit
): String {
    val sb = StringBuilder()
    sb.builderAction()
    return sb.toString()
}

fun main() {
    println(
        buildString {
            append("Hello, ")
            append("World!")
        }
    )   // Hello, World!
}
```

- `buildString()`의 목적이 `StringBuilder`를 텍스트로 채우는 것이므로 수신 객체 지정 람다를 사용해야 한다
- 람다의 인자 중 하나에 수신 객체라는 상태를 부여하면 이름을 명시하지 않아도 바로 사용할 수 있다

일반 함수 타입 대신 확장 함수 타입을 사용한 이유

- 일반 함수 타입의 경우 `(T) → R` 명시적인 참조를 통해 함수를 호출해야 한다
    - 즉 수신 객체 T의 멤버에 접근하려면 반드시 `it` 같은 변수명으로 접근해야 한다
- 반면 확장 함수 타입의 경우엔 `T.() → R` 은 암묵적 수신 객체 `T`를 통해 마치 내부 DSL처럼 바로 사용 가능하다

- 마치 StringBuilder 내부에 직접 들어가서 코드를 작성하는 느낌으로 사용 가능하다
- 확장 함수를 생각해보면 비슷하게 그 클래스 내부에서 호출하듯이 사용할 수 있었다

왜 암묵적 수신 객체라고 표현하지?

- 위의 예시에서 실제 내부 코드에서 실행되는 코드는 `sb.builderAction()` 부분이다
- 코틀린 내부적으로는 이 호출이 `builderAction.invode(sb)` 처럼 변환된다
  - 람다는 함수 객체이기 때문에 코틀린에서는 함수 타입의 객체를 마치 함수처럼 호출할 수 있도록 `invoke` 연산자 오버로딩을 제공하기 때문
  - 함수 타입 `StringBuilder.() -> Unit`은 사실상 다음과 같은 익명 객체

```
val builderAction = object : Function1<StringBuilder, Unit> {
    override fun invoke(receiver: StringBuilder) {
        // 람다 실행
    }
}
```

- 이 때 `sb`는 수신 객체로 바인딩되고, 람다 블록 내부에서는 `this`가 자동으로 `sb`가 되기 때문에 `append()`를 `this` 없이 바로 호출 가능했던 것이다
  - `sb`가 수신 객체로 바인딩되는 이유는 `invoke()`가 `sb`를 첫 번째 인자로 받아 람다의 컨텍스트 내부에 `this = sb`로 설정하기 때문이다
  - 그래서 수신 객체(`sb`)가 명시적으로 참조되진 않았지만 람다 내부에서 자동으로 `this`로 작동한다
  - 그래서 **"암묵적 수신 객체"**라고 부르는 것

스코프 함수 `apply`와 `with`의 수신 객체 지정 람다 활용

- 두 함수 모두 수신 객체 지정 람다를 인자로 사용해 람다 내부에서 `this` 생략하고 수신 객체(`T`)의 멤버에 바로 접근할 수 있도록 했다

`apply()`

```
public inline fun <T> T.apply(block: T.() → Unit): T {
    block()      // this.block()과 동일.
    return this  // 수신 객체(자기 자신) 반환
}
```

- `T`의 확장 함수이면서 수신 객체 지정 람다 (`T.() → Unit`)을 인자로 사용
- 즉, `apply`의 호출 대상도 `T`이고, 람다 안의 수신 객체도 `T`이기 때문에 람다 안에서 `T`의 멤버에 바로 접근해 `block`을 수행하고 수행된 자기 자신(`this`)를 그대로 반환한다
    - 여기서도 마찬가지로 block()은 내부적으로 컴파일러에 의해 `block.invoke()`로 변환될 것

with()

