Chapter#11 DSL 만들기

Features

- 영역 특화 언어 만들기
- 수신 객체 지정 람다 사용
- invoke 관례 사용
- 기존 코틀린 DSL 예제

DSL (영역 특화 언어)

- Domain-Specific-Language
- DSL을 사용해 표현력이 좋고 코틀린 다운 API를 설계하는 방법을 설명한다.
- DB 접근, HTML 생성, 테스트, 빌드 스크립트 작성, 안드로이드 UI 레이아웃 정의 등의 여러 작업에서 사용할 수 있다.

API에서 DSL로

- DSL을 알아보기 전에 우리가 해결하려는 문제에 대해 자세히 알아야 한다.
- 궁극적으로 목표는 `코드의 가독성`과 `유지 보수성`을 위함.
    - 즉, API 가 깔끔해야 한다는 뜻.

API에서 DSL로

- API가 깔끔하다는 말이 의미하는 것.

- 코드를 읽는 독자들이 어떤 일이 벌어질지 명확하게 이해할 수 있어야 한다.
    - 이름과 개념을 잘 선택하면 이런 목적을 달성할 수 있다.
- 코드가 간결해야 한다.
    - 불필요한 구문이나 번접한 준비 코드가 가능한 적어야 한다.
    - 즉, 깔끔한 API란 언어에 기본 내장된 기능과 거의 구분 할 수 없는 수준.

API에서 DSL로

- API가 깔끔하다는 말이 의미하는 것.

#. 일반 구문

```
StringUtil.capitalize(s)

1.to("one")

set.add(2)

map.get("key")

file.use({ file -> file.read() })

sb.append("yes")
sb.append("no")
```

#. 간결한 구문

```
s.capitalize()

1 to "one"

set += 2

map["key"]

file.use { it.read() }

with (sb) {
    append("yes")
    append("no")
}
```

API에서 DSL로

- 코틀린 DSL 맛보기

```
# 하루 전 날을 구하는 DSL
val yesterday = 1.days.ago

>>> 2018.12.11
```

```
# HTML Table을 생성하는 DSL
createHTML().table {
    tr {
        td { +"cell" }
    }
}

>>> <table>
>>>   <tr>
>>>     <td>cell</td>
>>>   </tr>
>>> </table>
```

> 11장에서는 위의 예제를 어떻게 구현하는지 살펴본다.

영역 특화 언어라는 개념

- DSL이라는 개념은 프로그래밍 언어라는 개념과 거의 마찬가지로 오래된 개념

- 범용 프로그래밍 언어
    - 컴퓨터로 풀 수 있는 모든 문제를 충분히 풀 수 있는 기능을 제공하는 것
    - 흔히 우리가 알고 있는 일반적인 프로그래밍 언어 (Java, Kotlin, C#, Python, Go....)
    - **명령적** (imperactive)
        - 어떤 연산을 완수하기 위해 필요한 각 단계를 순서대로 정확히 기술
- 영역 특화 언어
    - 특정 영역에 초점을 맞추고 그 영역에 필요하지 않은 기능을 없앤 것
    - 대표적인 예로 SQL과 정규식이 있다.
    - **선언적** (declarative)
        - 원하는 결과를 기술하기만 하고, 그 결과를 달성하기 위해 필요한 세부 실행은 언어를 해석하는 엔진에게 맡김

~~... 내용이 너무 길다. 나머지는 책을 읽어 보도록 유도하자...(p.478 ~ 479)~~

내부 DSL

- DSL의 단점을 해결하면서 DSL의 다른 이점을 살리기 위해 유명해지고 있는 개념
    - 그렇다면 우선 DSL의 단점을 알아보자
    - DSL의 한가지 단점으로, 범용 프로그래밍 언어로 만든 호스트 애플리케이션과 함께 조합하기가 어렵다.
    - ~~... 그만 알아보자~~ (p.479)
- 범용 프로그래밍 언어로 작성된 프로그램의 일부이며, 언어와 동일한 문법을 사용한다
- 즉, 내부 DSL은 다른 언어가 아니라 DSL의 핵심 장점을 유지하면서 주 언어를 특별한 방법으로 사용하는 것

