# 14장 코틀린 테스팅

테스트 프레임워크는 소프트웨어 개발 생태계에서 중요한 역할을 한다. 테스트 프레임워크는 소프트웨어 개발 생명주기 전반에서 소프트웨어 품질을 유지할 수 있도록 도움을 주는 재사용가능한 테스트 코드 작성을 돕는다. 잘 설계된 자바 상호운용성으로 인해 코틀린 개발자는 JVM 플랫폼을 대상으로 하는 JUnit, TestNG, Mockito 등의 수많은 테스트 도구를 활용할 수 있다.

하지만 코틀린 생태계에도 코틀린 언어를 활용해 간결하면서 표현력이 좋은 테스트 코드를 작성할 수 있게 도움을 주는 몇몇 프레임워크가 생기기 시작했다. 이번 장에서는 https://kotest.io/ 에서 찾을 수 있는 강력한 오픈소스 테스트 프레임워크인 코테스트(kotest)를 살펴본다[[1]](#footnote-1). 다음 세가지 주제를 살펴본다.

* 코테스트의 여러 명세 스타일(specification style)을 사용해 테스트 코드를 조직화하는 방법
* 매처(matcher), 인스펙터(inspector), 속성 기반 테스트를 위한 자동생성된 테스트 데이터셋(data set) 등을 사용해 테스트 단언문을 작성하는 방법
* 테스트 환경을 제대로 초기화하고 정리하는 방법과 테스트 설정을 제대로 제공하는 방법

먼저 인텔리J IDEA 프로젝트에서 코테스트를 설정하는 방법부터 살펴본다.

## 구조

* 코테스트 명세
* 단언문
* 픽스처와 설정

## 목표

코테스트 프레임워크가 제공하는 기능을 활용해 테스트 명세를 작성하는 방법을 배운다.

## 코테스트 명세

이번 절에서는 인텔리J IDEA 프로젝트에서 코테스트를 설정하는 방법과 코테스트 테스트 프레임워크가 제공하는 여러가지 테스트 레이아웃에 대해 살펴본다. 이번 장의 모든 예제는 코테스트 4.5.0을 기준으로 작성했다.

### 코테스트 시작하기

코테스트를 사용하려면 프로젝트 의존성에 코테스트를 추가해야 한다. 외부 의존성을 추가하는 방법에 대해서는 이미 *13장. 동시성*에서 코틀린 코루틴 라이브러리를 추가하는 방법을 통해 살펴봤다. 우선 Project Structure 대화창에서 추가할 의존성의 메이븐 좌표를 추가해야 한다. 메이븐 좌표로는 io.kotest:kotest-runner-junit5:4.5.0을 사용한다(그림 14.1을 보라).

메이븐이나 그레이들 같은 빌드 자동화 시스템을 사용중이라면 해당 빌드파일에 의존관계를 추가함으로써 코테스트를 사용할 수 있다.

그림 14.1: 코테스트 라이브러리 추가하기

라이브러리를 추가하고 나면 IDE가 프로젝트에 새 라이브러리 모듈을 추가하라고 권장한다. 다음 단계는 의존관계의 영역을 설정하는 것이다. 왼쪽의 뷰를 Modules로 바꾼 다음에, 관심 대상 모듈을 선택하고 Dependencies 탭을 열라. 의존관계로 새로 추가한 라이브러리의 영역이 Compile임을 알 수 있다. Compile은 프로덕션과 테스트 코드를 빌드할 때와 IDE에서 프로그램을 실행할 때 클래스패스안에 라이브러리가 들어간다는 뜻이다. 코테스트는 테스트를 할 때만 사용하므로 영역을 Test로 바꿔야 한다(그림 14.2).

그림 14.2: 의존관계 영역을 Test로 바꾸기

마지막 준비 단계는 테스트 소스 코드가 들어갈 디렉터리를 설정하는 것이다. 이미 설정된 테스트 소스코드 디렉터리가 없다면, Project 뷰에서 New | Directory를 선택해서 새 디렉터리(예: test)를 프로덕션 소스코드가 들어가는 src와 나란히 만들라. 이제 IDE에게 테스트 소스 코드의 루트를 지정할 때다. 이를 위해서는 새로 추가한 디렉터리를 오른쪽 클릭해서 Mark Directory as | Test Sources Root를 선택하라. IDEA가 이제 해당 디렉터리 안의 내용을 테스트에 사용할 소스 파일로 인식했음을 알려주는 뜻에서, 테스트 디렉터리의 색이 녹색으로 바뀐다.

코테스트와 인텔리J의 통합을 향상시켜주는 특별한 플러그인을 설치하면 좋다. Settings 대화창(File | Settings)의 Plugins 탭에서 kotest를 검색해 설치할 수 있다(그림 14.3). Install을 클릭한 후 다운로드와 설치가 끝나면 IDE를 재시작해야 한다.

그림 14.3: 인텔리J IDEA용 코테스트 플러그인 설치하기

이제는 13장까지 해왔던 것처럼 코드를 작성할 수 있다. test 디렉터리 안에 새 파일을 만들고 간단한 테스트 명세를 작성하자.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.StringSpec  
  
class NumbersTest : StringSpec({  
 "2 + 2 should be 4" { (2 + 2) shouldBe 4 }  
 "2 \* 2 should be 4" { (2 \* 2) shouldBe 4 }  
})

이 정의의 의미에 대해서는 나중에 설명한다. 하지만 이 테스트가 “2 + 2 should be 4”나 “2 \* 2 should be 4”라는 이름으로 몇몇 산술 계산의 결과가 같은지 검사한다는 사실을 알아챌 수 있을 것이다. 테스트를 실행하려면 코드 왼쪽에 있는 녹색 삼각형을 찾아보라. 녹색 삼각형 중 하나를 클릭하면 클릭한 위치에 있는 테스트만 실행하거나 전체 명세를 실행할 수 있다(그림 14.4).

그림 14.4: 코테스트 명세를 인텔리J에서 실행하기

이제 코테스트 기능을 살펴볼 준비가 됐다. 첫번째 주제는 테스트케이스를 조직화할 때 사용할 수 있는 다양한 테스트 명세 스타일이다.

## 명세 스타일

코테스트는 여러 명세 스타일을 지원한다. 어떤 스타일을 택하느냐에 따라 테스트 코드를 조직화하는 방법이 달라진다. 프로젝트에서 여러 스타일을 섞어 쓰거나, AbstractSpec 클래스나 AbstractSpec 클래스의 하위 클래스 중 하나인 AbstractStringSpec과 같은 클래스를 구현함으로써 여러분만의 명세 스타일을 만들어낼 수도 있다. 이번 절에서는 코테스트를 프로젝트에 추가하면 바로 쓸 수 있는 테스트 명세 스타일에 대해 살펴본다.