```
public inline fun <T, R> with(receiver: T, block: T.() → R): R {
    return receiver.block()
}
```

- 수신 객체 `T`와 그 `T`에 대한 수신 객체 지정 람다를 인자로 사용
- 수신 객체 지정 람다를 사용하므로 람다 내부에서 receiver의 멤버를 this 없이 바로 접근해 block을 실행한다
- `apply`와 비교했을 때
    - `T`의 확장함수가 아닌 T를 인자로 사용
    - 람다를 적용한 `T`를 반환하는 것이 아닌 람다의 결과를 반환

수신 객체 지정 람다를 HTML 빌더 안에서 사용

이름 결정 규칙이 각 람다의 수신 객체에 의해 결정된다

```
fun createAnotherTable() = createHTML().table {
    val numbers = mapOf(1 to "one", 2 to "two")
    for ((num, string) in numbers) {
        tr {
            td { +"$num" }
```

```
            td { +string }
        }
    }
}
```

- 각 람다는 자신의 본문에서 호출될 수 있는 함수들을 새로 추가한다
- 즉, `table` 함수에 넘겨진 람다 안에서는 `tr` 함수로 `<tr>` 태그를 만들 수 있지만 이 람다의 밖에서는 `tr` 함수에 접근할 수 없다
    - 마찬가지로 `td` 함수는 `tr` 람다 안에서만 접근 가능하다

만약 일반 람다 함수를 인자로 사용했다면?

- 태그를 생성할 때마다 it처럼 명시적인 이름을 정의해줘야만 해서 난잡한 코드가 된다

```
fun createAnotherTable() = createHTML().table {
    this@table.tr {
        (this@tr).td { +"cell" }
    }
}
```

람다의 내포 깊이가 깊어지면 수신 객체가 무엇인지 분명하지 않아 혼동이 올 수 있다

- 수신 객체 지정 람다가 다른 수신 객체 지정 람다 안에 들어가면 안쪽 람다에서 외부 람다에 정의된 수신 객체를 사용할 수 있다

```
createHTML().body {
    a {
        img {
            href = "https://..."    // a에 전달된 람다의 수신 객체의 href를 사용해버림
        }
    }
}
```

- 이를 막기 위해 코틀린은 `@DslMarker`로 내포된 람다에서 외부 람다의 수신 객체에 접근하지 못하도록 제한할 수 있게 한다

- 메타어노테이션 `@DslMarker`는 어노테이션에 적용할 수 있는 어노테이션으로 `HtmlTagMarker`에 사용되고 있다

```
@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
open class Tag
```

  - `IMG`와 `A` 태그에 `@HtmlTagMarker`를 붙여주면 `img` 람다(`this: IMG`) 안에서 `a` 람다의 수신 객체(`this: A`)를 직접 사용할 경우 컴파일 에러가 발생한다
  - `TABLE`, `TR`, `TD` 등 주요 태그들 전부 `@HtmlTagMarker`가 붙어 있다

태그를 생성하는 함수들은 수신 객체 지정 람다를 인자로 사용한다

```
fun table(init: TABLE.() -> Unit) = TABLE().apply(init)

fun tr(init: TR.() -> Unit) {
    val tr = TR()
    tr.init()
    children.add(tr)  // 바깥 태그의 자식 리스트에 추가
}

fun createTable() =
    table {
        tr {
                    // ..
        }
    }
```

- 각 태그에는 자식들에 대한 참조를 저장하는 children 리스트가 존재한다.
- 최상위 태그 `table`이 있을 때 `tr` 함수는 새로운 `TR` 인스턴스를 만들고 바깥 태그의 (`table`) 자식 리스트에 새로 만든 `TR` 인스턴스를 추가해야 한다

태그들을 생성하고 초기화한 뒤 바깥 태그의 자식으로 추가하는 로직을 모든 태그가 공유한다

- 상위 클래스 `Tag`에 `doInit` 메서드로 이 공통 로직을 뽑아낸다.
  1. 수신 객제 지정 람다 인자를 받아 호출해 생성 및 초기화하는 로직
  2. 생성한 인스턴스를 바깥 태그의 자식 리스트에 저장하는 로직