내부 DSL

- 외부 DSL과 내부 DSL 를 코드로 비교하기

- `Customer`와 `Country` 두개의 테이블이 있고,
- `Customer` 테이블에는 `Country` 테이블에 대한 참조가 있다
- 가장 많은 고객이 살고 있는 나라를 알아내기

- 외부 DSL

```
SELECT Country.name, COUNT(Customer.id)
FROM Country
JOIN Customer
  ON Country.id = Customer.country_id
GROUP BY Country.name
ORDER BY COUNT(Customer.id) DESC
LIMIT 1
```

- 내부 DSL (Exposed 프레임워크)

```
(Country join Customer)
    .slice(Country.name, Count(Customer.id))
    .selectAll()
    .groupBy(Country.name)
    .orderBy(Count(Customer.id), isAsc = false)
    .limit(1)
```

DSL의 구조

- DSL과 일반적인 API 사이를 구분하는 잘 정의된 일반적인 경계는 없다.
  - 그래서 `내 생각에 그건 DSL이야`와 같은 말을 쉽게 들을 수 있다고 한다.
- 하지만 DSL에는 일반 API에는 존재하지 않는 구조적인 특성이 있다.

DSL의 구조

- 전형적인 라이브러리
    - 여러 메서드들로 이루어져 있다
    - 클라이언트는 그런 메소드를 한번에 하나씩 호출함으로써 사용한다
    - 함수 호출 시퀀스에 아무런 구조가 없으며, 호출과 호출 사이에 아무런 맥락도 존재하지 않는다.
    - **명령-질의** (command-query)
- DSL
    - 메서드의 호출이 DSL 문법에 의해 정해지는 더 커다란 구조에 속함.
    - 코틀린의 DSL은 보통 `lambda` 를 중첩 시키거나, 메서드의 호출을 연쇄 시키는 방식으로 구조를 만든다.

~~... 이것도 역시 책을 읽어보자... (p.481 ~ 482)~~

DSL의 구조

- Gradle 의 의존성 관련 예시

```
// DSL
dependencies {
    implementation("junit:junit:4.11")
    implementation("com.google.inject.guice:4.1.0")
}

// 명령-질의
project.dependencies.add("implementation", "junit:junit:4.11")
project.dependencies.add("implementation", "com.google.inject.guice:4.1.0")
```