테스트케이스를 정의하려면 명세 클래스 중 하나를 상속해야만 한다. 그 후 클래스 생성자에 테스트를 주가하거나, 상위클래스 생성자에 전달하는 람다 안에 테스트를 추가한다. 테스트 자체를 정의하는 방법은 스타일에 따라 달라지며, 대부분의 경우 DSL과 비슷한 API를 통해 테스트를 정의한다. StringSpec 클래스를 사용하는 간단한 예제를 살펴보자.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.StringSpec  
  
class NumbersTest : StringSpec({  
 "2 + 2 should be 4" { (2 + 2) shouldBe 4 }  
 "2 \* 2 should be 4" { (2 \* 2) shouldBe 4 }  
})

StringSpec에서는 테스트에 대한 설명이 들어있는 문자열 뒤에 람다를 추가해서 개별 테스트를 작성한다. 여러분이 예상하는 것처럼 이는 StringSpec이 정의한 String.invoke() 확장 함수를 호출하는 것에 지나지 않는다. 예제에서 실제 검증 코드는 shouldBe 중위 연산자 함수를 사용하며, 이 함수는 수신객체로 받은 값과 인자로 받은 값이 일치하지 않으면 예외를 던진다. 이 함수는 매처 DSL에 속해있으며 이에 대해서는 다음 절에서 살펴본다.

StringSpec은 모든 테스트가 한 클래스 안에 들어가고 모든 테스트가 같은 수준에 정의되어 있는 평평한 구조를 만들어낸다. 테스트 블럭을 다른 블럭 안에 넣으려고 시도하면 프레임워크가 런타임에 예외를 발생시키면서 실패한다.

WordSpec 클래스를 사용하면 더 복잡한 테스트 레이아웃을 만들 수 있다. WordSpec을 가장 단순한 형태로 사용하면, 테스트를 정의하는 부분에서 2단계로 이뤄진 계층 구조를 만들 수 있다. 이때 각 테스트는 StringSpec과 비슷하며, should() 함수 호출에 의해 각각의 그룹으로 묶인다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.WordSpec  
  
class NumbersTest2 : WordSpec({  
 "1 + 2" should {  
 "be equal to 3" { (1 + 2) shouldBe 3 }  
 "be equal to 2 + 1" { (1 + 2) shouldBe (2 + 1) }  
 }  
})

추가로 should() 호출을 When()이나 when()으로 감싸면 테스트 계층을 3단계로 구성할 수도 있다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.WordSpec  
  
class NumbersTest2 : WordSpec({  
 "Addition" When {  
 "1 + 2" should {  
 "be equal to 3" { (1 + 2) shouldBe 3 }  
 "be equal to 2 + 1" { (1 + 2) shouldBe (2 + 1) }  
 }  
 }  
})

테스트 계층을 원하는대로 깊게 만들고 싶다면 어떻게 해야할까? FunSpec 클래스는 테스트 코드를 test() 함수 호출로 묶는다. 이 함수는 테스트에 대한 설명과 실행할 일시중단 함수를 인자로 받는다. StringSpec과 달리 이 스타일은 context 블럭으로 테스트를 한 그룹으로 묶을 수 있다.

[편집자께: NumbersTest3처럼 이름이 겹치는데 명세 방식이 다른 테스트클래스 이름은 번호를 다르게 붙였습니다.]

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.FunSpec  
  
class NumbersTest3 : FunSpec({  
 test("0 should be equal to 0") { 0 shouldBe 0 }  
 context("Arithmetic") {  
 context("Addition") {  
 test("2 + 2 should be 4") { (2 + 2) shouldBe 4 }  
 }  
 context("Multiplication") {  
 test("2 \* 2 should be 4") { (2 \* 2) shouldBe 4 }  
 }  
 }  
})

test와 context 블럭을 아무 깊이에서나 사용할 수 있다. 단, test 블럭을 test 블럭 안에 쓸 수는 없다.

IDE 팁:

IDE에서 실행하면 이런 다단계 테스트의 결과도 명세 블럭의 구조와 일치하는 계층적인 구조로 표시된다. 그림 14.5는 방금 본 테스트코드를 인텔리J IDEA에서 실행한 결과를 보여준다.

그림 14.5: 인텔리J가 다단계 테스트 결과 표시해주는 방법

ExpectSpec도 기본적으로 같지만 test() 대신 expect()를 사용하고, 추가로 최상위에 테스트를 위치시키지 못하게 한다(즉, 모든 테스트는 context() 블럭 안에 들어가야 한다).

DescribeSpec은 describe()/context() 블럭을 그룹을 짓는데 사용하고 it()을 내부에 테스트를 담기 위해 사용한다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.DescribeSpec  
  
class NumbersTest4 : DescribeSpec({  
 describe("Addition") {  
 context("1 + 2") {  
 it("should give 3") { (1 + 2) shouldBe 3 }  
 }  
 }  
})

[ShouldSpec의 경우 context를 사용해야 합니다. 문자열에 대한 invoke를 지원하지 않습니다.]

ShouldSpec은 FunSpec과 비슷한 레이아웃을 만들어낸다. 이 명세는 문맥 블럭을 그룹을 짓는데 사용하고, 말단에 테스트 블럭을 위치시킨다. 차이는 순전히 구문 차이일 뿐이다. 문맥 블럭을 만들기 위해서는 (DescribeSpec의 문맥 블럭과 비슷하게) 테스트를 설명하는 문자열에 대해 context() 호출을 사용하고, 테스트 블럭 자체는 should() 함수 호출로 정의한다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.ShouldSpec  
  
class NumbersTest5 : ShouldSpec({  
 should("be equal to 0") { 0 shouldBe 0 }  
 context("Addition") {  
 context("1 + 2") {  
 should("be equal to 3") { (1 + 2) shouldBe 3 }  
 should("be equal to 2 + 1") { (1 + 2) shouldBe (2 + 1) }  
 }  
 }  
})

비슷한 유형의 명세를 FreeSpec 클래스를 통해 구성할 수 있다. StringSpec과 비슷하게 이 명세도 문자열에 대한 invoke()를 사용해 테스트를 정의하며, -연산자(즉 minus 연산자 함수)를 통해 문맥을 소개한다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.FreeSpec  
  
class NumbersTest6 : FreeSpec({  
 "0 should be equal to 0" { 0 shouldBe 0 }  
 "Addition" - {  
 "1 + 2" - {  
 "1 + 2 should be equal to 3" { (1 + 2) shouldBe 3 }  
 "1 + 2 should be equal to 2 + 1" { (1 + 2) shouldBe (2 + 1) }  
 }  
 }  
})

코테스트는 거킨(Gherkin) 언어에서 영감을 얻은 BDD(행동 주도 개발) 명세 스타일도 지원한다. FeatureSpec에서는 feature 블럭에 의해 계층의 루트가 만들어지고, 그 안에는 구체적인 테스트를 구현하는 시나리오 블럭이 들어간다. feature 안에서 여러 시나리오(또는 그룹)을 묶어서 그룹으로 만들 때도 feature() 호출을 사용한다.