```
@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
open class Tag(val name: String) {
    private val children = mutableListOf<Tag>() // 모든 내포(자식) 태그 참조 저장

    protected fun <T: Tag> doInit(child: T, init: T.() -> Unit) {
        child.init()            // 1. 자식 태그 초기화
        children.add(child)     // 2. 자식 태그에 대한 참조 저장
    }

    override fun toString() = "<$name>${children.joinToString("")}</$name>"
}
```

- 모든 태그에는 내포 태그를 저장하는 `children` 리스트가 존재하고 문자열로 렌더링하는 `toStirng()` 메서드도 있다
  - 자기 이름을 태그 안에 넣고 모든 자식을 재귀적으로 문자열로 렌더링

이제 모든 태그에서 이 공통 로직 `doInit` 메서드를 사용한다

```
fun table(init: TABLE.() -> Unit) = TABLE().apply(init)

class TABLE : Tag("table") {
    fun tr(init: TR.() -> Unit) = doInit(TR(), init)
}

class TR : Tag("tr") {
```

```
    fun td(init: TD.() -> Unit) = doInit(TD(), init)
}

class TD : Tag("td")
```

태그 생성 함수 자체가 새로 생성한 태그를 직접 부모의 자식 리스트에 추가하기 때문에 태그를 동적으로 만들 수 있다

```
fun createAnotherTable() = table {
    for (i in 1..2) {
        tr {
            td {

            }
        }
    }
}

fun main() {
    println(createAnotherTable())
    // <table><tr><td></td></tr><tr><td></td></tr></table>
}
```

코틀린 빌더: 추상화와 재사용

코틀린 내부 DSL을 사용할 경우 일반 코드와 마찬가지로 반복되는 내부 DSL 코드 조각을 함수로 묶어 재사용할 수 있다

HTML 목차 생성 - 일반 코틀린 DSL 사용 버전

```
fun buildBookList() = createHTML().body {
    ul {
        li { a("#1") { +"Chapter 1" } }
        li { a("#2") { +"Chapter 2" } }
```

```
        li { a("#3") { +"Chapter 3" } }
    }

    h2 { id = "1"; +"Chapter 1" }
    p { +"챕터 1" }
    h2 { id = "2"; +"Chapter 2" }
    p { +"챕터 2" }
    h2 { id = "3"; +"Chapter 3" }
    p { +"챕터 3" }
}
```

HTML 목차 생성 - 도우미 함수 사용 버전

```
fun buildBookList2() = createHTML().body {
    listWithToc {
        item("Chapter 1", "챕터 1")
        item("Chapter 2", "챕터 2")
        item("Chapter 3", "챕터 3")
    }
}
```

- 도우미 함수를 정의해 세부 사항을 감추고 재사용해 깔끔한 사용이 가능하다

도우미 함수 구현

- 먼저 `LISTWITHTOC` 클래스를 직접 정의한다
    - `item` 함수로 모든 제목/본문 쌍을 추적하는 `entries` 프로퍼티를 제공한다

```
@HtmlTagMarker
class LISTWITHTOC {
    val entries = mutableListOf<Pair<String, String>>()
    fun item(headline: String, body: String) {
        entries += headline to body
    }
}
```

- 실제 `BODY`에 전달되는 람다 안에서 사용될 수 있도록 BODY의 확장함수로 만들어줘야 한다
  - `block()`으로 `item`들을 추가해 최종 `entries`를 구성하고 공통 로직을 구현한다
  - 이 때 `item()` 메서드를 `listwithToc`에 전달되는 람다 안에서 바로 호출하기 위해 수신 객체 지정 람다를 인자로 사용한다