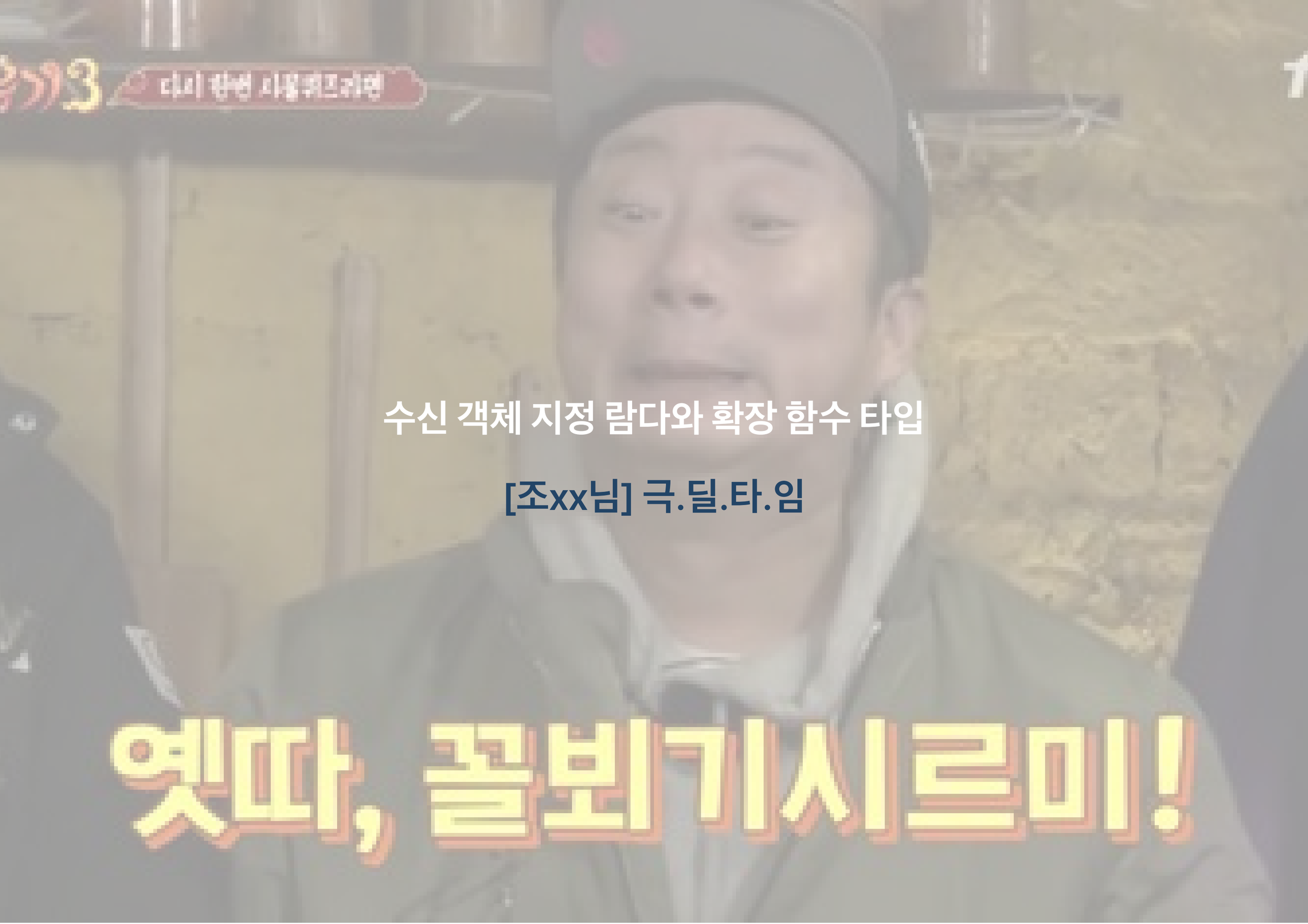
- 테스트 프레임워크 관련 예시

```
// 내부 DSL (kotlintest)
str should startWith("kot")

// jUnit의 API
assertTrue(str.startWith("kot"))
```

수신 객체 지정 람다와 확장 함수 타입

[조xx님] 극.딜.타.임



수신 객체 지정 람다와 확장 함수 타입

- 문제#1
  - 5장에서 `buildString`, `with`, `apply` 등의 표준 라이브러리 함수를 설명하면서 `수신 객체 지정 람다`에 대해서 간략이 소개 하였다고 합니다.
  - 과연, 현재 여러분들은 `수신 객체 지정 람다`에 대해 설명할 수 있으신가요?
- 문제#2
  - 다음 슬라이드부터 예제로 `buildString`이라는 함수를 구현 합니다.
  - 첫번째 예제로, `람다를 인자로 받는 함수를 사용하는 방법`으로 `buildString` 함수를 구현하는 샘플을 보게 됩니다.
  - 과연, 현재 여러분들은 `람다를 인자로 받는 함수`에 대해 `하나의 단어`로 표현할 수 있으신가요?

수신 객체 지정 람다와 확장 함수 타입

- 람다를 인자로 받는 `buildString()` 정의하기

- 함수 정의

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

- 함수 사용

```
val s = buildString {  // it -> { /*...*/ }
    it.append("Hello, ")
    it.append("World!")
}
```

> 위의 예시코드는 이해하기 쉽다.
> 단, 사용하기에는 불편하다.