[FeatureSpec에서 and를 지원하지 않습니다. feature를 또 써야 합니다.]

import io.kotlintest.shouldBe  
import io.kotlintest.specs.FeatureSpec  
  
class NumbersTest7 : FeatureSpec({  
 feature("Arithmetic") {  
 val x = 1  
 scenario("x is 1 at first") { x shouldBe 1 }  
 feature("increasing by") {  
 scenario("1 gives 2") { (x + 1) shouldBe 2 }  
 scenario("2 gives 3") { (x + 2) shouldBe 3 }  
 }  
 }  
})

BehaviorSpec 클래스가 구현하는 비슷한 스타일은 given()/Given(), when()/When(), then()/Then()이라는 함수로 구분되는 세가지 수준을 제공한다. and()와 And()를 통해 여러 when/then 블럭을 묶어서 그룹 수준을 추가할 수도 있다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.BehaviorSpec  
  
class NumbersTest8 : BehaviorSpec({  
 Given("Arithmetic") {  
 When("x is 1") {  
 val x = 1  
 And("increased by 1") {  
 Then("result is 2") { (x + 1) shouldBe 2 }  
 }  
 }  
 }  
})

이런 블럭을 사용하면 자연어(영어)에 가까운 테스트 설명을 작성할 수 있다("when x is 1 and increased by 1 then result is 2").

마지막으로 살펴볼 명세 스타일은 AnnotationSpec이다. 이 스타일은 DSL같은 테스트 명세를 사용하지 않고, 테스트 클래스 메서드에 붙인 @Test 애너테이션에 의존한다. 이 애너테이션은 JUnit이나 TestNG와 비슷하다.

import io.kotest.matchers.shouldBe  
import io.kotest.core.spec.style.AnnotationSpec  
  
class NumbersTest9 : AnnotationSpec() {  
 @Test fun `2 + 2 should be 4`() { (2 + 2) shouldBe 4 }  
 @Test fun `2 \* 2 should be 4`() { (2 \* 2) shouldBe 4 }  
}

테스트에 @Ignore를 붙이면 어떤 테스트를 비활성화 시킬 수도 있다.

## 단언문 매처

앞에서 본 예제 코드는 여러 명세 스타일을 사용하는 방법을 보여준다. 예제에서는 단순히 수신객체와 인자가 같은지 비교하는 단언 함수인 shouldBe를 사용했다. 이 함수는 코테스트 라이브러리가 제공하는 수많은 매처 중 한가지 예일 뿐이다.

매처는 일반 함수 호출이나 중위 연산사 형태로 사용할 수 있는 확장 함수로 정의된다. 모든 매처 이름은 shouldBe로 시작한다. 이 관습은 테스트 코드에서 shouldBeGreaterThanOrEqual처럼 읽기 좋은 이름을 지정하게 도와준다. 코테스트 문서에서 전체 내장 매처 목록을 볼 수 있다. 대부분, 매처 이름이 매처 자신의 역할을 잘 설명하고 매처를 직관적으로 사용할 수 있기 때문에 여기서는 매처 개개에 대한 예제에 초점을 맞추지 않는다. 이번 절에서는 여러분이 직접 매처를 작성해서 테스트 프레임워크를 확장하는 등 더 고급 주제를 다룬다.

커스텀 매처를 정의하려면 Matcher 인터페이스를 구현하고 이 인터페이스의 test() 메서드를 오버라이드해야 한다.

[Result객체 이름이 MatcherResult로 변경됐습니다.]

abstract fun test(value: T): MatcherResult

MatcherResult 객체는 매칭 결과를 표현한다. 이 클래스는 데이터 클래스로 다음과 같은 프로퍼티가 들어있다.

* passed : 단언문을 만족하는지(true) 만족하지 않는지(false)
* failureMessage : 단언문 실패를 보여주고 단언문을 성공시키려면 어떤 일을 해야 하는지 알려주는 메시지
* negatedFailureMessage : 매처를 반전시킨 버전을 사용했는데 매처가 실패하는 경우 표시해야 하는 메시지

예를 들어 주어진 수가 홀수인지 검사하는 매처를 만들어보자.

import io.kotest.matchers.Matcher  
import io.kotest.matchers.MatcherResult  
  
fun beOdd() = object : Matcher<Int> {  
 override fun test(value: Int): MatcherResult {  
 return MatcherResult(  
 value % 2 != 0,  
 "$value should be odd",  
 "$value should not be odd"  
 )  
 }  
}

이제 이 매처를 should()/shouldNot() 등의 내장 확장 함수에 넘겨서 단언문에 사용할 수 있다.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.matchers.\*  
  
class NumbersTestWithOddMatcher : StringSpec({  
 "5 is odd" { 5 should beOdd() }  
 "4 is not odd" { 4 shouldNot beOdd() }  
})

"beOdd"라는 이름을 선택한 이유는 만들어진 단언문을 읽을 때 자연스럽게 읽히게 하기 위함이다("should be odd"/"should not be odd").

Matcher 인터페이스의 구현은 자동으로 and/or/invert 연산을 지원한다. 이 연산들은 불리언 연산 규칙에 따라 매처를 합성해준다. 이런 연산을 활용해 복잡한 술어로 구성된 단언문을 구성할 수 있다. 다음 예제는 beOdd() 매처와 내장 positive() 매처를 합성한다.

"5 is positive odd" { 5 should (beOdd() and positive()) }

매처가 지원하는 연산으로 combine()이 있다. 이 연산을 사용하면 기존 매처에 타입 변환 함수를 추가함으로써 새로운 타입에 대한 매처를 만들어준다. 다음 함수는 beOdd() 매처를 재사용해 주어진 컬렉션의 길이가 홀수인지 검사한다.

fun beOddLength() = beOdd().compose<Collection<\*>> { it.size }

모든 매처를 should()/shouldNot() 함수를 통해 호출할 수 있지만, 내장 매처 중 대부분은 should로 시작하는 특화된 함수를 함께 제공한다. 예를 들어 다음 두 단언문은 같은 역할을 한다.

5 should beLessThan(10)  
5 shouldBeLessThan(10)

## 인스펙터

코테스트는 매처 외에도 인스펙터라는 매처와 관련된 개념을 지원한다. 인스펙터는 컬렉션 함수에 대한 확장 함수로, 주어진 단언문이 컬렉션 원소 중 어떤 그룹에 대해 성립하는지 검증할 수 있게 해준다.

* forAll()/forNone() : 단언문을 모든 원소가 만족하는지(forAll()), 어느 원소도 만족하지 않는지(forNone()) 검사한다
* forExactly(n) : 단언문을 정확히 n개의 원소가 만족하는지 검사한다. n = 1인 경우에 특화된 forOne() 함수도 있다
* forAtLeast(n)/forAtMost(n) : 단언문을 최소 n개의 원소가 만족하는지(forAtLeast(n)), 최대 n개의 원소가 만족하는지(forAtMost(n)) 검사한다. n = 1이라면 forAtLeastOne()/forAtMostOne()를 쓸 수 있고, forAny()도 쓸 수 있다. forAny()는 forAtLeastOne()과 같다.