```
fun BODY.listWithToc(block: LISTWITHTOC.() -> Unit) {
    val listWithToc = LISTWITHTOC()
    listWithToc.block()

    ul {
        for ((index, entry) in listWithToc.entries.withIndex()) {
            li { a("#$index") { +entry.first} }
        }
    }
    for ((index, entry) in listWithToc.entries.withIndex()) {
        h2 { id = "$index"; +entry.first }
        p { +entry.second }
    }
}
```

- 여기서 `ul`, `h2` 등의 함수를 호출할 수 있는 이유는 `listWithToc`이 `BODY`의 확장함수이기 때문이다

---

invoke 관례 활용

`invoke` 관례 복습

- 특별한 이름의 함수를 일반 함수 이름처럼 호출하지 않고 더 간결한 표기로 호출
- `operator` 변경자가 붙은 `invoke()`가 들어있는 클래스의 객체를 함수처럼 호출 가능

```
class Greater(val greeting: String) {
    operator fun invoke(name: String) {
```

```
        println("$greeting, $name!")
    }
}

fun main() {
    val bavarianGreeter = Greater("Servus")
    bavarianGreeter("Dmitry")  // Greeter 인스턴스를 함수처럼 호출 가능
    // Servus, Dmitry!
}
```

- `Greeter` 인스턴스를 함수처럼 호출 가능하고, 내부적으로는 `invoke()`로 컴파일된다
    - 즉, `bavarianGreeter("Dmitry")`가 `bavarianGreeter.invoke("Dmitry")`로 컴파일

invoke는 시그니처에 대한 요구사항이 없다

- 원하는 대로 파라미터의 개수나 타입을 지정할 수 있다
- 오버로딩도 가능

코틀린 람다의 호출 방식은 invoke 관례를 따른다

코틀린에서 람다는 일급 객체(함수도 값처럼 취급)로 사용할 수 있어서 일반 함수처럼 `()`로 호출 가능하다

```
val sum = { x: Int, y: Int -> x + y }
val result = sum(1, 2)
```

- 단순한 함수 호출처럼 보이지만 내부적으로는 `invoke` 메서드 호출로 컴파일 된다
- 람다는 컴파일 시에 `FunctionN` 인터페이스를 구현하는 익명 클래스로 변환되는데 이 인터페이스에는 이름이 나타내는 수만큼의 파라미터를 받는 `invoke()` 메서드가 정의되어 있다
- 따라서 실제로는 아래처럼 변환되어 동작한다

    ```
    val result = sum.invoke(1, 2)
    ```

- 코틀린 `invoke` 관례에 의해 함수 타입의 객체를 `()`로 호출해 내부적으로 `Invoke()`가 자동으로 호출되는 것

- 참고로 함수 타입이 `nullable`할 때에는 `lambda?.invoke()`처럼 세이프콜로 안전하게 호출할 필요가 있다
  - ex) `val f : ((Int) -> Int)? = null`

invoke 관례 활용 예시) Gradle 의존관계 선언

유연하게 내포된 블록 구조를 허용하는 동시에 평평한 함수 호출 구조도 제공하고 싶은 경우

```
// 평평한 함수 호출 구조
dependencies.implementation("org.springframework.boot:spring-boot-starter

// 내포된 블록 구조
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-reflect")
}
```

평평한 함수 호출은 그냥 함수로 정의하면 되고 내포된 블록 구조는 `dependencies` 안에 람다를 받는 `invoke` 메서드를 정의해주면 된다

- `dependencies` 객체는 `Dependencyhandler` 클래스의 인스턴스로 `implementation` 및 `invoke`가 정의되어 있다

```
class DependencyHandler {
    fun implementation(coordinate: String) {    // 일반적인 명령형 API
        println("Added dependency on $coordinate")
    }

    operator fun invoke(    // invoke로 DSL 형태의 API 제공
        body: DependencyHandler.() -> Unit
    ) {
        body()  // this가 수신 객체가 되므로 바로 body() 접근
    }
}
```

dependencies를 함수처럼 호출하면서 람다를 인자로 넘기면 일어나는 일