- 문제#1
  - 책의 내용을 참고하면 위의 예시 코드는 불편하다고 나옵니다.
  - 그렇다면, 여러분은 책을 보지 않고, 개발자의 입장에서 위의 예시 코드가 불편한 이유를 설명할 수 있나요?

수신 객체 지정 람다와 확장 함수 타입

- 수신 객체 지정 람다를 사용해 `buildString()` 정의하기

- 함수 정의

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

- 함수 사용

```
val s = buildString {
    this.append("Hello, ")
    append("World!")
}
```

수신 객체 지정 람다와 확장 OO OO

- 앞에서 살펴본 두개의 예제 코드의 차이점: 확장 OO OO

- 문제#1
  - 위에서 살펴본 두개의 예시코드는 모두 **함수의 매개변수로 함수를 전달 받고** 있습니다
  - 현재 **마지막 챕터**인 `11장` 을 살펴보고 있는 지금,
  - 여러분은 위에 **굵게 표시된 글자**를 **하나의 단어**로 표현할 수 있으신가요?

수신 객체 지정 람다와 확장 함수 타입

- 앞에서 살펴본 두개의 예제 코드의 차이점: 확장 함수 타입

- `buildString` 함수의 파라미터 타입을 선언할 때 **일반 함수 타입** 대신 **확장 함수 타입**을 사용했다.
- 확장 함수 타입 선언 방법
  - 람다의 파라미터 목록에 있던 수신 객체 타입을 파라미터 목록을 여는 괄호 앞으로 선언
    - `(StringBuilder) -> Unit` => `StringBuilder.() -> Unit`
  - 조금 더 복잡한 확장 함수 타입 예시
    - `String.(Int, Int) -> Int` => `(String, Int, Int) -> Int`

> 쉽게 이해한 내용으로는, `수신 객체 지정 람다` 는 `확장 함수`를 람다로 넘기는 것 😄

수신 객체 지정 람다와 확장 함수 타입

- 수신 객체 지정 람다를 변수에 저장하기

- 함수 정의

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

- 함수 사용

```
val appendExcl: StringBuilder.() -> Unit = { this.append("!") }

val stringBuilder = StringBuilder("Hi")
stringBuilder.appendExcl() // <-- appendExcl 을 확장함수처럼 호출 할 수 있다.

println(stringBuilder) // Hi!

println(buildString(appendExcl)) // <-- appendExcl 을 확장 함수 타입의 인자로 넘길 수 있다.
```

수신 객체 지정 람다와 확장 함수 타입

- 코틀린 표준 라이브러리의 예시

- 표준 라이브러리의 `buildString` 함수

```
fun buildString(builderAction: StringBuilder.() -> Unit) -> String =
    StringBuilder().apply(builderAction).toString()
```

- 표준 라이브러리의 `apply` 함수

```
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

- 표준 라이브러리의 `with` 함수

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()
```

수신 객체 지정 람다와 확장 함수 타입

- 수신 객체 지점 람다를 HTML 빌더 안에서 사용

- HTML을 만들기 위한 코틀린 DSL을 보통 `HTML Builder` 라고 부른다고 함
- 더 넓은 범위의 개념인 `Type-Safe Builder` 의 대표적인 예시

- 처음으로 `Builder` 라는 개념이 유명해진 곳은 그루비 커뮤니티 였다고 함
- `Builder` 를 사용하면 객체 계층 구조를 선언적으로 정의할 수 있는 장점이 있으며, HTML / XML / UI 컴포넌트 레이아웃 등을 정의할 때 유용하다고 한다.