이런 인스펙터를 사용하는 예제를 살펴보자.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.inspectors.\*  
import io.kotest.matchers.ints.shouldBeGreaterThanOrEqual  
import io.kotest.matchers.shouldBe  
  
class NumbersTestWithInspectors : StringSpec({  
 val numbers = Array(10) { it + 1 }  
  
 "all are non-negative" {  
 numbers.forAll { it shouldBeGreaterThanOrEqual 0 }  
 }  
  
 "none is zero" { numbers.forNone { it shouldBe 0 } }  
  
 "a single 10" { numbers.forOne { it shouldBe 10 } }  
  
 "at most one 0" { numbers.forAtMostOne { it shouldBe 0 } }  
  
 "at least one odd number" {  
 numbers.forAtLeastOne { it % 2 shouldBe 1 }  
 }  
  
 "at most five odd numbers" {  
 numbers.forAtMost(5) { it % 2 shouldBe 1 }  
 }  
  
 "at least three even numbers" {  
 numbers.forAtLeast(3) { it % 2 shouldBe 0 }  
 }  
  
 "some numbers are odd" { numbers.forAny { it % 2 shouldBe 1 } }  
  
 "some but not all numbers are even" {  
 numbers.forSome { it % 2 shouldBe 0 }  
 }  
  
 "exactly five numbers are even" {  
 numbers.forExactly(5) { it % 2 shouldBe 0 }  
 }  
})

## 예외 처리

코테스트는 어떤 코드가 지정된 예외에 의해 중단됐는지를 검사하는 shouldThrow() 단언문을 제공한다. 이 단언문은 try/catch로 예외를 잡아내는 방식을 간편하게 대신할 수 있다. 성공적인 경우 shouldThrow()는 여러분이 활용할 수 있도록 잡아낸 예외를 반환한다.

import io.kotest.matchers.string.shouldEndWith  
import io.kotest.assertions.throwables.shouldThrow  
import io.kotest.core.spec.style.StringSpec  
  
class ParseTest: StringSpec({  
 "invalid string" {  
 val e = shouldThrow<NumberFormatException>{ "abc".toInt() }  
 e.message shouldEndWith "\"abc\""  
 }  
})

코테스트에서 예외와 관련해 유용한 기능으로는 실패한 단언문이 던진 예외를 일시적으로 무시하는 기능이다. 이를 소프트 단언문(soft assertion)이라고 부른다. 여러분의 테스트가 여러 단언문으로 이뤄져 있고, 실패한 단언문을 모두 보고 싶은 경우 소프트 단언문이 유용하다. 일반적으로는 맨 처음 예외(단언문 실패 예외)가 발생한 시점에 테스트가 종료되기 때문에 모든 실패한 단언문을 볼 수는 없다. 코테스트에서는 assertSoftly 블럭을 사용해 이런 기본 동작을 우회할 수 있다. 이 블럭은 내부에서 발생한 AssertionError 예외를 잡아내서 누적시키면서 모든 단언문을 실행한다(다른 예외로 코드가 끝나는 경우는 제외). 블럭이 끝나면 assertSoftly는 누적시킨 모든 예외를 한 AssertionError에 넣고 호출한 쪽에 던진다. 예제를 살펴보자.

import io.kotest.assertions.assertSoftly  
import io.kotest.core.spec.style.StringSpec  
import io.kotest.inspectors.forAll  
import io.kotest.matchers.ints.shouldBeGreaterThan  
import io.kotest.matchers.ints.shouldBeLessThan  
  
class NumberTestWithForAll : StringSpec({  
 val numbers = Array(10) { it + 1 }  
 "invalid numbers" {  
 assertSoftly {  
 numbers.forAll { it shouldBeLessThan 5 }  
 numbers.forAll { it shouldBeGreaterThan 3 }  
 }  
 }  
})

assertSoftly()가 없다면 첫번째 forAll()이 실패하기 때문에 두번째 forAll() 단언문은 아예 실행되지도 않는다. 하지만 이 코드에서는 assertSoftly()가 있으므로 두 단언문 모두 실행되고 테스트가 예외와 함께 종료된다.

io.kotlintest.tables.MultiAssertionError:  
The following 9 assertions failed  
...

결과 메시지가 모든 실패를 나열하는 모습을 볼 수 있다.

### 비결정적 코드 테스트하기

몇번 시도를 거쳐야 테스트를 통과하는 경우가 있는 비결정적 코드를 다뤄야 한다면 타임 아웃과 반복 실행을 편리하게 처리할 수 있는 대안을 제공한다. eventually() 함수는 어떤 정해진 기간 안에 주어진 단언문을 만족시키는 경우가 한번이라도 생기는지 검사한다.

[eventually는 일시중단함수라서 코루틴 안에서만 호출되야 합니다. 문자열에 대한 invoke()를 하면 invoke가 코루틴 문맥을 제공해줍니다.]

import io.kotest.assertions.timing.eventually  
import io.kotest.core.spec.style.StringSpec  
import io.kotest.matchers.shouldBe  
import java.io.File  
import kotlin.time.Duration  
import kotlin.time.ExperimentalTime  
import kotlin.time.seconds  
  
@ExperimentalTime  
class StringSpecWithEventually : StringSpec({  
 "10초 안에 파일의 내용이 단 한줄인 경우가 있어야 함" {  
 eventually(Duration.seconds(10)) { // Duration.seconds(10)을 권장  
 // 주어진 기간동안 파일이 한 줄만 들어있는는 순간이 올 때까지 검사 (최대 10초)  
 File("data.txt").readLines().size shouldBe 1  
 }  
 }  
})

continually() 함수는 단언문이 최초 호출시 성립하고, 그 이후 지정한 기간동안 계속 성립한 채로 남아있는지 검사한다.

// import는 앞의 StringSpecWithEventually 예제와 동일함  
  
@ExperimentalTime  
class StringSpecWithEventually : StringSpec({  
 "10초동안 파일의 내용이 계속 한줄로 유지되야 함" {  
 continually(10.seconds) {  
 File("data.txt").readLines().size shouldBe 1  
 }  
 }  
})

## 속성 기반 테스트

[속성기반 테스트는 과거 버전과 많이 달라진 모양입니다. --;; 최신 버전에 맞춰 다시 썼습니다.]

코테스트는 여러분이 술어를 지정하면 코테스트가 자동으로 술어를 검증하기 위한 임의의 테스트 데이터를 생성해주는 속성 기반 테스트(property based test)를 지원한다. 이 기법은 수동으로 준비하고 유지하기기 어려운 큰 데이터 집합에 대해 성립해야 하는 조건을 테스트해야 하는 경우 유용하다. 단, 속성 기반 테스트를 사용하려면 kotest-property 모듈을 의존관계에 추가해야 한다. 메이븐 좌표는 "testImplementation 'io.kotest:kotest-property:4.5.0"이다.