```
fun main() {
        dependencies {
          implementation("org.jetbrains.kotlin:kotlin-reflect")
        }
}
```

1. 람다를 함수처럼 호출하면 `dependencies { ... }` → `dependencies.invoke { ... }`로 변환된다
2. invoke는 `DependencyHandler.() -> Unit` 타입의 수신 객체 지정 람다를 인자로 받는다
3. `body()` 호출 시 람다 블록 안에서 암묵적 수신 객체인 `this`는 `DependencyHandler`라서 바로 접근 가능하다
4. `implementation` 함수는 `DependencyHandler`의 멤버 함수니까 직접 호출된다

실전 코틀린 DSL

kotest에서 중위 호출을 활용한 방법

kotest를 사용하면 일반 영어처럼 읽히도록 코드를 짤 수 있다

```
s should startWith("K")
```

`should`가 중위 호출을 활용해 `Matcher<T>`를 받아서 테스트하도록 구현했기 때문이다.

```
infix fun <T> T.should(matcher: Matcher<T>) = matcher.test(this)
```

`Matcher<T>`는 값에 대한 단언문을 표현하는 제네릭 인터페이스이다

- `Matcher`를 구현한 `startWith`를 `should`의 인자로 지정한 것

```
interface Matcher<T> {
    fun test(value: T)
```

```
}

fun startWith(prefix: String): Matcher<String> {
    return object : Matcher<String> {
        override fun test(value: String) {
            if (!value.startsWith(prefix)) {
                throw AssertionError("Expected '$value' to start with '$prefix'")
            }
        }
    }
}
```

- `object : Matcher { ... }`로 익명 클래스로 `Matcher` 싱글톤 인스턴스 활용
- 이렇게 중위 호출과 `object`로 정의한 싱글톤 객체 인스턴스를 조합하면 간결한 DSL 설계 가능
- 정적 타입 지정 언어로 남기에 타입 안전하게 사용 가능하다는 장점도 있다

kotlinx.datetime에서 확장 함수를 활용한 방법

날짜와 시간을 간결하게 다루기 위한 DSL을 확장 함수 형태로 제공한다

```
val yesterday = now - 1.days
val later = now + 5.hours
```

날짜 조작 DSL

```
public inline val Int.days
        get() = toDuration(DurationUnit.DAYS)

public val Int.hours
        get() = toDuration(DurationUnit.HOURS)
```

Exposed에서 멤버 확장 함수를 활용한 방법

멤버 확장 ?

- 클래스 안에서 확장 함수와 확장 프로퍼티를 선언하는 기법
- 외부 타입을 확장하는 함수처럼 동작하면서도 해당 클래스 내부에서만 사용 가능한 멤버가 된다

멤버 확장 함수를 활용한 이유 (예시: `autoIncrement`)

```
class Table {
    fun Column<Int>.autoIncrement(): Column<Int>
    // ...
}
```

- `Column<Int>`을 확장하면서 자신의 수신 객체를 그대로 다시 반환하기 때문에 연쇄 호출할 수 있다
    - `Column<Int>`로 제한하는 효과도 있어 타입 안정성을 지켜준다
- `autoIncrement`는 Table 클래스의 멤버이기는 하지만 `Column`의 확장 함수이기도 한 멤버 확장 함수이다
- 멤버 확장 함수이기에 `Table`이라는 맥락 안에서만 호출될 수 있고, 그 맥락 안의 다른 멤버들에 접근할 수 있다
    - 외부 수신자인 `Table`에 대한 참조도 가지고 있고 확장 대상인 `Column<Int>`에 대한 참조도 가지고 있다
    - `this` 키워드로 수신 객체인 `Column<Int>`에 접근할 수 있고 `this` 없이 또는 `this@Table`로 `Table` 멤버에 접근이 가능하다

```
class Table {
    val columns = mutableListOf<Column<*>>()  // Table 내부 상태

    fun Column<Int>.autoIncrement(): Column<Int> {
        this.autoInc = true              // Column 내부 조작
        columns.add(this)                // Table 멤버에 접근도 가능
        return this
```

```
    }
}
```