수신 객체 지정 람다와 확장 함수 타입

- 수신 객체 지점 람다를 HTML 빌더 안에서 사용

- HTML의 `Table` 태그를 생성하는 예제

```
createHTML().table {
    tr {
        td { + "cell" }
    }
}
```

- 주의 깊게 봐야 될 점은, `수신 객체 지정 람다`가 이름 결정 규칙을 바꾼다는 것
  - `table` 함수에 넘겨진 람다에서는 `tr 함수`을 사용해 HTML의 `<tr>` 태그를 만들 수 있다
  - `tr` 함수에 넘겨진 람다에서는 `td 함수`를 사용해 HTML의 `<td>` 태그를 만들 수 있다
  - 즉, 이렇게 API를 설계하면 **의도하지 않아도 어쩔 수 없이 HTML 언어의 문법을 따르는 코드를 작성 할 수 있다**

수신 객체 지정 람다와 확장 함수 타입

- 수신 객체 지점 람다를 HTML 빌더 안에서 사용

- HTML 빌더를 위한 태그 클래스 뜯어보기

```
open class Tag

class TABLE : Tag {
    fun tr(init: TR.() -> Unit)
}

class TR : Tag {
    fun td(init: TD.() -> Unit)
}

class TD : Tag

// 여긴, 개인적으로 예상되는 코드 구조...?
class HTML private constructor() : Tag {
    fun table(init: TABLE.() -> Unit)

    companion object {
        fun newInstance(): HTML = HTML()
    }
}

fun createHTML(): HTML = HTML.newInstance()
```

> `TABLE` , `TR` , `TD` 는 모두 HTML 생성에 나타나면 안되는 유틸리티 클래스
> 그래서, 이름을 모두 대문자로 만들어 일반 클래스와 구분한다. (관례)
> 조금 더 자세한 코드가 궁금하면 `kotlinx.html` 코드를 살펴보는 것도 좋다.

수신 객체 지정 람다와 확장 함수 타입

- 코틀린 빌더: 추상화와 재사용을 가능하게 하는 도구

- 외부 DSL인 SQL이나 HTML은 별도 함수로 분리해 이름을 부여하기 어렵다.
- 하지만, 내부 DSL을 사용하면 일반 코드와 마찬가지로 반복되는 내부 DSL 코드 조각을 새 함수로 묶어서 재사용 할 수 있다.

수신 객체 지정 람다와 확장 함수 타입

- 부트스트랩 라이브러리를 사용해 드롭다운 메뉴 HTML 추가하기

- 기존 HTML

```
<div class="dropdown">
  <button class="btn dropdown-toggle">
    Dropdown
    <span class="caret"></span>
  </button>
  <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
    <li><a href="#">Another Action</a></li>
    <li role="separator" class="divider"></li>
    <li class="dropdown-header">Header</li>
    <li><a href="#">Separated link</a></li>
  </ul>
</div>
```

- kotlinx.html 사용

```
fun buildDropdown() = createHTML().div(classes = "dropdown") {
    button(classes = "btn dropdown-toggle") {
        + "Dropdown"
        span(classes = "caret")
    }

    ui(classes = "dropdown-menu") {
        li { a("#") { + "Action" } }
        li { a("#") { + "Another Action" } }
        li { role = "separator"; classes = setOf("divider") }
        li { classes = setOf("dropdown-header"); + "Header" }
        li { a("#") { + "Separated link" } }
    }
}
```

수신 객체 지정 람다와 확장 함수 타입

- 도우미 함수를 활용해 드롭다운 메뉴 만들기

- 도우미 함수를 활용해 만들어진 최종 코드

```
fun dropdownExample() = createHTML().dropdown {
    dropdownButton { + "Dropdown" }
    dropdownMenu {
        item("#", "Action")
        item("#", "Another Action")
        divider()
        dropdownHeader("Header")
        item("#", "Sperated link")
    }
}
```

> `kotlinx.html` 의 내부 DSL 코드 조각을 새로운 함수로 묶어서 깔끔하게 개선하고, 재사용 가능하도록 변경하였다.