예를 들어 두 수의 최솟값을 구하는 함수를 정의했다고 하자.

infix fun Int.min(n: Int) = if (this < n) this else n

결과가 두 객체(수신객체와 인자로 받은 객체)보다 항상 작거나 같은지 검사하고 싶다고 하자. 이를 위해서 단언문을 checkAll로 감쌀 수 있다.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.matchers.and  
import io.kotest.matchers.ints.beLessThanOrEqualTo  
import io.kotest.matchers.should  
import io.kotest.property.checkAll  
  
class NumberTestWithAssertAll : StringSpec({  
 "min" {  
 checkAll { a: Int, b: Int ->  
 (a min b).let {  
 it should (beLessThanOrEqualTo(a) and beLessThanOrEqualTo(b))  
 }  
 }  
 }  
}

코드를 실행하면 코테스트는 Int 쌍의 스트림을 생성해서 모든 쌍을 테스트의 단언문이나 식을 사용해 검사한다. 디폴트로 1000개의 테스트 데이터 집합 원소가 생성된다. 하지만 checkAll()에 원하는 데이터 집합의 크기를 넘길 수도 있다.

[assertNone은 현재 버전엔 없습니다. 어쩌면 저자가 과거 버전에서도 checkAll을 assertAll로 혼동했을 수도 있어요. 모든 단언문이 성립한다고 기술할 때 쓰는 assertAll이라는 매처가 있습니다. 그래서 여기 부분은 생략하고. forAll, forNone으로 넘어가면서 문장 순서를 좀 바꿨습니다.]

단언문 대신 불리언 값을 반환하는 식을 사용할 수도 있다. 이 경우, forAll 안에 불리언 값을 반환하는 람다를 넣으면 된다. 이 경우 생성한 모든 테스트 데이터에 대해 람다가 true를 반환해야 테스트가 성공한다. forAll과 반대로 모든 테스트 집합 원소에 대해 false를 반환할 때만 성공하는 검사가 필요하면 forNone을 사용할 수 있다.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.property.forAll  
  
class NumberTestWithAssertAll : StringSpec({  
 "min (단언문 대신 식 사용)" {  
 forAll{ a: Int, b: Int ->  
 (a min b).let { it <= a && it <= b }  
 }  
 }  
})

[생성기 관련 코드도 완전히 바뀌었습니다. --;;]

코테스트는 Int, Boolean, String 등 일반적으로 쓰이는 여러 타입에 대한 디폴트 생성기(generator)를 제공한다. 디폴트 생성기는 람다 파라미터의 타입에 대한 런타임 정보를 사용해 자동으로 생성기를 선택한다. 하지만 때로는 시스템이 선택해주는 생성기를 사용하고 싶지 않거나, 시스템이 지원하지 않는 타입의 데이터에 대한 속성 기반 테스트를 진행해야 할 때도 있다. 이런 경우에는 생성기를 직접 생성해야 한다. 코테스트 속성 기반 테스트의 생성기는 Gen이라는 추상 클래스를 상속하는데, 크게 임의의 값을 생성하는 생성기와 정해진 값을 모두 돌려주는 생성기로 나뉜다. 이 두 유형의 생성기를 표현하는 추상 클래스는 다음과 같다.

* Arb : Arb는 미리 하드코딩된 에지케이스(edge case)와 무한한 난수 샘플(random sample)을 생성해주는 생성기다. 생성기에 따라서는 에지케이스 테스트 데이터를 제공하지 않을 수도 있다. 테스트를 진행할 때는 디폴트로 2%는 에지케이스를, 98%는 난수 데이터를 발생시킨다.
* Exhaustive : Exhaustive는 주어진 공간에 속한 모든 데이터를 생성해준다. 해당 공간의 모든 값을 사용하는지 검사하고 싶을 때 Exhaustive 타입의 생성기가 유용하다.

직접 Arb나 Exhaustive를 상속해 추상 메서드를 구현하면 쉽게 생성기를 만들 수 있다. 예를 들어 *11장. 도메인 특화 언어*에서 만든 유리수를 표현하는 클래스에 대한 테스트를 작성해보자. 어떤 유리수를 자기 자신에게서 빼면 0이 나오는지 검사하고 싶다. 이런 경우 다음과 같이 Arb를 상속해서 임의의 유리수에 대한 테스트를 작성할 수 있다.

class RationalTest : StringSpec({  
 "Subtraction (Arb 사용)" {  
 forAll(  
 // 첫번째 인자로 Arb<Rational> 인스턴스를 넘김  
 object : Arb<Rational>() {  
 override fun edgecase(rs: RandomSource): Rational? = null // 에지 케이스 없음  
 override fun sample(rs: RandomSource): Sample<Rational> =  
 Sample(Rational.of(rs.random.nextInt(), rs.random.nextInt(0, Int.MAX\_VALUE)))  
 }  
 ){ a: Rational ->  
 (a - a).num == 0  
 }  
 }  
})

하지만 이보다 더 쉬운 방법이 있다. 바로 코테스트가 제공하는 기본 생성기를 조합하는 것이다. 우선, 코테스트에서 기본 제공하는 생성기 중 몇가지를 보면 다음과 같다.

* Arb.int(range), Arb.long(range), Arb.nats(range) ... : 범위(range)에 속한 수(이름을 보면 어떤 타입일지 추측이 가능하다)를 임의로 선택한다. 범위를 지정하지 않으면 이름이 암시하는 영역에 속하는 모든 수 값 중에 난수를 생성한다. 에지케이스를 제공하며, 보통 0,-1,+1,Int.MIN\_VALUE,Int.MAX\_VALUE 등의 값 중 의미가 있는 것이 선택된다.
* Exhaustive.ints(range), Exhaustive.longs(range) : 범위에 속한 모든 수를 테스트 데이터로 생성한다.
* Arb.string(range), Arb.stringPattern(pattern)... : 주어진 범위에 속하는 문자열이나 주어진 패턴(https://github.com/mifmif/Generex 에 있는 Generex로 기술함)에 부합하는 문자열을 생성한다.
* arb.orNull(), arb.orNull(nullProbability) : arb이 만들어내는 값에 널 값을 섞은 데이터를 생성한다. 널이될 확률을 지정할 수도 있다.

이런 제네레이터를 엮어서 다른 생성기를 만들거나, 제네레이터에서 값을 가져오는 연산을 제공한다. 우선 gen.next()을 통해 생성기로부터 다음 값을 가져올 수 있고, filter(), map(), flatMap(), merge() 등의 컬렉션 연산도 제공한다. 한편, 여러 제네레이터로부터 얻은 데이터를 한꺼번에 엮을 때 쓸 수 있는 bind() 메서드도 있다. 이런 연산을 통하면 기존 생성기를 조합한 새로운 생성기를 쉽게 얻을 수 있다.