invoke 관례를 사용한 더 유연한 블록 중첩

- invoke 관례를 사용하면 객체를 함수처럼 호출할 수 있음
- 단, 이 기능은 일상적으로 사용하라고 만든 기능은 아니라는 점을 유의

invoke 관례를 사용한 더 유연한 블록 중첩

- 문제#1
  - 우리는 이미 7장에서 코틀린의 관례에 대해 배웠다고 합니다.
  - 현재, 여러분은 `코틀린의 관례` 에 대해 설명할 수 있으신가요?
    > ~~저는 7장의 모임을 빠져서 설명 못해요...~~

invoke 관례를 사용한 더 유연한 블록 중첩

- invoke 관례: 함수처럼 호출할 수 있는 객체

- 코틀린의 관례란?
  - 특별한 이름이 붙은 함수를 일반적인 메소드 호출 구문으로 호출하는 대신, 더 간단한 다른 구문으로 호출 할 수 있게 지원해주는 기능
  - 대표적인 예로 인덱스 연산을 사용할 수 있게 해주는 `get` 함수에 대해 살펴봤다고 합니다.
  - `foo.get(index)` 대신, `foo[index]` 와 같은 구문으로 사용이 가능하다.
  - 이런 관례 기능을 사용할 수 있는 전제 조건은 `get` 함수가 Foo 클래스에 정의된 함수 이거나, Foo 클래스에 대해 정의된 확장 함수이어야 한다.
- invoke 관례
  - 위에서 설명한 `코틀린 관례` 와 마찬가지의 역할을 한다고 합니다.
  - 다만, `invoke` 는 `코틀린 관례` 를 사용할 때 사용하는 `각괄호` 대신 일반적인 `괄호` 를 사용한다.
  - `operator` 변경자가 붙은 `invoke` 함수를 클래스에 정의하면 객체를 함수처럼 호출 할 수 있다고 합니다.

invoke 관례를 사용한 더 유연한 블록 중첩

- invoke 관례: 함수처럼 호출할 수 있는 객체

- Greeter 클래스 정의

```
class Greeter(val greeting: String) {
    operator fun invoke(name: String) {
        println()
    }
}
```

- `invoke 관례`를 이용한 함수 호출

```
val greeter = Greeter("Servus")
// greeter.invoke("Dmitry")
greeter("Dmitry")
```