이 bind 연산과 Arb.int()를 엮어서 앞의 유리수 테스트를 다시 작성해보면 다음과 같다.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.property.\*  
import io.kotest.property.arbitrary.bind  
import io.kotest.property.arbitrary.int  
  
class RationalTest : StringSpec({  
 val rationalArb = Arb.bind(Arb.int(),Arb.int(0,Int.MAX\_VALUE)){x,y->Rational.of(x,y)}  
  
 "Subtraction (Arb.int()와 Arb.bind() 사용)" {  
 forAll(rationalArb){ a: Rational ->  
 (a - a).num == 0  
 }  
 }  
})

한편 좀 더 편리하게 Arb나 Exhaustive를 구현할 수 있는 빌더 함수도 있다. Arb를 구현할 때는 arbitrary()를 사용하고, Exhaustive를 구현할 때는 리스트 객체에 대해 exhaustive() 확장 함수를 호출하면 된다. 앞의 유리수 테스트를 arbitrary()를 사용해 다시 구현해보자.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.property.\*  
import io.kotest.property.arbitrary.bind  
import io.kotest.property.arbitrary.int  
  
class RationalTest : StringSpec({  
 val rationalArb2 = arbitrary { Rational.of(it.random.nextInt(), it.random.nextInt(0,Int.MAX\_VALUE)) }  
  
 "Subtraction (arbitrary() 사용)" {  
 forAll(rationalArb2){ a: Rational ->  
 (a - a).num == 0  
 }  
 }  
})

속성 기반 테스트 프레임워크가 제공하는 데이터나 커스텀 생성기 대신 고정된 데이터 집합을 사용해 테스트를 진행할 수도 있다. io.kotest.data 패키지에는 데이터 기반 테스트를 지원하는 클래스와 함수들이 정의되어 있다. 예를 들어 io.kotest.data.forAll()을 사용하면 가변 길이의 행(row) 형태 객체를 활용해 테스트를 진행할 수도 있다.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.data.forAll  
import io.kotest.data.row  
  
class DataDrivenTest : StringSpec({  
 "Minimum" {  
 forAll(  
 row(1, 1),  
 row(1, 2),  
 row(2, 1)  
 ) { a: Int, b: Int ->  
 (a min b).let { it <= a && it <= b }  
 }  
 }  
})

속성 기반 테스트(io.kotest.property)와 데이터 기반 테스트(io.kotest.data)의 패키지 경로 차이에 주의하라. 데이터 기반 테스트는 기본 코테스트 외에 별도의 의존성을 지정하지 않아도 사용할 수 있지만, 속성 기반 테스트는 의존성을 지정해야 한다는 점도 다르다.

행만 사용하는 대신, 구체적인 헤더(header)를 제공하는 테이블 형태의 객체를 사용할 수도 있다. 테스트가 실패했을 때 적절한 맥락을 알려주기 위해 헤더를 사용한다. 예제를 살펴보자.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.data.\*  
import io.kotest.matchers.ints.shouldBeGreaterThanOrEqual  
  
class DataDrivenTest2 : StringSpec({  
 "Minimum" {  
 forAll(  
 table(  
 headers("name", "age"),  
 row("John", 20),  
 row("Harry", 25),  
 row("Bob", 16)  
 )  
 ) { name, age ->  
 age shouldBeGreaterThanOrEqual 18  
 }  
 }  
})

이 테스트를 실행하면 다음과 같은 오류 메시지를 표시하면서 테스트가 실패한다.

Test failed for (name, "Bob"), (age, 16) with error 16 should be >= 18

이 방식은 forAll()과 forNone()을 모두 지원한다. 하지만 이 두 함수는 속성 기반 테스트의 forAll()/forNone()과 달리 Unit을 반환하는 람다를 받는다. 그래서 람다 내부에서 불리언 값을 반환하는 대신 매처를 사용해야 한다.

## 픽스처와 설정

### 픽스처 제공하기

실제 테스트를 진행하기 위해 필요한 환경과 자원을 초기화하고 테스트가 끝나면 정리해야 하는 경우가 있다(이런 테스트 환경을 테스트 픽스처라고 말한다). 코테스트에서는 TestListener 인터페이스를 구현함으로써 이런 픽스처를 지정할 수 있다. TestListener 인터페이스는 BeforeTestListener 등 개별 픽스처 인터페이스를 한데 모아둔 리스너이다. 다음 코드를 보자. 이 코드만 별도로 실행할 수 있도록 fixture라는 디렉터리를 만들고 그 아래 파일을 저장하라. 인텔리J IDEA의 프로젝트 창에서 해당 디렉터리를 오른쪽 클릭한 후 Run 'Tests in 'fixture''를 선택하면 fixture 패키지에 들어있는 테스트만 실행할 수 있다.

[편집자께: Spec의 구조가 바뀌면서 디스크립션 필드가 없어졌습니다. 그리고 이후 프로젝트 리스너 관련 설명을 위해 구조를 좀 바꿨습니다.]

package fixutre   
  
import io.kotest.core.listeners.\*  
import io.kotest.core.spec.AutoScan  
import io.kotest.core.spec.Spec  
import io.kotest.core.spec.style.FunSpec  
import io.kotest.core.test.TestCase  
import io.kotest.core.test.TestResult  
import io.kotest.matchers.shouldBe  
import kotlin.reflect.KClass  
  
  
object SpecLevelListener : TestListener {  
 override suspend fun prepareSpec(kclass: KClass<out Spec>) {  
 println("PrepareSpec(in SpecLevelListener): ${kclass.qualifiedName}")  
 }  
  
 override suspend fun beforeSpec(spec: Spec) {  
 println("BeforeSpec: ${spec.materializeRootTests().joinToString { it.testCase.displayName }}")  
 }  
  
 override suspend fun beforeTest(testCase: TestCase) {  
 println("BeforeTest: ${testCase.displayName}")  
 }  
  
 override suspend fun afterTest(testCase: TestCase, result: TestResult) {  
 println("AfterTest: ${testCase.displayName}")  
 }  
  
 override suspend fun afterSpec(spec: Spec) {  
 println("AfterSpec: ${spec.materializeRootTests().joinToString { it.testCase.displayName }}")  
 }  
  
 override suspend fun finalizeSpec(kclass: KClass<out Spec>, results: Map<TestCase, TestResult>) {  
 println("FinalizeSpec(in SpecLevelListener): ${kclass.qualifiedName}")  
 }  
}  
  
class NumbersTestWithFixture1 : FunSpec() {  
 init {  
 context("Addition") {  
 test("2 + 2") {  
 2 + 2 shouldBe 4  
 }  
 test("4 \* 4") {  
 4 + 4 shouldBe 8  
 }  
 }  
 }  
  
 override fun listeners() = listOf(SpecLevelListener)  
}  
  
class NumbersTestWithFixture2 : FunSpec() {  
 init {  
 context("Multiplication") {  
 test("2 \* 2") {  
 2 \* 2 shouldBe 4  
 }  
 test("4 \* 4") {  
 4 \* 4 shouldBe 16  
 }  
 }  
 }  
  
 override fun listeners() = listOf(SpecLevelListener)  
}

이 테스트 코드를 실행하면 다음과 같은 출력을 볼 수 있다.