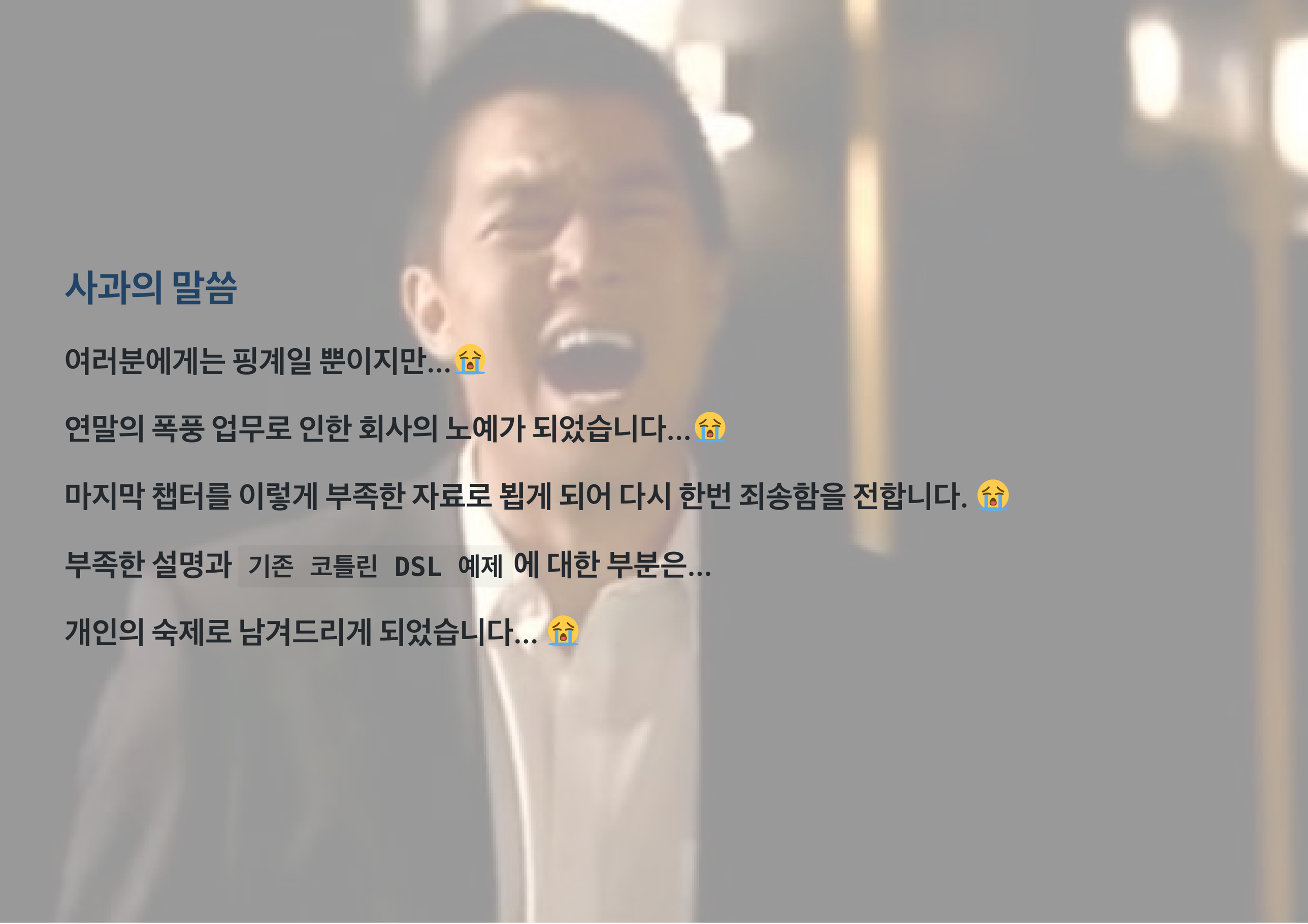
- `invoke 관례`를 통해 `Greeter` 인스터스를 함수처럼 호출할 수 있다.
- 미리 정해둔 이름을 사용한 함수를 통해서 긴 식 대신, 짧고 간결한 식을 쓸 수 있다.

> `invoke 관례`는 그리 특별할 것이 없다.

invoke 관례를 정리

- invoke 관례란?
  - 코틀린에서 미리 정해둔 이름의 함수일 뿐, 그리 특별한 것이 아니다.
  - 클래스 인스턴스를 마치 함수와 동일하게 사용하게 해주는 기능.
  - `inline` 하는 람다를 제외한 모든 람다는 `함수형 인터페이스`를 구현하는 클래스로 컴파일 된다. `(FunctionN)`
  - 즉, 코틀린에서 사용하는 람다는 결국 `invoke 관례` 기능을 통해, `FunctionN` 클래스를 함수처럼 쓰고 있는 것에 불과하다.

- 마지막 문제
  - 위에서 `inline` 이라는 키워드가 잠시 언급 되었는데요...
  - 현재, 여러분은 `inline` 키워드가 하는 역할에 대하여 설명할 수 있나요? 😝



사과의 말씀

여러분에게는 핑계일 뿐이지만... 😭

연말의 폭풍 업무로 인한 회사의 노예가 되었습니다... 😭

마지막 챕터를 이렇게 부족한 자료로 뵙게 되어 다시 한번 죄송함을 전합니다. 😭

부족한 설명과 `기존 코틀린 DSL 예제` 에 대한 부분은...

개인의 숙제로 남겨드리게 되었습니다... 😭