BeforeSpec: Addition  
BeforeTest: Addition  
BeforeTest: 2 + 2  
AfterTest: 2 + 2  
BeforeTest: 4 \* 4  
AfterTest: 4 \* 4  
AfterTest: Addition  
AfterSpec: Addition  
BeforeSpec: Multiplication  
BeforeTest: Multiplication  
BeforeTest: 2 \* 2  
AfterTest: 2 \* 2  
BeforeTest: 4 \* 4  
AfterTest: 4 \* 4  
AfterTest: Multiplication  
AfterSpec: Multiplication

[beforeSpec과 beforeTest가 언제 호출되는지에 대한 설명이 반대입니다.]

beforeSpec()와 beforeTest()의 차이는 (그리고 비슷하게 afterSpec()과 afterTest()의 차이도) beforeTest()은 테스트마다 실행되고, 테스트가 활성화된 경우에만 호출되는 반면 beforeSpec()은 어떤 명세가 인스턴스화 될 때 실행된다는 점에 있다. 따라서 테스트 함수가 실제 호출될 때 불려야 하는 준비/정리 코드는 beforeTest()/afterTest()를 쓰고, 명세 클래스의 인스턴스마다 하나씩 필요한 준비/정리 코드는 beforeSpec()/afterSpec()을 사용해야 한다. 다음 절에서는 설정을 사용해 개별 테스트를 비활성화하는 방법을 살펴본다. 한편 TestListener에 정의된 prepareSpec과 finalizeSpec은 개별 Spec안에서 listeners를 오버라이드하는 경우 정상작동하지 않음도 알 수 있다.

한편, 프로젝트(테스트를 실행할 때 모든 스팩 클래스가 모인 전체 테스트 모임을 프로젝트라고 부른다) 수준의 리스너인 beforeProject()/afterProject() 구현을 제공하고 싶다면 ProjectConfig 타입의 싱글턴 객체에 리스너를 등록해야 한다. 한편, 이 ProjectConfig 타입의 객체에 리스너를 등록할 때 prepareSpec()과 finalizeSpec() 메서드 구현도 함께 제공해보자. TestListener를 상속해서 이 두 메서드만 오버라이드하면 된다. 다음과 같이 리스너와 설정 객체를 정의할 수 있다.

import io.kotest.core.config.AbstractProjectConfig  
import io.kotest.core.listeners.\*  
import kotlin.reflect.KClass  
  
object MyProjectListener : ProjectListener, TestListener {  
 override val name: String = "MyProjectListener"  
  
 override suspend fun beforeProject() { println("Before project") }  
  
 override suspend fun afterProject() { println("After project") }  
  
 override suspend fun prepareSpec(kclass: KClass<out Spec>) {  
 println("PrepareSpec: ${kclass.qualifiedName}")  
 }  
  
 override suspend fun finalizeSpec(kclass: KClass<out Spec>, results: Map<TestCase, TestResult>) {  
 println("FinalizeSpec: ${kclass.qualifiedName}")  
 }  
}  
  
object ProjectConfig: AbstractProjectConfig() {  
 override fun listeners() = listOf(MyProjectListener)  
}

이렇게 리스너를 프로젝트 수준에서 설정하고 fixture 패키지의 테스트를 다시 실행해보면 다음과 같은 결과를 볼 수 있다.

Before project  
PrepareSpec: fixutre.NumbersTestWithFixture1  
(중간 생략. 이전 예제의 실행 예와 같음)  
FinalizeSpec: fixutre.NumbersTestWithFixture2  
After project

유용한 기능으로 AutoCloseable 인터페이스를 구현한 자원을 자동으로 해제해주는 기능이 있다. 이 기능을 작동시키려면 자원을 할당할 때 autoClose() 호출로 자원을 감싸야 한다.

import io.kotest.core.spec.style.FunSpec  
import io.kotest.matchers.shouldBe  
import java.io.FileReader  
  
class FileTest : FunSpec() {  
 val reader = autoClose(FileReader("data.txt"))  
   
 init {  
 test("Line count") {  
 reader.readLines().isNotEmpty() shouldBe true  
 }  
 }  
}

## 테스트 설정

코테스트는 테스트 환경을 설정할 수 있는 수단을 여러가지 제공한다. 특히 명세 함수가 제공하는 config() 함수를 통해 여러가지 테스트 실행 파라미터를 지정할 수 있다. config() 함수 사용법은 선택한 명세의 스타일에 따라 달라지지만, 일반적으로는 config()가 적용된 테스트 블럭의 동작을 변경한다.

import io.kotest.core.spec.style.BehaviorSpec  
import io.kotest.core.spec.style.ShouldSpec  
import io.kotest.core.spec.style.StringSpec  
import io.kotest.matchers.shouldBe  
  
class StringSpecWithConfig : StringSpec({  
 "2 + 2 should be 4".config(invocations = 10) { (2 + 2) shouldBe 4 }  
})  
class ShouldSpecWithConfig : ShouldSpec({  
 context("Addition") {  
 context("1 + 2") {  
 should("be equal to 3").config(threads = 2, invocations = 100) {  
 (1 + 2) shouldBe 3  
 }  
 should("be equal to 2 + 1").config(timeout = 1.minutes) {  
 (1 + 2) shouldBe (2 + 1)  
 }  
 }  
 }  
})  
  
class BehaviorSpecWithConfig : BehaviorSpec({  
 Given("Arithmetic") {  
 When("x is 1") {  
 val x = 1  
 And("increased by 1") {  
 then("result is 2").config(invocations = 100) {  
 (x + 1) shouldBe 2  
 }  
 }  
 }  
 }  
})

각각의 명시 스타일에 대한 문서에서 더 자세한 정보를 얻을 수 있다.

config() 함수를 사용해 조정할 수 있는 파라미터에 대해 살펴보자.

* invocations : 테스트 실행 회수. 모든 실행이 성공해야 테스트가 성공한 것으로 간주한다. 간헐적으로 실패하는 비결정적 테스트가 있을 때 이런 옵션이 유용하다
* threads : 테스트를 실행할 스레드 개수. invocations가 2 이상일 때만 이 옵션이 유용하다. 실행 회수가 1이라면 병렬화할 여지가 없다
* enabled : 테스트를 실행해야 할지 여부. false로 설정하면 테스트 실행을 비활성화한다
* timeout : 테스트 실행에 걸리는 최대 시간. 테스트 실행 시간이 이 타임아웃 값을 넘어서면 테스트가 종료되고 실패로 간주된다. 실행 회수 옵션과 마찬가지로 이 옵션도 비결정적 테스트에서 유용하다.

threads 옵션은 한 테스트케이스에 속한 개별 테스트를 병렬화할 때만 영향을 끼친다는 사실에 유의하라. 여러 테스트케이스를 병렬로 실행하고 싶다면 앞에서 이야기했던 AbstractProjectConfig를 사용해야 한다. AbstractProjectConfig의 parallelism 프로퍼티를 오버라이드해서 원하는 동시성 스레드 개수를 지정하라.

import io.kotest.core.config.AbstractProjectConfig  
  
object ProjectConfig : AbstractProjectConfig() {  
 override val parallelism = 4  
}

각 테스트를 개별적으로 설정하는 방법 외에도 defaultConfig() 함수를 오버라이드해서 한 명세에 속한 모든 테스트케이스의 설정을 한꺼번에 변경할 수도 있다.

import io.kotest.core.spec.style.StringSpec  
import io.kotest.core.test.TestCaseConfig  
import io.kotest.matchers.shouldBe  
  
class StringSpecWithConfig2 : StringSpec({  
 "2 + 2 should be 4" { (2 + 2) shouldBe 4 }  
}) {  
 override fun defaultConfig(): TestCaseConfig =  
 TestCaseConfig(invocations = 10, threads = 2)  
}

각 테스트에 대한 설정을 따로 명시하지 않으면 디폴트 설정 옵션이 자동으로 상속된다.

마지막으로 알려주고 싶은 코테스트의 기능으로 테스트 사이에 테스트케이스 인스턴스를 공유하는 방법을 지정하는 기능이 있다. 이를 격리 모드(isolation mode)라고 부른다. 디폴트로 모든 테스트 케이스는 한번만 인스턴스화되고 모든 테스트에 같은 인스턴스를 사용한다. 성능 측면에서는 이런 동작이 좋지만 일부 시나리오의 경우 이런 동작이 바람직하지 않을 수 있다. 테스트케이스에 속한 테스트들이 읽고 써야 하는 가변 상태가 있는 경우 인스턴스 공유가 문제가 될 수 있다. 이런 경우 테스트나 테스트 그룹을 시작할 때마다 테스트를 매번 인스턴스화하고 싶을 수도 있다. 이런 식의 동작(매번 인스턴스화)을 원한다면 isolationMode 프로퍼티를 오버라이드해야 한다. 이 프로퍼티는 IsolationMode 이넘에 속하는 세가지 상수 중 하나를 지정한다.

* SingleInstance : 테스트케이스의 인스턴스가 하나만 만들어진다. 이 옵션이 디폴트 동작이다
* InstancePerTest : 문맥이나 테스트 블럭이 실행될 때마다 테스트케이스의 새 인스턴스를 만든다
* InstancePerLeaf : (말단에 있는) 개별 테스트 블럭을 실행하기 전에 테스트가 인스턴스화된다

예제를 하나 살펴보자. 다음과 같은 FunSpec 스타일의 테스트 케이스가 있다고 하자.

import io.kotest.core.spec.style.FunSpec  
import io.kotest.matchers.shouldBe  
  
class IncTest : FunSpec() {  
 var x = 0  
 init {  
 context("Increment") {  
 println("Increment")  
 test("prefix") {  
 println("prefix")  
 ++x shouldBe 1  
 }  
 test("postfix") {  
 println("postfix")  
 x++ shouldBe 0  
 }  
 }  
 }  
}

이 테스트를 실행하면 두번째 테스트가 실패하는 모습을 볼 수 있다. 이유는 직전 테스트에서 대입한 값이 x 변수에 들어있기 때문이다. 다음 코드를 추가해 격리 모드를 InstancePerTest로 변경하자.

object IncTestProjectConfig : AbstractProjectConfig() {  
 override val isolationMode = IsolationMode.InstancePerTest  
}

두 테스트가 서로 다른 IncTest 인스턴스를 가지기 때문에 두 테스트 모두 통과한다. 표준 출력에 표시되는 메시지는 다음과 같다.

Increment  
Increment  
prefix  
Increment  
postfix

Increment가 세번 표시되는 이유는 IncTest가 세번 인스턴스화되기 때문이다. 첫번째는 문맥 블럭을 실행하기 위해서 초기화되고, 두번째는 문맥 블럭을 실행하고 그 안의 prefix 테스트를 실행하기 위해서, 세번째는 postfix 테스트를 실행하기 위해서 초기화된다(이 경우에도 먼저 문맥 블럭을 실행해야만 된다). 그 결과 문맥 블럭도 세번 실행된다.

격리 모드를 InstancePerLeaf로 하면 문맥 블럭 자체만 실행되는 일은 없어지고 개별 테스트를 실행할 때만 문맥 블럭을 실행한다. 그 결과 IncTest는 두번만 인스턴스화되며(한번은 prefix를 실행하기 위해서, 다른 한번은 postfix를 실행하기 위해서), 출력은 다음과 같다.

Increment  
prefix  
Increment  
postfix

지금까지 코테스트 프레임워크에 대해 다뤘다. 우리가 다룬 내용(그리고 우리가 다루지 않은 내용돌)에 대해 더 자세한 정보를 보고 싶은 독자는 다음 코테스트 문서를 참조하라.

* 깃헙: https://github.com/kotest/kotest
* 홈페이지: https://kotest.io

## 결론

이번 장에서는 코틀린을 사용해 작성된 애플리케이션을 위해 특별히 설계된 유명 오픈소스 테스트 프레임워크인 코테스트를 활용해 테스트 명세를 작성하는 기본적인 방법을 살펴봤다. 코테스트가 기본 제공하는 여러 스타일을 통해 테스트 명세를 작성하는 방법을 살펴보고, 매처와 인스펙터를 활용해 표현력이 풍부하고 읽기 쉬운 테스트를 작성하는 방법을 소개했으며, 테스트 데이터 집합을 기술하는 방법과 자동화된 속성 기반 테스트를 사용하는 방법을 보여줬다. 그리고 코테스트의 설정/정리 기능과 기본적인 테스트 관련 설정에 대해 설명했다. 이제 여러분은 직접 테스트 명세를 작성하고 코테스트나 다른 테스트 프레임워크가 제공하는 더 고급 기능을 활용할 수 있는 모든 지식을 갖췄다.

다음 장에서는 코틀린을 활용해 안드로이드 플랫폼 애플리케이션을 작성하는 방법에 대해 이야기한다. 안드로이드 스튜디오에서 프로젝트를 설정하는 방법과 기본적인 UI 및 액티비티 생명주기에 대해 설명하고, 안드로이드 확장고 안코(Anko) 프레임워크가 제공하는 여러 유용한 기능에 대해 소개한다.

## 질문

1. 코틀린을 지원하는 유명한 테스트 프레임워크의 개요를 설명하라.
2. 코테스트가 지원하는 명세 스타일에 대해 설명하라.
3. 매처란 무엇인가? 복잡한 단언문을 작성하기 위해 매처를 어떻게 조합하고 변환할 수 있는가?
4. 커스텀 코테스트 매처를 구현하는 방법을 설명하라.
5. shouldThrow() 함수에 대해 설명하라. 소프트 단언문이란 무엇인가?
6. 코테스트가 제공하는 컬렉션 인스펙터에 대해 설명하라.
7. eventually()와 continually() 함수의 의미에 대해 설명하라.
8. 코테스트에서 리스너를 사용해 테스트 관련 자원을 초기화하고 정리하는 방법에 대해 설명하라.
9. 코테스트에서 개별 테스트와 명세에 대한 설정을 어떻게 지정할 수 있는가? 전역 설정을 어떻게 지정할 수 있는가?
10. 코테스트에서 세가지 테스트케이스 격리 모드와 각각의 차이에 대해 설명하라.

1. 옮긴이 - 원서에는 코틀린테스트(KotlinTest)로 되어있었으나, 버전 4.2부터 코테스트로 이름이 변경됐다. [↑](#footnote-ref-1)